EK-LM3S2965 Firmware Development Package

USER'S GUIDE

SW-EK-LM3S2965-UG-10636

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 10636 of this document, last updated on March 28, 2013.

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® EK-LM3S2965 evaluation board is a platform that can be used for software development and to prototype a hardware design. It contains a Stellaris ARM® Cortex™-M3-based microcontroller, along with an OLED display, push buttons, a small speaker, and a CAN interface that can be used to exercise the peripherals on the microcontroller. Additionally, all of the microcontroller's pins are brought to unpopulated stake headers, allowing for easy connection to other hardware for the purposes of prototyping (after the stake headers have been populated by the customer).

This document describes the board-specific drivers and example applications that are provided for this development board.

2 Example Applications

The example applications show how to utilize features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the drivers provided by the peripheral driver library. These applications are intended for demonstration and as a starting point for new applications.

There is an IAR workspace file (`ek-lm3s2965.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`ek-lm3s2965.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/ek-lm3s2965` subdirectory of the firmware development package source distribution.

2.1 Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M3 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

2.2 Blinky (blinky)

A very simple example that blinks the on-board LED.

2.3 Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART and branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

2.4 Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of the boot loader. After being started by the boot loader, the application will configure the UART, wait for select button to be pressed, and then branch back to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

Both the boot loader and the application must be placed into flash. Once the boot loader is in flash, it can be used to program the application into flash as well. Then, the boot loader can be used to replace the application with another.

The boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

2.5 Boot Loader (boot_serial)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Stellaris microcontroller, utilizing either UART0, I2C0, SSI0, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses UART0 to load an application.

2.6 CAN device FIFO mode example (can_device_fifo)

This application uses the CAN controller in FIFO mode to communicate with the CAN FIFO mode example program that is running on the main board. The main board must have the can_fifo example program loaded for this example to function. This application will simply echo all data that it receives back in it's receive FIFO back out it's transmit FIFO.

2.7 CAN Device Board LED Application (can_device_led)

This simple application uses the two buttons on the board as a light switch. When the "up" button is pressed the status LED will turn on. When the "down" button is pressed, the status LED will turn off.

2.8 CAN Device Board Quickstart Application (can_device_qs)

This application uses the CAN controller to communicate with the evaluation board that is running the example game. It receives messages over CAN to turn on, turn off, or to pulse the LED on the device board. It also sends CAN messages when either of the up and down buttons are pressed or released.

2.9 CAN FIFO mode example (can_fifo)

This application uses the CAN controller in FIFO mode to communicate with the CAN device board. The CAN device board must have the can_device_fifo example program loaded and running prior to starting this application. This program expects the CAN device board to echo back the data that it receives and this application will then compare the data received with the transmitted data and look for differences. This application will then modify the data and continue transmitting and receiving data over the CAN bus indefinitely.

Note:
This application must be started after the can_device_fifo example has started on the CAN device board.

2.10 GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the select push button will toggle the pins between JTAG mode and GPIO mode. Because there is no debouncing of the push button (either in hardware or software), a button press will occasionally result in more than one mode change.

In this example, all five pins (PB7, PC0, PC1, PC2, and PC3) are switched, though the more typical use would be to change PB7 into a GPIO.

2.11 Graphics Example (graphics)

A simple application that displays scrolling text on the top line of the OLED display, along with a 4-bit gray scale image.

2.12 Hello World (hello)

A very simple “hello world” example. It simply displays “hello world” on the OLED and is a starting point for more complicated applications.

2.13 Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the OLED; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the

off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

2.14 MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

2.15 PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 25% duty cycle PWM signal and a 75% duty cycle PWM signal, both at 440 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

2.16 EK-LM3S2965 Quickstart Application (qs_ek-lm3s2965)

A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly.

The game is started by pressing the select push button on the right side of the board. During game play, the select push button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the board will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

If the CAN device board is attached and is running the can_device_qs application, the volume of the music and sound effects can be adjusted over CAN with the two push buttons on the target board. The LED on the CAN device board will track the state of the LED on the main board via CAN messages. The operation of the game will not be affected by the absence of the CAN device board.

Since the OLED display on the evaluation board has burn-in characteristics similar to a CRT, the application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user push button being pressed while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by pressing the

select push button. The select push button will then need to be pressed again to start the game.

2.17 Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

2.18 UART Echo (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

2.19 Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If the watchdog is not periodically fed, it will reset the system. Each time the watchdog is fed, the LED is inverted so that it is easy to see that it is being fed, which occurs once every second.

3 Development System Utilities

These are tools that run on the development system, not on the embedded target. They are provided to assist in the development of firmware for Stellaris microcontrollers.

These tools reside in the `tools` subdirectory of the firmware development package source distribution.

AES Key Expansion Utility

Usage:

```
aes_gen_key [OPTIONS] --keysize=[SIZE] --key=[KEYSTRING] [FILE]
```

Description:

Generates pre-expanded keys for AES encryption and decryption. It is designed to work in conjunction with the AES library code found in the StellarisWare directory `third_party/aes`. When using an AES key to perform encryption or decryption, the key must first be expanded into a larger table of values before the key can be used. This operation can be performed at run-time but takes time and uses space in RAM.

If the keys are fixed and known in advance, then it is possible to perform the expansion operation at build-time and the pre-expanded table can be built into the code. The advantages of doing this are that it saves time when the keys are used, and the expanded table is stored in non-volatile program memory (flash), which is usually less precious in a typical microcontroller application.

By default, the pre-expanded key is generated as a data array that can be used by reference in the application. It is also possible to generate the pre-expanded key as a code sequence. A function is generated that will copy the pre-expanded key to a caller supplied buffer. This does not save RAM space, but it makes the expanded key more secure. By making the key into pure code (versus data in flash), the Texas Instruments Stellaris OTP feature can be used to make the code execute only (no read). This means that the expanded key cannot be read from flash. It is only loaded into RAM during an encrypt or decrypt operation.

The length of a pre-set key is 44 words for 128-bit keys, 54 words for 192-bit keys, and 68 words for 256-bit keys; instruction-based versions are about two to four times as large in flash and require as much RAM as run-time expansion.

The source code for this utility is contained in `tools/aes_gen_key`, with a pre-built binary contained in `tools/bin`.

Arguments:

-a, --data generates expanded key as an array of data.

-x, --code generates expanded key as executable code.

-e, --encrypt generate expanded key for encryption.

-d, --decrypt generate expanded key for decryption.

-s, --keysize KEYSIZE size of the key in bits (128, 192, or 256).

-k, --key KEY key value in hexadecimal.

-v, --version show program version.

-h, --help display usage information.

The **--keysize** and **--key** arguments are mandatory. Only one each of **--data** or **--code**, and **--encrypt** or **--decrypt** should be used. If not specified otherwise then the default is **--data --encrypt**.

FILE is the name of the file that will be created containing the expanded key. This file will be in the form of a C header file and should be included in your application.

Example:

The following will generate an expanded 128-bit key for encryption, encoded as data and create a C header file named `enc_key.h`:

```
aes_gen_key --data --encrypt --keysize=128
            --key=112233445566778899AABBCCDDEEFF00 enc_key.h
```

The following will generate an expanded 128-bit key for decryption, encoded as a code function and create a C header file named `dec_key.h`:

```
aes_gen_key --code --decrypt --keysize=128
            --key=112233445566778899AABBCCDDEEFF00 dec_key.h
```

Serial Flash Downloader

Usage:

```
sflash [OPTION]... [INPUT FILE]
```

Description:

Downloads a firmware image to a Stellaris board using a UART connection to the Stellaris Serial Flash Loader or the Stellaris Boot Loader. This has the same capabilities as the serial download portion of the Stellaris Flash Programmer.

The source code for this utility is contained in `tools/sflash`, with a pre-built binary contained in `tools/bin`.

Arguments:

-b BAUD specifies the baud rate. If not specified, the default of 115,200 will be used.

-c PORT specifies the COM port. If not specified, the default of COM1 will be used.

-d disables auto-baud.

-h displays usage information.

-l FILENAME specifies the name of the boot loader image file.

-p ADDR specifies the address at which to program the firmware. If not specified, the default of 0 will be used.

-r ADDR specifies the address at which to start processor execution after the firmware has been downloaded. If not specified, the processor will be reset after the firmware has been downloaded.

-s SIZE specifies the size of the data packets used to download the firmware date. This must be a multiple of four between 8 and 252, inclusive. If using the Serial Flash Loader, the maximum value that can be used is 76. If using the Boot Loader, the maximum value that can be used is dependent upon the configuration of the Boot Loader. If not specified, the default of 8 will be used.

INPUT FILE specifies the name of the firmware image file.

Example:

The following will download a firmware image to the board over COM2 without auto-baud support:

```
sflash -c 2 -d image.bin
```

4 Display Driver

4.1 Introduction

The display driver provides a way to draw text and images on the 128x96 OLED display. The display can also be turned on or off as required in order to preserve the OLED display, which has the same image burn-in characteristics as a CRT display.

This driver is located in `boards/ek-lm3s2965/drivers`, with `rit128x96x4.c` containing the source code and `rit128x96x4.h` containing the API definitions for use by applications.

4.2 API Functions

Functions

- void RIT128x96x4Clear (void)
- void RIT128x96x4Disable (void)
- void RIT128x96x4DisplayOff (void)
- void RIT128x96x4DisplayOn (void)
- void RIT128x96x4Enable (unsigned long ulFrequency)
- void RIT128x96x4ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void RIT128x96x4Init (unsigned long ulFrequency)
- void RIT128x96x4StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY, unsigned char ucLevel)

4.2.1 Function Documentation

4.2.1.1 RIT128x96x4Clear

Clears the OLED display.

Prototype:

```
void
RIT128x96x4Clear(void)
```

Description:

This function will clear the display RAM. All pixels in the display will be turned off.

Returns:

None.

4.2.1.2 RIT128x96x4Disable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Disable(void)
```

Description:
This function initializes the SSI interface to the OLED display.

Returns:
None.

4.2.1.3 RIT128x96x4DisplayOff

Turns off the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOff(void)
```

Description:
This function will turn off the OLED display. This will stop the scanning of the panel and turn off the on-chip DC-DC converter, preventing damage to the panel due to burn-in (it has similar characters to a CRT in this respect).

Returns:
None.

4.2.1.4 RIT128x96x4DisplayOn

Turns on the OLED display.

Prototype:

```
void
RIT128x96x4DisplayOn(void)
```

Description:
This function will turn on the OLED display, causing it to display the contents of its internal frame buffer.

Returns:
None.

4.2.1.5 RIT128x96x4Enable

Enable the SSI component of the OLED display driver.

Prototype:

```
void
RIT128x96x4Enable(unsigned long ulFrequency)
```

Parameters:

ulFrequency specifies the SSI Clock Frequency to be used.

Description:

This function initializes the SSI interface to the OLED display.

Returns:

None.

4.2.1.6 RIT128x96x4ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
RIT128x96x4ImageDraw(const unsigned char *pucImage,
                     unsigned long ulX,
                     unsigned long ulY,
                     unsigned long ulWidth,
                     unsigned long ulHeight)
```

Parameters:

pucImage is a pointer to the image data.

ulX is the horizontal position to display this image, specified in columns from the left edge of the display.

ulY is the vertical position to display this image, specified in rows from the top of the display.

ulWidth is the width of the image, specified in columns.

ulHeight is the height of the image, specified in rows.

Description:

This function will display a bitmap graphic on the display. Because of the format of the display RAM, the starting column (*ulX*) and the number of columns (*ulWidth*) must be an integer multiple of two.

The image data is organized with the first row of image data appearing left to right, followed immediately by the second row of image data. Each byte contains the data for two columns in the current row, with the leftmost column being contained in bits 7:4 and the rightmost column being contained in bits 3:0.

For example, an image six columns wide and seven scan lines tall would be arranged as follows (showing how the twenty one bytes of the image would appear on the display):

```
+-------------------+-------------------+-------------------+
|      Byte 0       |      Byte 1       |      Byte 2       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 3       |      Byte 4       |      Byte 5       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
```

```
+---------+---------+---------+---------+---------+---------+
|      Byte 6       |      Byte 7       |      Byte 8       |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 9       |      Byte 10      |      Byte 11      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 12      |      Byte 13      |      Byte 14      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 15      |      Byte 16      |      Byte 17      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
|      Byte 18      |      Byte 19      |      Byte 20      |
+---------+---------+---------+---------+---------+---------+
| 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 | 7 6 5 4 | 3 2 1 0 |
+---------+---------+---------+---------+---------+---------+
```

Returns:
None.

4.2.1.7 RIT128x96x4Init

Initialize the OLED display.

Prototype:

```
void
RIT128x96x4Init(unsigned long ulFrequency)
```

Parameters:
ulFrequency specifies the SSI Clock Frequency to be used.

Description:
This function initializes the SSI interface to the OLED display and configures the SSD1329 controller on the panel.

Returns:
None.

4.2.1.8 RIT128x96x4StringDraw

Displays a string on the OLED display.

Prototype:

```
void
RIT128x96x4StringDraw(const char *pcStr,
                      unsigned long ulX,
                      unsigned long ulY,
                      unsigned char ucLevel)
```

Parameters:

pcStr is a pointer to the string to display.

ulX is the horizontal position to display the string, specified in columns from the left edge of the display.

ulY is the vertical position to display the string, specified in rows from the top edge of the display.

ucLevel is the 4-bit gray scale value to be used for displayed text.

Description:

This function will draw a string on the display. Only the ASCII characters between 32 (space) and 126 (tilde) are supported; other characters will result in random data being draw on the display (based on whatever appears before/after the font in memory). The font is mono-spaced, so characters such as “i” and “l” have more white space around them than characters such as “m” or “w”.

If the drawing of the string reaches the right edge of the display, no more characters will be drawn. Therefore, special care is not required to avoid supplying a string that is “too long” to display.

Note:

Because the OLED display packs 2 pixels of data in a single byte, the parameter *ulX* must be an even column number (for example, 0, 2, 4, and so on).

Returns:

None.

4.3 Programming Example

The following example shows how to use the display driver to display text on the OLED display.

```
//
// Initialize the OLED display with a 1 MHz interface clock.
//
RIT128x96x4Init(1000000);

//
// Write text on the display.
//
RIT128x96x4StringDraw("Hello", 0, 0, 15);
```

5 Command Line Processing Module

5.1 Introduction

The command line processor allows a simple command line interface to be made available in an application, for example via a UART. It takes a buffer containing a string (which must be obtained by the application) and breaks it up into a command and arguments (in traditional C “argc, argv” format). The command is then found in a command table and the corresponding function in the table is called to process the command.

This module is contained in `utils/cmdline.c`, with `utils/cmdline.h` containing the API definitions for use by applications.

5.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_INVALID_ARG
- CMDLINE_TOO_FEW_ARGS
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

5.2.1 Data Structure Documentation

5.2.1.1 tCmdLineEntry

Definition:

```
typedef struct
{
    const char *pcCmd;
    pfnCmdLine pfnCmd;
    const char *pcHelp;
}
tCmdLineEntry
```

Members:

pcCmd A pointer to a string containing the name of the command.

pfnCmd A function pointer to the implementation of the command.

pcHelp A pointer to a string of brief help text for the command.

Description:

Structure for an entry in the command list table.

5.2.2 Define Documentation

5.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

Defines the value that is returned if the command is not found.

5.2.2.2 CMDLINE_INVALID_ARG

Definition:

```
#define CMDLINE_INVALID_ARG
```

Description:

Defines the value that is returned if an argument is invalid.

5.2.2.3 CMDLINE_TOO_FEW_ARGS

Definition:

```
#define CMDLINE_TOO_FEW_ARGS
```

Description:

Defines the value that is returned if there are too few arguments.

5.2.2.4 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

Defines the value that is returned if there are too many arguments.

5.2.3 Function Documentation

5.2.3.1 CmdLineProcess

Process a command line string into arguments and execute the command.

Prototype:

```
int
CmdLineProcess(char *pcCmdLine)
```

Parameters:

pcCmdLine points to a string that contains a command line that was obtained by an application by some means.

Description:

This function will take the supplied command line string and break it up into individual arguments. The first argument is treated as a command and is searched for in the command table. If the command is found, then the command function is called and all of the command line arguments are passed in the normal argc, argv form.

The command table is contained in an array named `g_sCmdTable` which must be provided by the application.

Returns:

Returns **CMDLINE_BAD_CMD** if the command is not found, **CMDLINE_TOO_MANY_ARGS** if there are more arguments than can be parsed. Otherwise it returns the code that was returned by the command function.

5.2.4 Variable Documentation

5.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

This is the command table that must be provided by the application.

5.3 Programming Example

The following example shows how to process a command line.

```
//
// Code for the "foo" command.
//
int
ProcessFoo(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "bar" command.
//
int
ProcessBar(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "help" command.
//
int
ProcessHelp(int argc, char *argv[])
{
    //
    // Provide help.
    //
}

//
// The table of commands supported by this application.
//
tCmdLineEntry g_sCmdTable[] =
{
    { "foo", ProcessFoo, "The first command." },
    { "bar", ProcessBar, "The second command." },
    { "help", ProcessHelp, "Application help." }
};

//
// Read a process a command.
//
int
Test(void)
{
    unsigned char pucCmd[256];

    //
    // Retrieve a command from the user into pucCmd.
    //
    ...

    //
    // Process the command line.
    //
    return(CmdLineProcess(pucCmd));
}
```

6 CPU Usage Module

6.1 Introduction

The CPU utilization module uses one of the system timers and peripheral clock gating to determine the percentage of the time that the processor is being clocked. For the most part, the processor is executing code whenever it is being clocked (exceptions occur when the clocking is being configured, which only happens at startup, and when entering/exiting an interrupt handler, when the processor is performing stacking operations on behalf of the application).

The specified timer is configured to run when the processor is in run mode and to not run when the processor is in sleep mode. Therefore, the timer will only count when the processor is being clocked. Comparing the number of clocks the timer counted during a fixed period to the number of clocks in the fixed period provides the percentage utilization.

In order for this to be effective, the application must put the processor to sleep when it has no work to do (instead of busy waiting). If the processor never goes to sleep (either because of a continual stream of work to do or a busy loop), the processor utilization will be reported as 100%.

Since deep-sleep mode changes the clocking of the system, the computed processor usage may be incorrect if deep-sleep mode is utilized. The number of clocks the processor spends in run mode will be properly counted, but the timing period may not be accurate (unless extraordinary measures are taken to ensure timing period accuracy).

The accuracy of the computed CPU utilization depends upon the regularity with which CPUUsageTick() is called by the application. If the CPU usage is constant, but CPUUsageTick() is called sporadically, the reported CPU usage will fluctuate as well despite the fact that the CPU usage is actually constant.

This module is contained in `utils/cpu_usage.c`, with `utils/cpu_usage.h` containing the API definitions for use by applications.

6.2 API Functions

Functions

- void CPUUsageInit (unsigned long ulClockRate, unsigned long ulRate, unsigned long ulTimer)
- unsigned long CPUUsageTick (void)

6.2.1 Function Documentation

6.2.1.1 CPUUsageInit

Initializes the CPU usage measurement module.

Prototype:

```
void
CPUUsageInit(unsigned long ulClockRate,
             unsigned long ulRate,
             unsigned long ulTimer)
```

Parameters:

ulClockRate is the rate of the clock supplied to the timer module.

ulRate is the number of times per second that CPUUsageTick() is called.

ulTimer is the index of the timer module to use.

Description:

This function prepares the CPU usage measurement module for measuring the CPU usage of the application.

Returns:

None.

6.2.1.2 CPUUsageTick

Updates the CPU usage for the new timing period.

Prototype:

```
unsigned long
CPUUsageTick(void)
```

Description:

This function, when called at the end of a timing period, will update the CPU usage.

Returns:

Returns the CPU usage percentage as a 16.16 fixed-point value.

6.3 Programming Example

The following example shows how to use the CPU usage module to measure the CPU usage where the foreground simply burns some cycles.

```
//
// The CPU usage for the most recent time period.
//
unsigned long g_ulCPUUsage;

//
// Handles the SysTick interrupt.
```

```
//
void
SysTickIntHandler(void)
{
    //
    // Compute the CPU usage for the last time period.
    //
    g_ulCPUUsage = CPUUsageTick();
}

//
// The main application.
//
int
main(void)
{
    //
    // Initialize the CPU usage module, using timer 0.
    //
    CPUUsageInit(8000000, 100, 0);

    //
    // Initialize SysTick to interrupt at 100 Hz.
    //
    SysTickPeriodSet(8000000 / 100);
    SysTickIntEnable();
    SysTickEnable();

    //
    // Loop forever.
    //
    while(1)
    {
        //
        // Delay for a little bit so that CPU usage is not zero.
        //
        SysCtlDelay(100);

        //
        // Put the processor to sleep.
        //
        SysCtlSleep();
    }
}
```

7 CRC Module

7.1 Introduction

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

A CRC is useful for detecting errors that occur during the transmission of data over a communications channel or during storage in a memory (such as flash). However, a CRC does not provide protection against an intentional modification or tampering of the data.

This module is contained in `utils/crc.c`, with `utils/crc.h` containing the API definitions for use by applications.

7.2 API Functions

Functions

- unsigned short Crc16 (unsigned short usCrc, const unsigned char *pucData, unsigned long ulCount)
- unsigned short Crc16Array (unsigned long ulWordLen, const unsigned long *pulData)
- void Crc16Array3 (unsigned long ulWordLen, const unsigned long *pulData, unsigned short *pusCrc3)
- unsigned long Crc32 (unsigned long ulCrc, const unsigned char *pucData, unsigned long ulCount)
- unsigned char Crc8CCITT (unsigned char ucCrc, const unsigned char *pucData, unsigned long ulCount)

7.2.1 Function Documentation

7.2.1.1 Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

```
unsigned short
Crc16(unsigned short usCrc,
      const unsigned char *pucData,
      unsigned long ulCount)
```

Parameters:

usCrc is the starting CRC-16 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

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

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

```
usCrc = Crc16(0, pucData1, ulLen1);
usCrc = Crc16(usCrc, pucData2, ulLen2);
usCrc = Crc16(usCrc, pucData3, ulLen3);
```

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

Returns:

The CRC-16 of the input data.

7.2.1.2 Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

```
unsigned short
Crc16Array(unsigned long ulWordLen,
           const unsigned long *pulData)
```

Parameters:

ulWordLen is the length of the array in words (the number of bytes divided by 4).

pulData is a pointer to the data buffer.

Description:

This function is a wrapper around the running CRC-16 function, providing the CRC-16 for a single block of data.

Returns:

The CRC-16 of the input data.

7.2.1.3 Crc16Array3

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

Prototype:

```
void
Crc16Array3(unsigned long ulWordLen,
            const unsigned long *pulData,
            unsigned short *pusCrc3)
```

Parameters:

ulWordLen is the length of the array in words (the number of bytes divided by 4).

pulData is a pointer to the data buffer.

pusCrc3 is a pointer to an array in which to place the three CRC-16 values.

Description:

This function is used to calculate three CRC-16s of the input buffer; the first uses every byte from the array, the second uses only the even-index bytes from the array (in other words, bytes 0, 2, 4, etc.), and the third uses only the odd-index bytes from the array (in other words, bytes 1, 3, 5, etc.).

Returns:

None

7.2.1.4 Crc32

Calculates the CRC-32 of an array of bytes.

Prototype:

```
unsigned long
Crc32(unsigned long ulCrc,
      const unsigned char *pucData,
      unsigned long ulCount)
```

Parameters:

ulCrc is the starting CRC-32 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-32 of the input buffer. The CRC-32 is computed in a running fashion, meaning that the entire data block that is to have its CRC-32 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ulCrc** should be set to 0xFFFFFFFF. If, however, the entire block of data is not available, then **ulCrc** should be set to 0xFFFFFFFF for the first portion of the data, and then the returned value should be passed back in as **ulCrc** for the next portion of the data. Once all data has been passed to the function, the final CRC-32 can be obtained by inverting the last returned value.

For example, to compute the CRC-32 of a block that has been split into three pieces, use the following:

```
ulCrc = Crc32(0xFFFFFFFF, pucData1, ulLen1);
ulCrc = Crc32(ulCrc, pucData2, ulLen2);
ulCrc = Crc32(ulCrc, pucData3, ulLen3);
ulCrc ^= 0xFFFFFFFF;
```

Computing a CRC-32 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The accumulated CRC-32 of the input data.

7.2.1.5 Crc8CCITT

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

Prototype:

```
unsigned char
Crc8CCITT(unsigned char ucCrc,
          const unsigned char *pucData,
          unsigned long ulCount)
```

Parameters:

ucCrc is the starting CRC-8-CCITT value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

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

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

```
ucCrc = Crc8CCITT(0, pucData1, ulLen1);
ucCrc = Crc8CCITT(ucCrc, pucData2, ulLen2);
ucCrc = Crc8CCITT(ucCrc, pucData3, ulLen3);
```

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

Returns:

The CRC-8-CCITT of the input data.

7.3 Programming Example

The following example shows how to compute the CRC-16 of a buffer of data.

```
unsigned long ulIdx, ulValue;
unsigned char pucData[256];

//
// Fill pucData with some data.
```

```
//
for(ulIdx = 0; ulIdx < 256; ulIdx++)
{
    pucData[ulIdx] = ulIdx;
}

//
// Compute the CRC-16 of the data.
//
ulValue = Crc16(0, pucData, 256);
```

8 Flash Parameter Block Module

8.1 Introduction

The flash parameter block module provides a simple, fault-tolerant, persistent storage mechanism for storing parameter information for an application.

The FlashPBInit() function is used to initialize a parameter block. The primary conditions for the parameter block are that flash region used to store the parameter blocks must contain at least two erase blocks of flash to ensure fault tolerance, and the size of the parameter block must be an integral divisor of the the size of an erase block. FlashPBGet() and FlashPBSave() are used to read and write parameter block data into the parameter region. The only constraints on the content of the parameter block are that the first two bytes of the block are reserved for use by the read/write functions as a sequence number and checksum, respectively.

This module is contained in `utils/flash_pb.c`, with `utils/flash_pb.h` containing the API definitions for use by applications.

8.2 API Functions

Functions

- unsigned char * FlashPBGet (void)
- void FlashPBInit (unsigned long ulStart, unsigned long ulEnd, unsigned long ulSize)
- void FlashPBSave (unsigned char *pucBuffer)

8.2.1 Function Documentation

8.2.1.1 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

```
unsigned char *
FlashPBGet(void)
```

Description:

This function returns the address of the most recent parameter block that is stored in flash.

Returns:

Returns the address of the most recent parameter block, or NULL if there are no valid parameter blocks in flash.

8.2.1.2 FlashPBInit

Initializes the flash parameter block.

Prototype:

```
void
FlashPBInit(unsigned long ulStart,
            unsigned long ulEnd,
            unsigned long ulSize)
```

Parameters:

ulStart is the address of the flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash.

ulEnd is the address of the end of flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash (the first block that is NOT part of the flash memory to be used), or the address of the first word after the flash array if the last block of flash is to be used.

ulSize is the size of the parameter block when stored in flash; this must be a power of two less than or equal to the flash erase block size (typically 1024).

Description:

This function initializes a fault-tolerant, persistent storage mechanism for a parameter block for an application. The last several erase blocks of flash (as specified by *ulStart* and *ulEnd* are used for the storage; more than one erase block is required in order to be fault-tolerant.

A parameter block is an array of bytes that contain the persistent parameters for the application. The only special requirement for the parameter block is that the first byte is a sequence number (explained in FlashPBSave()) and the second byte is a checksum used to validate the correctness of the data (the checksum byte is the byte such that the sum of all bytes in the parameter block is zero).

The portion of flash for parameter block storage is split into N equal-sized regions, where each region is the size of a parameter block (*ulSize*). Each region is scanned to find the most recent valid parameter block. The region that has a valid checksum and has the highest sequence number (with special consideration given to wrapping back to zero) is considered to be the current parameter block.

In order to make this efficient and effective, three conditions must be met. The first is *ulStart* and *ulEnd* must be specified such that at least two erase blocks of flash are dedicated to parameter block storage. If not, fault tolerance can not be guaranteed since an erase of a single block will leave a window where there are no valid parameter blocks in flash. The second condition is that the size (*ulSize*) of the parameter block must be an integral divisor of the size of an erase block of flash. If not, a parameter block will end up spanning between two erase blocks of flash, making it more difficult to manage. The final condition is that the size of the flash dedicated to parameter blocks (*ulEnd* - *ulStart*) divided by the parameter block size (*ulSize*) must be less than or equal to 128. If not, it will not be possible in all cases to determine which parameter block is the most recent (specifically when dealing with the sequence number wrapping back to zero).

When the microcontroller is initially programmed, the flash blocks used for parameter block storage are left in an erased state.

This function must be called before any other flash parameter block functions are called.

Returns:

None.

8.2.1.3 FlashPBSave

Writes a new parameter block to flash.

Prototype:

```
void
FlashPBSave(unsigned char *pucBuffer)
```

Parameters:

pucBuffer is the address of the parameter block to be written to flash.

Description:

This function will write a parameter block to flash. Saving the new parameter blocks involves three steps:

- Setting the sequence number such that it is one greater than the sequence number of the latest parameter block in flash.
- Computing the checksum of the parameter block.
- Writing the parameter block into the storage immediately following the latest parameter block in flash; if that storage is at the start of an erase block, that block is erased first.

By this process, there is always a valid parameter block in flash. If power is lost while writing a new parameter block, the checksum will not match and the partially written parameter block will be ignored. This is what makes this fault-tolerant.

Another benefit of this scheme is that it provides wear leveling on the flash. Since multiple parameter blocks fit into each erase block of flash, and multiple erase blocks are used for parameter block storage, it takes quite a few parameter block saves before flash is re-written.

Returns:

None.

8.3 Programming Example

The following example shows how to use the flash parameter block module to read the contents of a flash parameter block.

```
unsigned char pucBuffer[16], *pucPB;

//
// Initialize the flash parameter block module, using the last two pages of
// a 64 KB device as the parameter block.
//
FlashPBInit(0xf800, 0x10000, 16);

//
// Read the current parameter block.
//
pucPB = FlashPBGet();
if(pucPB)
{
    memcpy(pucBuffer, pucPB);
}
```

9 Integer Square Root Module

9.1 Introduction

The integer square root module provides an integer version of the square root operation that can be used instead of the floating point version provided in the C library. The algorithm used is a derivative of the manual pencil-and-paper method that used to be taught in school, and is closely related to the pencil-and-paper division method that is likely still taught in school.

For full details of the algorithm, see the article by Jack W. Crenshaw in the February 1998 issue of Embedded System Programming. It can be found online at `http://www.embedded.com/98/9802fe2.htm`.

This module is contained in `utils/isqrt.c`, with `utils/isqrt.h` containing the API definitions for use by applications.

9.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

9.2.1 Function Documentation

9.2.1.1 isqrt

Compute the integer square root of an integer.

Prototype:

```
unsigned long
isqrt(unsigned long ulValue)
```

Parameters:

ulValue is the value whose square root is desired.

Description:

This function will compute the integer square root of the given input value. Since the value returned is also an integer, it is actually better defined as the largest integer whose square is less than or equal to the input value.

Returns:

Returns the square root of the input value.

9.3 Programming Example

The following example shows how to compute the square root of a number.

```
unsigned long ulValue;

//
// Get the square root of 52378.  The result returned will be 228, which is
// the largest integer less than or equal to the square root of 52378.
//
ulValue = isqrt(52378);
```

10 Ring Buffer Module

10.1 Introduction

The ring buffer module provides a set of functions allowing management of a block of memory as a ring buffer. This is typically used in buffering transmit or receive data for a communication channel but has many other uses including implementing queues and FIFOs.

This module is contained in `utils/ringbuf.c`, with `utils/ringbuf.h` containing the API definitions for use by applications.

10.2 API Functions

Functions

- void RingBufAdvanceRead (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- void RingBufAdvanceWrite (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- unsigned long RingBufContigFree (tRingBufObject *ptRingBuf)
- unsigned long RingBufContigUsed (tRingBufObject *ptRingBuf)
- tBoolean RingBufEmpty (tRingBufObject *ptRingBuf)
- void RingBufFlush (tRingBufObject *ptRingBuf)
- unsigned long RingBufFree (tRingBufObject *ptRingBuf)
- tBoolean RingBufFull (tRingBufObject *ptRingBuf)
- void RingBufInit (tRingBufObject *ptRingBuf, unsigned char *pucBuf, unsigned long ulSize)
- void RingBufRead (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)
- unsigned char RingBufReadOne (tRingBufObject *ptRingBuf)
- unsigned long RingBufSize (tRingBufObject *ptRingBuf)
- unsigned long RingBufUsed (tRingBufObject *ptRingBuf)
- void RingBufWrite (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ulLength)
- void RingBufWriteOne (tRingBufObject *ptRingBuf, unsigned char ucData)

10.2.1 Function Documentation

10.2.1.1 RingBufAdvanceRead

Remove bytes from the ring buffer by advancing the read index.

Prototype:

```
void
RingBufAdvanceRead(tRingBufObject *ptRingBuf,
                   unsigned long ulNumBytes)
```

Parameters:

ptRingBuf points to the ring buffer from which bytes are to be removed.

ulNumBytes is the number of bytes to be removed from the buffer.

Description:

This function advances the ring buffer read index by a given number of bytes, removing that number of bytes of data from the buffer. If *ulNumBytes* is larger than the number of bytes currently in the buffer, the buffer is emptied.

Returns:

None.

10.2.1.2 RingBufAdvanceWrite

Add bytes to the ring buffer by advancing the write index.

Prototype:

```
void
RingBufAdvanceWrite(tRingBufObject *ptRingBuf,
                    unsigned long ulNumBytes)
```

Parameters:

ptRingBuf points to the ring buffer to which bytes have been added.

ulNumBytes is the number of bytes added to the buffer.

Description:

This function should be used by clients who wish to add data to the buffer directly rather than via calls to RingBufWrite() or RingBufWriteOne(). It advances the write index by a given number of bytes. If the *ulNumBytes* parameter is larger than the amount of free space in the buffer, the read pointer will be advanced to cater for the addition. Note that this will result in some of the oldest data in the buffer being discarded.

Returns:

None.

10.2.1.3 RingBufContigFree

Returns number of contiguous free bytes available in a ring buffer.

Prototype:

```
unsigned long
RingBufContigFree(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of contiguous free bytes ahead of the current write pointer in the ring buffer.

Returns:
Returns the number of contiguous bytes available in the ring buffer.

10.2.1.4 RingBufContigUsed

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

Prototype:
```
unsigned long
RingBufContigUsed(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of contiguous bytes of data available in the ring buffer ahead of the current read pointer. This represents the largest block of data which does not straddle the buffer wrap.

Returns:
Returns the number of contiguous bytes available.

10.2.1.5 RingBufEmpty

Determines whether the ring buffer whose pointers and size are provided is empty or not.

Prototype:
```
tBoolean
RingBufEmpty(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is empty. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is empty or **false** otherwise.

10.2.1.6 RingBufFlush

Empties the ring buffer.

Prototype:

```
void
RingBufFlush(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

Discards all data from the ring buffer.

Returns:

None.

10.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:

```
unsigned long
RingBufFree(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to check.

Description:

This function returns the number of bytes available in the ring buffer.

Returns:

Returns the number of bytes available in the ring buffer.

10.2.1.8 RingBufFull

Determines whether the ring buffer whose pointers and size are provided is full or not.

Prototype:

```
tBoolean
RingBufFull(tRingBufObject *ptRingBuf)
```

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

This function is used to determine whether or not a given ring buffer is full. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:

Returns **true** if the buffer is full or **false** otherwise.

10.2.1.9 RingBufInit

Initialize a ring buffer object.

Prototype:

```
void
RingBufInit(tRingBufObject *ptRingBuf,
            unsigned char *pucBuf,
            unsigned long ulSize)
```

Parameters:

ptRingBuf points to the ring buffer to be initialized.

pucBuf points to the data buffer to be used for the ring buffer.

ulSize is the size of the buffer in bytes.

Description:

This function initializes a ring buffer object, preparing it to store data.

Returns:

None.

10.2.1.10 RingBufRead

Reads data from a ring buffer.

Prototype:

```
void
RingBufRead(tRingBufObject *ptRingBuf,
            unsigned char *pucData,
            unsigned long ulLength)
```

Parameters:

ptRingBuf points to the ring buffer to be read from.

pucData points to where the data should be stored.

ulLength is the number of bytes to be read.

Description:

This function reads a sequence of bytes from a ring buffer.

Returns:

None.

10.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:

```
unsigned char
RingBufReadOne(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf points to the ring buffer to be written to.

Description:
This function reads a single byte of data from a ring buffer.

Returns:
The byte read from the ring buffer.

10.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

Prototype:
```
unsigned long
RingBufSize(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the size of the ring buffer.

Returns:
Returns the size in bytes of the ring buffer.

10.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

Prototype:
```
unsigned long
RingBufUsed(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of bytes stored in the ring buffer.

Returns:
Returns the number of bytes stored in the ring buffer.

10.2.1.14 RingBufWrite

Writes data to a ring buffer.

Prototype:

```
void
RingBufWrite(tRingBufObject *ptRingBuf,
             unsigned char *pucData,
             unsigned long ulLength)
```

Parameters:

ptRingBuf points to the ring buffer to be written to.

pucData points to the data to be written.

ulLength is the number of bytes to be written.

Description:

This function write a sequence of bytes into a ring buffer.

Returns:

None.

10.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:

```
void
RingBufWriteOne(tRingBufObject *ptRingBuf,
                unsigned char ucData)
```

Parameters:

ptRingBuf points to the ring buffer to be written to.

ucData is the byte to be written.

Description:

This function writes a single byte of data into a ring buffer.

Returns:

None.

10.3 Programming Example

The following example shows how to pass data through the ring buffer.

```
char pcBuffer[128], pcData[16];
tRingBufObject sRingBuf;

//
// Initialize the ring buffer.
//
RingBufInit(&sRingBuf, pcBuffer, sizeof(pcBuffer));

//
// Write some data into the ring buffer.
//
RingBufWrite(&sRingBuf, "Hello World", 11);
```

```
//
// Read the data out of the ring buffer.
//
RingBufRead(&sRingBuf, pcData, 11);
```

11 Simple Task Scheduler Module

11.1 Introduction

The simple task scheduler module offers an easy way to implement applications which rely upon a group of functions being called at regular time intervals. The module makes use of an application-defined task table listing functions to be called. Each task is defined by a function pointer, a parameter that will be passed to that function, the period between consecutive calls to the function and a flag indicating whether that particular task is enabled.

The scheduler makes use of the SysTick counter and interrupt to track time and calls enabled functions when the appropriate period has elapsed since the last call to that function.

In addition to providing the task table `g_psSchedulerTable[]` to the module, the application must also define a global variable `g_ulSchedulerNumTasks` containing the number of task entries in the table. The module also requires exclusive access to the SysTick hardware and the application must hook the scheduler's SysTick interrupt handler to the appropriate interrupt vector. Although the scheduler owns SysTick, functions are provided to allow the current system time to be queried and to calculate elapsed time between two system time values or between an earlier time value and the present time.

All times passed to the scheduler or returned from it are expressed in terms of system ticks. The basic system tick rate is set by the application when it initializes the scheduler module.

This module is contained in `utils/scheduler.c`, with `utils/scheduler.h` containing the API definitions for use by applications.

11.2 API Functions

Data Structures

- tSchedulerTask

Functions

- unsigned long SchedulerElapsedTicksCalc (unsigned long ulTickStart, unsigned long ulTickEnd)
- unsigned long SchedulerElapsedTicksGet (unsigned long ulTickCount)
- void SchedulerInit (unsigned long ulTicksPerSecond)
- void SchedulerRun (void)
- void SchedulerSysTickIntHandler (void)
- void SchedulerTaskDisable (unsigned long ulIndex)
- void SchedulerTaskEnable (unsigned long ulIndex, tBoolean bRunNow)

- unsigned long SchedulerTickCountGet (void)

Variables

- tSchedulerTask g_psSchedulerTable[ ]
- unsigned long g_ulSchedulerNumTasks

11.2.1 Data Structure Documentation

11.2.1.1 tSchedulerTask

Definition:

```
typedef struct
{
    void (*pfnFunction)(void *);
    void *pvParam;
    unsigned long ulFrequencyTicks;
    unsigned long ulLastCall;
    tBoolean bActive;
}
tSchedulerTask
```

Members:

pfnFunction A pointer to the function which is to be called periodically by the scheduler.

pvParam The parameter which is to be passed to this function when it is called.

ulFrequencyTicks The frequency the function is to be called expressed in terms of system ticks. If this value is 0, the function will be called on every call to SchedulerRun.

ulLastCall Tick count when this function was last called. This field is updated by the scheduler.

bActive A flag indicating whether or not this task is active. If true, the function will be called periodically. If false, the function is disabled and will not be called.

Description:

The structure defining a function which the scheduler will call periodically.

11.2.2 Function Documentation

11.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

```
unsigned long
SchedulerElapsedTicksCalc(unsigned long ulTickStart,
                          unsigned long ulTickEnd)
```

Parameters:

ulTickStart is the system tick count for the start of the period.

ulTickEnd is the system tick count for the end of the period.

Description:
This function may be called by a client to determine the number of ticks which have elapsed between provided starting and ending tick counts. The function takes into account wrapping cases where the end tick count is lower than the starting count assuming that the ending tick count always represents a later time than the starting count.

Returns:
The number of ticks elapsed between the provided start and end counts.

11.2.2.2 SchedulerElapsedTicksGet

Returns the number of ticks elapsed since the provided tick count.

Prototype:
```
unsigned long
SchedulerElapsedTicksGet(unsigned long ulTickCount)
```

Parameters:
ulTickCount is the tick count from which to determine the elapsed time.

Description:
This function may be called by a client to determine how much time has passed since a particular tick count provided in the *ulTickCount* parameter. This function takes into account wrapping of the global tick counter and assumes that the provided tick count always represents a time in the past. The returned value will, of course, be wrong if the tick counter has wrapped more than once since the passed *ulTickCount*. As a result, please do not use this function if you are dealing with timeouts of 497 days or longer (assuming you use a 10mS tick period).

Returns:
The number of ticks elapsed since the provided tick count.

11.2.2.3 SchedulerInit

Initializes the task scheduler.

Prototype:
```
void
SchedulerInit(unsigned long ulTicksPerSecond)
```

Parameters:
ulTicksPerSecond sets the basic frequency of the SysTick interrupt used by the scheduler to determine when to run the various task functions.

Description:
This function must be called during application startup to configure the SysTick timer. This is used by the scheduler module to determine when each of the functions provided in the g_psSchedulerTable array is called.

The caller is responsible for ensuring that SchedulerSysTickIntHandler() has previously been installed in the SYSTICK vector in the vector table and must also ensure that interrupts are enabled at the CPU level.

Note that this call does not start the scheduler calling the configured functions. All function calls are made in the context of later calls to SchedulerRun(). This call merely configures the SysTick interrupt that is used by the scheduler to determine what the current system time is.

Returns:
None.

11.2.2.4 SchedulerRun

Instructs the scheduler to update its task table and make calls to functions needing called.

Prototype:
```
void
SchedulerRun(void)
```

Description:
This function must be called periodically by the client to allow the scheduler to make calls to any configured task functions if it is their time to be called. The call must be made at least as frequently as the most frequent task configured in the g_psSchedulerTable array.

Although the scheduler makes use of the SysTick interrupt, all calls to functions configured in *g_psSchedulerTable* are made in the context of SchedulerRun().

Returns:
None.

11.2.2.5 SchedulerSysTickIntHandler

Handles the SysTick interrupt on behalf of the scheduler module.

Prototype:
```
void
SchedulerSysTickIntHandler(void)
```

Description:
Applications using the scheduler module must ensure that this function is hooked to the SysTick interrupt vector.

Returns:
None.

11.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

Prototype:
```
void
SchedulerTaskDisable(unsigned long ulIndex)
```

Parameters:

ulIndex is the index of the task which is to be disabled in the global *g_psSchedulerTable* array.

Description:

This function marks one of the configured tasks as inactive and prevents SchedulerRun() from calling it. The task may be reenabled by calling SchedulerTaskEnable().

Returns:

None.

11.2.2.7 SchedulerTaskEnable

Enables a task and allows the scheduler to call it periodically.

Prototype:

```
void
SchedulerTaskEnable(unsigned long ulIndex,
                    tBoolean bRunNow)
```

Parameters:

ulIndex is the index of the task which is to be enabled in the global *g_psSchedulerTable* array.

bRunNow is **true** if the task is to be run on the next call to SchedulerRun() or **false** if one whole period is to elapse before the task is run.

Description:

This function marks one of the configured tasks as enabled and causes SchedulerRun() to call that task periodically. The caller may choose to have the enabled task run for the first time on the next call to SchedulerRun() or to wait one full task period before making the first call.

Returns:

None.

11.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

Prototype:

```
unsigned long
SchedulerTickCountGet(void)
```

Description:

This function may be called by a client to retrieve the current system time. The value returned is a count of ticks elapsed since the system last booted.

Returns:

Tick count since last boot.

11.2.3 Variable Documentation

11.2.3.1 g_psSchedulerTable

Definition:

```
tSchedulerTask g_psSchedulerTable[]
```

Description:

This global table must be populated by the client and contains information on each function that the scheduler is to call.

11.2.3.2 g_ulSchedulerNumTasks

Definition:

```
unsigned long g_ulSchedulerNumTasks
```

Description:

This global variable must be exported by the client. It must contain the number of entries in the g_psSchedulerTable array.

11.3 Programming Example

The following example shows how to use the task scheduler module. This code illustrates a simple application which toggles two LEDs at different rates and updates a scrolling text string on the display.

```
//*****************************************************************************
//
// Definition of the system tick rate.  This results in a tick period of 10mS.
//
//*****************************************************************************
#define TICKS_PER_SECOND 100

//*****************************************************************************
//
// Prototypes of functions which will be called by the scheduler.
//
//*****************************************************************************
static void ScrollTextBanner(void *pvParam);
static void ToggleLED(void *pvParam);

//*****************************************************************************
//
// This table defines all the tasks that the scheduler is to run, the periods
// between calls to those tasks, and the parameter to pass to the task.
//
//*****************************************************************************
tSchedulerTask g_psSchedulerTable[] =
{
    //
    // Scroll the text banner 1 character to the left.  This function is called
    // every 20 ticks (5 times per second).
    //
    { ScrollTextBanner, (void *)0, 20, 0, true},
```

```
    //
    // Toggle LED number 0 every 50 ticks (twice per second).
    //
    { ToggleLED, (void *)0, 50, 0, true},

    //
    // Toggle LED number 1 every 100 ticks (once per second).
    //
    { ToggleLED, (void *)1, 100, 0, true},
};

//*****************************************************************************
//
// The number of entries in the global scheduler task table.
//
//*****************************************************************************
unsigned long g_ulSchedulerNumTasks = (sizeof(g_psSchedulerTable) /
                                       sizeof(tSchedulerTask));

//*****************************************************************************
//
// This function is called by the scheduler to toggle one of two LEDs
//
//*****************************************************************************
static void
ToggleLED(void *pvParam)
{
    long lState;

    ulState = GPIOPinRead(LED_GPIO_BASE
                          (pvParam ? LED1_GPIO_PIN : LED0_GPIO_PIN));
    GPIOPinWrite(LED_GPIO_BASE, (pvParam ? LED1_GPIO_PIN : LED0_GPIO_PIN),
                 ~lState);
}

//*****************************************************************************
//
// This function is called by the scheduler to scroll a line of text on the
// display.
//
//*****************************************************************************
static void
ScrollTextBanner(void *pvParam)
{
    //
    // Left as an exercise for the reader.
    //
}

//*****************************************************************************
//
// Application main task.
//
//*****************************************************************************
int
main(void)
{
    //
    // Initialize system clock and any peripherals that are to be used.
    //
    SystemInit();

    //
    // Initialize the task scheduler and configure the SysTick to interrupt
    // 100 times per second.
```

```
    //
    SchedulerInit(TICKS_PER_SECOND);

    //
    // Turn on interrupts at the CPU level.
    //
    IntMasterEnable();

    //
    // Drop into the main loop.
    //
    while(1)
    {
        //
        // Tell the scheduler to call any periodic tasks that are due to be
        // called.
        //
        SchedulerRun();
    }
}
```

12 Sine Calculation Module

12.1 Introduction

This module provides a fixed-point sine function. The input angle is a 0.32 fixed-point value that is the percentage of 360 degrees. This has two benefits; the sine function does not have to handle angles that are outside the range of 0 degrees through 360 degrees (in fact, 360 degrees can not be represented since it would wrap to 0 degrees), and the computation of the angle can be simplified since it does not have to deal with wrapping at values that are not natural for binary arithmetic (such as 360 degrees or 2π radians).

A sine table is used to find the approximate value for a given input angle. The table contains 128 entries that range from 0 degrees through 90 degrees and the symmetry of the sine function is used to determine the value between 90 degrees and 360 degrees. The maximum error caused by this table-based approach is 0.00618, which occurs near 0 and 180 degrees.

This module is contained in `utils/sine.c`, with `utils/sine.h` containing the API definitions for use by applications.

12.2 API Functions

Defines

- cosine(ulAngle)

Functions

- long sine (unsigned long ulAngle)

12.2.1 Define Documentation

12.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

```
#define cosine(ulAngle)
```

Parameters:

ulAngle is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

Description:

This function computes the cosine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

Returns:

Returns the cosine of the angle, in 16.16 fixed point format.

12.2.2 Function Documentation

12.2.2.1 sine

Computes an approximation of the sine of the input angle.

Prototype:

```
long
sine(unsigned long ulAngle)
```

Parameters:

ulAngle is an angle expressed as a 0.32 fixed-point value that is the percentage of the way around a circle.

Description:

This function computes the sine for the given input angle. The angle is specified in 0.32 fixed point format, and is therefore always between 0 and 360 degrees, inclusive of 0 and exclusive of 360.

Returns:

Returns the sine of the angle, in 16.16 fixed point format.

12.3 Programming Example

The following example shows how to produce a sine wave with 7 degrees between successive values.

```
unsigned long ulValue;

//
// Produce a sine wave with each step being 7 degrees advanced from the
// previous.
//
for(ulValue = 0; ; ulValue += 0x04FA4FA4)
{
    //
    // Compute the sine at this angle and do something with the result.
    //
    sine(ulValue);
}
```

13 Software I2C Module

13.1 Introduction

The software I2C module uses a timer and two GPIO pins to create a slow-speed software I2C peripheral. Multiple software I2C modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software I2C module supports master mode only; multi-master support is not provided. A callback mechanism is used to simulate the interrupts that would be provided by a hardware I2C module.

The API for the software I2C module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware I2C module. The two notable differences are the function prefix being “SoftI2C” instead of “I2CMaster”, and the first argument of each API is a pointer to the tSoftI2C data structure instead of the base address of the hardware module.

Timing for the software I2C module is provided by the application. The SoftI2CTimerTick() function must be called on a periodic basis to provide the timing for the software I2C module. The timer tick function must be called at four times the desired I2C clock rate; for example, to operate the software I2C interface at 10 KHz, the tick function must be called at a 40 KHz rate. By having the application providing the timing, the timer resource can be flexible and multiple software I2C modules can be driven from a single timer resource. Alternatively, if the software I2C module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire I2C transaction has completed (maximizing both I2C clock speed and processor usage, but not requiring a timer).

The software I2C module requires two GPIO pins; one for SCL and one for SDA. The per-instance data structure is approximately 20 bytes in length (the actual length depends on how the structure is packed by the compiler).

As a point of reference, the following are some rough measurements of the processor usage of the software I2C module at various I2C clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software I2C module, processor clock speed, interrupt priority, and compiler.

I2C Clock	% Of Processor	Million Cycles Per Second
5 KHz	4.53	2.26
10 KHz	9.05	4.52
15 KHz	13.53	6.76
20 KHz	18.03	9.01
25 KHz	22.51	11.25
30 KHz	27.05	13.52
35 KHz	31.52	15.76
40 KHz	36.06	18.03
45 KHz	40.54	20.27
50 KHz	44.96	22.48

This module is contained in `utils/softi2c.c`, with `utils/softi2c.h` containing the API definitions for use by applications.

13.2 API Functions

Data Structures

- tSoftI2C

Functions

- tBoolean SoftI2CBusy (tSoftI2C *pI2C)
- void SoftI2CCallbackSet (tSoftI2C *pI2C, void (*pfnCallback)(void))
- void SoftI2CControl (tSoftI2C *pI2C, unsigned long ulCmd)
- unsigned long SoftI2CDataGet (tSoftI2C *pI2C)
- void SoftI2CDataPut (tSoftI2C *pI2C, unsigned char ucData)
- unsigned long SoftI2CErr (tSoftI2C *pI2C)
- void SoftI2CInit (tSoftI2C *pI2C)
- void SoftI2CIntClear (tSoftI2C *pI2C)
- void SoftI2CIntDisable (tSoftI2C *pI2C)
- void SoftI2CIntEnable (tSoftI2C *pI2C)
- tBoolean SoftI2CIntStatus (tSoftI2C *pI2C, tBoolean bMasked)
- void SoftI2CSCLGPIOSet (tSoftI2C *pI2C, unsigned long ulBase, unsigned char ucPin)
- void SoftI2CSDAGPIOSet (tSoftI2C *pI2C, unsigned long ulBase, unsigned char ucPin)
- void SoftI2CSlaveAddrSet (tSoftI2C *pI2C, unsigned char ucSlaveAddr, tBoolean bReceive)
- void SoftI2CTimerTick (tSoftI2C *pI2C)

13.2.1 Data Structure Documentation

13.2.1.1 tSoftI2C

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulSCLGPIO;
    unsigned long ulSDAGPIO;
    unsigned char ucFlags;
    unsigned char ucSlaveAddr;
    unsigned char ucData;
    unsigned char ucState;
    unsigned char ucCurrentBit;
    unsigned char ucIntMask;
    unsigned char ucIntStatus;
}
tSoftI2C
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware I2C implementation. This address can be set via a direct structure access or using the SoftI2CCallbackSet function.

ulSCLGPIO The address of the GPIO pin to be used for the SCL signal. This member can be set via a direct structure access or using the SoftI2CSCLGPIOSet function.

ulSDAGPIO The address of the GPIO pin to be used for the SDA signal. This member can be set via a direct structure access or using the SoftI2CSDAGPIOSet function.

ucFlags The flags that control the operation of the SoftI2C module. This member should not be accessed or modified by the application.

ucSlaveAddr The slave address that is currently being accessed. This member should not be accessed or modified by the application.

ucData The data that is currently being transmitted or received. This member should not be accessed or modified by the application.

ucState The current state of the SoftI2C state machine. This member should not be accessed or modified by the application.

ucCurrentBit The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.

ucIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

ucIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

Description:

This structure contains the state of a single instance of a SoftI2C module.

13.2.2 Function Documentation

13.2.2.1 SoftI2CBusy

Indicates whether or not the SoftI2C module is busy.

Prototype:

```
tBoolean
SoftI2CBusy(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function returns an indication of whether or not the SoftI2C module is busy transmitting or receiving data.

Returns:

Returns **true** if the SoftI2C module is busy; otherwise, returns **false**.

13.2.2.2 SoftI2CCallbackSet

Sets the callback used by the SoftI2C module.

Prototype:

```
void
SoftI2CCallbackSet(tSoftI2C *pI2C,
                   void (*pfnCallback)(void))
```

Parameters:

pI2C specifies the SoftI2C data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftI2C module.

Returns:

None.

13.2.2.3 SoftI2CControl

Controls the state of the SoftI2C module.

Prototype:

```
void
SoftI2CControl(tSoftI2C *pI2C,
               unsigned long ulCmd)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulCmd command to be issued to the SoftI2C module.

Description:

This function is used to control the state of the SoftI2C module send and receive operations. The *ucCmd* parameter can be one of the following values:

- **SOFTI2C_CMD_SINGLE_SEND**
- **SOFTI2C_CMD_SINGLE_RECEIVE**
- **SOFTI2C_CMD_BURST_SEND_START**
- **SOFTI2C_CMD_BURST_SEND_CONT**
- **SOFTI2C_CMD_BURST_SEND_FINISH**
- **SOFTI2C_CMD_BURST_SEND_ERROR_STOP**
- **SOFTI2C_CMD_BURST_RECEIVE_START**
- **SOFTI2C_CMD_BURST_RECEIVE_CONT**
- **SOFTI2C_CMD_BURST_RECEIVE_FINISH**
- **SOFTI2C_CMD_BURST_RECEIVE_ERROR_STOP**

Returns:

None.

13.2.2.4 SoftI2CDataGet

Receives a byte that has been sent to the SoftI2C module.

Prototype:

```
unsigned long
SoftI2CDataGet(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function reads a byte of data from the SoftI2C module that was received as a result of an appropriate call to SoftI2CControl().

Returns:

Returns the byte received by the SoftI2C module, cast as an unsigned long.

13.2.2.5 SoftI2CDataPut

Transmits a byte from the SoftI2C module.

Prototype:

```
void
SoftI2CDataPut(tSoftI2C *pI2C,
               unsigned char ucData)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ucData data to be transmitted from the SoftI2C module.

Description:

This function places the supplied data into SoftI2C module in preparation for being transmitted via an appropriate call to SoftI2CControl().

Returns:

None.

13.2.2.6 SoftI2CErr

Gets the error status of the SoftI2C module.

Prototype:

```
unsigned long
SoftI2CErr(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function is used to obtain the error status of the SoftI2C module send and receive operations.

Returns:
Returns the error status, as one of **SOFTI2C_ERR_NONE**, **SOFTI2C_ERR_ADDR_ACK**, or **SOFTI2C_ERR_DATA_ACK**.

13.2.2.7 SoftI2CInit

Initializes the SoftI2C module.

Prototype:
```
void
SoftI2CInit(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
This function initializes operation of the SoftI2C module. After successful initialization of the SoftI2C module, the software I2C bus is in the idle state.

Returns:
None.

13.2.2.8 SoftI2CIntClear

Clears the SoftI2C “interrupt”.

Prototype:
```
void
SoftI2CIntClear(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
The SoftI2C “interrupt” source is cleared, so that it no longer asserts. This function must be called in the “interrupt” handler to keep it from being called again immediately on exit.

Returns:
None.

13.2.2.9 SoftI2CIntDisable

Disables the SoftI2C “interrupt”.

Prototype:
```
void
SoftI2CIntDisable(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
Disables the SoftI2C “interrupt” source.

Returns:
None.

13.2.2.10 SoftI2CIntEnable

Enables the SoftI2C “interrupt”.

Prototype:
```
void
SoftI2CIntEnable(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
Enables the SoftI2C “interrupt” source.

Returns:
None.

13.2.2.11 SoftI2CIntStatus

Gets the current SoftI2C “interrupt” status.

Prototype:
```
tBoolean
SoftI2CIntStatus(tSoftI2C *pI2C,
                 tBoolean bMasked)
```

Parameters:
pI2C specifies the SoftI2C data structure.
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 SoftI2C 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.

13.2.2.12 SoftI2CSCLGPIOSet

Sets the GPIO pin to be used as the SoftI2C SCL signal.

Prototype:

```
void
SoftI2CSCLGPIOSet(tSoftI2C *pI2C,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftI2C SCL signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

13.2.2.13 SoftI2CSDAGPIOSet

Sets the GPIO pin to be used as the SoftI2C SDA signal.

Prototype:

```
void
SoftI2CSDAGPIOSet(tSoftI2C *pI2C,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftI2C SDA signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

13.2.2.14 SoftI2CSlaveAddrSet

Sets the address that the SoftI2C module places on the bus.

Prototype:

```
void
SoftI2CSlaveAddrSet(tSoftI2C *pI2C,
```

```
                        unsigned char ucSlaveAddr,
                        tBoolean bReceive)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ucSlaveAddr 7-bit slave address

bReceive flag indicating the type of communication with the slave.

Description:

This function sets the address that the SoftI2C module places on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address indicates that the SoftI2C moudle is initiating a read from the slave; otherwise the address indicates that the SoftI2C module is initiating a write to the slave.

Returns:

None.

13.2.2.15 SoftI2CTimerTick

Performs the periodic update of the SoftI2C module.

Prototype:

```
void
SoftI2CTimerTick(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function performs the periodic, time-based updates to the SoftI2C module. The transmission and reception of data over the SoftI2C link is performed by the state machine in this function.

This function must be called at four times the desired SoftI2C clock rate. For example, to run the SoftI2C clock at 10 KHz, this function must be called at a 40 KHz rate.

Returns:

None.

13.3 Programming Example

The following example shows how to configure the software I2C module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software I2C.
//
tSoftI2C g_sI2C;

//
// The timer tick function.
```

```
//
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software I2C timer tick function.
    //
    SoftI2CTimerTick(&g_sI2C);
}

//
// The callback function for the software I2C.  This function is equivalent
// to the interrupt handler for a hardware I2C.
//
void
I2CCallback(void)
{
    //
    // Clear the interrupt.
    //
    SoftI2CIntClear(&g_sI2C);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the software I2C and send some data.
//
void
TestSoftI2C(void)
{
    //
    // Clear the software I2C instance data.
    //
    memset(&g_sI2C, 0, sizeof(g_sI2C));

    //
    // Set the callback function used for this software I2C.
    //
    SoftI2CCallbackSet(&g_sI2C, I2CCallback);

    //
    // Configure the pins used for the software I2C.  This example uses
    // pins PD0 and PE1.
    //
    SoftI2CSCLGPIOSet(&g_sI2C, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftI2CSDAGPIOSet(&g_sI2C, GPIO_PORTE_BASE, GPIO_PIN_1);

    //
    // Enable the GPIO modules that contains the GPIO pins to be used by
    // the software I2C.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    //
    // Initalize the software I2C module.
    //
```

```
    SoftI2CInit(&g_sI2C);

    //
    // Configure the timer used to generate the timing for the software
    // I2C.  The interface will be run at 10 KHz, requiring a timer tick
    // at 40 KHz.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 40000);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntEnable(INT_TIMER0A);
    TimerEnable(TIMER0_BASE, TIMER_A);

    //
    // Enable the software I2C interrupt.
    //
    SoftI2CIntEnable(&g_sI2C);

    //
    // Send a single byte to the slave device.
    //
    SoftI2CSlaveAddrSet(&g_sI2C, 0x55, 0);
    SoftI2CDataPut(&g_sI2C, 0xaa);
    SoftI2CControl(&g_sI2C, SOFTI2C_CMD_SINGLE_SEND);

    //
    // Wait until the software I2C is idle.  The completion interrupt will
    // be sent to the callback function prior to exiting this loop.
    //
    while(SoftI2CBusy(&g_sI2C))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware I2C module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware I2C.
//
void
I2C0IntHandler(void)
{
    //
    // Clear the asserted interrupt sources.
    //
    I2CMasterIntClear(I2C0_MASTER_BASE);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the hardware I2C and send some data.
//
void
TestI2C(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the I2C, as well as the I2C module.
    //
```

```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

    //
    // Configure the GPIO pins for use by the I2C module.
    //
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);

    //
    // Initalize the hardware I2C module.
    //
    I2CMasterInitExpClk(I2C0_MASTER_BASE, SysCtlClockGet(), false);

    //
    // Enable the hardware I2C.
    //
    I2CMasterEnable(I2C0_MASTER_BASE);

    //
    // Enable the interrupt in the hardware I2C.
    //
    I2CMasterIntEnable(I2C0_MASTER_BASE);
    IntEnable(INT_I2C0);

    //
    // Write some data into the hardware I2C transmit FIFO.
    //
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, 0x55, 0);
    I2CMasterDataPut(I2C0_MASTER_BASE, 0xaa);
    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);

    //
    // Wait until the hardware I2C is idle.  The interrupt will be sent to
    // the interrupt handler prior to exiting this loop.
    //
    while(I2CBusy(I2C0_MASTER_BASE))
    {
    }
}
```

14 Software SSI Module

14.1 Introduction

The software SSI module uses a timer and a few GPIO pins to create a slow-speed software SSI peripheral. Multiple software SSI modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software SSI module supports the Motorola® SPI™ formats with 4 to 16 data bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware SSI module.

The API for the software SSI module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware SSI module. The two notable difference are the function prefix being “SoftSSI” instead of “SSI”, and the first argument of each API is a pointer to the tSoftSSI data structure instead of the base address of the hardware module.

Timing for the software SSI module is provided by the application. The SoftSSITimerTick() function must be called on a periodic basis to provide the timing for the software SSI module. The timer tick function must be called at twice the desired SSI clock rate; for example, to operate the software SSI interface at 10 KHz, the tick function must be called at a 20 KHz rate. By having the application providing the timing, the timer resource to be used is flexible and multiple software SSI modules can be driven from a single timer resource. Alternatively, if the software SSI module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire SSI transaction has completed (maximizing both SSI clock speed and processor usage, but not requiring a timer).

The software SSI module requires a few as two and as many as four GPIO pins. The following table shows the possible pin usages for the software SSI module:

Fss	Clk	Tx	Rx	Pins	Description
	yes	yes		2	transmit only
yes	yes	yes		3	
	yes		yes	2	receive only
yes	yes		yes	3	
	yes	yes	yes	3	transmit and receive
yes	yes	yes	yes	4	

For the cases where Fss is not used, it is up to the application to control that signal (either via a separately-controlled GPIO, or by being tied to ground in the hardware).

The per-instance data structure is approximately 52 bytes in length (the actual length will depend upon how the structure is packed by the compiler in use).

As a point of reference, the following are some rough measurements of the processor usage of the software SSI module at various SSI clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending upon how the application uses the software SSI module, processor clock speed, interrupt priority, and compiler in use.

SSI Clock	% Of Processor	Million Cycles Per Second
10 KHz	5.26	2.63
20 KHz	10.48	5.24
30 KHz	15.68	7.84
40 KHz	20.90	10.45
50 KHz	26.10	13.05
60 KHz	31.38	15.69
70 KHz	36.54	18.27
80 KHz	41.79	20.89
90 KHz	47.06	23.53
100 KHz	52.17	26.08

This module is contained in `utils/softssi.c`, with `utils/softssi.h` containing the API definitions for use by applications.

14.2 API Functions

Data Structures

- tSoftSSI

Functions

- tBoolean SoftSSIBusy (tSoftSSI *pSSI)
- void SoftSSICallbackSet (tSoftSSI *pSSI, void (*pfnCallback)(void))
- void SoftSSIClkGPIOSet (tSoftSSI *pSSI, unsigned long ulBase, unsigned char ucPin)
- void SoftSSIConfigSet (tSoftSSI *pSSI, unsigned char ucProtocol, unsigned char ucBits)
- tBoolean SoftSSIDataAvail (tSoftSSI *pSSI)
- void SoftSSIDataGet (tSoftSSI *pSSI, unsigned long *pulData)
- long SoftSSIDataGetNonBlocking (tSoftSSI *pSSI, unsigned long *pulData)
- void SoftSSIDataPut (tSoftSSI *pSSI, unsigned long ulData)
- long SoftSSIDataPutNonBlocking (tSoftSSI *pSSI, unsigned long ulData)
- void SoftSSIDisable (tSoftSSI *pSSI)
- void SoftSSIEnable (tSoftSSI *pSSI)
- void SoftSSIFssGPIOSet (tSoftSSI *pSSI, unsigned long ulBase, unsigned char ucPin)
- void SoftSSIIntClear (tSoftSSI *pSSI, unsigned long ulIntFlags)
- void SoftSSIIntDisable (tSoftSSI *pSSI, unsigned long ulIntFlags)
- void SoftSSIIntEnable (tSoftSSI *pSSI, unsigned long ulIntFlags)
- unsigned long SoftSSIIntStatus (tSoftSSI *pSSI, tBoolean bMasked)
- void SoftSSIRxBufferSet (tSoftSSI *pSSI, unsigned short *pusRxBuffer, unsigned short usLen)
- void SoftSSIRxGPIOSet (tSoftSSI *pSSI, unsigned long ulBase, unsigned char ucPin)
- tBoolean SoftSSISpaceAvail (tSoftSSI *pSSI)
- void SoftSSITimerTick (tSoftSSI *pSSI)
- void SoftSSITxBufferSet (tSoftSSI *pSSI, unsigned short *pusTxBuffer, unsigned short usLen)
- void SoftSSITxGPIOSet (tSoftSSI *pSSI, unsigned long ulBase, unsigned char ucPin)

14.2.1 Data Structure Documentation

14.2.1.1 tSoftSSI

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulFssGPIO;
    unsigned long ulClkGPIO;
    unsigned long ulTxGPIO;
    unsigned long ulRxGPIO;
    unsigned short *pusTxBuffer;
    unsigned short *pusRxBuffer;
    unsigned short usTxBufferLen;
    unsigned short usTxBufferRead;
    unsigned short usTxBufferWrite;
    unsigned short usRxBufferLen;
    unsigned short usRxBufferRead;
    unsigned short usRxBufferWrite;
    unsigned short usTxData;
    unsigned short usRxData;
    unsigned char ucFlags;
    unsigned char ucBits;
    unsigned char ucState;
    unsigned char ucCurrentBit;
    unsigned char ucIntMask;
    unsigned char ucIntStatus;
    unsigned char ucIdleCount;
}
tSoftSSI
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware SSI implementation. This address can be set via a direct structure access or using the SoftSSICallbackSet function.

ulFssGPIO The address of the GPIO pin to be used for the Fss signal. If this member is zero, the Fss signal is not generated. This member can be set via a direct structure access or using the SoftSSIFssGPIOSet function.

ulClkGPIO The address of the GPIO pin to be used for the Clk signal. This member can be set via a direct structure access or using the SoftSSIClkGPIOSet function.

ulTxGPIO The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the SoftSSITxGPIOSet function.

ulRxGPIO The address of the GPIO pin to be used for the Rx signal. If this member is zero, the Rx signal is not read. This member can be set via a direct structure access or using the SoftSSIRxGPIOSet function.

pusTxBuffer The address of the data buffer used for the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

pusRxBuffer The address of the data buffer used for the receive FIFO. This member can be set via a direct structure access or using the SoftSSIRxBufferSet function.

usTxBufferLen The length of the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

usTxBufferRead The index into the transmit FIFO of the next word to be transmitted. This member should be initialized to zero, but should not be accessed or modified by the application.

usTxBufferWrite The index into the transmit FIFO of the next location to store data into the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.

usRxBufferLen The length of the receive FIFO. This member can be set via a direct structure access or using the SoftSSIRxBufferSet function.

usRxBufferRead The index into the receive FIFO of the next word to be read from the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.

usRxBufferWrite The index into the receive FIFO of the location to store the next word received. This member should be initialized to zero, but should not be accessed or modified by the application.

usTxData The word that is currently being transmitted. This member should not be accessed or modified by the application.

usRxData The word that is currently being received. This member should not be accessed or modified by the application.

ucFlags The flags that control the operation of the SoftSSI module. This member should not be accessed or modified by the application.

ucBits The number of data bits in each SoftSSI frame, which also specifies the width of each data item in the transmit and receive FIFOs. This member can be set via a direct structure access or using the SoftSSIConfigSet function.

ucState The current state of the SoftSSI state machine. This member should not be accessed or modified by the application.

ucCurrentBit The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.

ucIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

ucIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

ucIdleCount The number of tick counts that the SoftSSI module has been idle with data stored in the receive FIFO, which is used to generate the receive timeout interrupt. This member should not be accessed or modified by the application.

Description:

This structure contains the state of a single instance of a SoftSSI module.

14.2.2 Function Documentation

14.2.2.1 SoftSSIBusy

Determines whether the SoftSSI transmitter is busy or not.

Prototype:

```
tBoolean
SoftSSIBusy(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

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

Returns:

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

14.2.2.2 SoftSSICallbackSet

Sets the callback used by the SoftSSI module.

Prototype:

```
void
SoftSSICallbackSet(tSoftSSI *pSSI,
                   void (*pfnCallback)(void))
```

Parameters:

pSSI specifies the SoftSSI data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftSSI module.

Returns:

None.

14.2.2.3 SoftSSIClkGPIOSet

Sets the GPIO pin to be used as the SoftSSI Clk signal.

Prototype:

```
void
SoftSSIClkGPIOSet(tSoftSSI *pSSI,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Clk signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

14.2.2.4 SoftSSIConfigSet

Sets the configuration of a SoftSSI module.

Prototype:

```
void
SoftSSIConfigSet(tSoftSSI *pSSI,
                 unsigned char ucProtocol,
                 unsigned char ucBits)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ucProtocol specifes the data transfer protocol.

ucBits specifies the number of bits transferred per frame.

Description:

This function configures the data format of a SoftSSI module. The *ucProtocol* parameter can be one of the following values: **SOFTSSI_FRF_MOTO_MODE_0**, **SOFTSSI_FRF_MOTO_MODE_1**, **SOFTSSI_FRF_MOTO_MODE_2**, or **SOFTSSI_FRF_MOTO_MODE_3**. These frame formats imply the following polarity and phase configurations:

```
Polarity Phase         Mode
  0       0    SOFTSSI_FRF_MOTO_MODE_0
  0       1    SOFTSSI_FRF_MOTO_MODE_1
  1       0    SOFTSSI_FRF_MOTO_MODE_2
  1       1    SOFTSSI_FRF_MOTO_MODE_3
```

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

Returns:

None.

14.2.2.5 SoftSSIDataAvail

Determines if there is any data in the receive FIFO.

Prototype:

```
tBoolean
SoftSSIDataAvail(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function determines if there is any data available to be read from 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.

14.2.2.6 SoftSSIDataGet

Gets a data element from the SoftSSI receive FIFO.

Prototype:

```
void
SoftSSIDataGet(tSoftSSI *pSSI,
               unsigned long *pulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pulData is a pointer to a storage location for data that was received over the SoftSSI interface.

Description:

This function gets received data from the receive FIFO of the specified SoftSSI 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 SoftSSIConfigSet(). 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.

14.2.2.7 SoftSSIDataGetNonBlocking

Gets a data element from the SoftSSI receive FIFO.

Prototype:

```
long
SoftSSIDataGetNonBlocking(tSoftSSI *pSSI,
                          unsigned long *pulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pulData is a pointer to a storage location for data that was received over the SoftSSI interface.

Description:

This function gets received data from the receive FIFO of the specified SoftSSI 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 SoftSSIConfigSet(). 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 SoftSSI receive FIFO.

14.2.2.8 SoftSSIDataPut

Puts a data element into the SoftSSI transmit FIFO.

Prototype:

```
void
SoftSSIDataPut(tSoftSSI *pSSI,
               unsigned long ulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulData is the data to be transmitted over the SoftSSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SoftSSI module.

Note:

The upper 32 - N bits of the *ulData* are discarded, where N is the data width as configured by SoftSSIConfigSet(). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ulData* are discarded.

Returns:

None.

14.2.2.9 SoftSSIDataPutNonBlocking

Puts a data element into the SoftSSI transmit FIFO.

Prototype:

```
long
SoftSSIDataPutNonBlocking(tSoftSSI *pSSI,
                          unsigned long ulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulData is the data to be transmitted over the SoftSSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SoftSSI 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, where N is the data width as configured by SoftSSIConfigSet(). 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.

14.2.2.10 SoftSSIDisable

Disables the SoftSSI module.

Prototype:

```
void
SoftSSIDisable(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function disables operation of the SoftSSI module. If a data transfer is in progress, it is finished before the module is fully disabled.

Returns:

None.

14.2.2.11 SoftSSIEnable

Enables the SoftSSI module.

Prototype:

```
void
SoftSSIEnable(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function enables operation of the SoftSSI module. The SoftSSI module must be configured before it is enabled.

Returns:

None.

14.2.2.12 SoftSSIFssGPIOSet

Sets the GPIO pin to be used as the SoftSSI Fss signal.

Prototype:

```
void
SoftSSIFssGPIOSet(tSoftSSI *pSSI,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Fss signal. If there is not a GPIO pin allocated for Fss, the SoftSSI module does not assert/deassert the Fss signal, leaving it to the application either to do manually or to not do at all if the slave device has Fss tied to ground.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

14.2.2.13 SoftSSIIntClear

Clears SoftSSI "interrupt" sources.

Prototype:

```
void
SoftSSIIntClear(tSoftSSI *pSSI,
                unsigned long ulIntFlags)
```

Parameters:

pSSI specifies the SoftSSI data structure.

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

Description:

The specified SoftSSI "interrupt" sources are cleared so that they no longer assert. This function must be called in the "interrupt" handler to keep the "interrupt" from being recognized again immediately upon exit. The *ulIntFlags* parameter is the logical OR of any of the **SOFTSSI_TXEOT**, **SOFTSSI_RXTO**, and **SOFTSSI_RXOR** values.

Returns:

None.

14.2.2.14 SoftSSIIntDisable

Disables individual SoftSSI "interrupt" sources.

Prototype:

```
void
SoftSSIIntDisable(tSoftSSI *pSSI,
                  unsigned long ulIntFlags)
```

Parameters:

pSSI specifies the SoftSSI data structure.

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

Description:

Disables the indicated SoftSSI "interrupt" sources. The *ulIntFlags* parameter can be any of the **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, or **SOFTSSI_RXOR** values.

Returns:
None.

14.2.2.15 SoftSSIIntEnable

Enables individual SoftSSI "interrupt" sources.

Prototype:
```
void
SoftSSIIntEnable(tSoftSSI *pSSI,
                 unsigned long ulIntFlags)
```

Parameters:
pSSI specifies the SoftSSI data structure.
ulIntFlags is a bit mask of the "interrupt" sources to be enabled.

Description:
Enables the indicated SoftSSI "interrupt" sources. Only the sources that are enabled can be reflected to the callback function; disabled sources do not result in a callback. The *ulIntFlags* parameter can be any of the **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, or **SOFTSSI_RXOR** values.

Returns:
None.

14.2.2.16 SoftSSIIntStatus

Gets the current "interrupt" status.

Prototype:
```
unsigned long
SoftSSIIntStatus(tSoftSSI *pSSI,
                 tBoolean bMasked)
```

Parameters:
pSSI specifies the SoftSSI data structure.
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 SoftSSI module. Either the raw "interrupt" status or the status of "interrupts" that are allowed to reflect to the callback can be returned.

Returns:
The current "interrupt" status, enumerated as a bit field of **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, and **SOFTSSI_RXOR**.

14.2.2.17 SoftSSIRxBufferSet

Sets the receive FIFO buffer for a SoftSSI module.

Prototype:

```
void
SoftSSIRxBufferSet(tSoftSSI *pSSI,
                   unsigned short *pusRxBuffer,
                   unsigned short usLen)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pusRxBuffer is the address of the receive FIFO buffer.

usLen is the size, in 16-bit half-words, of the receive FIFO buffer.

Description:

This function sets the address and size of the receive FIFO buffer and also resets the read and write pointers, marking the receive FIFO as empty. When the buffer pointer and length are configured as zero, all data received from the slave device is discarded. This capability is useful when there is no GPIO pin allocated for the Rx signal.

Returns:

None.

14.2.2.18 SoftSSIRxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Rx signal.

Prototype:

```
void
SoftSSIRxGPIOSet(tSoftSSI *pSSI,
                 unsigned long ulBase,
                 unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Rx signal. If there is not a GPIO pin allocated for Rx, the SoftSSI module does not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

14.2.2.19 SoftSSISpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:

```
tBoolean
SoftSSISpaceAvail(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function determines if 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.

14.2.2.20 SoftSSITimerTick

Performs the periodic update of the SoftSSI module.

Prototype:

```
void
SoftSSITimerTick(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function performs the periodic, time-based updates to the SoftSSI module. The transmission and reception of data over the SoftSSI link is performed by the state machine in this function.

This function must be called at twice the desired SoftSSI clock rate. For example, to run the SoftSSI clock at 10 KHz, this function must be called at a 20 KHz rate.

Returns:

None.

14.2.2.21 SoftSSITxBufferSet

Sets the transmit FIFO buffer for a SoftSSI module.

Prototype:

```
void
SoftSSITxBufferSet(tSoftSSI *pSSI,
                   unsigned short *pusTxBuffer,
                   unsigned short usLen)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pusTxBuffer is the address of the transmit FIFO buffer.

usLen is the size, in 16-bit half-words, of the transmit FIFO buffer.

Description:

This function sets the address and size of the transmit FIFO buffer and also resets the read and write pointers, marking the transmit FIFO as empty.

Returns:

None.

14.2.2.22 SoftSSITxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Tx signal.

Prototype:

```
void
SoftSSITxGPIOSet(tSoftSSI *pSSI,
                 unsigned long ulBase,
                 unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

14.3 Programming Example

The following example shows how to configure the software SSI module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software SSI.
//
tSoftSSI g_sSSI;

//
// The buffer used to hold the transmit data.
//
unsigned short g_pusTxBuffer[8];

//
// The timer tick function.
//
```

```
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software SSI timer tick function.
    //
    SoftSSITimerTick(&g_sSSI);
}

//
// The callback function for the software SSI.  This function is equivalent
// to the interrupt handler for a hardware SSI.
//
void
SSICallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftSSIIntStatus(&g_sSSI, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftSSIIntClear(&g_sSSI, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software SSI and send some data.
//
void
TestSoftSSI(void)
{
    //
    // Clear the software SSI instance data.
    //
    memset(&g_sSSI, 0, sizeof(g_sSSI));

    //
    // Set the callback function used for this software SSI.
    //
    SoftSSICallbackSet(&g_sSSI, SSICallback);

    //
    // Configure the pins used for the software SSI.  This example uses
    // pins PD0, PE1, and PF2.
    //
    SoftSSIFssGPIOSet(&g_sSSI, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftSSIClkGPIOSet(&g_sSSI, GPIO_PORTE_BASE, GPIO_PIN_1);
    SoftSSITxGPIOSet(&g_sSSI, GPIO_PORTF_BASE, GPIO_PIN_2);

    //
    // Configure the data buffer used as the transmit FIFO.
    //
```

```
    SoftSSITxBufferSet(&g_sSSI, g_pusTxBuffer, 8);

    //
    // Enable the GPIO modules that contains the GPIO pins to be used by
    // the software SSI.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    //
    // Initalize the software SSI module, using mode 3 and 8 data bits.
    //
    SoftSSIConfigSet(&g_sSSI, SOFTSSI_FRF_MOTO_MODE_3, 8);

    //
    // Enable the software SSI.
    //
    SoftSSIEnable(&g_sSSI);

    //
    // Configure the timer used to generate the timing for the software
    // SSI.  The interface will be run at 10 KHz, requiring a timer tick
    // at 20 KHz.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 20000);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntEnable(INT_TIMER0A);
    TimerEnable(TIMER0_BASE, TIMER_A);

    //
    // Enable the transmit FIFO half full interrupt in the software SSI.
    //
    SoftSSIIntEnable(&g_sSSI, SOFTSSI_TXFF);

    //
    // Write some data into the software SSI transmit FIFO.
    //
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);

    //
    // Wait until the software SSI is idle.  The transmit FIFO half full
    // interrupt will be sent to the callback function prior to exiting
    // this loop.
    //
    while(SoftSSIBusy(&g_sSSI))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware SSI module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware SSI.
//
void
SSI0IntHandler(void)
```

```
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SSIIntStatus(SSI0_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    SSIIntClear(SSI0_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware SSI and send some data.
//
void
TestSSI(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the SSI, as well as the SSI module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

    //
    // Configure the GPIO pins for use by the SSI module.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_3 |
                                     GPIO_PIN_4 | GPIO_PIN_5));

    //
    // Initalize the hardware SSI module, using mode 3 and 8 data bits.
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
                       SSI_MODE_MASTER, 10000, 8);

    //
    // Enable the hardware SSI.
    //
    SSIEnable(SSI0_BASE);

    //
    // Enable the transmit FIFO half full interrupt in the hardware SSI.
    //
    SSIIntEnable(SSI0_BASE, SSI_TXFF);
    IntEnable(INT_SSI0);

    //
    // Write some data into the hardware SSI transmit FIFO.
    //
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);

    //
```

```
    // Wait until the hardware SSI is idle.  The transmit FIFO half full
    // interrupt will be sent to the interrupt handler prior to exiting
    // this loop.
    //
    while(SSIBusy(SSI0_BASE))
    {
    }
}
```

15 Software UART Module

15.1 Introduction

The software UART module uses two timers and a two GPIO pins to create a software UART peripheral. Multiple software UART modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers, and processor cycles to execute the code). The software UART module supports five through eight data bits, a varity of parity modes (odd, even, one, zero, and none), and one or two stop bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware UART module.

The API for the software UART module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware UART module. The two notable difference are the function prefix being “SoftUART” instead of “UART”, and the first argument of each API is a pointer to the tSoftUART data structure instead of the base address of the hardware module.

The software UART transmitter and receiver are handled independently (because of the asynchronous nature of the two). As a result, there are separate timers for each, and if only one is required then the other does not need to be utilized.

Timing for the software UART transmitter is provided by the application. The SoftUARTTxTimerTick() function must be called on a periodic basis to provide the timing for the software UART transmitter. The timer tick function must be called at the desired UART baud rate; for example, to operate the software UART transmitter at 38,400 baud, the tick function must be called at a 38,400 Hz rate. Because the application provides the timing, the timer resource can be flexible and multiple software UART transmitters can be driven from a single timer resource.

Timing for the software UART receiver is also provided by the application. Initially, the Rx pin is configured by the software UART module for a GPIO edge interrupt. The GPIO edge interrupt handler must be provided by the application (so that it can be shared with other possible GPIO interrupts on that port). When the interrupt occurs, a timer must be started at the desired baud rate (i.e. for 38,400 baud, it must run at 38,400 Hz) and the SoftUARTRxTick() function must be called. Then, whenever the timer interrupt occurs, the SoftUARTRxTick() function must be called. The timer is disabled whenver SoftUARTRxTick() indicates that it is no longer needed. Because the application provides the timing, the timer resource can beflexible. However, each software UART receiver must have its own timer resource.

The software UART module requires one or two GPIO pins. The following table shows the possible pin usages for the software UART module:

Tx	Rx	Pins	Description
yes		1	transmit only
	yes	1	receive only
yes	yes	2	transmit and receive

The per-instance data structure is approximately 52 bytes in length (the actual length depends on how the structure is packed by the compiler in use).

The following table shows some approximate measurements of the processor usage of the software UART module at various baud rates with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software UART module, processor clock speed, interrupt priority, and compiler in use.

UART Baud Rate	% Of Processor	Million Cycles Per Second
9600	5.32	2.66
14400	7.99	3.99
19200	10.65	5.32
28800	15.96	7.98
38400	21.28	10.64
57600	32.00	16.00
115200	64.04	32.02

This module is contained in `utils/softuart.c`, with `utils/softuart.h` containing the API definitions for use by applications.

15.2 API Functions

Data Structures

- tSoftUART

Functions

- void SoftUARTBreakCtl (tSoftUART *pUART, tBoolean bBreakState)
- tBoolean SoftUARTBusy (tSoftUART *pUART)
- void SoftUARTCallbackSet (tSoftUART *pUART, void (*pfnCallback)(void))
- long SoftUARTCharGet (tSoftUART *pUART)
- long SoftUARTCharGetNonBlocking (tSoftUART *pUART)
- void SoftUARTCharPut (tSoftUART *pUART, unsigned char ucData)
- tBoolean SoftUARTCharPutNonBlocking (tSoftUART *pUART, unsigned char ucData)
- tBoolean SoftUARTCharsAvail (tSoftUART *pUART)
- void SoftUARTConfigGet (tSoftUART *pUART, unsigned long *pulConfig)
- void SoftUARTConfigSet (tSoftUART *pUART, unsigned long ulConfig)
- void SoftUARTDisable (tSoftUART *pUART)
- void SoftUARTEnable (tSoftUART *pUART)
- void SoftUARTFIFOLevelGet (tSoftUART *pUART, unsigned long *pulTxLevel, unsigned long *pulRxLevel)
- void SoftUARTFIFOLevelSet (tSoftUART *pUART, unsigned long ulTxLevel, unsigned long ulRxLevel)
- void SoftUARTInit (tSoftUART *pUART)
- void SoftUARTIntClear (tSoftUART *pUART, unsigned long ulIntFlags)
- void SoftUARTIntDisable (tSoftUART *pUART, unsigned long ulIntFlags)
- void SoftUARTIntEnable (tSoftUART *pUART, unsigned long ulIntFlags)

- unsigned long SoftUARTIntStatus (tSoftUART *pUART, tBoolean bMasked)
- unsigned long SoftUARTParityModeGet (tSoftUART *pUART)
- void SoftUARTParityModeSet (tSoftUART *pUART, unsigned long ulParity)
- void SoftUARTRxBufferSet (tSoftUART *pUART, unsigned short *pusRxBuffer, unsigned short usLen)
- void SoftUARTRxErrorClear (tSoftUART *pUART)
- unsigned long SoftUARTRxErrorGet (tSoftUART *pUART)
- void SoftUARTRxGPIOSet (tSoftUART *pUART, unsigned long ulBase, unsigned char ucPin)
- unsigned long SoftUARTRxTick (tSoftUART *pUART, tBoolean bEdgeInt)
- tBoolean SoftUARTSpaceAvail (tSoftUART *pUART)
- void SoftUARTTxBufferSet (tSoftUART *pUART, unsigned char *pucTxBuffer, unsigned short usLen)
- void SoftUARTTxGPIOSet (tSoftUART *pUART, unsigned long ulBase, unsigned char ucPin)
- void SoftUARTTxTimerTick (tSoftUART *pUART)

15.2.1 Data Structure Documentation

15.2.1.1 tSoftUART

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulTxGPIO;
    unsigned long ulRxGPIOPort;
    unsigned char *pucTxBuffer;
    unsigned short *pusRxBuffer;
    unsigned short usTxBufferLen;
    unsigned short usTxBufferRead;
    unsigned short usTxBufferWrite;
    unsigned short usTxBufferLevel;
    unsigned short usRxBufferLen;
    unsigned short usRxBufferRead;
    unsigned short usRxBufferWrite;
    unsigned short usRxBufferLevel;
    unsigned short usIntStatus;
    unsigned short usIntMask;
    unsigned short usConfig;
    unsigned char ucFlags;
    unsigned char ucTxState;
    unsigned char ucTxNext;
    unsigned char ucTxData;
    unsigned char ucRxPin;
    unsigned char ucRxState;
    unsigned char ucRxData;
    unsigned char ucRxFlags;
    unsigned char ucRxStatus;
}
tSoftUART
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware UART implementation. This address can be set via a direct structure access or using the SoftUARTCallbackSet function.

ulTxGPIO The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the SoftUARTTxGPIOSet function.

ulRxGPIOPort The address of the GPIO port to be used for the Rx signal. This member can be set via a direct structure access or using the SoftUARTRxGPIOSet function.

pucTxBuffer The address of the data buffer used for the transmit buffer. This member can be set via a direct structure access or using the SoftUARTTxBufferSet function.

pusRxBuffer The address of the data buffer used for the receive buffer. This member can be set via a direct structure access or using the SoftUARTRxBufferSet function.

usTxBufferLen The length of the transmit buffer. This member can be set via a direct structure access or using the SoftUARTTxBufferSet function.

usTxBufferRead The index into the transmit buffer of the next character to be transmitted. This member should not be accessed or modified by the application.

usTxBufferWrite The index into the transmit buffer of the next location to store a character into the buffer. This member should not be accessed or modified by the application.

usTxBufferLevel The transmit buffer level at which the transmit interrupt is asserted. This member should not be accessed or modified by the application.

usRxBufferLen The length of the receive buffer. This member can be set via a direct structure access or using the SoftUARTRxBufferSet function.

usRxBufferRead The index into the receive buffer of the next character to be read from the buffer. This member should not be accessed or modified by the application.

usRxBufferWrite The index into the receive buffer of the lcoation to store the next character received. This member should not be accessed or modified by the application.

usRxBufferLevel The receive buffer level at which the receive interrupt is asserted. This member should not be accessed or modified by the application.

usIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

usIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

usConfig The configuration of the SoftUART module. This member can be set via the SoftUARTConfigSet and SoftUARTFIFOLevelSet functions.

ucFlags The flags that control the operation of the SoftUART module. This member should not be be accessed or modified by the application.

ucTxState The current state of the SoftUART transmit state machine. This member should not be accessed or modified by the application.

ucTxNext The value that is written to the Tx pin at the start of the next transmit timer tick. This member should not be accessed or modified by the application.

ucTxData The character that is currently be sent via the Tx pin. This member should not be accessed or modified by the application.

ucRxPin The GPIO pin to be used for the Rx signal. This member can be set via a direct structure access or using the SoftUARTRxGPIOSet function.

ucRxState The current state of the SoftUART receive state machine. This member should not be accessed or modified by the application.

ucRxData The character that is currently being received via the Rx pin. This member should not be accessed or modified by the application.

ucRxFlags The flags that indicate any errors that have occurred during the reception of the current character via the Rx pin. This member should not be accessed or modified by the application.

ucRxStatus The receive error status. This member should only be accessed via the SoftUARTRxErrorGet and SoftURATRxErrorClear functions.

Description:
This structure contains the state of a single instance of a SoftUART module.

15.2.2 Function Documentation

15.2.2.1 SoftUARTBreakCtl

Causes a BREAK to be sent.

Prototype:
```
void
SoftUARTBreakCtl(tSoftUART *pUART,
                 tBoolean bBreakState)
```

Parameters:
pUART specifies the SoftUART data structure.

bBreakState controls the output level.

Description:
Calling this function with *bBreakState* set to **true** asserts a break condition on the SoftUART. 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.

15.2.2.2 SoftUARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:
```
tBoolean
SoftUARTBusy(tSoftUART *pUART)
```

Parameters:
pUART specifies the SoftUART data structure.

Description:
Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit buffer 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.

15.2.2.3 SoftUARTCallbackSet

Sets the callback used by the SoftUART module.

Prototype:

```
void
SoftUARTCallbackSet(tSoftUART *pUART,
                    void (*pfnCallback)(void))
```

Parameters:

pUART specifies the SoftUART data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftUART module.

Returns:

None.

15.2.2.4 SoftUARTCharGet

Waits for a character from the specified port.

Prototype:

```
long
SoftUARTCharGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

Gets a character from the receive buffer 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*.

15.2.2.5 SoftUARTCharGetNonBlocking

Receives a character from the specified port.

Prototype:

```
long
SoftUARTCharGetNonBlocking(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

Gets a character from the receive buffer for the specified port.

Returns:

Returns the character read from the specified port, cast as a *long*. A **-1** isreturned if there are no characters present in the receive buffer. The SoftUARTCharsAvail() function should be called before attempting to call this function.

15.2.2.6 SoftUARTCharPut

Waits to send a character from the specified port.

Prototype:

```
void
SoftUARTCharPut(tSoftUART *pUART,
                unsigned char ucData)
```

Parameters:

pUART specifies the SoftUART data structure.

ucData is the character to be transmitted.

Description:

Sends the character *ucData* to the transmit buffer for the specified port. If there is no space available in the transmit buffer, this function waits until there is space available before returning.

Returns:

None.

15.2.2.7 SoftUARTCharPutNonBlocking

Sends a character to the specified port.

Prototype:

```
tBoolean
SoftUARTCharPutNonBlocking(tSoftUART *pUART,
                           unsigned char ucData)
```

Parameters:

pUART specifies the SoftUART data structure.

ucData is the character to be transmitted.

Description:

Writes the character *ucData* to the transmit buffer 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 buffer or **false** if there was no space available in the transmit buffer.

15.2.2.8 SoftUARTCharsAvail

Determines if there are any characters in the receive buffer.

Prototype:

```
tBoolean
SoftUARTCharsAvail(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function returns a flag indicating whether or not there is data available in the receive buffer.

Returns:

Returns **true** if there is data in the receive buffer or **false** if there is no data in the receive buffer.

15.2.2.9 SoftUARTConfigGet

Gets the current configuration of a UART.

Prototype:

```
void
SoftUARTConfigGet(tSoftUART *pUART,
                  unsigned long *pulConfig)
```

Parameters:

pUART specifies the SoftUART data structure.

pulConfig is a pointer to storage for the data format.

Description:

Returns the data format of the SoftUART. The data format returned in *pulConfig* is enumerated the same as the *ulConfig* parameter of SoftUARTConfigSet().

Returns:

None.

15.2.2.10 SoftUARTConfigSet

Sets the configuration of a SoftUART module.

Prototype:

```
void
SoftUARTConfigSet(tSoftUART *pUART,
                  unsigned long ulConfig)
```

Parameters:

pUART specifies the SoftUART data structure.

ulConfig is the data format for the port (number of data bits, number of stop bits, and parity).

Description:

This function configures the SoftUART for operation in the specified data format, as specified 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. **SOFTUART_CONFIG_WLEN_8**, **SOFTUART_CONFIG_WLEN_7**, **SOFTUART_CONFIG_WLEN_6**, and **SOFTUART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **SOFTUART_CONFIG_STOP_ONE** and **SOFTUART_CONFIG_STOP_TWO** select one or two stop bits (respectively). **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, and **SOFTUART_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).

Returns:

None.

15.2.2.11 SoftUARTDisable

Disables the SoftUART.

Prototype:

```
void
SoftUARTDisable(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function disables the SoftUART after waiting for it to become idle.

Returns:

None.

15.2.2.12 SoftUARTEnable

Enables the SoftUART.

Prototype:

```
void
SoftUARTEnable(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function enables the SoftUART, allowing data to be transmitted and received.

Returns:

None.

15.2.2.13 SoftUARTFIFOLevelGet

Gets the buffer level at which “interrupts” are generated.

Prototype:

```
void
SoftUARTFIFOLevelGet(tSoftUART *pUART,
                     unsigned long *pulTxLevel,
                     unsigned long *pulRxLevel)
```

Parameters:

pUART specifies the SoftUART data structure.

pulTxLevel is a pointer to storage for the transmit buffer 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 buffer 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 buffer level at which transmit and receive “interrupts” are generated.

Returns:

None.

15.2.2.14 SoftUARTFIFOLevelSet

Sets the buffer level at which “interrupts” are generated.

Prototype:

```
void
SoftUARTFIFOLevelSet(tSoftUART *pUART,
                     unsigned long ulTxLevel,
                     unsigned long ulRxLevel)
```

Parameters:

pUART specifies the SoftUART data structure.

ulTxLevel is the transmit buffer “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 buffer “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 buffer level at which transmit and receive “interrupts” are generated.

Returns:

None.

15.2.2.15 SoftUARTInit

Initializes the SoftUART module.

Prototype:

```
void
SoftUARTInit(tSoftUART *pUART)
```

Parameters:

pUART specifies the soft UART data structure.

Description:

This function initializes the data structure for the SoftUART module, putting it into the default configuration.

Returns:

None.

15.2.2.16 SoftUARTIntClear

Clears SoftUART "interrupt" sources.

Prototype:

```
void
SoftUARTIntClear(tSoftUART *pUART,
                 unsigned long ulIntFlags)
```

Parameters:

pUART specifies the SoftUART data structure.

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

Description:

The specified SoftUART "interrupt" sources are cleared, so that they no longer assert. This function must be called in the callback function to keep the "interrupt" from being recognized again immediately upon exit.

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

Returns:

None.

15.2.2.17 SoftUARTIntDisable

Disables individual SoftUART "interrupt" sources.

Prototype:

```
void
SoftUARTIntDisable(tSoftUART *pUART,
                   unsigned long ulIntFlags)
```

Parameters:

pUART specifies the SoftUART data structure.

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

Description:

Disables the indicated SoftUART "interrupt" sources. Only the sources that are enabled can be reflected to the SoftUART callback.

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

Returns:

None.

15.2.2.18 SoftUARTIntEnable

Enables individual SoftUART "interrupt" sources.

Prototype:

```
void
SoftUARTIntEnable(tSoftUART *pUART,
                  unsigned long ulIntFlags)
```

Parameters:

pUART specifies the SoftUART data structure.

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

Description:

Enables the indicated SoftUART "interrupt" sources. Only the sources that are enabled can be reflected to the SoftUART callback.

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

- **SOFTUART_INT_OE** - Overrun Error "interrupt"
- **SOFTUART_INT_BE** - Break Error "interrupt"
- **SOFTUART_INT_PE** - Parity Error "interrupt"
- **SOFTUART_INT_FE** - Framing Error "interrupt"
- **SOFTUART_INT_RT** - Receive Timeout "interrupt"
- **SOFTUART_INT_TX** - Transmit "interrupt"
- **SOFTUART_INT_RX** - Receive "interrupt"

Returns:

None.

15.2.2.19 SoftUARTIntStatus

Gets the current SoftUART "interrupt" status.

Prototype:

```
unsigned long
SoftUARTIntStatus(tSoftUART *pUART,
                  tBoolean bMasked)
```

Parameters:

pUART specifies the SoftUART data structure.

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 SoftUART. Either the raw "interrupt" status or the status of "interrupts" that are allowed to reflect to the SoftUART callback can be returned.

Returns:

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

15.2.2.20 SoftUARTParityModeGet

Gets the type of parity currently being used.

Prototype:

```
unsigned long
SoftUARTParityModeGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

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 **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, or **SOFTUART_CONFIG_PAR_ZERO**.

15.2.2.21 SoftUARTParityModeSet

Sets the type of parity.

Prototype:

```
void
SoftUARTParityModeSet(tSoftUART *pUART,
                      unsigned long ulParity)
```

Parameters:

pUART specifies the SoftUART data structure.

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 **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, or **SOFTUART_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.

15.2.2.22 SoftUARTRxBufferSet

Sets the receive buffer for a SoftUART module.

Prototype:
```
void
SoftUARTRxBufferSet(tSoftUART *pUART,
                    unsigned short *pusRxBuffer,
                    unsigned short usLen)
```

Parameters:
pUART specifies the SoftUART data structure.
pusRxBuffer is the address of the receive buffer.
usLen is the size, in 16-bit half-words, of the receive buffer.

Description:
This function sets the address and size of the receive buffer. It also resets the read and write pointers, marking the receive buffer as empty.

Returns:
None.

15.2.2.23 SoftUARTRxErrorClear

Clears all reported receiver errors.

Prototype:
```
void
SoftUARTRxErrorClear(tSoftUART *pUART)
```

Parameters:
pUART specifies the SoftUART data structure.

Description:
This function is used to clear all receiver error conditions reported via SoftUARTRxErrorGet(). 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.

15.2.2.24 SoftUARTRxErrorGet

Gets current receiver errors.

Prototype:

```
unsigned long
SoftUARTRxErrorGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

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 SoftUARTCharGet() or SoftUARTCharGetNonBlocking() with the exception that the overrun error is set immediately when the overrun occurs rather than when a character is next read.

Returns:

Returns a logical OR combination of the receiver error flags, **SOFTUART_RXERROR_FRAMING**, **SOFTUART_RXERROR_PARITY**, **SOFTUART_RXERROR_BREAK** and **SOFTUART_RXERROR_OVERRUN**.

15.2.2.25 SoftUARTRxGPIOSet

Sets the GPIO pin to be used as the SoftUART Rx signal.

Prototype:

```
void
SoftUARTRxGPIOSet(tSoftUART *pUART,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pUART specifies the SoftUART data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used when the SoftUART must sample the Rx signal. If there is not a GPIO pin allocated for Rx, the SoftUART module will not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

15.2.2.26 SoftUARTRxTick

Performs the periodic update of the SoftUART receiver.

Prototype:

```
unsigned long
SoftUARTRxTick(tSoftUART *pUART,
               tBoolean bEdgeInt)
```

Parameters:

pUART specifies the SoftUART data structure.

bEdgeInt should be **true** if this function is being called because of a GPIO edge interrupt and **false** if it is being called because of a timer interrupt.

Description:

This function performs the periodic, time-based updates to the SoftUART receiver. The reception of data to the SoftUART is performed by the state machine in this function.

This function must be called by the GPIO interrupt handler, and then periodically at the desired SoftUART baud rate. For example, to run the SoftUART at 115,200 baud, this function must be called at a 115,200 Hz rate.

Returns:

Returns **SOFTUART_RXTIMER_NOP** if the receive timer should continue to operate or **SOFTUART_RXTIMER_END** if it should be stopped.

15.2.2.27 SoftUARTSpaceAvail

Determines if there is any space in the transmit buffer.

Prototype:

```
tBoolean
SoftUARTSpaceAvail(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function returns a flag indicating whether or not there is space available in the transmit buffer.

Returns:

Returns **true** if there is space available in the transmit buffer or **false** if there is no space available in the transmit buffer.

15.2.2.28 SoftUARTTxBufferSet

Sets the transmit buffer for a SoftUART module.

Prototype:

```
void
SoftUARTTxBufferSet(tSoftUART *pUART,
                    unsigned char *pucTxBuffer,
                    unsigned short usLen)
```

Parameters:

pUART specifies the SoftUART data structure.

pucTxBuffer is the address of the transmit buffer.

usLen is the size, in 8-bit bytes, of the transmit buffer.

Description:
This function sets the address and size of the transmit buffer. It also resets the read and write pointers, marking the transmit buffer as empty.

Returns:
None.

15.2.2.29 SoftUARTTxGPIOSet

Sets the GPIO pin to be used as the SoftUART Tx signal.

Prototype:
```
void
SoftUARTTxGPIOSet(tSoftUART *pUART,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:
pUART specifies the SoftUART data structure.
ulBase is the base address of the GPIO module.
ucPin is the bit-packed representation of the pin to use.

Description:
This function sets the GPIO pin that is used when the SoftUART must assert the Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

15.2.2.30 SoftUARTTxTimerTick

Performs the periodic update of the SoftUART transmitter.

Prototype:
```
void
SoftUARTTxTimerTick(tSoftUART *pUART)
```

Parameters:
pUART specifies the SoftUART data structure.

Description:
This function performs the periodic, time-based updates to the SoftUART transmitter. The transmission of data from the SoftUART is performed by the state machine in this function.

This function must be called at the desired SoftUART baud rate. For example, to run the SoftUART at 115,200 baud, this function must be called at a 115,200 Hz rate.

Returns:
None.

15.3 Programming Example

The following example shows how to configure the software UART module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software UART.
//
tSoftUART g_sUART;

//
// The buffer used to hold the transmit data.
//
unsigned char g_pucTxBuffer[16];

//
// The buffer used to hold the receive data.
//
unsigned short g_pusRxBuffer[16];

//
// The number of processor clocks in the time period of a single bit on the
// software UART interface.
//
unsigned long g_ulBitTime;

//
// The transmit timer tick function.
//
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART transmit timer tick function.
    //
    SoftUARTTxTimerTick(&g_sUART);
}

//
// The receive timer tick function.
//
void
Timer0BIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART receive timer tick function, and see if the
    // timer should be disabled.
    //
    if(SoftUARTRxTick(&g_sUART, false) == SOFTUART_RXTIMER_END)
    {
        //
        // Disable the timer interrupt since the software UART doesn't need
        // it any longer.
        //
```

```
        TimerDisable(TIMER0_BASE, TIMER_B);
    }
}

//
// The interrupt handler for the software UART GPIO edge interrupt.
//
void
GPIOIntHandler(void)
{
    //
    // Configure the software UART receive timer so that it samples at the
    // mid-bit time of this character.
    //
    TimerDisable(TIMER0_BASE, TIMER_B);
    TimerLoadSet(TIMER0_BASE, TIMER_B, g_ulBitTime);
    TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    TimerEnable(TIMER0_BASE, TIMER_B);

    //
    // Call the software UART receive timer tick function.
    //
    SoftUARTRxTick(&g_sUART, true);
}

//
// The callback function for the software UART.  This function is
// equivalent to the interrupt handler for a hardware UART.
//
void
UARTCallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftUARTIntStatus(&g_sUART, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftUARTIntClear(&g_sUART, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software UART and send some data.
//
void
TestSoftUART(void)
{
    //
    // Initialize the software UART instance data.
    //
    SoftUARTInit(&g_sUART);

    //
    // Set the callback function used for this software UART.
    //
    SoftUARTCallbackSet(&g_sUART, UARTCallback);
```

```
//
// Configure the pins used for the software UART.  This example uses
// pins PD0 and PE1.
//
SoftUARTTxGPIOSet(&g_sUART, GPIO_PORTD_BASE, GPIO_PIN_0);
SoftUARTRxGPIOSet(&g_sUART, GPIO_PORTE_BASE, GPIO_PIN_1);

//
// Configure the data buffers used as the transmit and receive buffers.
//
SoftUARTTxBufferSet(&g_sUART, g_pucTxBuffer, 16);
SoftUARTRxBufferSet(&g_sUART, g_pusRxBuffer, 16);

//
// Enable the GPIO modules that contains the GPIO pins to be used by
// the software UART.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//
// Configure the software UART module: 8 data bits, no parity, and one
// stop bit.
//
SoftUARTConfigSet(&g_sUART,
                  (SOFTUART_CONFIG_WLEN_8 | SOFTUART_CONFIG_PAR_NONE |
                   SOFTUART_CONFIG_STOP_ONE));

//
// Compute the bit time for 38,400 baud.
//
g_ulBitTime = (SysCtlClockGet() / 38400) - 1;

//
// Configure the timers used to generate the timing for the software
// UART.  The interface in this example is run at 38,400 baud,
// requiring a timer tick at 38,400 Hz.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE,
               (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_PERIODIC |
                TIMER_CFG_B_PERIODIC));
TimerLoadSet(TIMER0_BASE, TIMER_A, g_ulBitTime);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT | TIMER_TIMB_TIMEOUT);
TimerEnable(TIMER0_BASE, TIMER_A);

//
// Set the priorities of the interrupts associated with the software
// UART.  The receiver is higher priority than the transmitter, and the
// receiver edge interrupt is higher priority than the receiver timer
// interrupt.
//
IntPrioritySet(INT_GPIOE, 0x00);
IntPrioritySet(INT_TIMER0B, 0x40);
IntPrioritySet(INT_TIMER0A, 0x80);

//
// Enable the interrupts associated with the software UART.
//
IntEnable(INT_GPIOE);
IntEnable(INT_TIMER0A);
IntEnable(INT_TIMER0B);

//
// Enable the transmit FIFO half full interrupt in the software UART.
//
```

```
    SoftUARTIntEnable(&g_sUART, SOFTUART_INT_TX);

    //
    // Write some data into the software UART transmit FIFO.
    //
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);

    //
    // Wait until the software UART is idle.  The transmit FIFO half full
    // interrupt is sent to the callback function prior to exiting this
    // loop.
    //
    while(SoftUARTBusy(&g_sUART))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware UART module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware UART.
//
void
UART0IntHandler(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = UARTIntStatus(UART0_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    UARTIntClear(UART0_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware UART and send some data.
//
void
TestUART(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the UART, as well as the UART module.
    //
```

```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure the GPIO pins for use by the UART module.
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Initalize the hardware UART module: 8 data bits, no parity, one stop
    // bit, and 38,400 baud rate.
    //
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_PAR_NONE |
                         UART_CONFIG_STOP_ONE));

    //
    // Enable the transmit FIFO half full interrupt in the hardware UART.
    //
    UARTIntEnable(UART0_BASE, UART_INT_TX);
    IntEnable(INT_UART0);

    //
    // Write some data into the hardware UART transmit FIFO.
    //
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);

    //
    // Wait until the hardware UART is idle.  The transmit FIFO half full
    // interrupt is sent to the interrupt handler prior to exiting this
    // loop.
    //
    while(UARTBusy(UART0_BASE))
    {
    }
}
```

16 Micro Standard Library Module

16.1 Introduction

The micro standard library module provides a set of small implementations of functions normally found in the C library. These functions provide reduced or greatly reduced functionality in order to remain small while still being useful for most embedded applications.

The following functions are provided, along with the C library equivalent:

Function	C library equivalent
usprintf	sprintf
usnprintf	snprintf
uvsnprintf	vsnprintf
ustrnicmp	strnicmp
ustrtoul	strtoul
ustrstr	strstr
ulocaltime	localtime

This module is contained in `utils/ustdlib.c`, with `utils/ustdlib.h` containing the API definitions for use by applications.

16.2 API Functions

Data Structures

- tTime

Functions

- void ulocaltime (unsigned long ulTime, tTime *psTime)
- unsigned long umktime (tTime *psTime)
- int urand (void)
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)
- int usprintf (char *pcBuf, const char *pcString,...)
- void usrand (unsigned long ulSeed)
- int ustrcasecmp (const char *pcStr1, const char *pcStr2)
- int ustrcmp (const char *pcStr1, const char *pcStr2)
- int ustrlen (const char *pcStr)
- int ustrncmp (const char *pcStr1, const char *pcStr2, int iCount)

- char * ustrncpy (char *pcDst, const char *pcSrc, int iNum)
- int ustrnicmp (const char *pcStr1, const char *pcStr2, int iCount)
- char * ustrstr (const char *pcHaystack, const char *pcNeedle)
- unsigned long ustrtoul (const char *pcStr, const char **ppcStrRet, int iBase)
- int uvsnprintf (char *pcBuf, unsigned long ulSize, const char *pcString, va_list vaArgP)

16.2.1 Data Structure Documentation

16.2.1.1 tTime

Definition:

```
typedef struct
{
    unsigned short usYear;
    unsigned char ucMon;
    unsigned char ucMday;
    unsigned char ucWday;
    unsigned char ucHour;
    unsigned char ucMin;
    unsigned char ucSec;
}
tTime
```

Members:

usYear The number of years since 0 AD.

ucMon The month, where January is 0 and December is 11.

ucMday The day of the month.

ucWday The day of the week, where Sunday is 0 and Saturday is 6.

ucHour The number of hours.

ucMin The number of minutes.

ucSec The number of seconds.

Description:

A structure that contains the broken down date and time.

16.2.2 Function Documentation

16.2.2.1 ulocaltime

Converts from seconds to calendar date and time.

Prototype:

```
void
ulocaltime(unsigned long ulTime,
           tTime *psTime)
```

Parameters:

ulTime is the number of seconds.

psTime is a pointer to the time structure that is filled in with the broken down date and time.

Description:
This function converts a number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch) into the equivalent month, day, year, hours, minutes, and seconds representation.

Returns:
None.

16.2.2.2 umktime

Converts calendar date and time to seconds.

Prototype:
```
unsigned long
umktime(tTime *psTime)
```

Parameters:
psTime is a pointer to the time structure that is filled in with the broken down date and time.

Description:
This function converts the date and time represented by the *psTime* structure pointer to the number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch).

Returns:
Returns the calendar time and date as seconds. If the conversion was not possible then the function returns (unsigned long)(-1).

16.2.2.3 urand

Generate a new (pseudo) random number

Prototype:
```
int
urand(void)
```

Description:
This function is very similar to the C library `rand()` function. It will generate a pseudo-random number sequence based on the seed value.

Returns:
A pseudo-random number will be returned.

16.2.2.4 usnprintf

A simple snprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:
```
int
usnprintf(char *pcBuf,
          unsigned long ulSize,
          const char *pcString,
          ...)
```

Parameters:

pcBuf is the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `sprintf()` function. Only the following formatting characters are supported:

- %c to print a character
- %d or %i to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %i, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeros instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The function will copy at most *ulSize* - 1 characters into the buffer *pcBuf*. One space is reserved in the buffer for the null termination character.

The function will return the number of characters that would be converted as if there were no limit on the buffer size. Therefore it is possible for the function to return a count that is greater than the specified buffer size. If this happens, it means that the output was truncated.

Returns:

Returns the number of characters that were to be stored, not including the NULL termination character, regardless of space in the buffer.

16.2.2.5 usprintf

A simple sprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:

```
int
usprintf(char *pcBuf,
         const char *pcString,
         ...)
```

Parameters:

pcBuf is the buffer where the converted string is stored.

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `sprintf()` function. Only the following formatting characters are supported:

- %c to print a character
- %d or %i to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %i, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeros instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The caller must ensure that the buffer *pcBuf* is large enough to hold the entire converted string, including the null termination character.

Returns:

Returns the count of characters that were written to the output buffer, not including the NULL termination character.

16.2.2.6 usrand

Set the random number generator seed.

Prototype:

```
void
usrand(unsigned long ulSeed)
```

Parameters:

ulSeed is the new seed value to use for the random number generator.

Description:

This function is very similar to the C library `srand()` function. It will set the seed value used in the `urand()` function.

Returns:

None

16.2.2.7 ustrcasecmp

Compares two strings without regard to case.

Prototype:

```
int
ustrcasecmp(const char *pcStr1,
            const char *pcStr2)
```

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

This function is very similar to the C library `strcasecmp()` function. It compares two strings without regard to case. The comparison ends if a terminating NULL character is found in either string. In this case, the shorter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *pcStr1* is less than *pcStr2* and 1 if *pcStr1* is greater than *pcStr2*.

16.2.2.8 ustrcmp

Compares two strings.

Prototype:

```
int
ustrcmp(const char *pcStr1,
        const char *pcStr2)
```

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

This function is very similar to the C library `strcmp()` function. It compares two strings, taking case into account. The comparison ends if a terminating NULL character is found in either string. In this case, the shorter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *pcStr1* is less than *pcStr2* and 1 if *pcStr1* is greater than *pcStr2*.

16.2.2.9 ustrlen

Retruns the length of a null-terminated string.

Prototype:

```
int
ustrlen(const char *pcStr)
```

Parameters:

pcStr is a pointer to the string whose length is to be found.

Description:

This function is very similar to the C library `strlen()` function. It determines the length of the null-terminated string passed and returns this to the caller.

This implementation assumes that single byte character strings are passed and will return incorrect values if passed some UTF-8 strings.

Returns:

Returns the length of the string pointed to by *pcStr*.

16.2.2.10 ustrncmp

Compares two strings.

Prototype:

```
int
ustrncmp(const char *pcStr1,
         const char *pcStr2,
         int iCount)
```

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

This function is very similar to the C library `strncmp()` function. It compares at most *iCount* characters of two strings taking case into account. The comparison ends if a terminating NULL character is found in either string before *iCount* characters are compared. In this case, the shorter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *pcStr1* is less than *pcStr2* and 1 if *pcStr1* is greater than *pcStr2*.

16.2.2.11 ustrncpy

Copies a certain number of characters from one string to another.

Prototype:

```
char *
ustrncpy(char *pcDst,
         const char *pcSrc,
         int iNum)
```

Parameters:

pcDst is a pointer to the destination buffer into which characters are to be copied.

pcSrc is a pointer to the string from which characters are to be copied.

iNum is the number of characters to copy to the destination buffer.

Description:

This function copies at most *iNum* characters from the string pointed to by *pcSrc* into the buffer pointed to by *pcDst*. If the end of *pcSrc* is found before *iNum* characters have been copied, remaining characters in *pcDst* will be padded with zeroes until *iNum* characters have been written. Note that the destination string will only be NULL terminated if the number of characters to be copied is greater than the length of *pcSrc*.

Returns:

Returns *pcDst*.

16.2.2.12 ustrnicmp

Compares two strings without regard to case.

Prototype:

```
int
ustrnicmp(const char *pcStr1,
          const char *pcStr2,
          int iCount)
```

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

This function is very similar to the C library `strnicmp()` function. It compares at most *iCount* characters of two strings without regard to case. The comparison ends if a terminating NULL character is found in either string before *iCount* characters are compared. In this case, the shorter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *pcStr1* is less than *pcStr2* and 1 if *pcStr1* is greater than *pcStr2*.

16.2.2.13 ustrstr

Finds a substring within a string.

Prototype:

```
char *
ustrstr(const char *pcHaystack,
        const char *pcNeedle)
```

Parameters:

pcHaystack is a pointer to the string that will be searched.

pcNeedle is a pointer to the substring that is to be found within *pcHaystack*.

Description:
This function is very similar to the C library `strstr()` function. It scans a string for the first instance of a given substring and returns a pointer to that substring. If the substring cannot be found, a NULL pointer is returned.

Returns:
Returns a pointer to the first occurrence of *pcNeedle* within *pcHaystack* or NULL if no match is found.

16.2.2.14 ustrtoul

Converts a string into its numeric equivalent.

Prototype:
```
unsigned long
ustrtoul(const char *pcStr,
         const char **ppcStrRet,
         int iBase)
```

Parameters:
pcStr is a pointer to the string containing the integer.
ppcStrRet is a pointer that will be set to the first character past the integer in the string.
iBase is the radix to use for the conversion; can be zero to auto-select the radix or between 2 and 16 to explicitly specify the radix.

Description:
This function is very similar to the C library `strtoul()` function. It scans a string for the first token (that is, non-white space) and converts the value at that location in the string into an integer value.

Returns:
Returns the result of the conversion.

16.2.2.15 uvsnprintf

A simple vsnprintf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:
```
int
uvsnprintf(char *pcBuf,
           unsigned long ulSize,
           const char *pcString,
           va_list vaArgP)
```

Parameters:
pcBuf points to the buffer where the converted string is stored.
ulSize is the size of the buffer.
pcString is the format string.
vaArgP is the list of optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `vsnprintf()` function. Only the following formatting characters are supported:

- %c to print a character
- %d or %i to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %d, %i, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeroes instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

The *ulSize* parameter limits the number of characters that will be stored in the buffer pointed to by *pcBuf* to prevent the possibility of a buffer overflow. The buffer size should be large enough to hold the expected converted output string, including the null termination character.

The function will return the number of characters that would be converted as if there were no limit on the buffer size. Therefore it is possible for the function to return a count that is greater than the specified buffer size. If this happens, it means that the output was truncated.

Returns:

Returns the number of characters that were to be stored, not including the NULL termination character, regardless of space in the buffer.

16.3 Programming Example

The following example shows how to use some of the micro standard library functions.

```
unsigned long ulValue;
char pcBuffer[32];
tTime sTime;

//
// Convert the number in pcBuffer (previous read from somewhere) into an
// integer.  Note that this supports converting decimal values (such as
// 4583), octal values (such as 036583), and hexadecimal values (such as
// 0x3425).
//
ulValue = ustrtoul(pcBuffer, 0, 0);

//
// Convert that integer from a number of seconds into a broken down date.
```

```
//
ulocaltime(ulValue, &sTime);

//
// Print out the corresponding time of day in military format.
//
usprintf(pcBuffer, "%02d:%02d", sTime.ucHour, sTime.ucMin);
```

17 UART Standard IO Module

17.1 Introduction

The UART standard IO module provides a simple interface to a UART that is similar to the standard IO package available in the C library. Only a very small subset of the normal functions are provided; UARTprintf() is an equivalent to the C library printf() function and UARTgets() is an equivalent to the C library fgets() function.

This module is contained in `utils/uartstdio.c`, with `utils/uartstdio.h` containing the API definitions for use by applications.

17.1.1 Unbuffered Operation

Unbuffered operation is selected by not defining **UART_BUFFERED** when building the UART standard IO module. In unbuffered mode, calls to the module will not return until the operation has been completed. So, for example, a call to UARTprintf() will not return until the entire string has be placed into the UART's FIFO. If it is not possible for the function to complete its operation immediately, it will busy wait.

17.1.2 Buffered Operation

Buffered operation is selected by defining **UART_BUFFERED** when building the UART standard IO module. In buffered mode, there is a larger UART data FIFO in SRAM that extends the size of the hardware FIFO. Interrupts from the UART are used to transfer data between the SRAM buffer and the hardware FIFO. It is the responsibility of the application to ensure that UARTStdioIntHandler() is called when the UART interrupt occurs; typically this is accomplished by placing it in the vector table in the startup code for the application.

In addition providing a larger UART buffer, the behavior of UARTprintf() is slightly modified. If the output buffer is full, UARTprintf() will discard the remaining characters from the string instead of waiting until space becomes available in the buffer. If this behavior is not desired, UARTFlushTx() may be called to ensure that the transmit buffer is emptied prior to adding new data via UARTprintf() (though this will not work if the string to be printed is larger than the buffer).

UARTPeek() can be used to determine whether a line end is present prior to calling UARTgets() if non-blocking operation is required. In cases where the buffer supplied on UARTgets() fills before a line termination character is received, the call will return with a full buffer.

17.2 API Functions

Functions

- void UARTEchoSet (tBoolean bEnable)
- void UARTFlushRx (void)
- void UARTFlushTx (tBoolean bDiscard)
- unsigned char UARTgetc (void)
- int UARTgets (char *pcBuf, unsigned long ulLen)
- int UARTPeek (unsigned char ucChar)
- void UARTprintf (const char *pcString,...)
- int UARTRxBytesAvail (void)
- void UARTStdioConfig (unsigned long ulPortNum, unsigned long ulBaud, unsigned long ulSrcClock)
- void UARTStdioInit (unsigned long ulPortNum)
- void UARTStdioInitExpClk (unsigned long ulPortNum, unsigned long ulBaud)
- void UARTStdioIntHandler (void)
- int UARTTxBytesFree (void)
- int UARTwrite (const char *pcBuf, unsigned long ulLen)

17.2.1 Function Documentation

17.2.1.1 UARTEchoSet

Enables or disables echoing of received characters to the transmitter.

Prototype:

```
void
UARTEchoSet(tBoolean bEnable)
```

Parameters:

bEnable must be set to **true** to enable echo or **false** to disable it.

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to control whether or not received characters are automatically echoed back to the transmitter. By default, echo is enabled and this is typically the desired behavior if the module is being used to support a serial command line. In applications where this module is being used to provide a convenient, buffered serial interface over which application-specific binary protocols are being run, however, echo may be undesirable and this function can be used to disable it.

Returns:

None.

17.2.1.2 UARTFlushRx

Flushes the receive buffer.

Prototype:

```
void
UARTFlushRx(void)
```

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to discard any data received from the UART but not yet read using UARTgets().

Returns:

None.

17.2.1.3 UARTFlushTx

Flushes the transmit buffer.

Prototype:

```
void
UARTFlushTx(tBoolean bDiscard)
```

Parameters:

bDiscard indicates whether any remaining data in the buffer should be discarded (**true**) or transmitted (**false**).

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to flush the transmit buffer, either discarding or transmitting any data received via calls to UARTprintf() that is waiting to be transmitted. On return, the transmit buffer will be empty.

Returns:

None.

17.2.1.4 UARTgetc

Read a single character from the UART, blocking if necessary.

Prototype:

```
unsigned char
UARTgetc(void)
```

Description:

This function will receive a single character from the UART and store it at the supplied address.

In both buffered and unbuffered modes, this function will block until a character is received. If non-blocking operation is required in buffered mode, a call to UARTRxAvail() may be made to determine whether any characters are currently available for reading.

Returns:
Returns the character read.

17.2.1.5 UARTgets

A simple UART based get string function, with some line processing.

Prototype:
```
int
UARTgets(char *pcBuf,
         unsigned long ulLen)
```

Parameters:
pcBuf points to a buffer for the incoming string from the UART.
ulLen is the length of the buffer for storage of the string, including the trailing 0.

Description:
This function will receive a string from the UART input and store the characters in the buffer pointed to by *pcBuf*. The characters will continue to be stored until a termination character is received. The termination characters are CR, LF, or ESC. A CRLF pair is treated as a single termination character. The termination characters are not stored in the string. The string will be terminated with a 0 and the function will return.

In both buffered and unbuffered modes, this function will block until a termination character is received. If non-blocking operation is required in buffered mode, a call to UARTPeek() may be made to determine whether a termination character already exists in the receive buffer prior to calling UARTgets().

Since the string will be null terminated, the user must ensure that the buffer is sized to allow for the additional null character.

Returns:
Returns the count of characters that were stored, not including the trailing 0.

17.2.1.6 UARTPeek

Looks ahead in the receive buffer for a particular character.

Prototype:
```
int
UARTPeek(unsigned char ucChar)
```

Parameters:
ucChar is the character that is to be searched for.

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to look ahead in the receive buffer for a particular character and report its position if found. It is typically used to determine whether a complete line of user input is available, in which case ucChar should be set to CR ('\r') which is used as the line end marker in the receive buffer.

Returns:

Returns -1 to indicate that the requested character does not exist in the receive buffer. Returns a non-negative number if the character was found in which case the value represents the position of the first instance of *ucChar* relative to the receive buffer read pointer.

17.2.1.7 UARTprintf

A simple UART based printf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:

```
void
UARTprintf(const char *pcString,
           ...)
```

Parameters:

pcString is the format string.

... are the optional arguments, which depend on the contents of the format string.

Description:

This function is very similar to the C library `fprintf()` function. All of its output will be sent to the UART. Only the following formatting characters are supported:

- %c to print a character
- %d or %i to print a decimal value
- %s to print a string
- %u to print an unsigned decimal value
- %x to print a hexadecimal value using lower case letters
- %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
- %p to print a pointer as a hexadecimal value
- %% to print out a % character

For %s, %d, %i, %u, %p, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, “%8d” will use eight characters to print the decimal value with spaces added to reach eight; “%08d” will use eight characters as well but will add zeroes instead of spaces.

The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.

Returns:

None.

17.2.1.8 UARTRxBytesAvail

Returns the number of bytes available in the receive buffer.

Prototype:

```
int
UARTRxBytesAvail(void)
```

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to determine the number of bytes of data currently available in the receive buffer.

Returns:

Returns the number of available bytes.

17.2.1.9 UARTStdioConfig

Configures the UART console.

Prototype:

```
void
UARTStdioConfig(unsigned long ulPortNum,
                unsigned long ulBaud,
                unsigned long ulSrcClock)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

ulBaud is the bit rate that the UART is to be configured to use.

ulSrcClock is the frequency of the source clock for the UART module.

Description:

This function will configure the specified serial port to be used as a serial console. The serial parameters are set to the baud rate specified by the *ulBaud* parameter and use 8 bit, no parity, and 1 stop bit.

This function must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). This function assumes that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

17.2.1.10 UARTStdioInit

Initializes the UART console.

Prototype:

```
void
UARTStdioInit(unsigned long ulPortNum)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

Description:

This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 115200, 8-N-1. An application wishing to use a different baud rate may call UARTStdioInitExpClk() instead of this function.

This function or UARTStdioInitExpClk() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

17.2.1.11 UARTStdioInitExpClk

Initializes the UART console and allows the baud rate to be selected.

Prototype:

```
void
UARTStdioInitExpClk(unsigned long ulPortNum,
                    unsigned long ulBaud)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

ulBaud is the bit rate that the UART is to be configured to use.

Description:

This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 8-N-1 and the bit rate set according to the value of the *ulBaud* parameter.

This function or UARTStdioInit() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function. An application wishing to use 115,200 baud may call UARTStdioInit() instead of this function but should not call both functions.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

17.2.1.12 UARTStdioIntHandler

Handles UART interrupts.

Prototype:

```
void
UARTStdioIntHandler(void)
```

Description:
This function handles interrupts from the UART. It will copy data from the transmit buffer to the UART transmit FIFO if space is available, and it will copy data from the UART receive FIFO to the receive buffer if data is available.

Returns:
None.

17.2.1.13 UARTTxBytesFree

Returns the number of bytes free in the transmit buffer.

Prototype:
```
int
UARTTxBytesFree(void)
```

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to determine the amount of space currently available in the transmit buffer.

Returns:
Returns the number of free bytes.

17.2.1.14 UARTwrite

Writes a string of characters to the UART output.

Prototype:
```
int
UARTwrite(const char *pcBuf,
          unsigned long ulLen)
```

Parameters:
pcBuf points to a buffer containing the string to transmit.
ulLen is the length of the string to transmit.

Description:
This function will transmit the string to the UART output. The number of characters transmitted is determined by the *ulLen* parameter. This function does no interpretation or translation of any characters. Since the output is sent to a UART, any LF (/n) characters encountered will be replaced with a CRLF pair.

Besides using the *ulLen* parameter to stop transmitting the string, if a null character (0) is encountered, then no more characters will be transmitted and the function will return.

In non-buffered mode, this function is blocking and will not return until all the characters have been written to the output FIFO. In buffered mode, the characters are written to the UART transmit buffer and the call returns immediately. If insufficient space remains in the transmit buffer, additional characters are discarded.

Returns:
Returns the count of characters written.

17.3 Programming Example

The following example shows how to use the UART standard IO module to write a string to the UART “console”.

```
//
// Configure the appropriate pins as UART pins; in this case, PA0/PA1 are
// used for UART0.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Initialize the UART standard IO module.
//
UARTStdioInit(0);

//
// Print a string.
//
UARTprintf("Hello world!\n");
```

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
TI E2E Community Home Page	e2e.ti.com		

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