EK-LM3S1968 Firmware Development Package

USER'S GUIDE

SW-EK-LM3S1968-UG-6075

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 6075 of this document, last updated on June 04, 2010.

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® EK-LM3S1968 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, and a small speaker 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-lm3s1968.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-lm3s1968.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-lm3s1968` subdirectory of the firmware development package source distribution.

2.1 AES Pre-expanded Key (aes_expanded_key)

This example shows how to use pre-expanded keys to encrypt some plaintext, and then decrypt it back to the original message. Using pre-expanded keys avoids the need to perform the expansion at run-time. This example also uses cipher block chaining (CBC) mode instead of the simpler ECB mode.

2.2 AES Normal Key (aes_set_key)

This example shows how to set an encryption key and then use that key to encrypt some plaintext. It then sets the decryption key and decrypts the previously encrypted block back to plaintext.

2.3 Audio Playback (audio)

This example application plays audio via the Class-D amplifier and speaker. The same source audio clip is provided in both PCM and ADPCM format so that the audio quality can be compared.

2.4 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).

2.5 Blinky (blinky)

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

2.6 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.

2.7 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.

2.8 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.

2.9 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.

2.10 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.

2.11 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.

2.12 Hibernate Example (hibernate)

An example to demonstrate the use of the Hibernation module. The user can put the microcontroller in hibernation by pressing the select button. The microcontroller will then wake on its own after 5 seconds, or immediately if the user presses the select button again. The program keeps a count of the number of times it has entered hibernation. The value of the counter is stored in the battery backed memory of the Hibernation module so that it can be retrieved when the microcontroller wakes.

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-LM3S1968 Quickstart Application (qs_ek-lm3s1968)

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.

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 processor will enter hibernation mode, and the red LED will turn on. Hibernation mode 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 (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
```

Audio Converter

Usage:

```
converter [OPTION]... [INPUT FILE]
```

Description:

Converts a file containing raw 16-bit mono PCM data into an C array. The input data is assumed to be 16-bit, signed, mono PCM data at the desired sample rate and with no header (in other words, raw PCM data). This utility can encode it as either 8-bit PCM or 4-bit IMA ADPCM data, resulting in a 50% or 75% reduction in size (respectively).

In 8-bit unsigned PCM format, each byte represents a single sample. This format provides 8 bits of resolution in the output stream. In IMA ADPCM format, each byte represents two encoded samples. After the decoding process, the output stream has approximately 14 bits of resolution. There will generally be little to no degradation of audio quality when using IMA ADPCM, though it varies based on the audio clip.

The output is a C array definition that can be placed into a source or header file and passed to a compiler. The contents of the array can be passed to the audio playback function of a board's class-D audio driver in order to playback the audio file. The sample rate of the input file must match the sample rate expected by the class-D audio driver in use.

In order to produce a raw 16-bit mono PCM file, some pre-processing is likely required. Since most audio files have headers, they must be stripped as well. There are numerous open source and commercial audio editors that are capable of performing these conversions, with `sox` (`http://sox.sourceforge.net`) being an open source tool that is powerful and easy to use. The following `sox` command will take an input audio file and convert it into the correct format (assuming that an 8 KHz sample rate is required):

```
sox foo.wav -t raw -r 8000 -c 1 -s 2 foo.raw polyphase
```

`sox` will sample rate convert the audio (`polyphase` selects a higher quality sample rate conversion algorithm), mix a stereo channel pair to get to mono, and convert the sample size to 16 bits (each step only if the input file is not already in the specified format). If may be helpful (and/or necessary) to also include `vol {factor}` before `polyphase` in order to increase or

decrease the volume of the waveform. If `sox` reports that clipping has occurred, the volume needs to be reduced to prevent the clipping.

The same steps can be performed using other audio editor software.

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

Arguments:

-a specifies that the audio should be encoded using ADPCM.

-c COUNT specifies the number of audio samples to place into the output file.

-h displays usage information.

-n NAME specifies the name of the C array in the output file.

-o FILENAME specifies the name of the output file.

-p specifies that the audio should be encoded using PCM.

-s SKIP specifies the number of audio samples at the beginning of the file to skip.

INPUT FILE specifies the name of the input PCM file. If no input file is specified, the input PCM is read from standard input.

Example:

The following will encode a 16-bit mono PCM file into IMA ADPCM and place the result into a C array called `g_pucFoo`:

```
converter -a -n g_pucFoo -o foo.h foo.raw
```

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 Class-D Audio Driver

4.1 Introduction

The board has a Class-D audio amplifier connected to a small magnetic speaker that can be used to playback digital audio waveforms. The audio driver uses a carrier frequency of 64 KHz to playback 8 KHz digital audio waveforms, which can be in either 8-bit unsigned PCM format or IMA ADPCM format.

When running the processor at 50 MHz, the 64 KHz carrier frequency results in approximately 9.5 bits of resolution in the PWM output. Running the processor at slower rates will reduce the PWM resolution, and therefore the audio quality, so for best audio quality it is recommended that the processor be run at 50 MHz.

In 8-bit unsigned PCM format, each byte represents a single sample. This format provides 8 bits of resolution in the output stream and it takes 8000 bytes per second of audio.

In IMA ADPCM format, each byte represents two encoded samples. After the decoding process, the output stream has approximately 14 bits of resolution and it takes 4000 bytes per second of audio. There will generally be little to no degradation of audio quality when using IMA ADPCM.

The `converter` utility can be used to prepare audio files for playback by this audio driver.

This driver is located in `boards/ek-lm3s1968/drivers`, with `class-d.c` containing the source code and `class-d.h` containing the API definitions for use by applications.

4.2 API Functions

Functions

- tBoolean ClassDBusy (void)
- void ClassDInit (unsigned long ulPWMClock)
- void ClassDPlayADPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPlayPCM (const unsigned char *pucBuffer, unsigned long ulLength)
- void ClassDPWMHandler (void)
- void ClassDStop (void)
- void ClassDVolumeDown (unsigned long ulVolume)
- void ClassDVolumeSet (unsigned long ulVolume)
- void ClassDVolumeUp (unsigned long ulVolume)

4.2.1 Function Documentation

4.2.1.1 ClassDBusy

Determines if the Class-D audio driver is busy.

Prototype:

```
tBoolean
ClassDBusy(void)
```

Description:

This function determines if the Class-D audio driver is busy, either performing the startup or shutdown ramp for the speaker or playing an audio stream.

Returns:

Returns **true** if the Class-D audio driver is busy and **false** otherwise.

4.2.1.2 ClassDInit

Initializes the Class-D audio driver.

Prototype:

```
void
ClassDInit(unsigned long ulPWMClock)
```

Parameters:

ulPWMClock is the rate of the clock supplied to the PWM module.

Description:

This function initializes the Class-D audio driver, preparing it to output audio data to the speaker.

The PWM module clock should be as high as possible; lower clock rates reduces the quality of the produced audio. For the best quality audio, the PWM module should be clocked at 50 MHz.

Note:

In order for the Class-D audio driver to function properly, the Class-D audio driver interrupt handler (ClassDPWMHandler()) must be installed into the vector table for the PWM1 interrupt.

Returns:

None.

4.2.1.3 ClassDPlayADPCM

Plays a buffer of 8 KHz IMA ADPCM data.

Prototype:

```
void
ClassDPlayADPCM(const unsigned char *pucBuffer,
                unsigned long ulLength)
```

Parameters:

pucBuffer is a pointer to the buffer containing the IMA ADPCM encoded data.

ulLength is the number of bytes in the buffer.

Description:

This function starts playback of a stream of IMA ADPCM encoded data. The data is decoded as needed and therefore does not require a large buffer in SRAM. This provides a 2:1 compression ratio relative to raw 8-bit PCM with little to no loss in audio quality.

Returns:

None.

4.2.1.4 ClassDPlayPCM

Plays a buffer of 8 KHz, 8-bit, unsigned PCM data.

Prototype:

```
void
ClassDPlayPCM(const unsigned char *pucBuffer,
              unsigned long ulLength)
```

Parameters:

pucBuffer is a pointer to the buffer containing 8-bit, unsigned PCM data.

ulLength is the number of bytes in the buffer.

Description:

This function starts playback of a stream of 8-bit, unsigned PCM data. Since the data is unsigned, a value of 128 represents the mid-point of the speaker's travel (that is, corresponds to no DC offset).

Returns:

None.

4.2.1.5 ClassDPWMHandler

Handles the PWM1 interrupt.

Prototype:

```
void
ClassDPWMHandler(void)
```

Description:

This function responds to the PWM1 interrupt, updating the duty cycle of the output waveform in order to produce sound. It is the application's responsibility to ensure that this function is called in response to the PWM1 interrupt, typically by installing it in the vector table as the handler for the PWM1 interrupt.

Returns:

None.

4.2.1.6 ClassDStop

Stops playback of the current audio stream.

Prototype:

```
void
ClassDStop(void)
```

Description:

This function immediately stops playback of the current audio stream. As a result, the output is changed directly to the mid-point, possibly resulting in a pop or click. It is then ramped down to no output, eliminating the current draw through the Class-D amplifier and speaker.

Returns:

None.

4.2.1.7 ClassDVolumeDown

Decreases the volume of the audio playback.

Prototype:

```
void
ClassDVolumeDown(unsigned long ulVolume)
```

Parameters:

ulVolume is the amount by which to decrease the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:

This function decreases the volume of the audio playback relative to the current volume.

Returns:

None.

4.2.1.8 ClassDVolumeSet

Sets the volume of the audio playback.

Prototype:

```
void
ClassDVolumeSet(unsigned long ulVolume)
```

Parameters:

ulVolume is the volume of the audio playback, specified as a value between 0 (for silence) and 256 (for full volume).

Description:

This function sets the volume of the audio playback. Setting the volume to 0 will mute the output, while setting the volume to 256 will play the audio stream without any volume adjustment (that is, full volume).

Returns:
None.

4.2.1.9 ClassDVolumeUp

Increases the volume of the audio playback.

Prototype:
```
void
ClassDVolumeUp(unsigned long ulVolume)
```

Parameters:
ulVolume is the amount by which to increase the volume of the audio playback, specified as a value between 0 (for no adjustment) and 256 maximum adjustment).

Description:
This function increases the volume of the audio playback relative to the current volume.

Returns:
None.

4.3 Programming Example

The following example shows how to use the Class-D audio driver. This assumes that the vector table in the startup code has ClassDPWMHandler() listed as the handler for the PWM1 interrupt.

```
unsigned char pucBuffer[256];

//
// Initialize the Class-D driver.
//
ClassDInit(SysCtlClockGet());

//
// Fill pucBuffer with PCM audio data.
//
...

//
// Play the audio data.
//
ClassDPlayPCM(pucBuffer, sizeof(pucBuffer));
```

5 Display Driver

5.1 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-lm3s1968/drivers`, with `rit128x96x4.c` containing the source code and `rit128x96x4.h` containing the API definitions for use by applications.

5.2 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)

5.2.1 Function Documentation

5.2.1.1 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.

5.2.1.2 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.

5.2.1.3 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.

5.2.1.4 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.

5.2.1.5 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.

5.2.1.6 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.

5.2.1.7 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.

5.2.1.8 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.

5.3 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);
```

6 Command Line Processing Module

6.1 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.

6.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

6.2.1 Data Structure Documentation

6.2.1.1 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.

6.2.2 Define Documentation

6.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

6.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

6.2.3 Function Documentation

6.2.3.1 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.

6.2.4 Variable Documentation

6.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

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

6.3 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));
}
```

7 CPU Usage Module

7.1 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.

7.2 API Functions

Functions

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

7.2.1 Function Documentation

7.2.1.1 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.

7.2.1.2 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.

7.3 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();
    }
}
```

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 Sine Calculation Module

11.1 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.

11.2 API Functions

Functions

- long sine (unsigned long ulAngle)

11.2.1 Function Documentation

11.2.1.1 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.

11.3 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);
}
```

12 Micro Standard Library Module

12.1 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.

12.2 API Functions

Data Structures

- tTime

Functions

- void ulocaltime (unsigned long ulTime, tTime *psTime)
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)
- int usprintf (char *pcBuf, const char *pcString,...)
- int ustrcasecmp (const char *pcStr1, const char *pcStr2)
- 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)

12.2.1 Data Structure Documentation

12.2.1.1 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.

12.2.2 Function Documentation

12.2.2.1 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.

12.2.2.2 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 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, %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.

12.2.2.3 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 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, %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.

12.2.2.4 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*.

12.2.2.5 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*.

12.2.2.6 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.

12.2.2.7 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.

12.2.2.8 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 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, %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.

12.3 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);
```

13 UART Standard IO Module

13.1 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.

13.1.1 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.

13.1.2 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.

13.2 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 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)

13.2.1 Function Documentation

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 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.

13.2.1.4 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.

13.2.1.5 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.

13.2.1.6 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.

13.2.1.7 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 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, %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.

13.2.1.8 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.

13.2.1.9 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.

13.2.1.10 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.

13.2.1.11 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.

13.2.1.12 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.

13.2.1.13 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.

13.3 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**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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