LM3S9U81 ROM

USER'S GUIDE

ROM-LM3S9U81-UG-461

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.ti.com/stellaris

Revision Information

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

Table of Contents

1 Introduction

The LM3S9U81 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_FIRESTORM_RA2
#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 23.

2 Boot Loader

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
- Ethernet using standard network protocols

Since the boot loader has no knowledge of the frequency of the attached crystal, or in fact if one is even present, it operates entirely from the internal oscillator. This is a 16 MHz clock, with an accuracy of +/- 1%.

The LM Flash Programmer GUI can be used to download an application via the boot loader over the UART or Ethernet 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. Since 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 Ethernet Interface

When using the Ethernet interface to communicate with the boot loader, the BOOTP and TFTP protocols are utilized. By using standard protocols, the boot loader will co-exist in a normal Ethernet environment without causing any problems (other than using a small amount of network bandwidth).

The bootstrap protocol (BOOTP), a predecessor to the DHCP protocol, is used to discover the IP address of the client, the IP address of the server, and the name of the firmware image to use. BOOTP uses UDP/IP packets to communicate between the client and the server; the boot loader acts as the client. First, it will send a BOOTP request using a broadcast message. When the server receives the request, it will reply, thereby informing the client of its IP address, the IP address of the server, and the name of the firmware image. Once this reply is received, the BOOTP protocol has completed.

Then, the trivial file transfer protocol (TFTP) is used to transfer the firmware image from the server to the client. TFTP also uses UDP/IP packets to communicate between the client and the server, and the boot loader also acts as the client in this protocol. As each data block is received, it is programmed into flash. Once all data blocks are received and programmed, the device is reset, causing it to start running the new firmware image.

The Ethernet controller will be configured to use the MAC address stored in the USER0/UART1 data registers, or if one is not programmed in USER0/USER1 it will use the default MAC address of 00:1a:b6:00:64:00. When there is data in USER0/USER1, it will be interpreted as a MAC address of U0B0:U0B1:U0B2:U1B0:U1B1:U1B2 (where U0B0 is USER0 bits 7-0, or byte 0, U0B1 is USER0 bits 15-8, or byte 1, and so on).

Note:
When using the Ethernet update, the boot loader can only program images to the beginning of memory since there is no mechanism in BOOTP to specify the address to program the image.

The following IETF specifications define the protocols used by the Ethernet update mechanism:

- RFC951 (`http://tools.ietf.org/html/rfc951.html`) defines the bootstrap protocol.
- RFC1350 (`http://tools.ietf.org/html/rfc1350.html`) defines the trivial file transfer protocol.

3 AES Data Tables

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

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)

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 sixteen 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)
- unsigned long ROM_ADCResolutionGet (unsigned long ulBase)
- void ROM_ADCResolutionSet (unsigned long ulBase, unsigned long ulResolution)
- 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 will be 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_ADCResolutionGet

Gets the setting of ADC resolution.

Prototype:

```
unsigned long
ROM_ADCResolutionGet(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_ADCResolutionGet` is a function pointer located at `ROM_ADCTABLE[27]`.

Parameters:

ulBase is the base address of the ADC module.

Description:

The ADC resolution is returned as one of **ADC_RES_12BIT** or **ADC_RES_10BIT**.

Returns:

The current setting of the ADC resolution.

5.2.1.19 ROM_ADCResolutionSet

Selects the ADC resolution.

Prototype:

```
void
ROM_ADCResolutionSet(unsigned long ulBase,
                     unsigned long ulResolution)
```

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_ADCResolutionSet` is a function pointer located at `ROM_ADCTABLE[26]`.

Parameters:

ulBase is the base address of the ADC module.
ulResolution is the ADC bit resolution.

Description:

The ADC resolution is set as specified by *ulResolution*. It must be one of **ADC_RES_12BIT** or **ADC_RES_10BIT**.

Returns:

None.

5.2.1.20 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_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.

5.2.1.21 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.

5.2.1.22 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.

5.2.1.23 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.

5.2.1.24 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.

5.2.1.25 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.

5.2.1.26 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_CH15**), 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_CH15** 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 will be returned by the ADC. Additionally, if differential mode is selected when the temperature sensor is being sampled, undefined results will be returned by the ADC.

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

Returns:

None.

5.2.1.27 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.

5.2.1.28 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)

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, will be 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 will be 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 will be (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 will be 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 will be 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 will be 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-16

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 needs to be validated. A CRC is preferred over a simple checksum (for example, XOR all bits) because it catches changes more readily.

There are a two CRC calculation routines available. Both implement the standard CRC-16 (also known as CRC-16-IBM) polynomial:

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

The first function, ROM_Crc16Array(), performs a CRC-16 calculation across all the bytes in the input data array. The other function, ROM_Crc16Array3(), performs 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 since 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_Crc16Array (unsigned long ulWordLen, unsigned long *pulData)
- void ROM_Crc16Array3 (unsigned long ulWordLen, unsigned long *pulData, unsigned short *pusCrc3)

7.2.1 Function Documentation

7.2.1.1 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.

7.2.1.2 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

8 Ethernet Controller

8.1 Introduction

The Stellaris Ethernet controller consists of a fully integrated media access controller (MAC) and a network physical (PHY) interface device. The Ethernet controller conforms to IEEE 802.3 specifications and fully supports 10BASE-T and 100BASE-TX standards.

The Ethernet API provides the set of functions required to implement an interrupt-driven Ethernet driver for this Ethernet controller. Functions are provided to configure and control the MAC, to access the register set on the PHY, to transmit and receive Ethernet packets, and to configure and control the interrupts that are available.

For any application, the ROM_EthernetInitExpClk() function must be called first to prepare the Ethernet controller for operation. This function will configure the Ethernet controller options that are based on system parameters, such as the system clock speed.

Once initialized, access to the PHY is available via the ROM_EthernetPHYRead() and ROM_EthernetPHYWrite() functions. By default, the PHY will auto-negotiate the line speed and duplex modes. For most applications, this will be sufficient. If a special configuration is required, the PHY read and write functions can be used to reconfigure the PHY to the desired mode of operation.

The MAC must also be configured using the ROM_EthernetConfigSet() function. The parameters for this function will allow the configuration of options such as Promiscuous Mode, Multicast Reception, Transmit Data Length Padding, and so on. The ROM_EthernetConfigGet() function can be used to query the current configuration of the Ethernet MAC.

The MAC address, used for incoming packet filtering, must also be programmed using the ROM_EthernetMACAddrSet() function. The current value can be queried using the ROM_EthernetMACAddrGet() function.

When configuration has been completed, the Ethernet controller can be enabled using the ROM_EthernetEnable() function. When getting ready to terminate operations on the Ethernet controller, the ROM_EthernetDisable() function may be called.

After the Ethernet controller has been enabled, Ethernet frames can be transmitted and received using the ROM_EthernetPacketPut() and ROM_EthernetPacketGet() functions. Care must be taken when using these functions, as they are blocking functions, and will not return until data is available (for RX) or buffer space is available (for TX). The ROM_EthernetSpaceAvail() and ROM_EthernetPacketAvail() functions can be called to determine if there is room for a TX packet or if there is an RX packet available prior to calling these blocking functions. Alternatively, the ROM_EthernetPacketGetNonBlocking() and ROM_EthernetPacketPutNonBlocking() functions will return immediately if a packet cannot be processed. Otherwise, the packet will be processed normally.

When developing a mapping layer for a TCP/IP stack, you may wish to use the interrupt capability of the Ethernet controller. The ROM_EthernetIntEnable() and ROM_EthernetIntDisable() functions are used to manipulate the individual interrupt sources available in the Ethernet controller (for exam-

ple, RX Error, TX Complete). The ROM_EthernetIntStatus() and ROM_EthernetIntClear() functions would be used to query the active interrupts to determine which process to service, and to clear the indicated interrupts prior to returning from the registered ISR.

8.2 Functions

Functions

- unsigned long ROM_EthernetConfigGet (unsigned long ulBase)
- void ROM_EthernetConfigSet (unsigned long ulBase, unsigned long ulConfig)
- void ROM_EthernetDisable (unsigned long ulBase)
- void ROM_EthernetEnable (unsigned long ulBase)
- void ROM_EthernetInitExpClk (unsigned long ulBase, unsigned long ulEthClk)
- void ROM_EthernetIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_EthernetIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_EthernetIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long ROM_EthernetIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_EthernetMACAddrGet (unsigned long ulBase, unsigned char *pucMACAddr)
- void ROM_EthernetMACAddrSet (unsigned long ulBase, unsigned char *pucMACAddr)
- tBoolean ROM_EthernetPacketAvail (unsigned long ulBase)
- long ROM_EthernetPacketGet (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long ROM_EthernetPacketGetNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long ROM_EthernetPacketPut (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- long ROM_EthernetPacketPutNonBlocking (unsigned long ulBase, unsigned char *pucBuf, long lBufLen)
- void ROM_EthernetPHYPowerOff (unsigned long ulBase)
- void ROM_EthernetPHYPowerOn (unsigned long ulBase)
- unsigned long ROM_EthernetPHYRead (unsigned long ulBase, unsigned char ucRegAddr)
- void ROM_EthernetPHYWrite (unsigned long ulBase, unsigned char ucRegAddr, unsigned long ulData)
- tBoolean ROM_EthernetSpaceAvail (unsigned long ulBase)
- void ROM_UpdateEthernet (void)

8.2.1 Function Documentation

8.2.1.1 ROM_EthernetConfigGet

Gets the current configuration of the Ethernet controller.

Prototype:

```
unsigned long
ROM_EthernetConfigGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetConfigGet` is a function pointer located at `ROM_ETHERNETTABLE[3]`.

Parameters:

ulBase is the base address of the controller.

Description:

This function will query the control registers of the Ethernet controller and return a bit-mapped configuration value.

See also:

The description of the ROM_EthernetConfigSet() function provides detailed information for the bit-mapped configuration values that will be returned.

Returns:

Returns the bit-mapped Ethernet controller configuration value.

8.2.1.2 ROM_EthernetConfigSet

Sets the configuration of the Ethernet controller.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetConfigSet` is a function pointer located at `ROM_ETHERNETTABLE[2]`.

Parameters:

ulBase is the base address of the controller.

ulConfig is the configuration for the controller.

Description:

After the ROM_EthernetInitExpClk() function has been called, this API function can be used to configure the various features of the Ethernet controller.

The Ethernet controller provides three control registers that are used to configure the controller's operation. The transmit control register provides settings to enable full duplex operation, to auto-generate the frame check sequence, and to pad the transmit packets to the minimum length as required by the IEEE standard. The receive control register provides settings to enable reception of packets with bad frame check sequence values and to enable multi-cast or promiscuous modes.

The *ulConfig* parameter is the logical OR of the following values:

- **ETH_CFG_RX_BADCRCDIS** - Disable reception of packets with a bad CRC
- **ETH_CFG_RX_PRMSEN** - Enable promiscuous mode reception (all packets)
- **ETH_CFG_RX_AMULEN** - Enable reception of multicast packets

- **ETH_CFG_TX_DPLXEN** - Enable full duplex transmit mode
- **ETH_CFG_TX_CRCEN** - Enable transmit with auto CRC generation
- **ETH_CFG_TX_PADEN** - Enable padding of transmit data to minimum size

These bit-mapped values are programmed into the transmit, receive, and/or timestamp control register.

Returns:
None.

8.2.1.3 ROM_EthernetDisable

Disables the Ethernet controller.

Prototype:
```
void
ROM_EthernetDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetDisable` is a function pointer located at `ROM_ETHERNETTABLE[7]`.

Parameters:
ulBase is the base address of the controller.

Description:
When terminating operations on the Ethernet interface, this function should be called. This function will disable the transmitter and receiver, and will clear out the receive FIFO.

Returns:
None.

8.2.1.4 ROM_EthernetEnable

Enables the Ethernet controller for normal operation.

Prototype:
```
void
ROM_EthernetEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetEnable` is a function pointer located at `ROM_ETHERNETTABLE[6]`.

Parameters:
ulBase is the base address of the controller.

Description:

Once the Ethernet controller has been configured using the ROM_EthernetConfigSet() function and the MAC address has been programmed using the ROM_EthernetMACAddrSet() function, this API function can be called to enable the controller for normal operation.

This function will enable the controller's transmitter and receiver, and will reset the receive FIFO.

Returns:

None.

8.2.1.5 ROM_EthernetInitExpClk

Initializes the Ethernet controller for operation.

Prototype:

```
void
ROM_EthernetInitExpClk(unsigned long ulBase,
                       unsigned long ulEthClk)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetInitExpClk` is a function pointer located at `ROM_ETHERNETTABLE[1]`.

Parameters:

ulBase is the base address of the controller.

ulEthClk is the rate of the clock supplied to the Ethernet module.

Description:

This function will prepare the Ethernet controller for first time use in a given hardware/software configuration. This function should be called before any other Ethernet API functions are called.

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

Note:

If the device configuration is changed (for example, the system clock is reprogrammed to a different speed), then the Ethernet controller must be disabled by calling the ROM_EthernetDisable() function and the controller must be reinitialized by calling the ROM_EthernetInitExpClk() function again. After the controller has been reinitialized, the controller should be reconfigured using the appropriate Ethernet API calls.

Returns:

None.

8.2.1.6 ROM_EthernetIntClear

Clears Ethernet interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetIntClear` is a function pointer located at `ROM_ETHERNETTABLE[0]`.

Parameters:

ulBase is the base address of the controller.

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

Description:

The specified Ethernet 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_EthernetIntEnable().

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.

8.2.1.7 ROM_EthernetIntDisable

Disables individual Ethernet interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetIntDisable` is a function pointer located at `ROM_ETHERNETTABLE[15]`.

Parameters:

ulBase is the base address of the controller.

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

Description:

Disables the indicated Ethernet 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_EthernetIntEnable().

Returns:
None.

8.2.1.8 ROM_EthernetIntEnable

Enables individual Ethernet interrupt sources.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetIntEnable` is a function pointer located at `ROM_ETHERNETTABLE[14]`.

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

Description:
Enables the indicated Ethernet 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:

- **ETH_INT_PHY** - An interrupt from the PHY has occurred. The integrated PHY supports a number of interrupt conditions. The PHY register, PHY_MR17, must be read to determine which PHY interrupt has occurred. This register can be read using the ROM_EthernetPHYRead() API function.
- **ETH_INT_MDIO** - This interrupt indicates that a transaction on the management interface has completed successfully.
- **ETH_INT_RXER** - This interrupt indicates that an error has occurred during reception of a frame. This error can indicate a length mismatch, a CRC failure, or an error indication from the PHY.
- **ETH_INT_RXOF** - This interrupt indicates that a frame has been received that exceeds the available space in the RX FIFO.
- **ETH_INT_TX** - This interrupt indicates that the packet stored in the TX FIFO has been successfully transmitted.
- **ETH_INT_TXER** - This interrupt indicates that an error has occurred during the transmission of a packet. This error can be either a retry failure during the back-off process, or an invalid length stored in the TX FIFO.
- **ETH_INT_RX** - This interrupt indicates that one (or more) packets are available in the RX FIFO for processing.

Returns:
None.

8.2.1.9 ROM_EthernetIntStatus

Gets the current Ethernet interrupt status.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetIntStatus` is a function pointer located at `ROM_ETHERNETTABLE[16]`.

Parameters:

ulBase is the base address of the controller.

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 Ethernet controller. 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_EthernetIntEnable().

8.2.1.10 ROM_EthernetMACAddrGet

Gets the MAC address of the Ethernet controller.

Prototype:

```
void
ROM_EthernetMACAddrGet(unsigned long ulBase,
                       unsigned char *pucMACAddr)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetMACAddrGet` is a function pointer located at `ROM_ETHERNETTABLE[5]`.

Parameters:

ulBase is the base address of the controller.

pucMACAddr is the pointer to the location in which to store the array of MAC-48 address octets.

Description:

This function will read the currently programmed MAC address into the *pucMACAddr* buffer.

See also:

Refer to ROM_EthernetMACAddrSet() API description for more details about the MAC address format.

Returns:
None.

8.2.1.11 ROM_EthernetMACAddrSet

Sets the MAC address of the Ethernet controller.

Prototype:
```
void
ROM_EthernetMACAddrSet(unsigned long ulBase,
                       unsigned char *pucMACAddr)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetMACAddrSet` is a function pointer located at `ROM_ETHERNETTABLE[4]`.

Parameters:
ulBase is the base address of the controller.
pucMACAddr is the pointer to the array of MAC-48 address octets.

Description:
This function will program the IEEE-defined MAC-48 address specified in *pucMACAddr* into the Ethernet controller. This address is used by the Ethernet controller for hardware-level filtering of incoming Ethernet packets (when promiscuous mode is not enabled).

The MAC-48 address is defined as 6 octets, illustrated by the following example address. The numbers are shown in hexadecimal format.

AC-DE-48-00-00-80

In this representation, the first three octets (AC-DE-48) are the Organizationally Unique Identifier (OUI). This is a number assigned by the IEEE to an organization that requests a block of MAC addresses. The last three octets (00-00-80) are a 24-bit number managed by the OUI owner to uniquely identify a piece of hardware within that organization that is to be connected to the Ethernet.

In this representation, the octets are transmitted from left to right, with the “AC” octet being transmitted first and the “80” octet being transmitted last. Within an octet, the bits are transmitted LSB to MSB. For this address, the first bit to be transmitted would be “0”, the LSB of “AC”, and the last bit to be transmitted would be “1”, the MSB of “80”.

Returns:
None.

8.2.1.12 ROM_EthernetPacketAvail

Check for packet available from the Ethernet controller.

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPacketAvail` is a function pointer located at `ROM_ETHERNETTABLE[8]`.

Parameters:

ulBase is the base address of the controller.

Description:

The Ethernet controller provides a register that contains the number of packets available in the receive FIFO. When the last bytes of a packet are successfully received (that is, the frame check sequence bytes), the packet count is incremented. Once the packet has been fully read (including the frame check sequence bytes) from the FIFO, the packet count will be decremented.

Returns:

Returns **true** if there are one or more packets available in the receive FIFO, including the current packet being read, and **false** otherwise.

8.2.1.13 ROM_EthernetPacketGet

Waits for a packet from the Ethernet controller.

Prototype:

```
long
ROM_EthernetPacketGet(unsigned long ulBase,
                      unsigned char *pucBuf,
                      long lBufLen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPacketGet` is a function pointer located at `ROM_ETHERNETTABLE[11]`.

Parameters:

ulBase is the base address of the controller.
pucBuf is the pointer to the packet buffer.
lBufLen is the maximum number of bytes to be read into the buffer.

Description:

This function reads a packet from the receive FIFO of the controller and places it into *pucBuf*. The function will wait until a packet is available in the FIFO. Then the function will read the entire packet from the receive FIFO. If there are more bytes in the packet than will fit into *pucBuf* (as specified by *lBufLen*), the function will return the negated length of the packet and the buffer will contain *lBufLen* bytes of the packet. Otherwise, the function will return the length of the packet that was read and *pucBuf* will contain the entire packet (excluding the frame check sequence bytes).

Note:

This function is blocking and will not return until a packet arrives.

Returns:

Returns the negated packet length **-n** if the packet is too large for *pucBuf*, and returns the packet length **n** otherwise.

8.2.1.14 ROM_EthernetPacketGetNonBlocking

Receives a packet from the Ethernet controller.

Prototype:

```
long
ROM_EthernetPacketGetNonBlocking(unsigned long ulBase,
                                 unsigned char *pucBuf,
                                 long lBufLen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPacketGetNonBlocking` is a function pointer located at `ROM_ETHERNETTABLE[10]`.

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is the maximum number of bytes to be read into the buffer.

Description:

This function reads a packet from the receive FIFO of the controller and places it into *pucBuf*. If no packet is available the function will return immediately. Otherwise, the function will read the entire packet from the receive FIFO. If there are more bytes in the packet than will fit into *pucBuf* (as specified by *lBufLen*), the function will return the negated length of the packet and the buffer will contain *lBufLen* bytes of the packet. Otherwise, the function will return the length of the packet that was read and *pucBuf* will contain the entire packet (excluding the frame check sequence bytes).

Note:

This function will return immediately if no packet is available.

Returns:

Returns **0** if no packet is available, the negated packet length **-n** if the packet is too large for *pucBuf*, and the packet length **n** otherwise.

8.2.1.15 ROM_EthernetPacketPut

Waits to send a packet from the Ethernet controller.

Prototype:

```
long
ROM_EthernetPacketPut(unsigned long ulBase,
                      unsigned char *pucBuf,
                      long lBufLen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPacketPut` is a function pointer located at `ROM_ETHERNETTABLE[13]`.

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is number of bytes in the packet to be transmitted.

Description:

This function writes *lBufLen* bytes of the packet contained in *pucBuf* into the transmit FIFO of the controller and then activates the transmitter for this packet. This function will wait until the transmit FIFO is empty. Once space is available, the function will return once *lBufLen* bytes of the packet have been placed into the FIFO and the transmitter has been started. The function will not wait for the transmission to complete. The function will return the negated *lBufLen* if the length is larger than the space available in the transmit FIFO.

Note:

This function blocks and will wait until space is available for the transmit packet before returning.

Returns:

Returns the negated packet length **-lBufLen** if the packet is too large for FIFO, and the packet length **lBufLen** otherwise.

8.2.1.16 ROM_EthernetPacketPutNonBlocking

Sends a packet to the Ethernet controller.

Prototype:

```
long
ROM_EthernetPacketPutNonBlocking(unsigned long ulBase,
                                 unsigned char *pucBuf,
                                 long lBufLen)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPacketPutNonBlocking` is a function pointer located at `ROM_ETHERNETTABLE[12]`.

Parameters:

ulBase is the base address of the controller.

pucBuf is the pointer to the packet buffer.

lBufLen is number of bytes in the packet to be transmitted.

Description:

This function writes *lBufLen* bytes of the packet contained in *pucBuf* into the transmit FIFO of the controller and then activates the transmitter for this packet. If no space is available in the FIFO, the function will return immediately. If space is available, the function will return once *lBufLen* bytes of the packet have been placed into the FIFO and the transmitter has been started. The function will not wait for the transmission to complete. The function will return the negated *lBufLen* if the length is larger than the space available in the transmit FIFO.

Note:

This function does not block and will return immediately if no space is available for the transmit packet.

Returns:

Returns **0** if no space is available in the transmit FIFO, the negated packet length **-lBufLen** if the packet is too large for FIFO, and the packet length **lBufLen** otherwise.

8.2.1.17 ROM_EthernetPHYPowerOff

Powers off the Ethernet PHY.

Prototype:

```
void
ROM_EthernetPHYPowerOff(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPHYPowerOff` is a function pointer located at `ROM_ETHERNETTABLE[21]`.

Parameters:

ulBase is the base address of the controller.

Description:

This function will power off the Ethernet PHY, reducing the current consuption of the device. While in the powered off state, the Ethernet controller will be unable to connect to the Ethernet.

Returns:

None.

8.2.1.18 ROM_EthernetPHYPowerOn

Powers on the Ethernet PHY.

Prototype:

```
void
ROM_EthernetPHYPowerOn(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPHYPowerOn` is a function pointer located at `ROM_ETHERNETTABLE[22]`.

Parameters:

ulBase is the base address of the controller.

Description:

This function will power on the Ethernet PHY, enabling it return to normal operation. By default, the PHY is powered on, so this function only needs to be called if ROM_EthernetPHYPowerOff() has previously been called.

Returns:

None.

8.2.1.19 ROM_EthernetPHYRead

Reads from a PHY register.

Prototype:

```
unsigned long
ROM_EthernetPHYRead(unsigned long ulBase,
                    unsigned char ucRegAddr)
```

ROM Location:

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

Parameters:

ulBase is the base address of the controller.
ucRegAddr is the address of the PHY register to be accessed.

Description:

This function will return the contents of the PHY register specified by *ucRegAddr*.

Returns:

Returns the 16-bit value read from the PHY.

8.2.1.20 ROM_EthernetPHYWrite

Writes to the PHY register.

Prototype:

```
void
ROM_EthernetPHYWrite(unsigned long ulBase,
                     unsigned char ucRegAddr,
                     unsigned long ulData)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetPHYWrite` is a function pointer located at `ROM_ETHERNETTABLE[17]`.

Parameters:

ulBase is the base address of the controller.
ucRegAddr is the address of the PHY register to be accessed.
ulData is the data to be written to the PHY register.

Description:

This function will write the *ulData* to the PHY register specified by *ucRegAddr*.

Returns:

None.

8.2.1.21 ROM_EthernetSpaceAvail

Checks for packet space available in the Ethernet controller.

Prototype:

```
tBoolean
ROM_EthernetSpaceAvail(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_EthernetSpaceAvail` is a function pointer located at `ROM_ETHERNETTABLE[9]`.

Parameters:

ulBase is the base address of the controller.

Description:

The Ethernet controller's transmit FIFO is designed to support a single packet at a time. After the packet has been written into the FIFO, the transmit request bit must be set to enable the transmission of the packet. Only after the packet has been transmitted can a new packet be written into the FIFO. This function will simply check to see if a packet is in progress. If so, there is no space available in the transmit FIFO.

Returns:

Returns **true** if a space is available in the transmit FIFO, and **false** otherwise.

8.2.1.22 ROM_UpdateEthernet

Starts an update over the Ethernet interface.

Prototype:

```
void
ROM_UpdateEthernet(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_ETHERNETTABLE` is an array of pointers located at `ROM_APITABLE[15]`.
`ROM_UpdateEthernet` is a function pointer located at `ROM_ETHERNETTABLE[19]`.

Description:

Calling this function commences an update of the firmware via the Ethernet interface. This function assumes that the Ethernet interface has already been configured, had its MAC address programmed, and is currently operational. The BOOTP requests that are generated will have the server name field set to "stellaris".

Returns:

Never returns.

9 External Peripheral Interface (EPI)

9.1 Introduction

The EPI API provides functions to use the EPI module available in the Stellaris microcontroller. The EPI module provides a physical interface for external peripherals and memories. The EPI can be configured to support several types of external interfaces and different sized address and data buses.

Some features of the EPI module are:

- configurable interface modes including SDRAM, HostBus, and simple read/write protocols
- configurable address and data sizes
- configurable bus cycle timing
- blocking and non-blocking reads and writes
- FIFO for streaming reads
- interrupt and uDMA support

The function ROM_EPIModeSet() is used to select the interface mode. The clock divider is set with the ROM_EPIDividerSet() function which will determine the speed of the external bus. The external device is mapped into the processor memory or peripheral space using the ROM_EPIAddressMapSet() function.

Once the mode is selected, the interface is configured with one of the configuration functions. If SDRAM mode was chosen, the function ROM_EPIConfigSDRAMSet() is used to configure the SDRAM interface. If Host-bus 8 mode was chosen, the function ROM_EPIConfigHB8Set() is used to configure the Host-bus 8 interface. If Host-bus 16 mode was chosen, the function ROM_EPIConfigHB16Set() is used to configure the Host-bus 16 interface. If general-purpose mode was chosen, then the function ROM_EPIConfigGPModeSet() is used to configure the general-purpose interface.

After the mode has been selected and configured, then the device can be accessed by reading and writing to the memory or peripheral address space that was programmed with ROM_EPIAddressMapSet().

There are more sophisticated ways to use the read/write interface. When an application is writing to the mapped memory or peripheral space, the writes will stall the processor until the write to the external interface is completed. However, the EPI contains an internal transaction FIFO and can buffer up to 4 pending writes without stalling the processor. Prior to writing, the application can test to see if the EPI can take more write operations without stalling the processor by using the function ROM_EPIWriteFIFOCountGet() which will return the number of non-blocking writes that can be made.

For efficient reads from the external device, the EPI contains a programmable read FIFO. This can be used to set a starting address and a count, and the FIFO will perform sequential reads from the device and store the values in the FIFO. The application can then periodically drain the FIFO either by polling, or by interrupts, or by using the uDMA controller. A

non-blocking read is configured by using the function ROM_EPINonBlockingReadConfigure(). The read operation is started with ROM_EPINonBlockingReadStart() and can be stopped by calling ROM_EPINonBlockingReadStop(). The function ROM_EPINonBlockingReadCount() can be used to determine the number of items remaining to be read, while the function ROM_EPINonBlockingReadAvail() returns the number of items in the FIFO that can be read immediately without stalling. There are 3 functions available for reading data from the FIFO and into a buffer provided by the application. These functions are ROM_EPINonBlockingReadGet32(), ROM_EPINonBlockingReadGet16(), ROM_EPINonBlockingReadGet8(), to read the data from the FIFO as 32-bit, 16-bit, or 8-bit data items.

The read FIFO and write transaction FIFO can be configured with the function ROM_EPIFIFOConfig(). This function is used to set the FIFO trigger levels, and to enable error interrupts to be generated when a read or write is stalled.

Interrupts are enabled or disabled with the functions ROM_EPIIntEnable() and ROM_EPIIntDisable(). The interrupt status can be read by calling ROM_EPIIntStatus(). If there is an error interrupt pending, the cause of the error can be determined with the function ROM_EPIIntErrorStatus(). The error can then be cleared with ROM_EPIIntErrorClear().

9.2 Functions

Functions

- void ROM_EPIAddressMapSet (unsigned long ulBase, unsigned long ulMap)
- void ROM_EPIConfigGPModeSet (unsigned long ulBase, unsigned long ulConfig, unsigned long ulFrameCount, unsigned long ulMaxWait)
- void ROM_EPIConfigHB16Set (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMaxWait)
- void ROM_EPIConfigHB8Set (unsigned long ulBase, unsigned long ulConfig, unsigned long ulMaxWait)
- void ROM_EPIConfigSDRAMSet (unsigned long ulBase, unsigned long ulConfig, unsigned long ulRefresh)
- void ROM_EPIDividerSet (unsigned long ulBase, unsigned long ulDivider)
- void ROM_EPIFIFOConfig (unsigned long ulBase, unsigned long ulConfig)
- void ROM_EPIIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_EPIIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_EPIIntErrorClear (unsigned long ulBase, unsigned long ulErrFlags)
- unsigned long ROM_EPIIntErrorStatus (unsigned long ulBase)
- unsigned long ROM_EPIIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_EPIModeSet (unsigned long ulBase, unsigned long ulMode)
- unsigned long ROM_EPINonBlockingReadAvail (unsigned long ulBase)
- void ROM_EPINonBlockingReadConfigure (unsigned long ulBase, unsigned long ulChannel, unsigned long ulDataSize, unsigned long ulAddress)
- unsigned long ROM_EPINonBlockingReadCount (unsigned long ulBase, unsigned long ulChannel)
- unsigned long ROM_EPINonBlockingReadGet16 (unsigned long ulBase, unsigned long ulCount, unsigned short *pusBuf)
- unsigned long ROM_EPINonBlockingReadGet32 (unsigned long ulBase, unsigned long ulCount, unsigned long *pulBuf)

- unsigned long ROM_EPINonBlockingReadGet8 (unsigned long ulBase, unsigned long ulCount, unsigned char *pucBuf)
- void ROM_EPINonBlockingReadStart (unsigned long ulBase, unsigned long ulChannel, unsigned long ulCount)
- void ROM_EPINonBlockingReadStop (unsigned long ulBase, unsigned long ulChannel)
- unsigned long ROM_EPIWriteFIFOCountGet (unsigned long ulBase)

9.2.1 Function Documentation

9.2.1.1 ROM_EPIAddressMapSet

Configures the address map for the external interface.

Prototype:

```
void
ROM_EPIAddressMapSet(unsigned long ulBase,
                     unsigned long ulMap)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIAddressMapSet` is a function pointer located at `ROM_EPITABLE[7]`.

Parameters:

ulBase is the EPI module base address.

ulMap is the address mapping configuration.

Description:

This function is used to configure the address mapping for the external interface. This determines the base address of the external memory or device within the processor peripheral and/or memory space.

The parameter *ulMap* is the logical OR of the following:

- **EPI_ADDR_PER_SIZE_256B**, **EPI_ADDR_PER_SIZE_64KB**, **EPI_ADDR_PER_SIZE_16MB**, or **EPI_ADDR_PER_SIZE_512MB** to choose a peripheral address space of 256 bytes, 64 Kbytes, 16 Mbytes or 512 Mbytes
- **EPI_ADDR_PER_BASE_NONE**, **EPI_ADDR_PER_BASE_A**, or **EPI_ADDR_PER_BASE_C** to choose the base address of the peripheral space as none, 0xA0000000, or 0xC0000000
- **EPI_ADDR_RAM_SIZE_256B**, **EPI_ADDR_RAM_SIZE_64KB**, **EPI_ADDR_RAM_SIZE_16MB**, or **EPI_ADDR_RAM_SIZE_512MB** to choose a RAM address space of 256 bytes, 64 Kbytes, 16 Mbytes or 512 Mbytes
- **EPI_ADDR_RAM_BASE_NONE**, **EPI_ADDR_RAM_BASE_6**, or **EPI_ADDR_RAM_BASE_8** to choose the base address of the RAM space as none, 0x60000000, or 0x80000000

Returns:

None.

9.2.1.2 ROM_EPIConfigGPModeSet

Configures the interface for general-purpose mode operation.

Prototype:

```
void
ROM_EPIConfigGPModeSet(unsigned long ulBase,
                       unsigned long ulConfig,
                       unsigned long ulFrameCount,
                       unsigned long ulMaxWait)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIConfigGPModeSet` is a function pointer located at `ROM_EPITABLE[4]`.

Parameters:

ulBase is the EPI module base address.

ulConfig is the interface configuration.

ulFrameCount is the frame size in clocks, if the frame signal is used (0-15).

ulMaxWait is the maximum number of external clocks to wait when the external clock enable is holding off the transaction (0-255).

Description:

This function is used to configure the interface when used in general-purpose operation as chosen with the function ROM_EPIModeSet(). The parameter *ulConfig* is the logical OR of any of the following:

- **EPI_GPMODE_CLKPIN** - interface clock is output on a pin
- **EPI_GPMODE_CLKGATE** - clock is stopped when there is no transaction, otherwise it is free-running
- **EPI_GPMODE_RDYEN** - the external peripheral drives an iRDY signal into pin EPI0S27. If absent, the peripheral is assumed to be ready at all times. This flag may only be used with a free-running clock (**EPI_GPMODE_CLKGATE** is absent).
- **EPI_GPMODE_FRAMEPIN** - framing signal is emitted on a pin
- **EPI_GPMODE_FRAME50** - framing signal is 50/50 duty cycle, otherwise it is a pulse
- **EPI_GPMODE_READWRITE** - read and write strobes are emitted on pins
- **EPI_GPMODE_WRITE2CYCLE** - a two cycle write is used, otherwise a single-cycle write is used
- **EPI_GPMODE_READ2CYCLE** - a two cycle read is used, otherwise a single-cycle read is used
- **EPI_GPMODE_ASIZE_NONE**, **EPI_GPMODE_ASIZE_4**, **EPI_GPMODE_ASIZE_12**, or **EPI_GPMODE_ASIZE_20** to choose no address bus, or and address bus size of 4, 12, or 20 bits
- **EPI_GPMODE_DSIZE_8**, **EPI_GPMODE_DSIZE_16**, **EPI_GPMODE_DSIZE_24**, or **EPI_GPMODE_DSIZE_32** to select a data bus size of 8, 16, 24, or 32 bits
- **EPI_GPMODE_WORD_ACCESS** - use Word Access mode to route bytes to the correct byte lanes allowing data to be stored in the upper bits of the word when necessary.

The parameter *ulFrameCount* is the number of clocks used to form the framing signal, if the framing signal is used. The behavior depends on whether the frame signal is a pulse or a

50/50 duty cycle. This value is not used if the framing signal is not enabled with the option **EPI_GPMODE_FRAMEPIN**.

The parameter *ulMaxWait* is used if the external clock enable is turned on with the **EPI_GPMODE_CLKENA** option is used. In the case that external clock enable is used, this parameter determines the maximum number of clocks to wait when the external clock enable signal is holding off a transaction. A value of 0 means to wait forever. If a non-zero value is used and exceeded, an interrupt will occur and the transaction aborted.

Returns:

None.

9.2.1.3 ROM_EPIConfigHB16Set

Configures the interface for Host-bus 16 operation.

Prototype:

```
void
ROM_EPIConfigHB16Set(unsigned long ulBase,
                     unsigned long ulConfig,
                     unsigned long ulMaxWait)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIConfigHB16Set` is a function pointer located at `ROM_EPITABLE[6]`.

Parameters:

ulBase is the EPI module base address.

ulConfig is the interface configuration.

ulMaxWait is the maximum number of external clocks to wait if a FIFO ready signal is holding off the transaction.

Description:

This function is used to configure the interface when used in Host-bus 16 operation as chosen with the function ROM_EPIModeSet(). The parameter *ulConfig* is the logical OR of any of the following:

- one of **EPI_HB16_MODE_ADMUX**, **EPI_HB16_MODE_ADDEMUX**, **EPI_HB16_MODE_SRAM**, or **EPI_HB16_MODE_FIFO** to select the HB16 mode
- **EPI_HB16_USE_TXEMPTY** - enable TXEMPTY signal with FIFO
- **EPI_HB16_USE_RXFULL** - enable RXFULL signal with FIFO
- **EPI_HB16_WRHIGH** - use active high write strobe, otherwise it is active low
- **EPI_HB16_RDHIGH** - use active high read strobe, otherwise it is active low
- one of **EPI_HB16_WRWAIT_0**, **EPI_HB16_WRWAIT_1**, **EPI_HB16_WRWAIT_2**, or **EPI_HB16_WRWAIT_3** to select the number of write wait states (default is 0 wait states)
- one of **EPI_HB16_RDWAIT_0**, **EPI_HB16_RDWAIT_1**, **EPI_HB16_RDWAIT_2**, or **EPI_HB16_RDWAIT_3** to select the number of read wait states (default is 0 wait states)
- **EPI_HB16_WORD_ACCESS** - use Word Access mode to route bytes to the correct byte lanes allowing data to be stored in bits [31:8]. If absent, all data transfers use bits [7:0].

- **EPI_HB16_BSEL** - enables byte selects. In this mode, two EPI signals operate as byte selects allowing 8-bit transfers. If this flag is not specified, data must be read and written using only 16-bit transfers.
- **EPI_HB16_CSBAUD_DUAL** - use different baud rates when accessing devices on each CSn. CS0n uses the baud rate specified by the lower 16 bits of the divider passed to ROM_EPIDividerSet() and CS1n uses the divider passed in the upper 16 bits. If this option is absent, both chip selects use the baud rate resulting from the divider in the lower 16 bits of the parameter passed to ROM_EPIDividerSet().
- one of **EPI_HB16_CSCFG_CS**, **EPI_HB16_CSCFG_ALE**, **EPI_HB16_CSCFG_DUAL_CS** or **EPI_HB16_CSCFG_ALE_DUAL_CS**. **EPI_HB16_CSCFG_CS** sets EPI30 to operate as a Chip Select (CSn) signal. **EPI_HB16_CSCFG_ALE** sets EPI30 to operate as an address latch (ALE). **EPI_HB16_CSCFG_DUAL_CS** sets EPI30 to operate as CS0n and EPI27 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map. **EPI_HB16_CSCFG_ALE_DUAL_CS** sets EPI30 as an address latch (ALE), EPI27 as CS0n and EPI26 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.

The parameter *ulMaxWait* is used if the FIFO mode is chosen. If a FIFO is used along with RXFULL or TXEMPTY ready signals, then this parameter determines the maximum number of clocks to wait when the transaction is being held off by by the FIFO using one of these ready signals. A value of 0 means to wait forever.

Returns:

None.

9.2.1.4 ROM_EPIConfigHB8Set

Configures the interface for Host-bus 8 operation.

Prototype:

```
void
ROM_EPIConfigHB8Set(unsigned long ulBase,
                    unsigned long ulConfig,
                    unsigned long ulMaxWait)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIConfigHB8Set` is a function pointer located at `ROM_EPITABLE[5]`.

Parameters:

ulBase is the EPI module base address.

ulConfig is the interface configuration.

ulMaxWait is the maximum number of external clocks to wait if a FIFO ready signal is holding off the transaction.

Description:

This function is used to configure the interface when used in Host-bus 8 operation as chosen with the function ROM_EPIModeSet(). The parameter *ulConfig* is the logical OR of any of the following:

- one of **EPI_HB8_MODE_ADMUX**, **EPI_HB8_MODE_ADDEMUX**, **EPI_HB8_MODE_SRAM**, or **EPI_HB8_MODE_FIFO** to select the HB8 mode
- **EPI_HB8_USE_TXEMPTY** - enable TXEMPTY signal with FIFO
- **EPI_HB8_USE_RXFULL** - enable RXFULL signal with FIFO
- **EPI_HB8_WRHIGH** - use active high write strobe, otherwise it is active low
- **EPI_HB8_RDHIGH** - use active high read strobe, otherwise it is active low
- one of **EPI_HB8_WRWAIT_0**, **EPI_HB8_WRWAIT_1**, **EPI_HB8_WRWAIT_2**, or **EPI_HB8_WRWAIT_3** to select the number of write wait states (default is 0 wait states)
- one of **EPI_HB8_RDWAIT_0**, **EPI_HB8_RDWAIT_1**, **EPI_HB8_RDWAIT_2**, or **EPI_HB8_RDWAIT_3** to select the number of read wait states (default is 0 wait states)
- **EPI_HB8_WORD_ACCESS** - use Word Access mode to route bytes to the correct byte lanes allowing data to be stored in bits [31:8]. If absent, all data transfers use bits [7:0].
- **EPI_HB8_CSBAUD_DUAL** - use different baud rates when accessing devices on each CSn. CS0n uses the baud rate specified by the lower 16 bits of the divider passed to ROM_EPIDividerSet() and CS1n uses the divider passed in the upper 16 bits. If this option is absent, both chip selects use the baud rate resulting from the divider in the lower 16 bits of the parameter passed to ROM_EPIDividerSet().
- one of **EPI_HB8_CSCFG_CS**, **EPI_HB8_CSCFG_ALE**, **EPI_HB8_CSCFG_DUAL_CS** or **EPI_HB8_CSCFG_ALE_DUAL_CS**. **EPI_HB8_CSCFG_CS** sets EPI30 to operate as a Chip Select (CSn) signal. **EPI_HB8_CSCFG_ALE** sets EPI30 to operate as an address latch (ALE). **EPI_HB8_CSCFG_DUAL_CS** sets EPI30 to operate as CS0n and EPI27 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map. **EPI_HB8_CSCFG_ALE_DUAL_CS** sets EPI30 as an address latch (ALE), EPI27 as CS0n and EPI26 as CS1n with the asserted chip select determined from the most significant address bit for the respective external address map.

The parameter *ulMaxWait* is used if the FIFO mode is chosen. If a FIFO is used along with RXFULL or TXEMPTY ready signals, then this parameter determines the maximum number of clocks to wait when the transaction is being held off by by the FIFO using one of these ready signals. A value of 0 means to wait forever.

Returns:

None.

9.2.1.5 ROM_EPIConfigSDRAMSet

Configures the SDRAM mode of operation.

Prototype:

```
void
ROM_EPIConfigSDRAMSet(unsigned long ulBase,
                      unsigned long ulConfig,
                      unsigned long ulRefresh)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIConfigSDRAMSet` is a function pointer located at `ROM_EPITABLE[3]`.

Parameters:

ulBase is the EPI module base address.

ulConfig is the SDRAM interface configuration.

ulRefresh is the refresh count in core clocks (0-2047).

Description:

This function is used to configure the SDRAM interface, when the SDRAM mode is chosen with the function ROM_EPIModeSet(). The parameter *ulConfig* is the logical OR of several sets of choices:

The processor core frequency must be specified with one of the following:

- **EPI_SDRAM_CORE_FREQ_0_15** - core clock is 0 MHz < clk <= 15 MHz
- **EPI_SDRAM_CORE_FREQ_15_30** - core clock is 15 MHz < clk <= 30 MHz
- **EPI_SDRAM_CORE_FREQ_30_50** - core clock is 30 MHz < clk <= 50 MHz
- **EPI_SDRAM_CORE_FREQ_50_100** - core clock is 50 MHz < clk <= 100 MHz

The low power mode is specified with one of the following:

- **EPI_SDRAM_LOW_POWER** - enter low power, self-refresh state
- **EPI_SDRAM_FULL_POWER** - normal operating state

The SDRAM device size is specified with one of the following:

- **EPI_SDRAM_SIZE_64MBIT** - 64 Mbit device (8 MB)
- **EPI_SDRAM_SIZE_128MBIT** - 128 Mbit device (16 MB)
- **EPI_SDRAM_SIZE_256MBIT** - 256 Mbit device (32 MB)
- **EPI_SDRAM_SIZE_512MBIT** - 512 Mbit device (64 MB)

The parameter *ulRefresh* sets the refresh counter in units of core clock ticks. It is an 11-bit value with a range of 0 - 2047 counts.

Returns:

None.

9.2.1.6 ROM_EPIDividerSet

Sets the clock divider for the EPI module.

Prototype:

```
void
ROM_EPIDividerSet(unsigned long ulBase,
                  unsigned long ulDivider)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIDividerSet` is a function pointer located at `ROM_EPITABLE[2]`.

Parameters:

ulBase is the EPI module base address.

ulDivider is the value of the clock divider to be applied to the external interface (0-65535).

Description:

This functions sets the clock divider(s) that will be used to determine the clock rate of the external interface. The *ulDivider* value is used to derive the EPI clock rate from the system clock based upon the following formula.

EPIClock = (Divider == 0) ? SysClk : (SysClk / (((Divider / 2) + 1) * 2))

For example, a divider value of 1 results in an EPI clock rate of half the system clock, value of 2 or 3 yield one quarter of the system clock and a value of 4 results in one sixth of the system clock rate.

In cases where a dual chip select mode is in use and different clock rates are required for each chip select, the *ulDivider* parameter must contain two dividers. The lower 16 bits define the divider to be used with CS0n and the upper 16 bits define the divider for CS1n.

Returns:

None.

9.2.1.7 ROM_EPIFIFOConfig

Configures the read FIFO.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIFIFOConfig` is a function pointer located at `ROM_EPITABLE[16]`.

Parameters:

ulBase is the EPI module base address.

ulConfig is the FIFO configuration.

Description:

This function configures the FIFO trigger levels and error generation. The parameter *ulConfig* is the logical OR of the following:

- **EPI_FIFO_CONFIG_WTFULLERR** - enables an error interrupt when a write is attempted and the write FIFO is full
- **EPI_FIFO_CONFIG_RSTALLERR** - enables an error interrupt when a read is stalled due to an interleaved write or other reason
- **EPI_FIFO_CONFIG_TX_EMPTY**, **EPI_FIFO_CONFIG_TX_1_4**, **EPI_FIFO_CONFIG_TX_1_2**, or **EPI_FIFO_CONFIG_TX_3_4** to set the TX FIFO trigger level to empty, 1/4, 1/2, or 3/4 level
- **EPI_FIFO_CONFIG_RX_1_8**, **EPI_FIFO_CONFIG_RX_1_4**, **EPI_FIFO_CONFIG_RX_1_2**, **EPI_FIFO_CONFIG_RX_3_4**, **EPI_FIFO_CONFIG_RX_7_8**, or **EPI_FIFO_CONFIG_RX_FULL** to set the RX FIFO trigger level to 1/8, 1/4, 1/2, 3/4, 7/8 or full level

Returns:

None.

9.2.1.8 ROM_EPIIntDisable

Disables EPI interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIIntDisable` is a function pointer located at `ROM_EPITABLE[19]`.

Parameters:

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

Description:

This function disables the specified EPI sources for interrupt generation. The *ulIntFlags* parameter can be the logical OR of any of the following values: **EPI_INT_RXREQ**, **EPI_INT_TXREQ**, or **I2S_INT_ERR**.

Returns:

Returns None.

9.2.1.9 ROM_EPIIntEnable

Enables EPI interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIIntEnable` is a function pointer located at `ROM_EPITABLE[18]`.

Parameters:

ulBase is the EPI module base address.
ulIntFlags is a bit mask of the interrupt sources to be enabled.

Description:

This function enables the specified EPI sources to generate interrupts. The *ulIntFlags* parameter can be the logical OR of any of the following values:

- **EPI_INT_TXREQ** - transmit FIFO is below the trigger level
- **EPI_INT_RXREQ** - read FIFO is above the trigger level
- **EPI_INT_ERR** - an error condition occurred

Returns:

Returns None.

9.2.1.10 ROM_EPIIntErrorClear

Clears pending EPI error sources.

Prototype:

```
void
ROM_EPIIntErrorClear(unsigned long ulBase,
                     unsigned long ulErrFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIIntErrorClear` is a function pointer located at `ROM_EPITABLE[21]`.

Parameters:

ulBase is the EPI module base address.

ulErrFlags is a bit mask of the error sources to be cleared.

Description:

This function clears the specified pending EPI errors. The *ulErrFlags* parameter can be the logical OR of any of the following values: **EPI_INT_ERR_WTFULL**, **EPI_INT_ERR_RSTALL**, or **EPI_INT_ERR_TIMEOUT**.

Returns:

Returns None.

9.2.1.11 ROM_EPIIntErrorStatus

Gets the EPI error interrupt status.

Prototype:

```
unsigned long
ROM_EPIIntErrorStatus(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIIntErrorStatus` is a function pointer located at `ROM_EPITABLE[20]`.

Parameters:

ulBase is the EPI module base address.

Description:

This function returns the error status of the EPI. If the return value of the function ROM_EPIIntStatus() has the flag **EPI_INT_ERR** set, then this function can be used to determine the cause of the error.

This function returns a bit mask of error flags, which can be the logical OR of any of the following:

- **EPI_INT_ERR_WTFULL** - occurs when a write stalled when the transaction FIFO was full
- **EPI_INT_ERR_RSTALL** - occurs when a read stalled

- **EPI_INT_ERR_TIMEOUT** - occurs when the external clock enable held off a transaction longer than the configured maximum wait time

Returns:

Returns the interrupt error flags as the logical OR of any of the following: **EPI_INT_ERR_WTFULL**, **EPI_INT_ERR_RSTALL**, or **EPI_INT_ERR_TIMEOUT**.

9.2.1.12 ROM_EPIIntStatus

Gets the EPI interrupt status.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIIntStatus` is a function pointer located at `ROM_EPITABLE[0]`.

Parameters:

ulBase is the EPI module base address.

bMasked is set **true** to get the masked interrupt status, or **false** to get the raw interrupt status.

Description:

This function returns the EPI interrupt status. It can return either the raw or masked interrupt status.

Returns:

Returns the masked or raw EPI interrupt status, as a bit field of any of the following values: **EPI_INT_TXREQ**, **EPI_INT_RXREQ**, or **EPI_INT_ERR**

9.2.1.13 ROM_EPIModeSet

Sets the usage mode of the EPI module.

Prototype:

```
void
ROM_EPIModeSet(unsigned long ulBase,
               unsigned long ulMode)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIModeSet` is a function pointer located at `ROM_EPITABLE[1]`.

Parameters:

ulBase is the EPI module base address.

ulMode is the usage mode of the EPI module.

Description:

This functions sets the operating mode of the EPI module. The parameter *ulMode* must be one of the following:

- **EPI_MODE_GENERAL** - use for general-purpose mode operation
- **EPI_MODE_SDRAM** - use with SDRAM device
- **EPI_MODE_HB8** - use with host-bus 8-bit interface
- **EPI_MODE_HB16** - use with host-bus 16-bit interface
- **EPI_MODE_DISABLE** - disable the EPI module

Selection of any of the above modes will enable the EPI module, except for **EPI_MODE_DISABLE** which should be used to disable the module.

Returns:

None.

9.2.1.14 ROM_EPINonBlockingReadAvail

Get the count of items available in the read FIFO.

Prototype:

```
unsigned long
ROM_EPINonBlockingReadAvail(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadAvail` is a function pointer located at `ROM_EPITABLE[12]`.

Parameters:

ulBase is the EPI module base address.

Description:

This function gets the number of items that are available to read in the read FIFO. The read FIFO is filled by a non-blocking read transaction which is configured by the functions ROM_EPINonBlockingReadConfigure() and ROM_EPINonBlockingReadStart().

Returns:

The number of items available to read in the read FIFO.

9.2.1.15 ROM_EPINonBlockingReadConfigure

Configures a non-blocking read transaction.

Prototype:

```
void
ROM_EPINonBlockingReadConfigure(unsigned long ulBase,
                                unsigned long ulChannel,
                                unsigned long ulDataSize,
                                unsigned long ulAddress)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadConfigure` is a function pointer located at `ROM_EPITABLE[8]`.

Parameters:

ulBase is the EPI module base address.

ulChannel is the read channel (0 or 1).

ulDataSize is the size of the data items to read.

ulAddress is the starting address to read.

Description:

This function is used to configure a non-blocking read channel for a transaction. Two channels are available which can be used in a ping-pong method for continuous reading. It is not necessary to use both channels to perform a non-blocking read.

The parameter *ulDataSize* is one of **EPI_NBCONFIG_SIZE_8**, **EPI_NBCONFIG_SIZE_16**, or **EPI_NBCONFIG_SIZE_32** for 8-bit, 16-bit, or 32-bit sized data transfers.

The parameter *ulAddress* is the starting address for the read, relative to the external device. The start of the device is address 0.

Once configured, the non-blocking read is started by calling ROM_EPINonBlockingReadStart(). If the addresses to be read from the device are in a sequence, it is not necessary to call this function multiple times. Until it is changed, the EPI module will remember the last address that was used for a non-blocking read (per channel).

Returns:

None.

9.2.1.16 ROM_EPINonBlockingReadCount

Get the count remaining for a non-blocking transaction.

Prototype:

```
unsigned long
ROM_EPINonBlockingReadCount(unsigned long ulBase,
                            unsigned long ulChannel)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadCount` is a function pointer located at `ROM_EPITABLE[11]`.

Parameters:

ulBase is the EPI module base address.

ulChannel is the read channel (0 or 1).

Description:

This function gets the remaining count of items for a non-blocking read transaction.

Returns:

The number of items remaining in the non-blocking read transaction.

9.2.1.17 ROM_EPINonBlockingReadGet16

Read available data from the read FIFO, as 16-bit data items.

Prototype:

```
unsigned long
ROM_EPINonBlockingReadGet16(unsigned long ulBase,
                            unsigned long ulCount,
                            unsigned short *pusBuf)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadGet16` is a function pointer located at `ROM_EPITABLE[14]`.

Parameters:

ulBase is the EPI module base address.

ulCount is the maximum count of items to read.

pusBuf is the caller supplied buffer where the read data should be stored.

Description:

This function reads 16-bit data items from the read FIFO and stores the values in a caller supplied buffer. The function will read and store data from the FIFO until there is no more data in the FIFO or the maximum count is reached as specified in the parameter *ulCount*. The actual count of items will be returned.

Returns:

The number of items read from the FIFO.

9.2.1.18 ROM_EPINonBlockingReadGet32

Read available data from the read FIFO, as 32-bit data items.

Prototype:

```
unsigned long
ROM_EPINonBlockingReadGet32(unsigned long ulBase,
                            unsigned long ulCount,
                            unsigned long *pulBuf)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadGet32` is a function pointer located at `ROM_EPITABLE[13]`.

Parameters:

ulBase is the EPI module base address.

ulCount is the maximum count of items to read.

pulBuf is the caller supplied buffer where the read data should be stored.

Description:

This function reads 32-bit data items from the read FIFO and stores the values in a caller supplied buffer. The function will read and store data from the FIFO until there is no more

data in the FIFO or the maximum count is reached as specified in the parameter *ulCount*. The actual count of items will be returned.

Returns:
The number of items read from the FIFO.

9.2.1.19 ROM_EPINonBlockingReadGet8

Read available data from the read FIFO, as 8-bit data items.

Prototype:
```
unsigned long
ROM_EPINonBlockingReadGet8(unsigned long ulBase,
                           unsigned long ulCount,
                           unsigned char *pucBuf)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadGet8` is a function pointer located at `ROM_EPITABLE[15]`.

Parameters:
ulBase is the EPI module base address.
ulCount is the maximum count of items to read.
pucBuf is the caller supplied buffer where the read data should be stored.

Description:
This function reads 8-bit data items from the read FIFO and stores the values in a caller supplied buffer. The function will read and store data from the FIFO until there is no more data in the FIFO or the maximum count is reached as specified in the parameter *ulCount*. The actual count of items will be returned.

Returns:
The number of items read from the FIFO.

9.2.1.20 ROM_EPINonBlockingReadStart

Starts a non-blocking read transaction.

Prototype:
```
void
ROM_EPINonBlockingReadStart(unsigned long ulBase,
                            unsigned long ulChannel,
                            unsigned long ulCount)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadStart` is a function pointer located at `ROM_EPITABLE[9]`.

Parameters:

ulBase is the EPI module base address.

ulChannel is the read channel (0 or 1).

ulCount is the number of items to read (1-4095).

Description:

This function starts a non-blocking read that was previously configured with the function ROM_EPINonBlockingReadConfigure(). Once this function is called, the EPI module will begin reading data from the external device into the read FIFO. The EPI will stop reading when the FIFO fills up and resume reading when the application drains the FIFO, until the total specified count of data items has been read.

Once a read transaction is completed and the FIFO drained, another transaction can be started from the next address by calling this function again.

Returns:

None.

9.2.1.21 ROM_EPINonBlockingReadStop

Stops a non-blocking read transaction.

Prototype:

```
void
ROM_EPINonBlockingReadStop(unsigned long ulBase,
                           unsigned long ulChannel)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPINonBlockingReadStop` is a function pointer located at `ROM_EPITABLE[10]`.

Parameters:

ulBase is the EPI module base address.

ulChannel is the read channel (0 or 1).

Description:

This function cancels a non-blocking read transaction that is already in progress.

Returns:

None.

9.2.1.22 ROM_EPIWriteFIFOCountGet

Reads the number of empty slots in the write transaction FIFO.

Prototype:

```
unsigned long
ROM_EPIWriteFIFOCountGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_EPITABLE` is an array of pointers located at `ROM_APITABLE[23]`.
`ROM_EPIWriteFIFOCountGet` is a function pointer located at `ROM_EPITABLE[17]`.

Parameters:

ulBase is the EPI module base address.

Description:

This function returns the number of slots available in the transaction FIFO. It can be used in a polling method to avoid attempting a write that would stall.

Returns:

The number of empty slots in the transaction FIFO.

10 Flash

10.1 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.

10.2 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)

10.2.1 Function Documentation

10.2.1.1 ROM_FlashErase

Erases a block of flash.

Prototype:

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

ROM Location:

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

Parameters:

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

Description:

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

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

Returns:

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

10.2.1.2 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.

10.2.1.3 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.

10.2.1.4 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.

10.2.1.5 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**.

10.2.1.6 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.

10.2.1.7 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.

10.2.1.8 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.

10.2.1.9 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.

10.2.1.10 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.

10.2.1.11 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.

10.2.1.12 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.

10.2.1.13 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.

10.2.1.14 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.

11 GPIO

11.1 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).

11.2 Functions

Functions

- unsigned long ROM_GPIODirModeGet (unsigned long ulPort, unsigned char ucPin)
- void ROM_GPIODirModeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulPinIO)

- unsigned long ROM_GPIOIntTypeGet (unsigned long ulPort, unsigned char ucPin)
- void ROM_GPIOIntTypeSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulIntType)
- void ROM_GPIOPadConfigGet (unsigned long ulPort, unsigned char ucPin, unsigned long *pulStrength, unsigned long *pulPinType)
- void ROM_GPIOPadConfigSet (unsigned long ulPort, unsigned char ucPins, unsigned long ulStrength, unsigned long ulPinType)
- void ROM_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_GPIOPinTypeEPI (unsigned long ulPort, unsigned char ucPins)
- void ROM_GPIOPinTypeEthernetLED (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_GPIOPinTypeI2S (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)

11.2.1 Function Documentation

11.2.1.1 ROM_GPIODirModeGet

Gets the direction and mode of a pin.

Prototype:

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

ROM Location:

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

Parameters:

ulPort is the base address of the GPIO port.

ucPin is the pin number.

Description:

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

Returns:

Returns one of the enumerated data types described for ROM_GPIODirModeSet().

11.2.1.2 ROM_GPIODirModeSet

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

Prototype:

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

ROM Location:

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

Parameters:

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

Description:

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

The 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 will be programmed as a software controlled input, **GPIO_DIR_MODE_OUT** specifies that the pin will be programmed as a software controlled output, and **GPIO_DIR_MODE_HW** specifies that the pin will be placed under hardware control.

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

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.

11.2.1.3 ROM_GPIOIntTypeGet

Gets the interrupt type for a pin.

Prototype:

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

ROM Location:

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

Parameters:

ulPort is the base address of the GPIO port.

ucPin is the pin number.

Description:

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

Returns:

Returns one of the enumerated data types described for ROM_GPIOIntTypeSet().

11.2.1.4 ROM_GPIOIntTypeSet

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

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.
`ROM_GPIOIntTypeSet` is a function pointer located at `ROM_GPIOTABLE[3]`.

Parameters:

ulPort is the base address of the GPIO port.

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

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.

11.2.1.5 ROM_GPIOPadConfigGet

Gets the pad configuration for a pin.

Prototype:

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.
`ROM_GPIOPadConfigGet` is a function pointer located at `ROM_GPIOTABLE[6]`.

Parameters:
ulPort is the base address of the GPIO port.
ucPin is the pin number.
pulStrength is a pointer to storage for the output drive strength.
pulPinType is a pointer to storage for the output drive type.

Description:
This function gets the pad configuration for a specified pin on the selected GPIO port. The values returned in *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

11.2.1.6 ROM_GPIOPadConfigSet

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

Prototype:

```
void
ROM_GPIOPadConfigSet(unsigned long ulPort,
                     unsigned char ucPins,
                     unsigned long ulStrength,
                     unsigned long ulPinType)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.
`ROM_GPIOPadConfigSet` is a function pointer located at `ROM_GPIOTABLE[5]`.

Parameters:

ulPort is the base address of the GPIO port.

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

ulStrength specifies the output drive strength.

ulPinType specifies the pin type.

Description:

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

The 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.

11.2.1.7 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.

11.2.1.8 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.

11.2.1.9 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.

11.2.1.10 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.

11.2.1.11 ROM_GPIOPinIntStatus

Gets interrupt status for the specified GPIO port.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_GPIOTABLE` is an array of pointers located at `ROM_APITABLE[4]`.
`ROM_GPIOPinIntStatus` is a function pointer located at `ROM_GPIOTABLE[9]`.

Parameters:

ulPort is the base address of the GPIO port.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

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

Returns:

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

11.2.1.12 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.

11.2.1.13 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.

11.2.1.14 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.

11.2.1.15 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.

11.2.1.16 ROM_GPIOPinTypeEPI

Configures pin(s) for use by the external peripheral interface.

Prototype:

```
void
ROM_GPIOPinTypeEPI(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_GPIOPinTypeEPI` is a function pointer located at `ROM_GPIOTABLE[29]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

The external peripheral interface pins must be properly configured for the external peripheral interface to function correctly. This function provides a typica configuration for those pin(s); other configurations may work as well depending upon the board setup (for exampe, 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 an external peripheral interface pin; it only configures an external peripheral interface pin for proper operation.

Returns:

None.

11.2.1.17 ROM_GPIOPinTypeEthernetLED

Configures pin(s) for use by the Ethernet peripheral as LED signals.

Prototype:

```
void
ROM_GPIOPinTypeEthernetLED(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_GPIOPinTypeEthernetLED` is a function pointer located at `ROM_GPIOTABLE[27]`.

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

The Ethernet peripheral provides two signals that can be used to drive an LED (e.g. for link status/activity). This function provides a typical configuration for the pins.

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

Note:

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

Returns:

None.

11.2.1.18 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.

11.2.1.19 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.

11.2.1.20 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.

11.2.1.21 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.

11.2.1.22 ROM_GPIOPinTypeI2S

Configures pin(s) for use by the I2S peripheral.

Prototype:

```
void
ROM_GPIOPinTypeI2S(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_GPIOPinTypeI2S is a function pointer located at ROM_GPIOTABLE[25].

Parameters:

ulPort is the base address of the GPIO port.

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

Description:

Some I2S pins must be properly configured for the I2S peripheral to function correctly. This function provides a typical configuration for the digital I2S 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 I2S pin; it only configures a I2S pin for proper operation.

Returns:

None.

11.2.1.23 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.

11.2.1.24 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.

11.2.1.25 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.

11.2.1.26 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.

11.2.1.27 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.

11.2.1.28 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.

12 Inter-Integrated Circuit (I2C)

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_I2CMasterIntDisable (unsigned long ulBase)
- void ROM_I2CMasterIntEnable (unsigned long ulBase)
- tBoolean ROM_I2CMasterIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_I2CMasterSlaveAddrSet (unsigned long ulBase, unsigned char ucSlaveAddr, tBoolean bReceive)
- unsigned long ROM_I2CSlaveDataGet (unsigned long ulBase)
- void ROM_I2CSlaveDataPut (unsigned long ulBase, unsigned char ucData)

- void ROM_I2CSlaveDisable (unsigned long ulBase)
- void ROM_I2CSlaveEnable (unsigned long ulBase)
- void ROM_I2CSlaveInit (unsigned long ulBase, unsigned char ucSlaveAddr)
- void ROM_I2CSlaveIntClear (unsigned long ulBase)
- void ROM_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 will be set up to transfer data at 400 kbps; otherwise, it will be set up to transfer data at 100 kbps.

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

Returns:

None.

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_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.

12.2.1.12 ROM_I2CMasterIntEnable

Enables the I2C Master interrupt.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CMasterIntEnable` is a function pointer located at `ROM_I2CTABLE[7]`.

Parameters:

ulBase is the base address of the I2C Master module.

Description:

Enables the I2C Master interrupt source.

Returns:

None.

12.2.1.13 ROM_I2CMasterIntStatus

Gets the current I2C Master interrupt status.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CMasterIntStatus` is a function pointer located at `ROM_I2CTABLE[11]`.

Parameters:

ulBase is the base address of the I2C Master module.

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

Description:

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

Returns:

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

12.2.1.14 ROM_I2CMasterSlaveAddrSet

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

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CMasterSlaveAddrSet` is a function pointer located at `ROM_I2CTABLE[15]`.

Parameters:

ulBase is the base address of the I2C Master module.

ucSlaveAddr 7-bit slave address

bReceive flag indicating the type of communication with the slave

Description:

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

Returns:

None.

12.2.1.15 ROM_I2CSlaveDataGet

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

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveDataGet` is a function pointer located at `ROM_I2CTABLE[23]`.

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

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

Returns:

Returns the byte received from by the I2C Slave, cast as an unsigned long.

12.2.1.16 ROM_I2CSlaveDataPut

Transmits a byte from the I2C Slave.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveDataPut` is a function pointer located at `ROM_I2CTABLE[22]`.

Parameters:

ulBase is the base address of the I2C Slave module.

ucData data to be transmitted from the I2C Slave

Description:

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

Returns:

None.

12.2.1.17 ROM_I2CSlaveDisable

Disables the I2C slave block.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveDisable` is a function pointer located at `ROM_I2CTABLE[6]`.

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

This will disable operation of the I2C slave block.

Returns:

None.

12.2.1.18 ROM_I2CSlaveEnable

Enables the I2C Slave block.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveEnable` is a function pointer located at `ROM_I2CTABLE[4]`.

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

This will enable operation of the I2C Slave block.

Returns:

None.

12.2.1.19 ROM_I2CSlaveInit

Initializes the I2C Slave block.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveInit` is a function pointer located at `ROM_I2CTABLE[2]`.

Parameters:

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

Description:

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

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

Returns:

None.

12.2.1.20 ROM_I2CSlaveIntClear

Clears I2C Slave interrupt sources.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveIntClear` is a function pointer located at `ROM_I2CTABLE[14]`.

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

The I2C Slave interrupt source is cleared, so that it no longer asserts. This must be done in the interrupt handler to keep it from being called again immediately upon exit.

Note:

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.

12.2.1.21 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.

12.2.1.22 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.

12.2.1.23 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.

12.2.1.24 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.

12.2.1.25 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.

12.2.1.26 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.

12.2.1.27 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().

12.2.1.28 ROM_I2CSlaveStatus

Gets the I2C Slave module status

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2CTABLE` is an array of pointers located at `ROM_APITABLE[3]`.
`ROM_I2CSlaveStatus` is a function pointer located at `ROM_I2CTABLE[21]`.

Parameters:

ulBase is the base address of the I2C Slave module.

Description:

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

- **I2C_SLAVE_ACT_NONE**
- **I2C_SLAVE_ACT_RREQ**
- **I2C_SLAVE_ACT_TREQ**
- **I2C_SLAVE_ACT_RREQ_FBR**

Returns:

Returns **I2C_SLAVE_ACT_NONE** to indicate that no action has been requested of the I2C Slave module, **I2C_SLAVE_ACT_RREQ** to indicate that an I2C master has sent data to the I2C Slave module, **I2C_SLAVE_ACT_TREQ** to indicate that an I2C master has requested that the I2C Slave module send data, and **I2C_SLAVE_ACT_RREQ_FBR** to indicate that an I2C master has sent data to the I2C slave and the first byte following the slave's own address has been received.

12.2.1.29 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 Inter-IC Sound (I2S)

13.1 Introduction

The I2S API provides functions to use the I2S peripheral in the Stellaris microcontroller. The I2S peripheral provides an interface for serial transfer of variable sized data samples, typically for audio or analog applications. The I2S peripheral automatically handles left and right channels in audio data.

The I2S peripheral contains two modules, one for transmit and one for receive. These two modules can be independently configured for clock time base and data format.

Some features of the I2S peripheral are:

- independently configurable transmit and receive modules
- 8 sample pair FIFOs
- adjustable FIFO service request levels
- interrupt on FIFO service request or error
- DMA interface
- adjustable time base for clocking
- clock slave or master
- left justified, right justified, and I2S format modes
- adjustable sample data size
- adjustable wire word size
- single or dual channel (stereo/mono)

The I2S peripheral contains a transmit and receive module, which are generally the same in terms of configuration. Use ROM_I2SRxConfigSet() or ROM_I2STxConfigSet() to configure the receive or transmit module format and mode. Once configured, the transmit or receive module must be enabled using ROM_I2STxEnable() or ROM_I2SRxEnable(). The module can be later disabled with ROM_I2STxDisable() or ROM_I2SRxDisable().

If you want to use interrupts or DMA to service the I2S FIFO, then the FIFO trigger level must be set using ROM_I2SRxFIFOLimitSet() or ROM_I2STxFIFOLimitSet().

Use the function ROM_I2STxDataPut() to write data to the I2S transmit FIFO. This function will block until there is space in the FIFO. To avoid blocking, use the function ROM_I2STxDataPutNonBlocking() instead. Likewise, the functions ROM_I2SRxDataGet() and ROM_I2SRxDataGetNonBlocking() are used to read data from the receive FIFO.

There are several functions that can be used to query the status of the I2S peripheral. The functions ROM_I2SRxFIFOLevelGet() and ROM_I2STxFIFOLevelGet() can be used to read the number of samples in the receive or transmit FIFO.

There is a master clock that is used to derive the serial bit clock (SCLK) and the left-right word clock (LRCLK) timings. The master clock can be sourced from the microcontroller's internal PLL or from an external pin. The master clock source is configured with the function ROM_I2SMasterClockSelect(). This function will configure both the transmit and receive module. If the internal PLL is used, then the master clock rate must be set using ROM_SysCtlI2SMClkSet().

Interrupts for the transmit and receive modules are configured together since there is one interrupt for both. Interrupts are enabled or disabled using ROM_I2SIntEnable() and ROM_I2SIntDisable(). The interrupt status can be read using ROM_I2SIntStatus() from within the interrupt handler, or non-interrupt code. When in the interrupt handler, the pending interrupts must be cleared using ROM_I2SIntClear().

13.2 Functions

Functions

- void ROM_I2SIntClear (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2SIntDisable (unsigned long ulBase, unsigned long ulIntFlags)
- void ROM_I2SIntEnable (unsigned long ulBase, unsigned long ulIntFlags)
- unsigned long ROM_I2SIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_I2SMasterClockSelect (unsigned long ulBase, unsigned long ulMClock)
- void ROM_I2SRxConfigSet (unsigned long ulBase, unsigned long ulConfig)
- void ROM_I2SRxDataGet (unsigned long ulBase, unsigned long *pulData)
- long ROM_I2SRxDataGetNonBlocking (unsigned long ulBase, unsigned long *pulData)
- void ROM_I2SRxDisable (unsigned long ulBase)
- void ROM_I2SRxEnable (unsigned long ulBase)
- unsigned long ROM_I2SRxFIFOLevelGet (unsigned long ulBase)
- unsigned long ROM_I2SRxFIFOLimitGet (unsigned long ulBase)
- void ROM_I2SRxFIFOLimitSet (unsigned long ulBase, unsigned long ulLevel)
- void ROM_I2STxConfigSet (unsigned long ulBase, unsigned long ulConfig)
- void ROM_I2STxDataPut (unsigned long ulBase, unsigned long ulData)
- long ROM_I2STxDataPutNonBlocking (unsigned long ulBase, unsigned long ulData)
- void ROM_I2STxDisable (unsigned long ulBase)
- void ROM_I2STxEnable (unsigned long ulBase)
- unsigned long ROM_I2STxFIFOLevelGet (unsigned long ulBase)
- unsigned long ROM_I2STxFIFOLimitGet (unsigned long ulBase)
- void ROM_I2STxFIFOLimitSet (unsigned long ulBase, unsigned long ulLevel)
- void ROM_I2STxRxConfigSet (unsigned long ulBase, unsigned long ulConfig)
- void ROM_I2STxRxDisable (unsigned long ulBase)
- void ROM_I2STxRxEnable (unsigned long ulBase)

13.2.1 Function Documentation

13.2.1.1 ROM_I2SIntClear

Clears pending I2S interrupt sources.

Prototype:

```
void
ROM_I2SIntClear(unsigned long ulBase,
                unsigned long ulIntFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SIntClear` is a function pointer located at `ROM_I2STABLE[23]`.

Parameters:

ulBase is the I2S module base address.

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

Description:

This function clears the specified pending I2S interrupts. This must be done in the interrupt handler to keep the handler from being called again immediately upon exit. The *ulIntFlags* parameter can be the logical OR of any of the following values: **I2S_INT_RXERR**, **I2S_INT_RXREQ**, **I2S_INT_TXERR**, or **I2S_INT_TXREQ**.

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:

Returns None.

13.2.1.2 ROM_I2SIntDisable

Disables I2S interrupt sources.

Prototype:

```
void
ROM_I2SIntDisable(unsigned long ulBase,
                  unsigned long ulIntFlags)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SIntDisable` is a function pointer located at `ROM_I2STABLE[22]`.

Parameters:

ulBase is the I2S module base address.

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

Description:

This function disables the specified I2S sources for interrupt generation. The *ulIntFlags* parameter can be the logical OR of any of the following values: **I2S_INT_RXERR**, **I2S_INT_RXREQ**, **I2S_INT_TXERR**, or **I2S_INT_TXREQ**.

Returns:
Returns None.

13.2.1.3 ROM_I2SIntEnable

Enables I2S interrupt sources.

Prototype:
```
void
ROM_I2SIntEnable(unsigned long ulBase,
                 unsigned long ulIntFlags)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SIntEnable` is a function pointer located at `ROM_I2STABLE[21]`.

Parameters:
ulBase is the I2S module base address.
ulIntFlags is a bit mask of the interrupt sources to be enabled.

Description:
This function enables the specified I2S sources to generate interrupts. The *ulIntFlags* parameter can be the logical OR of any of the following values:

- **I2S_INT_RXERR** for receive errors
- **I2S_INT_RXREQ** for receive FIFO service requests
- **I2S_INT_TXERR** for transmit errors
- **I2S_INT_TXREQ** for transmit FIFO service requests

Returns:
Returns None.

13.2.1.4 ROM_I2SIntStatus

Gets the I2S interrupt status.

Prototype:
```
unsigned long
ROM_I2SIntStatus(unsigned long ulBase,
                 tBoolean bMasked)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SIntStatus` is a function pointer located at `ROM_I2STABLE[0]`.

Parameters:
ulBase is the I2S module base address.
bMasked is set **true** to get the masked interrupt status, or **false** to get the raw interrupt status.

Description:
This function returns the I2S interrupt status. It can return either the raw or masked interrupt status.

Returns:
Returns the masked or raw I2S interrupt status, as a bit field of any of the following values: **I2S_INT_RXERR**, **I2S_INT_RXREQ**, **I2S_INT_TXERR**, or **I2S_INT_TXREQ**

13.2.1.5 ROM_I2SMasterClockSelect

Selects the source of the master clock, internal or external.

Prototype:
```
void
ROM_I2SMasterClockSelect(unsigned long ulBase,
                         unsigned long ulMClock)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SMasterClockSelect` is a function pointer located at `ROM_I2STABLE[20]`.

Parameters:
ulBase is the I2S module base address.
ulMClock is the logical OR of the master clock configuration choices.

Description:
This function selects whether the master clock is sourced from the device internal PLL, or comes from an external pin. The I2S serial bit clock (SCLK) and left-right word clock (LRCLK) are derived from the I2S master clock. The transmit and receive modules can be configured independently. The *ulMClock* parameter is chosen from the following:

- one of **I2S_TX_MCLK_EXT** or **I2S_TX_MCLK_INT**
- one of **I2S_RX_MCLK_EXT** or **I2S_RX_MCLK_INT**

Returns:
Returns None.

13.2.1.6 ROM_I2SRxConfigSet

Configures the I2S receive module.

Prototype:
```
void
ROM_I2SRxConfigSet(unsigned long ulBase,
                   unsigned long ulConfig)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxConfigSet` is a function pointer located at `ROM_I2STABLE[13]`.

Parameters:

ulBase is the I2S module base address.

ulConfig is the logical OR of the configuration options.

Description:

This function is used to configure the options for the I2S receive channel. The parameter *ulConfig* is the logical OR of the following options:

- **I2S_CONFIG_FORMAT_I2S** for standard I2S format, **I2S_CONFIG_FORMAT_LEFT_JUST** for left justified format, or **I2S_CONFIG_FORMAT_RIGHT_JUST** for right justified format.
- **I2S_CONFIG_SCLK_INVERT** to invert the polarity of the serial bit clock.
- **I2S_CONFIG_MODE_DUAL** for dual channel stereo, **I2S_CONFIG_MODE_COMPACT_16** for 16-bit compact stereo mode, **I2S_CONFIG_MODE_COMPACT_8** for 8-bit compact stereo mode, or **I2S_CONFIG_MODE_MONO** for single channel mono format.
- **I2S_CONFIG_CLK_MASTER** or **I2S_CONFIG_CLK_SLAVE** to select whether the I2S receiver is the clock master or slave.
- **I2S_CONFIG_SAMPLE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per sample.
- **I2S_CONFIG_WIRE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per word that are transferred on the data line.

Returns:

None.

13.2.1.7 ROM_I2SRxDataGet

Reads data samples from the I2S receive FIFO with blocking.

Prototype:

```
void
ROM_I2SRxDataGet(unsigned long ulBase,
                 unsigned long *pulData)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxDataGet` is a function pointer located at `ROM_I2STABLE[11]`.

Parameters:

ulBase is the I2S module base address.

pulData points to storage for the returned I2S sample data.

Description:

This function reads a single channel sample or combined left-right samples from the I2S receive FIFO. The format of the sample is determined by the configuration that was used with the function ROM_I2SRxConfigSet(). If the receive mode is **I2S_MODE_DUAL_STEREO** then the returned value contains either the left or right sample. The left and right sample alternate with each read from the FIFO, left sample first. If the receive mode is **I2S_MODE_COMPACT_STEREO_16** or **I2S_MODE_COMPACT_STEREO_8**, then the returned data contains both the left and right samples. If the receive mode is **I2S_MODE_SINGLE_MONO** then the returned data contains the single channel sample.

For the compact modes, both the left and right samples are read at the same time. If 16-bit compact mode is used, then the least significant 16 bits contain the left sample, and the most significant 16 bits contain the right sample. If 8-bit compact mode is used, then the lower 8 bits contain the left sample, and the next 8 bits contain the right sample, with the upper 16 bits unused.

If there is no data in the receive FIFO, then this function will wait in a polling loop until data is available.

Returns:

None.

13.2.1.8 ROM_I2SRxDataGetNonBlocking

Reads data samples from the I2S receive FIFO without blocking.

Prototype:

```
long
ROM_I2SRxDataGetNonBlocking(unsigned long ulBase,
                            unsigned long *pulData)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxDataGetNonBlocking` is a function pointer located at `ROM_I2STABLE[12]`.

Parameters:

ulBase is the I2S module base address.

pulData points to storage for the returned I2S sample data.

Description:

This function reads a single channel sample or combined left-right samples from the I2S receive FIFO. The format of the sample is determined by the configuration that was used with the function ROM_I2SRxConfigSet(). If the receive mode is **I2S_MODE_DUAL_STEREO** then the received data contains either the left or right sample. The left and right sample alternate with each read from the FIFO, left sample first. If the receive mode is **I2S_MODE_COMPACT_STEREO_16** or **I2S_MODE_COMPACT_STEREO_8**, then the received data contains both the left and right samples. If the receive mode is **I2S_MODE_SINGLE_MONO** then the received data contains the single channel sample.

For the compact modes, both the left and right samples are read at the same time. If 16-bit compact mode is used, then the least significant 16 bits contain the left sample, and the most significant 16 bits contain the right sample. If 8-bit compact mode is used, then the lower 8 bits contain the left sample, and the next 8 bits contain the right sample, with the upper 16 bits unused.

If there is no data in the receive FIFO, then this function will return immediately without reading any data from the FIFO.

Returns:

The number of elements read from the I2S receive FIFO (1 or 0).

13.2.1.9 ROM_I2SRxDisable

Disables the I2S receive module for operation.

Prototype:

```
void
ROM_I2SRxDisable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxDisable` is a function pointer located at `ROM_I2STABLE[10]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function disables the receive module for operation. The module should be disabled before configuration. When the module is disabled, no data will be clocked in regardless of the signals on the I2S interface.

Returns:

None.

13.2.1.10 ROM_I2SRxEnable

Enables the I2S receive module for operation.

Prototype:

```
void
ROM_I2SRxEnable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxEnable` is a function pointer located at `ROM_I2STABLE[9]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function enables the receive module for operation. The module should be enabled after configuration. When the module is disabled, no data will be clocked in regardless of the signals on the I2S interface.

Returns:

None.

13.2.1.11 ROM_I2SRxFIFOLevelGet

Gets the number of samples in the receive FIFO.

Prototype:

```
unsigned long
ROM_I2SRxFIFOLevelGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxFIFOLevelGet` is a function pointer located at `ROM_I2STABLE[16]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function is used to get the number of samples in the receive FIFO. For the purposes of measuring the FIFO level, a left-right sample pair counts as 2, whether the mode is dual or compact stereo. When mono mode is used, internally the mono sample is still treated as a sample pair, so a single mono sample counts as 2. Since the FIFO always deals with sample pairs, normally the level will be an even number from 0 to 16. If dual stereo mode is used and only the left sample has been read without reading the matching right sample, then the FIFO level will be an odd value. If the FIFO level is odd, it indicates a left-right sample mismatch.

Returns:

Returns the number of samples in the transmit FIFO, which will normally be an even number.

13.2.1.12 ROM_I2SRxFIFOLimitGet

Gets the current setting of the FIFO service request level.

Prototype:

```
unsigned long
ROM_I2SRxFIFOLimitGet(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxFIFOLimitGet` is a function pointer located at `ROM_I2STABLE[15]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function is used to get the value of the receive FIFO service request level. This value is set using the ROM_I2SRxFIFOLimitSet() function.

Returns:

Returns the current value of the FIFO service request limit.

13.2.1.13 ROM_I2SRxFIFOLimitSet

Sets the FIFO level at which a service request is generated.

Prototype:

```
void
ROM_I2SRxFIFOLimitSet(unsigned long ulBase,
                      unsigned long ulLevel)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2SRxFIFOLimitSet` is a function pointer located at `ROM_I2STABLE[14]`.

Parameters:

ulBase is the I2S module base address.

ulLevel is the FIFO service request limit.

Description:

This function is used to set the receive FIFO fullness level at which a service request will occur. The service request is used to generate an interrupt or a DMA transfer request. The receive FIFO will generate a service request when the number of items in the FIFO is greater than the level specified in the *ulLevel* parameter. For example, if *ulLevel* is 4, then a service request will be generated when there are more than 4 samples available in the receive FIFO.

For the purposes of counting the FIFO level, a left-right sample pair counts as 2, whether the mode is dual or compact stereo. When mono mode is used, internally the mono sample is still treated as a sample pair, so a single mono sample counts as 2. Since the FIFO always deals with sample pairs, the level must be an even number from 0 to 16. The minimum value is 0, which will cause a service request when there is any data available in the FIFO. The maximum value is 16, which disables the service request (because there cannot be more than 16 items in the FIFO).

Returns:

None.

13.2.1.14 ROM_I2STxConfigSet

Configures the I2S transmit module.

Prototype:

```
void
ROM_I2STxConfigSet(unsigned long ulBase,
                   unsigned long ulConfig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxConfigSet` is a function pointer located at `ROM_I2STABLE[5]`.

Parameters:

ulBase is the I2S module base address.

ulConfig is the logical OR of the configuration options.

Description:

This function is used to configure the options for the I2S transmit channel. The parameter *ulConfig* is the logical OR of the following options:

- **I2S_CONFIG_FORMAT_I2S** for standard I2S format, **I2S_CONFIG_FORMAT_LEFT_JUST** for left justified format, or **I2S_CONFIG_FORMAT_RIGHT_JUST** for right justified format.
- **I2S_CONFIG_SCLK_INVERT** to invert the polarity of the serial bit clock.
- **I2S_CONFIG_MODE_DUAL** for dual channel stereo, **I2S_CONFIG_MODE_COMPACT_16** for 16-bit compact stereo mode, **I2S_CONFIG_MODE_COMPACT_8** for 8-bit compact stereo mode, or **I2S_CONFIG_MODE_MONO** for single channel mono format.
- **I2S_CONFIG_CLK_MASTER** or **I2S_CONFIG_CLK_SLAVE** to select whether the I2S transmitter is the clock master or slave.
- **I2S_CONFIG_SAMPLE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per sample.
- **I2S_CONFIG_WIRE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per word that are transferred on the data line.
- **I2S_CONFIG_EMPTY_ZERO** or **I2S_CONFIG_EMPTY_REPEAT** to select whether the module transmits zeroes or repeats the last sample when the FIFO is empty.

Returns:

None.

13.2.1.15 ROM_I2STxDataPut

Writes data samples to the I2S transmit FIFO with blocking.

Prototype:

```
void
ROM_I2STxDataPut(unsigned long ulBase,
                 unsigned long ulData)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxDataPut` is a function pointer located at `ROM_I2STABLE[3]`.

Parameters:

ulBase is the I2S module base address.

ulData is the single or dual channel I2S data.

Description:

This function writes a single channel sample or combined left-right samples to the I2S transmit FIFO. The format of the sample is determined by the configuration that was used with the function ROM_I2STxConfigSet(). If the transmit mode is **I2S_MODE_DUAL_STEREO** then the *ulData* parameter contains either the left or right sample. The left and right sample alternate with each write to the FIFO, left sample first. If the transmit mode is **I2S_MODE_COMPACT_STEREO_16** or **I2S_MODE_COMPACT_STEREO_8**, then the *ulData* parameter contains both the left and right samples. If the transmit mode is **I2S_MODE_SINGLE_MONO** then the *ulData* parameter contains the single channel sample.

For the compact modes, both the left and right samples are written at the same time. If 16-bit compact mode is used, then the least significant 16 bits contain the left sample, and the most significant 16 bits contain the right sample. If 8-bit compact mode is used, then the lower 8 bits contain the left sample, and the next 8 bits contain the right sample, with the upper 16 bits unused.

If there is no room in the transmit FIFO, then this function will wait in a polling loop until the data can be written.

Returns:
None.

13.2.1.16 ROM_I2STxDataPutNonBlocking

Writes data samples to the I2S transmit FIFO without blocking.

Prototype:
```
long
ROM_I2STxDataPutNonBlocking(unsigned long ulBase,
                            unsigned long ulData)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxDataPutNonBlocking` is a function pointer located at `ROM_I2STABLE[4]`.

Parameters:
ulBase is the I2S module base address.
ulData is the single or dual channel I2S data.

Description:
This function writes a single channel sample or combined left-right samples to the I2S transmit FIFO. The format of the sample is determined by the configuration that was used with the function ROM_I2STxConfigSet(). If the transmit mode is **I2S_MODE_DUAL_STEREO** then the *ulData* parameter contains either the left or right sample. The left and right sample alternate with each write to the FIFO, left sample first. If the transmit mode is **I2S_MODE_COMPACT_STEREO_16** or **I2S_MODE_COMPACT_STEREO_8**, then the *ulData* parameter contains both the left and right samples. If the transmit mode is **I2S_MODE_SINGLE_MONO** then the *ulData* parameter contains the single channel sample.

For the compact modes, both the left and right samples are written at the same time. If 16-bit compact mode is used, then the least significant 16 bits contain the left sample, and the most significant 16 bits contain the right sample. If 8-bit compact mode is used, then the lower 8 bits contain the left sample, and the next 8 bits contain the right sample, with the upper 16 bits unused.

If there is no room in the transmit FIFO, then this function will return immediately without writing any data to the FIFO.

Returns:
The number of elements written to the I2S transmit FIFO (1 or 0).

13.2.1.17 ROM_I2STxDisable

Disables the I2S transmit module for operation.

Prototype:

```
void
ROM_I2STxDisable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxDisable` is a function pointer located at `ROM_I2STABLE[2]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function disables the transmit module for operation. The module should be disabled before configuration. When the module is disabled, no data or clocks will be generated on the I2S signals.

Returns:

None.

13.2.1.18 ROM_I2STxEnable

Enables the I2S transmit module for operation.

Prototype:

```
void
ROM_I2STxEnable(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxEnable` is a function pointer located at `ROM_I2STABLE[1]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function enables the transmit module for operation. The module should be enabled after configuration. When the module is disabled, no data or clocks will be generated on the I2S signals.

Returns:

None.

13.2.1.19 ROM_I2STxFIFOLevelGet

Gets the number of samples in the transmit FIFO.

Prototype:

```
unsigned long
ROM_I2STxFIFOLevelGet(unsigned long ulBase)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_I2STABLE is an array of pointers located at ROM_APITABLE[22].
ROM_I2STxFIFOLevelGet is a function pointer located at ROM_I2STABLE[8].

Parameters:

ulBase is the I2S module base address.

Description:

This function is used to get the number of samples in the transmit FIFO. For the purposes of measuring the FIFO level, a left-right sample pair counts as 2, whether the mode is dual or compact stereo. When mono mode is used, internally the mono sample is still treated as a sample pair, so a single mono sample counts as 2. Since the FIFO always deals with sample pairs, normally the level will be an even number from 0 to 16. If dual stereo mode is used and only the left sample has been written without the matching right sample, then the FIFO level will be an odd value. If the FIFO level is odd, it indicates a left-right sample mismatch.

Returns:

Returns the number of samples in the transmit FIFO, which will normally be an even number.

13.2.1.20 ROM_I2STxFIFOLimitGet

Gets the current setting of the FIFO service request level.

Prototype:

```
unsigned long
ROM_I2STxFIFOLimitGet(unsigned long ulBase)
```

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_I2STABLE is an array of pointers located at ROM_APITABLE[22].
ROM_I2STxFIFOLimitGet is a function pointer located at ROM_I2STABLE[7].

Parameters:

ulBase is the I2S module base address.

Description:

This function is used to get the value of the transmit FIFO service request level. This value is set using the ROM_I2STxFIFOLimitSet() function.

Returns:

Returns the current value of the FIFO service request limit.

13.2.1.21 ROM_I2STxFIFOLimitSet

Sets the FIFO level at which a service request is generated.

Prototype:

```
void
ROM_I2STxFIFOLimitSet(unsigned long ulBase,
                      unsigned long ulLevel)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxFIFOLimitSet` is a function pointer located at `ROM_I2STABLE[6]`.

Parameters:

ulBase is the I2S module base address.

ulLevel is the FIFO service request limit.

Description:

This function is used to set the transmit FIFO fullness level at which a service request will occur. The service request is used to generate an interrupt or a DMA transfer request. The transmit FIFO will generate a service request when the number of items in the FIFO is less than the level specified in the *ulLevel* parameter. For example, if *ulLevel* is 8, then a service request will be generated when there are less than 8 samples remaining in the transmit FIFO.

For the purposes of counting the FIFO level, a left-right sample pair counts as 2, whether the mode is dual or compact stereo. When mono mode is used, internally the mono sample is still treated as a sample pair, so a single mono sample counts as 2. Since the FIFO always deals with sample pairs, the level must be an even number from 0 to 16. The maximum value is 16, which will cause a service request when there is any room in the FIFO. The minimum value is 0, which disables the service request.

Returns:

None.

13.2.1.22 ROM_I2STxRxConfigSet

Configures the I2S transmit and receive modules.

Prototype:

```
void
ROM_I2STxRxConfigSet(unsigned long ulBase,
                     unsigned long ulConfig)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxRxConfigSet` is a function pointer located at `ROM_I2STABLE[19]`.

Parameters:

ulBase is the I2S module base address.

ulConfig is the logical OR of the configuration options.

Description:

This function is used to configure the options for the I2S transmit and receive channels with identical parameters. The parameter *ulConfig* is the logical OR of the following options:

- **I2S_CONFIG_FORMAT_I2S** for standard I2S format, **I2S_CONFIG_FORMAT_LEFT_JUST** for left justified format, or **I2S_CONFIG_FORMAT_RIGHT_JUST** for right justified format.
- **I2S_CONFIG_SCLK_INVERT** to invert the polarity of the serial bit clock.

- **I2S_CONFIG_MODE_DUAL** for dual channel stereo, **I2S_CONFIG_MODE_COMPACT_16** for 16-bit compact stereo mode, **I2S_CONFIG_MODE_COMPACT_8** for 8-bit compact stereo mode, or **I2S_CONFIG_MODE_MONO** for single channel mono format.
- **I2S_CONFIG_CLK_MASTER** or **I2S_CONFIG_CLK_SLAVE** to select whether the I2S transmitter is the clock master or slave.
- **I2S_CONFIG_SAMPLE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per sample.
- **I2S_CONFIG_WIRE_SIZE_32**, **_24**, **_20**, **_16**, or **_8** to select the number of bits per word that are transferred on the data line.
- **I2S_CONFIG_EMPTY_ZERO** or **I2S_CONFIG_EMPTY_REPEAT** to select whether the module transmits zeroes or repeats the last sample when the FIFO is empty.

Returns:
None.

13.2.1.23 ROM_I2STxRxDisable

Disables the I2S transmit and receive modules.

Prototype:
```
void
ROM_I2STxRxDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxRxDisable` is a function pointer located at `ROM_I2STABLE[18]`.

Parameters:
ulBase is the I2S module base address.

Description:
This function simultaneously disables the transmit and receive modules. When the module is disabled, no data or clocks will be generated on the I2S signals.

Returns:
None.

13.2.1.24 ROM_I2STxRxEnable

Enables the I2S transmit and receive modules for operation.

Prototype:
```
void
ROM_I2STxRxEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_I2STABLE` is an array of pointers located at `ROM_APITABLE[22]`.
`ROM_I2STxRxEnable` is a function pointer located at `ROM_I2STABLE[17]`.

Parameters:

ulBase is the I2S module base address.

Description:

This function simultaneously enables the transmit and receive modules for operation, providing a synchronized SCLK and LRCLK. The module should be enabled after configuration. When the module is disabled, no data or clocks will be generated on the I2S signals.

Returns:

None.

14 Interrupt Controller (NVIC)

14.1 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.

14.2 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)

14.2.1 Function Documentation

14.2.1.1 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.

14.2.1.2 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.

14.2.1.3 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.

14.2.1.4 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.

14.2.1.5 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.

14.2.1.6 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.

14.2.1.7 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.

14.2.1.8 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.

14.2.1.9 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.

14.2.1.10 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 will be 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.

14.2.1.11 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 will be masked.

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

Smaller numbers correspond to higher interrupt priorities. 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 will be 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.

14.2.1.12 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:
ullnterrupt 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.

15 Memory Protection Unit (MPU)

15.1 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().

15.2 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)

15.2.1 Function Documentation

15.2.1.1 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.

15.2.1.2 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 will be 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.

15.2.1.3 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().

15.2.1.4 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.

15.2.1.5 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 will be applied and access violations will cause a memory management fault.

Returns:
None.

15.2.1.6 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 will be preserved in the attributes that are saved.

Returns:

None.

15.2.1.7 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.

16 Synchronous Serial Interface (SSI)

16.1 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.

16.2 Functions

Functions

- tBoolean ROM_SSIBusy (unsigned long ulBase)
- 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)

16.2.1 Function Documentation

16.2.1.1 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.

16.2.1.2 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 will be the same as the processor clock. This will be the value returned by ROM_SysCtlClockGet(), or it can be explicitly hard-coded if it is constant and known (to save the code/execution overhead of a call to ROM_SysCtlClockGet()).

Returns:

None.

16.2.1.3 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.

16.2.1.4 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.

16.2.1.5 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.

16.2.1.6 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.

16.2.1.7 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.

16.2.1.8 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.

16.2.1.9 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.

16.2.1.10 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.

16.2.1.11 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.

16.2.1.12 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.

16.2.1.13 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.

16.2.1.14 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**.

16.2.1.15 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.

17 System Control

17.1 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 3.579545 MHz and 16.384 MHz (that is, the standard crystal frequencies in that range). When direct clocking with an external oscillator or the main oscillator, the frequency is constrained to between 0 Hz and 100 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.

17.2 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_SysCtlDelay (unsigned long ulCount)
- unsigned long ROM_SysCtlFlashSizeGet (void)
- void ROM_SysCtlGPIOAHBDisable (unsigned long ulGPIOPeripheral)
- void ROM_SysCtlGPIOAHBEnable (unsigned long ulGPIOPeripheral)
- unsigned long ROM_SysCtlI2SMClkSet (unsigned long ulInputClock, unsigned long ulMClk)
- void ROM_SysCtlIntClear (unsigned long ulInts)
- void ROM_SysCtlIntDisable (unsigned long ulInts)
- void ROM_SysCtlIntEnable (unsigned long ulInts)
- unsigned long ROM_SysCtlIntStatus (tBoolean bMasked)
- unsigned long ROM_SysCtlLDOGet (void)
- void ROM_SysCtlLDOSet (unsigned long ulVoltage)
- void ROM_SysCtlPeripheralClockGating (tBoolean bEnable)
- void ROM_SysCtlPeripheralDeepSleepDisable (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralDeepSleepEnable (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralDisable (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralEnable (unsigned long ulPeripheral)
- tBoolean ROM_SysCtlPeripheralPresent (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralReset (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralSleepDisable (unsigned long ulPeripheral)
- void ROM_SysCtlPeripheralSleepEnable (unsigned long ulPeripheral)
- tBoolean ROM_SysCtlPinPresent (unsigned long ulPin)
- void ROM_SysCtlReset (void)
- void ROM_SysCtlResetCauseClear (unsigned long ulCauses)
- unsigned long ROM_SysCtlResetCauseGet (void)
- void ROM_SysCtlSleep (void)
- unsigned long ROM_SysCtlSRAMSizeGet (void)
- void ROM_SysCtlUSBPLLDisable (void)
- void ROM_SysCtlUSBPLLEnable (void)

17.2.1 Function Documentation

17.2.1.1 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; will be one of **SYSCTL_ADCSPEED_1MSPS**, **SYSCTL_ADCSPEED_500KSPS**, **SYSCTL_ADCSPEED_250KSPS**, or **SYSCTL_ADCSPEED_125KSPS**.

17.2.1.2 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.

17.2.1.3 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.

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.

17.2.1.4 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**, or **SYSCTL_OSC_INT30**.

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

To clock the system from an external source (such as an external crystal oscillator), use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the main oscillator, use **SYSCTL_USE_OSC** | **SYSCTL_OSC_MAIN**. To clock the system from the PLL, use **SYSCTL_USE_PLL** | **SYSCTL_OSC_MAIN**, and select the appropriate crystal with one of the **SYSCTL_XTAL_xxx** values.

Note:

If selecting the PLL as the system clock source (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.

17.2.1.5 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.

17.2.1.6 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.

17.2.1.7 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.

17.2.1.8 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.

17.2.1.9 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.

17.2.1.10 ROM_SysCtlI2SMClkSet

Sets the MCLK frequency provided to the I2S module.

Prototype:

```
unsigned long
ROM_SysCtlI2SMClkSet(unsigned long ulInputClock,
                     unsigned long ulMClk)
```

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_SysCtlI2SMClkSet` is a function pointer located at `ROM_SYSCTLTABLE[33]`.

Parameters:

ulInputClock is the input clock to the MCLK divider. If this is zero, the value is computed from the current PLL configuration.

ulMClk is the desired MCLK frequency. If this is zero, MCLK output is disabled.

Description:

This function sets the dividers to provide MCLK to the I2S module. A MCLK divider will be chosen that produces the MCLK frequency that is the closest possible to the requested frequency, which may be above or below the requested frequency.

The actual MCLK frequency will be returned. It is the responsibility of the application to determine if the selected MCLK is acceptable; in general the human ear can not discern the frequency difference if it is within 0.3% of the desired frequency (though there is a very small percentage of the population that can discern lower frequency deviations).

Returns:

Returns the actual MCLK frequency.

17.2.1.11 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.

17.2.1.12 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.

17.2.1.13 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.

17.2.1.14 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**.

17.2.1.15 ROM_SysCtlLDOGet

Gets the output voltage of the LDO.

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

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

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

Returns:
Returns the current voltage of the LDO; will be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

17.2.1.16 ROM_SysCtlLDOSet

Sets the output voltage of the LDO.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlLDOSet` is a function pointer located at `ROM_SYSCTLTABLE[17]`.

Parameters:
ulVoltage is the required output voltage from the LDO. Must be one of **SYSCTL_LDO_2_25V**, **SYSCTL_LDO_2_30V**, **SYSCTL_LDO_2_35V**, **SYSCTL_LDO_2_40V**, **SYSCTL_LDO_2_45V**, **SYSCTL_LDO_2_50V**, **SYSCTL_LDO_2_55V**, **SYSCTL_LDO_2_60V**, **SYSCTL_LDO_2_65V**, **SYSCTL_LDO_2_70V**, or **SYSCTL_LDO_2_75V**.

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

Returns:
None.

17.2.1.17 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.

17.2.1.18 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:
None.

17.2.1.19 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:
None.

17.2.1.20 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:

None.

17.2.1.21 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

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.

17.2.1.22 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.

17.2.1.23 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:

None.

17.2.1.24 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:

None.

17.2.1.25 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_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_I2C0**, **SYSCTL_PERIPH_I2C1**, **SYSCTL_PERIPH_I2S0**, **SYSCTL_PERIPH_SSI0**, **SYSCTL_PERIPH_SSI1**, **SYSCTL_PERIPH_TIMER0**, **SYSCTL_PERIPH_TIMER1**, **SYSCTL_PERIPH_TIMER2**, **SYSCTL_PERIPH_TIMER3**, **SYSCTL_PERIPH_UART0**, **SYSCTL_PERIPH_UART1**, **SYSCTL_PERIPH_UART2**, **SYSCTL_PERIPH_UDMA**, **SYSCTL_PERIPH_USB0**, **SYSCTL_PERIPH_WDOG0**, or **SYSCTL_PERIPH_WDOG1**.

Returns:

None.

17.2.1.26 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_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.

17.2.1.27 ROM_SysCtlReset

Resets the device.

Prototype:

```
void
ROM_SysCtlReset(void)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlReset` is a function pointer located at `ROM_SYSCTLTABLE[19]`.

Description:

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

Returns:

This function does not return.

17.2.1.28 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.

17.2.1.29 ROM_SysCtlResetCauseGet

Gets the reason for a reset.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSCTLTABLE` is an array of pointers located at `ROM_APITABLE[13]`.
`ROM_SysCtlResetCauseGet` is a function pointer located at `ROM_SYSCTLTABLE[21]`.

Description:

This function will return the reason(s) for a reset. Since the reset reasons are sticky until either cleared by software or an external reset, multiple reset reasons may be returned if multiple resets have occurred. The reset reason will be a logical OR of **SYSCTL_CAUSE_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.

17.2.1.30 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.

17.2.1.31 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.

17.2.1.32 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.

17.2.1.33 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.

18 System Tick (SysTick)

18.1 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.

18.2 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)

18.2.1 Function Documentation

18.2.1.1 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.

18.2.1.2 ROM_SysTickEnable

Enables the SysTick counter.

Prototype:
```
void
ROM_SysTickEnable(void)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_SYSTICKTABLE` is an array of pointers located at `ROM_APITABLE[10]`.
`ROM_SysTickEnable` is a function pointer located at `ROM_SYSTICKTABLE[1]`.

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

Note:
Calling this function will cause the SysTick counter to (re)commence counting from its current value. The counter is not automatically reloaded with the period as specified in a previous call to ROM_SysTickPeriodSet(). If an immediate reload is required, 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.

18.2.1.3 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.

18.2.1.4 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.

18.2.1.5 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.

18.2.1.6 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.

18.2.1.7 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.

19 Timer

19.1 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.

19.2 Functions

Functions

- void ROM_TimerConfigure (unsigned long ulBase, unsigned long ulConfig)
- 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)
- void ROM_TimerLoadSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long ROM_TimerMatchGet (unsigned long ulBase, unsigned long ulTimer)
- void ROM_TimerMatchSet (unsigned long ulBase, unsigned long ulTimer, unsigned long ulValue)
- unsigned long ROM_TimerPrescaleGet (unsigned long ulBase, unsigned long ulTimer)
- 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)

19.2.1 Function Documentation

19.2.1.1 ROM_TimerConfigure

Configures the timer(s).

Prototype:

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

ROM Location:

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

Parameters:

ulBase is the base address of the timer module.

ulConfig is the configuration for the timer.

Description:

This function configures the operating mode of the timer(s). The timer module is disabled before being configured, and is left in the disabled state. The configuration is specified in *ulConfig* as one of the following values:

- **TIMER_CFG_32_BIT_OS** - 32-bit one-shot timer
- **TIMER_CFG_32_BIT_OS_UP** - 32-bit one-shot timer that counts up instead of down
- **TIMER_CFG_32_BIT_PER** - 32-bit periodic timer
- **TIMER_CFG_32_BIT_PER_UP** - 32-bit periodic timer that counts up instead of down
- **TIMER_CFG_32_RTC** - 32-bit real time clock timer
- **TIMER_CFG_16_BIT_PAIR** - Two 16-bit timers

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

- **TIMER_CFG_A_ONE_SHOT** - 16-bit one-shot timer
- **TIMER_CFG_A_ONE_SHOT_UP** - 16-bit one-shot timer that counts up instead of down
- **TIMER_CFG_A_PERIODIC** - 16-bit periodic timer
- **TIMER_CFG_A_PERIODIC_UP** - 16-bit periodic timer that counts up instead of down
- **TIMER_CFG_A_CAP_COUNT** - 16-bit edge count capture
- **TIMER_CFG_A_CAP_COUNT_UP** - 16-bit edge count capture that counts up instead of down
- **TIMER_CFG_A_CAP_TIME** - 16-bit edge time capture
- **TIMER_CFG_A_CAP_TIME_UP** - 16-bit edge time capture that counts up instead of down
- **TIMER_CFG_A_PWM** - 16-bit PWM output

Similarly, the second timer is configured by setting *ulConfig* to the result of a logical OR operation between one of the corresponding **TIMER_CFG_B_**$*$ values and *ulConfig*.

Returns:
None.

19.2.1.2 ROM_TimerControlLevel

Controls the output level.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerControlLevel` is a function pointer located at `ROM_TIMERTABLE[4]`.

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.
bInvert specifies the output level.

Description:
This function sets the PWM output level for the specified timer. If the *bInvert* parameter is **true**, then the timer's output will be made active low; otherwise, it will be made active high.

Returns:
None.

19.2.1.3 ROM_TimerControlStall

Controls the stall handling.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerControlStall` is a function pointer located at `ROM_TIMERTABLE[7]`.

Parameters:
ulBase is the base address of the timer module.
ulTimer specifies the timer(s) to be adjusted; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.
bStall specifies the response to a stall signal.

Description:
This function controls the stall response for the specified timer. If the *bStall* parameter is **true**, then the timer will stop counting if the processor enters debug mode; otherwise the timer will keep running while in debug mode.

Returns:
None.

19.2.1.4 ROM_TimerControlTrigger

Enables or disables the trigger output.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerControlTrigger` is a function pointer located at `ROM_TIMERTABLE[5]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

bEnable specifies the desired trigger state.

Description:

This function controls the trigger output for the specified timer. If the *bEnable* parameter is **true**, then the timer's output trigger is enabled; otherwise it is disabled.

Returns:

None.

19.2.1.5 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.

19.2.1.6 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.

19.2.1.7 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.

19.2.1.8 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.

19.2.1.9 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.

19.2.1.10 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.

19.2.1.11 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().

19.2.1.12 ROM_TimerLoadGet

Gets the timer load value.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerLoadGet` is a function pointer located at `ROM_TIMERTABLE[15]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:

This function gets the currently programmed interval load value for the specified timer.

Returns:

Returns the load value for the timer.

19.2.1.13 ROM_TimerLoadSet

Sets the timer load value.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerLoadSet` is a function pointer located at `ROM_TIMERTABLE[14]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

ulValue is the load value.

Description:

This function sets the timer load value; if the timer is running then the value will be immediately loaded into the timer.

Returns:

None.

19.2.1.14 ROM_TimerMatchGet

Gets the timer match value.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerMatchGet` is a function pointer located at `ROM_TIMERTABLE[18]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:

This function gets the match value for the specified timer.

Returns:

Returns the match value for the timer.

19.2.1.15 ROM_TimerMatchSet

Sets the timer match value.

Prototype:

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

ROM Location:

ROM_APITABLE is an array of pointers located at 0x0100.0010.
ROM_TIMERTABLE is an array of pointers located at ROM_APITABLE[11].
ROM_TimerMatchSet is a function pointer located at ROM_TIMERTABLE[17].

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

ulValue is the match value.

Description:

This function sets the match value for a timer. This is used in capture count mode to determine when to interrupt the processor and in PWM mode to determine the duty cycle of the output signal.

Returns:

None.

19.2.1.16 ROM_TimerPrescaleGet

Get the timer prescale value.

Prototype:

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

ROM Location:

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

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**.

Description:

This function gets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:

The value of the timer prescaler.

19.2.1.17 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 16-bit mode that uses the counter match and prescaler, the prescale match effectively extends the range of the counter to 24-bits.

Returns:

The value of the timer prescale match.

19.2.1.18 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; must be between 0 and 255, inclusive.

Description:

This function sets the value of the input clock prescaler match value. When in a 16-bit mode that uses the counter match and the prescaler, the prescale match effectively extends the range of the counter to 24-bits.

Returns:

None.

19.2.1.19 ROM_TimerPrescaleSet

Set the timer prescale value.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerPrescaleSet` is a function pointer located at `ROM_TIMERTABLE[10]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer(s) to adjust; must be one of **TIMER_A**, **TIMER_B**, or **TIMER_BOTH**.

ulValue is the timer prescale value; must be between 0 and 255, inclusive.

Description:

This function sets the value of the input clock prescaler. The prescaler is only operational when in 16-bit mode and is used to extend the range of the 16-bit timer modes.

Returns:

None.

19.2.1.20 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.

19.2.1.21 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.

19.2.1.22 ROM_TimerValueGet

Gets the current timer value.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_TIMERTABLE` is an array of pointers located at `ROM_APITABLE[11]`.
`ROM_TimerValueGet` is a function pointer located at `ROM_TIMERTABLE[16]`.

Parameters:

ulBase is the base address of the timer module.

ulTimer specifies the timer; must be one of **TIMER_A** or **TIMER_B**. Only **TIMER_A** should be used when the timer is configured for 32-bit operation.

Description:

This function reads the current value of the specified timer.

Returns:

Returns the current value of the timer.

20 UART

20.1 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

20.2 Functions

Functions

- 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)
- 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)

20.2.1 Function Documentation

20.2.1.1 ROM_UARTBreakCtl

Causes a BREAK to be sent.

Prototype:

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

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_UARTTABLE` is an array of pointers located at `ROM_APITABLE[1]`.
`ROM_UARTBreakCtl` is a function pointer located at `ROM_UARTTABLE[16]`.

Parameters:

ulBase is the base address of the UART port.

bBreakState controls the output level.

Description:

Calling this function with *bBreakState* set to **true** 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.

20.2.1.2 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.

20.2.1.3 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*.

20.2.1.4 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.

20.2.1.5 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.

20.2.1.6 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.

20.2.1.7 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.

20.2.1.8 ROM_UARTConfigGetExpClk

Gets the current configuration of a UART.

Prototype:

```
void
ROM_UARTConfigGetExpClk(unsigned long ulBase,
                        unsigned long ulUARTClk,
                        unsigned long *pulBaud,
                        unsigned long *pulConfig)
```

ROM Location:

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

Parameters:

ulBase is the base address of the UART port.

ulUARTClk is the rate of the clock supplied to the UART module.

pulBaud is a pointer to storage for the baud rate.

pulConfig is a pointer to storage for the data format.

Description:

The baud rate and data format for the UART is determined, given an explicitly provided peripheral clock (hence the ExpClk suffix). The returned baud rate is the actual baud rate; it may not be the exact baud rate requested or an "official" baud rate. The data format returned in *pulConfig* is enumerated the same as the *ulConfig* parameter of ROM_UARTConfigSetExpClk().

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

Returns:

None.

20.2.1.9 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 will be the same as the processor clock. This will be the value returned by ROM_SysCtlClockGet(), or it can be explicitly hard-coded if it is constant and known (to save the code/execution overhead of a call to ROM_SysCtlClockGet()).

Returns:

None.

20.2.1.10 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.

20.2.1.11 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.

20.2.1.12 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.

20.2.1.13 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.

20.2.1.14 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.

20.2.1.15 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.

20.2.1.16 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.

20.2.1.17 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.

20.2.1.18 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.

20.2.1.19 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.

20.2.1.20 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.

20.2.1.21 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.

20.2.1.22 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_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.

20.2.1.23 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().

20.2.1.24 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**.

20.2.1.25 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.

20.2.1.26 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.

20.2.1.27 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**.

20.2.1.28 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.

20.2.1.29 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 will be **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 will be **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**.

20.2.1.30 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.

20.2.1.31 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.

21 uDMA Controller

21.1 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.

21.2 Functions

Functions

- 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)

21.2.1 Function Documentation

21.2.1.1 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.2 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.3 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.4 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.

21.2.1.5 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.6 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 will be 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**

- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.7 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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.

21.2.1.8 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 will be **UDMA_MODE_STOP**.

Returns:

Returns the transfer mode of the specified channel and control structure, which will be one of the following values: **UDMA_MODE_STOP**, **UDMA_MODE_BASIC**, **UDMA_MODE_AUTO**, **UDMA_MODE_PINGPONG**, **UDMA_MODE_MEM_SCATTER_GATHER**, or **UDMA_MODE_PER_SCATTER_GATHER**.

21.2.1.9 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_SEC_CHANNEL_EPI0RX**
- **UDMA_SEC_CHANNEL_EPI0TX**
- **UDMA_CHANNEL_ETH0RX**
- **UDMA_CHANNEL_ETH0TX**
- **UDMA_CHANNEL_I2S0RX**
- **UDMA_CHANNEL_I2S0TX**
- **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 will be signaled on the uDMA dedicated interrupt. If a peripheral channel is used, then the completion will be signaled on the peripheral's interrupt.

Returns:
None.

21.2.1.10 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 will be memory scatter-gather transfer)

Description:

This function is used to configure a channel for scatter-gather mode. The caller must have already set up a task list, and pass a pointer to the start of the task list as the *pvTaskList* parameter. The *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.

21.2.1.11 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_**) will be 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_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**
- **UDMA_DEF_I2S0RX_SEC_RESERVED**
- **UDMA_DEF_I2S0TX_SEC_RESERVED**

Returns:
None.

21.2.1.12 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_**) will be 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_TMR1A_SEC_EPI0RX**
- **UDMA_DEF_TMR1B_SEC_EPI0TX**
- **UDMA_DEF_SSI1RX_SEC_ADC10**
- **UDMA_DEF_SSI1TX_SEC_ADC11**
- **UDMA_DEF_RESERVED_SEC_ADC12**
- **UDMA_DEF_RESERVED_SEC_ADC13**

Returns:

None.

21.2.1.13 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 will be returned. If the transfer is complete, then 0 will be returned.

Returns:

Returns the number of items remaining to transfer.

21.2.1.14 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 will be unpredictable, including the possibility of undesired data transfers to or from memory or peripherals. For BASIC and AUTO modes, it is safe to make changes when the channel is disabled, or the 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.

21.2.1.15 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.

21.2.1.16 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.

21.2.1.17 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.

21.2.1.18 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.

21.2.1.19 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.

21.2.1.20 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.

21.2.1.21 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.

22 USB Controller

22.1 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);
```

22.2 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.
    //
    ...
}
```

22.3 Functions

Functions

- 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_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_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_USBPHYPowerOff (unsigned long ulBase)
- void ROM_USBPHYPowerOn (unsigned long ulBase)

22.3.1 Function Documentation

22.3.1.1 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.

22.3.1.2 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.

22.3.1.3 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.

22.3.1.4 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.

22.3.1.5 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 will be written with the maximum packet size for this endpoint.
pulFlags is a pointer which will be 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.

22.3.1.6 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.

22.3.1.7 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.

22.3.1.8 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 will be issued on the IN portion of this endpoint. If

the *ulFlags* parameter is **USB_EP_DEV_OUT** then the stall will be issued on the OUT portion of this endpoint.

Note:
This function should only be called in device mode.

Returns:
None.

22.3.1.9 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 will 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.

22.3.1.10 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.

22.3.1.11 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.

22.3.1.12 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 will be 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 will be 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.

22.3.1.13 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.

22.3.1.14 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.

22.3.1.15 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.

22.3.1.16 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.

22.3.1.17 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.

22.3.1.18 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.

22.3.1.19 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.

22.3.1.20 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.

22.3.1.21 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.

22.3.1.22 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 will be 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.

22.3.1.23 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.

22.3.1.24 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.

22.3.1.25 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.

22.3.1.26 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 will be 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.

22.3.1.27 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.

22.3.1.28 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 will be set to the DATA0 state, and if it is **true** it will be 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.

22.3.1.29 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 will 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.

22.3.1.30 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.

22.3.1.31 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.

22.3.1.32 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.

22.3.1.33 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.

22.3.1.34 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.

22.3.1.35 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.

22.3.1.36 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.

22.3.1.37 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.

22.3.1.38 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.

22.3.1.39 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 will be signaled when the status packet has been received.

Returns:

None.

22.3.1.40 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.

22.3.1.41 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.

22.3.1.42 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**.

22.3.1.43 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.

22.3.1.44 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 will be 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.

22.3.1.45 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.

22.3.1.46 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.

22.3.1.47 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 will be 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.

22.3.1.48 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.

22.3.1.49 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.

22.3.1.50 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.

22.3.1.51 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.

22.3.1.52 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.

22.3.1.53 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**.

22.3.1.54 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.

22.3.1.55 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 will be unable to operate.

Returns:

None.

22.3.1.56 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.

23 Watchdog Timer

23.1 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.

23.2 Functions

Functions

- void ROM_WatchdogEnable (unsigned long ulBase)
- void ROM_WatchdogIntClear (unsigned long ulBase)
- void ROM_WatchdogIntEnable (unsigned long ulBase)
- unsigned long ROM_WatchdogIntStatus (unsigned long ulBase, tBoolean bMasked)
- void ROM_WatchdogLock (unsigned long ulBase)
- tBoolean ROM_WatchdogLockState (unsigned long ulBase)
- unsigned long ROM_WatchdogReloadGet (unsigned long ulBase)
- void ROM_WatchdogReloadSet (unsigned long ulBase, unsigned long ulLoadVal)
- void ROM_WatchdogResetDisable (unsigned long ulBase)
- void ROM_WatchdogResetEnable (unsigned long ulBase)
- tBoolean ROM_WatchdogRunning (unsigned long ulBase)
- void ROM_WatchdogStallDisable (unsigned long ulBase)
- void ROM_WatchdogStallEnable (unsigned long ulBase)
- void ROM_WatchdogUnlock (unsigned long ulBase)

- unsigned long ROM_WatchdogValueGet (unsigned long ulBase)

23.2.1 Function Documentation

23.2.1.1 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.

23.2.1.2 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.

23.2.1.3 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.

23.2.1.4 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.

23.2.1.5 ROM_WatchdogLock

Enables the watchdog timer lock mechanism.

Prototype:

```
void
ROM_WatchdogLock(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogLock` is a function pointer located at `ROM_WATCHDOGTABLE[5]`.

Parameters:

ulBase is the base address of the watchdog timer module.

Description:

Locks out write access to the watchdog timer configuration registers.

Returns:

None.

23.2.1.6 ROM_WatchdogLockState

Gets the state of the watchdog timer lock mechanism.

Prototype:

```
tBoolean
ROM_WatchdogLockState(unsigned long ulBase)
```

ROM Location:

`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogLockState` is a function pointer located at `ROM_WATCHDOGTABLE[7]`.

Parameters:

ulBase is the base address of the watchdog timer module.

Description:
Returns the lock state of the watchdog timer registers.

Returns:
Returns **true** if the watchdog timer registers are locked, and **false** if they are not locked.

23.2.1.7 ROM_WatchdogReloadGet

Gets the watchdog timer reload value.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogReloadGet` is a function pointer located at `ROM_WATCHDOGTABLE[9]`.

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This function gets the value that is loaded into the watchdog timer when the count reaches zero for the first time.

See also:
ROM_WatchdogReloadSet()

Returns:
None.

23.2.1.8 ROM_WatchdogReloadSet

Sets the watchdog timer reload value.

Prototype:
```
void
ROM_WatchdogReloadSet(unsigned long ulBase,
                      unsigned long ulLoadVal)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogReloadSet` is a function pointer located at `ROM_WATCHDOGTABLE[8]`.

Parameters:
ulBase is the base address of the watchdog timer module.
ulLoadVal is the load value for the watchdog timer.

Description:
This function sets the value to load into the watchdog timer when the count reaches zero for the first time; if the watchdog timer is running when this function is called, then the value will be immediately loaded into the watchdog timer counter. If the *ulLoadVal* parameter is 0, then an interrupt is immediately generated.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
ROM_WatchdogLock(), ROM_WatchdogUnlock(), ROM_WatchdogReloadGet()

Returns:
None.

23.2.1.9 ROM_WatchdogResetDisable

Disables the watchdog timer reset.

Prototype:
```
void
ROM_WatchdogResetDisable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogResetDisable` is a function pointer located at `ROM_WATCHDOGTABLE[4]`.

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Disables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
ROM_WatchdogLock(), ROM_WatchdogUnlock()

Returns:
None.

23.2.1.10 ROM_WatchdogResetEnable

Enables the watchdog timer reset.

Prototype:
```
void
ROM_WatchdogResetEnable(unsigned long ulBase)
```

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogResetEnable` is a function pointer located at `ROM_WATCHDOGTABLE[3]`.

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
Enables the capability of the watchdog timer to issue a reset to the processor upon a second timeout condition.

Note:
This function will have no effect if the watchdog timer has been locked.

See also:
ROM_WatchdogLock(), ROM_WatchdogUnlock()

Returns:
None.

23.2.1.11 ROM_WatchdogRunning

Determines if the watchdog timer is enabled.

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

ROM Location:
`ROM_APITABLE` is an array of pointers located at `0x0100.0010`.
`ROM_WATCHDOGTABLE` is an array of pointers located at `ROM_APITABLE[12]`.
`ROM_WatchdogRunning` is a function pointer located at `ROM_WATCHDOGTABLE[1]`.

Parameters:
ulBase is the base address of the watchdog timer module.

Description:
This will check to see if the watchdog timer is enabled.

Returns:
Returns **true** if the watchdog timer is enabled, and **false** if it is not.

23.2.1.12 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.

23.2.1.13 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.

23.2.1.14 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.

23.2.1.15 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**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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