EK-EVALBOT Firmware Development Package

USER'S GUIDE

SW-EK-EVALBOT-UG-9107

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 9107 of this document, last updated on June 07, 2012.

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® EK-EVALBOT evaluation board is a platform that can be used for software development and fun exploration of the features of a small robot. It can also be used as a guide for custom board design using a Stellaris microcontroller.

The EK-EVALBOT includes a Stellaris ARM® Cortex™-M3-based microcontroller and the following features:

- Mechanical components assembled by user
- Stellaris® LM3S9B92 microcontroller
- MicroSD card connector
- I2S audio codec with speaker
- USB Host and Device connectors
- RJ45 Ethernet connector
- Bright 96 x 16 blue OLED display
- On-board In-Circuit Debug Interface (ICDI)
- Battery power (3 AA batteries) or power through ICDI USB cable
- Wireless communication expansion port
- Robot features
  - Two DC gear-motors provide drive and steering
  - Opto-sensors detect wheel rotation with 22.5 degree resolution
  - Sensors for “bump” detection

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 EK-EVALBOT evaluation board. Examples are included to show how to use the motors, process sensor input, show information on the display, and play audio sounds using the speaker.

A number of drivers are provided to make it easier to use the features of the EVALBOT. These drivers also contain low-level code that make use of the Stellaris peripheral driver library and utilities.

There is an IAR workspace file (`ek-evalbot.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-evalbot.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-evalbot` subdirectory of the firmware development package source distribution.

2.1 EVALBOT Controlled by an eZ430-Chronos Watch (chronos_drive)

This application allows the EVALBOT to be driven under radio control from a 915 MHz eZ430-Chronos sport watch. To run it, you need both the sport watch (running its default firmware) and a "CC1101 Evaluation Module 868-915", part number CC1101EMK868-915. Both of these can be ordered from estore.ti.com or other Texas Instruments part distributors.

To run the demonstration:

1. Place the EVALBOT within a flat, enclosed space then press the "On/Reset" button. You should see scrolling text appear on its OLED display. If you wait 5 seconds at this step and without doing anything else, the robot will start driving, making random turns or turning away from anything it bumps.
2. Hold the Chronos watch level and repeatedly press the bottom left button on the Chronos watch until you see "ACC" displayed, then press the watch's bottom right button to enable the radio.
3. After a few seconds, the EVALBOT links with the watch and stops driving. The display also indicates that it is connected to a Chronos. If this does not occur within 5 seconds, turn the Chronos radio off by pressing the bottom right button again and then resetting the EVALBOT using the "On/Reset" button.
4. Once the watch and EVALBOT are connected, drive the robot by tilting the watch. Tilting the watch forward and backward controls the direction - tilting forward moves the robot forward, and tilting backward moves it in reverse. The speed is controlled by the amount of tilt. When reversing, the robot beeps.
5. To turn the robot, when it is moving in forward or reverse, tilt the watch left or right to initiate a turn.

While controlling the EVALBOT, the watch buttons perform the following actions:

Top Left Stop the EVALBOT.

Bottom Left Restart the EVALBOT.

Top Right Sound the EVALBOT horn.

Bottom Right Turn Chronos radio on or off.

When controlling the EVALBOT, maximum speed is achieved with the watch face oriented vertically (90 degrees from the normal, "flat" viewing position). Since the accelerometer calibration varies slightly from watch to watch, the example contains a calibration mode. Press "Switch 1" on EVALBOT to enter calibration then move the watch through the full range of movement you want to use to control the robot. Once you have finished the movement, press "Switch 2" to resum normal operation with the control inputs scaled appropriately.

The application can be easily recompiled to operate using either the 433 MHz or 868 Mhz version of the eZ430-Chronos. To operate with a 433 MHz watch, a CC1101EMK433 is required in place of the CC1101EMK868-915 since the antenna design is different between these two frequency bands.

To compile for different frequencies, modify the `simpliciti-config.h` file to ensure that labels are defined as follows and then rebuild the application.

For 915 MHz operation (as used in the USA):

```
#define ISM_US #undef ISM_LF #undef ISM_EU
```

For 868 MHz operation (as used in Europe):

```
#undef ISM_US #undef ISM_LF #define ISM_EU
```

For 433 MHz operation (as used in Japan):

```
#undef ISM_US #define ISM_LF #undef ISM_EU
```

2.2 Motor Demo (motor_demo)

This example application demonstrates the use of the motors. The buttons and bump sensors are used to start, stop, and reverse the motors. It uses the system tick timer (SysTick) as a time reference for button debouncing and blinking LEDs.

When this example is running, the display shows a message identifying the example. The two user buttons on the right are used to control the motors. The top button controls the left motor and the bottom button controls the right motor. When a motor control button is pressed, the motor runs in the forward direction. When the button is pressed a second time, the motor pauses. Pressing the button a third time causes the motor to run in reverse. This cycle can be repeated by continuing to press the button.

When a motor is running, either forward or reverse, pressing the bump sensor pauses the motor. When the bump sensor is released, the motor resumes running in the same direction.

2.3 EVALBOT Autonomous Operation (qs-autonomous)

This application controls the EVALBOT, allowing it to operate autonomously. When first turned on, the EVALBOT shows messages on the display and toggles the two LEDs. Upon pressing button 1, the robot drives in a straight line. If one of the bumper sensors is activated, EVALBOT stops

immediately, rotates a random angle, and then resumes driving forward. While driving forward, the EVALBOT also makes turns after a random duration of time. Pressing button 2 stops the motion.

2.4 Simple Display (simple_display)

This example application demonstrates the use of the display and LEDs on the EK-EVALBOT by printing a series of messages on the display and blinking the LEDs. It uses the system tick timer (SysTick) as a time reference.

2.5 Sound Demo (sound_demo)

This example application demonstrates the use of the EVALBOT audio system by cycling through a set of audio clips and playing them using the buttons and bump sensors.

When the application starts, a message displays for several seconds. Once the message is finished, the display shows the name of the first audio clip. You can cycle through the available audio clips by pressing the left and right bump sensors. The name of each clip selected is shown on the display.

A clip can be played by pressing the front-most, right user button. While the clip is playing, it can be stopped by pressing the rear-most, right user button.

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 DAC Driver

4.1 Introduction

The EVALBOT DAC driver provides functions to utilize the audio DAC included as part of the EVALBOT board. The functions provide a way to initialize. enable and disable the DAC, and control the sound volume. The DAC API is used by the sound driver, so if the sound driver is used, then the application does not need to directly call the DAC driver.

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

4.2 API Functions

Functions

- void DACClassDDis (void)
- void DACClassDEn (void)
- tBoolean DACInit (void)
- unsigned long DACVolumeGet (void)
- void DACVolumeSet (unsigned long ulVolume)

4.2.1 Function Documentation

4.2.1.1 DACClassDDis

Disables the class D amplifier in the DAC.

Prototype:

```
void
DACClassDDis(void)
```

Description:

This function disables the class D amplifier in the DAC.

Returns:

None.

4.2.1.2 DACClassDEn

Enables the class D amplifier in the DAC.

Prototype:

```
void
DACClassDEn(void)
```

Description:
This function enables the class D amplifier in the DAC.

Returns:
None.

4.2.1.3 DACInit

Initializes the TLV320AIC3107 DAC.

Prototype:

```
tBoolean
DACInit(void)
```

Description:
This function initializes the I2C interface and the TLV320AIC3107 DAC. It must be called prior to any other API in the DAC module.

Returns:
Returns **true** on success or **false** on failure.

4.2.1.4 DACVolumeGet

Returns the current DAC volume setting.

Prototype:

```
unsigned long
DACVolumeGet(void)
```

Description:
This function may be called to determine the current DAC volume setting. The value returned is expressed as a percentage of full volume.

Returns:
Returns the current volume setting as a percentage between 0 and 100 inclusive.

4.2.1.5 DACVolumeSet

Sets the volume on the DAC.

Prototype:

```
void
DACVolumeSet(unsigned long ulVolume)
```

Parameters:

ulVolume is the volume to set, specified as a percentage between 0% (silence) and 100% (full volume), inclusive.

Description:

This function adjusts the audio output volume to the specified percentage. The adjusted volume will not go above 100% (full volume).

Returns:

None.

4.3 Programming Example

The following example shows how to use the DAC driver to initialize the DAC and set the volume.

```
//
// Initialize the EVALBOT DAC
//
DACInit();

//
// Set the volume to 50%
//
DACVolumeSet(50);
```

5 Display Driver

5.1 Introduction

The display driver provides a way to draw text and images on the 96x16 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-evalbot/drivers`, with `display96x16x1.c` containing the source code and `display96x16x1.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

- void Display96x16x1Clear (void)
- void Display96x16x1ClearLine (unsigned long ulY)
- void Display96x16x1DisplayOff (void)
- void Display96x16x1DisplayOn (void)
- void Display96x16x1ImageDraw (const unsigned char *pucImage, unsigned long ulX, unsigned long ulY, unsigned long ulWidth, unsigned long ulHeight)
- void Display96x16x1Init (tBoolean bFast)
- void Display96x16x1StringDraw (const char *pcStr, unsigned long ulX, unsigned long ulY)
- void Display96x16x1StringDrawCentered (const char *pcStr, unsigned long ulY, tBoolean bClear)
- void Display96x16x1StringDrawLen (const char *pcStr, unsigned long ulLen, unsigned long ulX, unsigned long ulY)

5.2.1 Function Documentation

5.2.1.1 Display96x16x1Clear

Clears the OLED display.

Prototype:

```
void
Display96x16x1Clear(void)
```

Description:

This function clears the OLED display, turning all pixels off.

Returns:
None.

5.2.1.2 Display96x16x1ClearLine

Clears a single line on the OLED display.

Prototype:
```
void
Display96x16x1ClearLine(unsigned long ulY)
```

Parameters:
ulY is the display line to be cleared, 0 or 1.

Description:
This function will clear one text line of the display.

Returns:
None.

5.2.1.3 Display96x16x1DisplayOff

Turns off the OLED display.

Prototype:
```
void
Display96x16x1DisplayOff(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 Display96x16x1DisplayOn

Turns on the OLED display.

Prototype:
```
void
Display96x16x1DisplayOn(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 Display96x16x1ImageDraw

Displays an image on the OLED display.

Prototype:

```
void
Display96x16x1ImageDraw(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 eight scan line blocks from the top of the display (that is, only 0 and 1 are valid).

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

ulHeight is the height of the image, specified in eight row blocks (that is, only 1 and 2 are valid).

Description:

This function will display a bitmap graphic on the display. The image to be displayed must be a multiple of eight scan lines high (that is, one row) and will be drawn at a vertical position that is a multiple of eight scan lines (that is, scan line zero or scan line eight, corresponding to row zero or row one).

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 the eight scan lines of the column, with the top scan line being in the least significant bit of the byte and the bottom scan line being in the most significant bit of the byte.

For example, an image four columns wide and sixteen scan lines tall would be arranged as follows (showing how the eight bytes of the image would appear on the display):

```
+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 0 | 6 |  | 1 | 6 |  | 2 | 6 |  | 3 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
+-------+  +-------+  +-------+  +-------+

+-------+  +-------+  +-------+  +-------+
|   | 0 |  |   | 0 |  |   | 0 |  |   | 0 |
| B | 1 |  | B | 1 |  | B | 1 |  | B | 1 |
| y | 2 |  | y | 2 |  | y | 2 |  | y | 2 |
| t | 3 |  | t | 3 |  | t | 3 |  | t | 3 |
| e | 4 |  | e | 4 |  | e | 4 |  | e | 4 |
|   | 5 |  |   | 5 |  |   | 5 |  |   | 5 |
| 4 | 6 |  | 5 | 6 |  | 6 | 6 |  | 7 | 6 |
|   | 7 |  |   | 7 |  |   | 7 |  |   | 7 |
+-------+  +-------+  +-------+  +-------+
```

Returns:
None.

5.2.1.6 Display96x16x1Init

Initialize the OLED display.

Prototype:
```
void
Display96x16x1Init(tBoolean bFast)
```

Parameters:
bFast is a boolean that is *true* if the I2C interface should be run at 400 kbps and *false* if it should be run at 100 kbps.

Description:
This function initializes the I2C interface to the OLED display and configures the SSD0303 or SSD1300 controller on the panel.

Returns:
None.

5.2.1.7 Display96x16x1StringDraw

Displays a string on the OLED display.

Prototype:
```
void
Display96x16x1StringDraw(const char *pcStr,
                         unsigned long ulX,
                         unsigned long ulY)
```

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 eight scan line blocks from the top of the display (that is, only 0 and 1 are valid).

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.

Returns:
None.

5.2.1.8 Display96x16x1StringDrawCentered

Draws a string horizontally centered on the OLED display.

Prototype:

```
void
Display96x16x1StringDrawCentered(const char *pcStr,
                                 unsigned long ulY,
                                 tBoolean bClear)
```

Parameters:

pcStr points to the NULL terminated string to be displayed.

ulY is the vertical position of the string specified in terms of 8 pixel character cells. Valid values are 0 and 1.

bClear is **true** if all uncovered areas of the display line are to be cleared or **false** if they are to be left unaffected.

Description:

This function displays a string centered on a given line of the OLED display and optionally clears sections of the display line to the left and right of the provided string.

Returns:

None.

5.2.1.9 Display96x16x1StringDrawLen

Displays a length-restricted string on the OLED display.

Prototype:

```
void
Display96x16x1StringDrawLen(const char *pcStr,
                            unsigned long ulLen,
                            unsigned long ulX,
                            unsigned long ulY)
```

Parameters:

pcStr is a pointer to the string to display.

ulLen is the number of characters 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 eight scan line blocks from the top of the display (that is, only 0 and 1 are valid).

Description:

This function will draw a specified number of characters of 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.

This function is similar to Display96x16x1SringDraw() except that the// length of the string to display can be specified.

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 using the fast interface clock option.
//
Display96x16x1Init(true);

//
// Write text on the display.
//
Display96x16x1StringDraw("Hello", 0, 0);
```

6 I/O Driver

6.1 Introduction

The I/O driver provides functions to make it easy to use the push buttons and LEDs on the EVALBOT board. The button state can be queried, and the LEDs can be turned on and off.

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

6.2 API Functions

Enumerations

- tButton
- tLED

Functions

- void LED_Off (tLED eLED)
- void LED_On (tLED eLED)
- void LED_Toggle (tLED eLED)
- void LEDsInit (void)
- void PushButtonDebouncer (void)
- tBoolean PushButtonGetDebounced (tButton eButton)
- tBoolean PushButtonGetStatus (tButton eButton)
- void PushButtonsInit (void)

6.2.1 Enumeration Documentation

6.2.1.1 tButton

Description:
This enumerated type defines the two user switches on EVALBOT.

Enumerators:

BUTTON_1 Switch 1 nearest the front on the right side of EVALBOT.

BUTTON_2 Switch 2 nearest the back on the right side of EVALBOT.

6.2.1.2 tLED

Description:
This enumerated type defines either one or both EVALBOT LEDs. It is used by functions which change the state of the LEDs.

Enumerators:
BOTH_LEDS Both LEDs will be affected.
LED_1 LED 1 on the right side of the EVALBOT will be affected.
LED_2 LED 2 on the left side of the EVALBOT will be affected.

6.2.2 Function Documentation

6.2.2.1 LED_Off

Turn one or both of the EVALBOT LEDs off.

Prototype:
```
void
LED_Off(tLED eLED)
```

Parameters:
eLED indicates the LED or LEDs to turn off. Valid values are *LED_1*, *LED_2* or *BOTH_LEDS*.

Description:
This function may be used to turn off either one or both of the LEDs on the EVALBOT. Callers must ensure that they have previously called LEDsInit().

Returns:
None.

6.2.2.2 LED_On

Turn one or both of the EVALBOT LEDs on.

Prototype:
```
void
LED_On(tLED eLED)
```

Parameters:
eLED indicates the LED or LEDs to turn on. Valid values are *LED_1*, *LED_2* or *BOTH_LEDS*.

Description:
This function may be used to light either one or both of the LEDs on the EVALBOT. Callers must ensure that they have previously called LEDsInit().

Returns:
None.

6.2.2.3 LED_Toggle

Toggle one or both of the EVALBOT LEDs.

Prototype:

```
void
LED_Toggle(tLED eLED)
```

Parameters:

eLED indicates the LED or LEDs to toggle. Valid values are *LED_1*, *LED_2* or *BOTH_LEDS*.

Description:

This function may be used to toggle either one or both of the LEDs on the EVALBOT. If the LED is currently lit, it will be turned off and vice verse. Callers must ensure that they have previously called LEDsInit().

Returns:

None.

6.2.2.4 LEDsInit

Initializes the EVALBOT's LEDs

Prototype:

```
void
LEDsInit(void)
```

Description:

This function must be called to initialize the GPIO pins used to control EVALBOT's LEDs prior to calling LED_Off(), LED_On() or LED_Toggle().

Returns:

None.

6.2.2.5 PushButtonDebouncer

Debounces the EVALBOT push buttons when called periodically.

Prototype:

```
void
PushButtonDebouncer(void)
```

Description:

If button debouncing is used, this function should be called periodically, for example every 10 ms. It will check the buttons state and save a debounced state that can be read by the application at any time.

Returns:

None.

6.2.2.6 PushButtonGetDebounced

Get the debounced state of a push button on the EVALBOT.

Prototype:

```
tBoolean
PushButtonGetDebounced(tButton eButton)
```

Parameters:

eButton is the ID of the push button to query. Valid values are *BUTTON_1* and *BUTTON_2*.

Description:

This function may be called to determine the debounced state of one of the two user switches on EVALBOT. Prior to calling it, an application must ensure that it has called PushButtonsInit(), and that the function PushButtonDebouncer() is called periodically by the application to keep the debounce processing running.

Returns:

Returns **false** if the push button is pressed or **true** if it is not pressed.

6.2.2.7 PushButtonGetStatus

Get the status of a push button on the EVALBOT.

Prototype:

```
tBoolean
PushButtonGetStatus(tButton eButton)
```

Parameters:

eButton is the ID of the push button to query. Valid values are *BUTTON_1* and *BUTTON_2*.

Description:

This function may be called to determine the state of one of the two user switches on EVALBOT. Prior to calling it, an application must ensure that it has called PushButtonsInit().

Returns:

Returns **false** if the push button is pressed or **true** if it is not pressed.

6.2.2.8 PushButtonsInit

Initializes the EVALBOT's push buttons.

Prototype:

```
void
PushButtonsInit(void)
```

Description:

This function must be called prior to PushButtonGetStatus() to configure the GPIOs used to support the user switches on EVALBOT.

Returns:

None.

6.3 Programming Example

The following example shows how to use the I/O driver to read a button and turn on an LED.

```
//
// Initialize the LEDs.
//
LEDsInit();

//
// Initialize the push buttons
//
PushButtonsInit();

//
// Turn on one of the LEDs
//
LED_On(LED_1);

//
// Read the button status
//
bStatus = PushButtonGetStatus(BUTTON_2);
```

7 Motor Driver

7.1 Introduction

The motor driver provides functions to control the EVALBOT motors. Functions are provided to initialize the motor drivers, and to control the speed and direction.

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

7.2 API Functions

Enumerations

- tDirection
- tSide

Functions

- void MotorDir (tSide ucMotor, tDirection eDirection)
- void MotorRun (tSide ucMotor)
- void MotorsInit (void)
- void MotorSpeed (tSide ucMotor, unsigned short usPercent)
- void MotorStop (tSide ucMotor)

7.2.1 Enumeration Documentation

7.2.1.1 tDirection

Description:
The enumerated type defining motor drive directions.

Enumerators:
FORWARD Run the motor in the forward direction.
REVERSE Run the motor in the reverse direction.

7.2.1.2 tSide

Description:
The enumerated type defining one of the two EVALBOT motors.

Enumerators:
LEFT_SIDE Defines the left side motor.
RIGHT_SIDE Defines the right side motor.

7.2.2 Function Documentation

7.2.2.1 MotorDir

Configures the DMOS Motor Driver to drive the motor in the required direction.

Prototype:
```
void
MotorDir(tSide ucMotor,
         tDirection eDirection)
```

Parameters:
ucMotor determines which motor's direction should be set. Valid values are *LEFT_SIDE* or *RIGHT_SIDE*.
eDirection sets the motor direction. Valid values are *FORWARD* or *REVERSE*

Description:
This function may be used to set the drive direction for one of the motors.

Returns:
None.

7.2.2.2 MotorRun

Starts the motor.

Prototype:
```
void
MotorRun(tSide ucMotor)
```

Parameters:
ucMotor determines which motor should be started. Valid values are *LEFT_SIDE* or *RIGHT_SIDE*.

Description:
This function will start either the right or left motor depending upon the value of the *ucMotor* parameter. The motor duty cycle will be the last value passed to the MotorSpeed() function for this motor.

Returns:
None.

7.2.2.3 MotorsInit

Initializes peripherals used to control the two EVALBOT motors.

Prototype:

```
void
MotorsInit(void)
```

Description:
This function must be called before any other API in this file. It initializes the GPIO pins and PWMs used to drive the two motors on the EVALBOT.

Returns:
None.

7.2.2.4 MotorSpeed

Sets the motor to be driven at the requested duty cycle.

Prototype:

```
void
MotorSpeed(tSide ucMotor,
           unsigned short usPercent)
```

Parameters:
ucMotor determines which motor's duty cycle is to be set. Valid values are *LEFT_SIDE* or *RIGHT_SIDE*.
usPercent Percent of the maximum speed to drive the motor in 8.8 fixed point format. This value must be less than (100 << 8).

Description:
This function can be called to set the duty cycle of one or other of the motors.

Returns:
None.

Note:
The duty cycle and motor speed are not the same thing, although there is a relation.

7.2.2.5 MotorStop

Stops the motor.

Prototype:

```
void
MotorStop(tSide ucMotor)
```

Parameters:
ucMotor determines which motor should be stopped. Valid values are *LEFT_SIDE* or *RIGHT_SIDE*.

Description:
This function will stop either the right or left motor depending upon the value of the *ucMotor* parameter.

Returns:
None.

7.3 Programming Example

The following example shows how to use the motor driver to control a motor.

```
//
// Initialize the motor driver
//
MotorsInit();

//
// Set the motor direction
//
MotorDir(LEFT_SIDE, FORWARD);

//
// Set the motor control level lto 65% duty cycle.
// Percent is specified in 8.8 fixed point format.
//
MotorSpeed(LEFT_SIDE, (65 << 8));

//
// Make the motor start running
//
MotorRun(LEFT_SIDE);
```

8 Sensors Driver

8.1 Introduction

The sensors driver provides functions to access the sensors on the EVALBOT boards. The EVALBOT has two sets of sensors. The first set is the bumper switches. This driver provides a way to query the state of the bumper switches to detect when the EVALBOT bumps into an obstacle. The second set of sensors is the wheel rotation sensors. This driver can be configured to call back a function that is supplied by the application, whenever wheel rotation is detected. The application can use this callback to determine the wheel speed.

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

8.2 API Functions

Enumerations

- tBumper
- tWheel

Functions

- void BumpSensorDebouncer (void)
- tBoolean BumpSensorGetDebounced (tBumper eBumper)
- tBoolean BumpSensorGetStatus (tBumper eBumper)
- void BumpSensorsInit (void)
- void WheelSensorDisable (void)
- void WheelSensorEnable (void)
- void WheelSensorIntDisable (tWheel eWheel)
- void WheelSensorIntEnable (tWheel eWheel)
- void WheelSensorIntHandler (void)
- void WheelSensorsInit (void (*pfnCallback)(tWheel eWheel))

8.2.1 Enumeration Documentation

8.2.1.1 tBumper

Description:

The enumerated type defining bumper left and right.

Enumerators:

BUMP_LEFT Left bump sensor.

BUMP_RIGHT Right bump sensor.

8.2.1.2 tWheel

Description:

The enumerated type defining wheel sensor left and right.

Enumerators:

WHEEL_LEFT Left wheel sensor.

WHEEL_RIGHT Right wheel sensor.

8.2.2 Function Documentation

8.2.2.1 BumpSensorDebouncer

Debounces the EVALBOT sensor switches when called periodically.

Prototype:

```
void
BumpSensorDebouncer(void)
```

Description:

If bump sensor debouncing is used, this function should be called periodically, for example every 10 ms. It will check the bump sensor state and save a debounced state that can be read by the application at any time.

Returns:

None.

8.2.2.2 BumpSensorGetDebounced

Gets the debounced state of a bump sensors on the board.

Prototype:

```
tBoolean
BumpSensorGetDebounced(tBumper eBumper)
```

Parameters:

eBumper identifies the sensor whose state is to be returned. Valid values are *BUMP_RIGHT* and *BUMP_LEFT*.

Description:

This function may be used to determine the debounced state of one or other of the EVALBOT's front bump sensors. If this function is used, then the application must periodically call the function BumpSensorDebouncer().

Returns:

Returns *true* if the sensor is closed or *false* if it is open.

8.2.2.3 BumpSensorGetStatus

Gets the status of a bump sensors on the board.

Prototype:

```
tBoolean
BumpSensorGetStatus(tBumper eBumper)
```

Parameters:

eBumper identifies the sensor whose state is to be returned. Valid values are *BUMP_RIGHT* and *BUMP_LEFT*.

Description:

This function may be used to determine the current state of one or other of the EVALBOT's front bump sensors.

Returns:

Returns *true* if the sensor is closed or *false* if it is open.

8.2.2.4 BumpSensorsInit

Initializes the board's bump sensors.

Prototype:

```
void
BumpSensorsInit(void)
```

Description:

This function must be called before any attempt to read the board bump sensors. It configures the GPIO ports used by the sensors.

Returns:

None.

8.2.2.5 WheelSensorDisable

Disables the LEDs for both EVALBOT wheel sensors.

Prototype:

```
void
WheelSensorDisable(void)
```

Description:

This function disables the LEDs used by both EVALBOT wheel sensors. When the sensors are disabled, no notification of sensor pulses will be made to the *pfnCallback* function passed to WheelSensorsInit() for that wheel. The sensors may be reenabled by calling WheelSensorEnable().

Returns:

None.

8.2.2.6 WheelSensorEnable

Enables the LEDs for both EVALBOT wheel sensors.

Prototype:

```
void
WheelSensorEnable(void)
```

Description:

This function enables the LEDs used by both EVALBOT wheel sensors. When the sensors are enabled, notification of sensor pulses will be made to the *pfnCallback* function passed to WheelSensorsInit() for that wheel assuming sensor interrupts for a given wheel have also been enabled by a previous call to WheelSensorIntEnable(). The sensors may be disabled by calling WheelSensorDisable().

Returns:

None.

8.2.2.7 WheelSensorIntDisable

Disables the interrupts from an infrared wheel sensor.

Prototype:

```
void
WheelSensorIntDisable(tWheel eWheel)
```

Parameters:

eWheel defines which wheel sensor interrupt to disable. Valid values are *WHEEL_LEFT* and *WHEEL_RIGHT*.

Description:

This function disables the wheel sensor interrupt from one EVALBOT wheel. When a sensor interrupt is disabled, no further callbacks will be made to the *pfnCallback* function passed to WheelSensorsInit() for that wheel. Interrupts may be reenabled by calling WheelSensorIntEnable().

Returns:

None.

8.2.2.8 WheelSensorIntEnable

Enables the interrupts from an infrared wheel sensor.

Prototype:

```
void
WheelSensorIntEnable(tWheel eWheel)
```

Parameters:

eWheel defines which wheel sensor interrupt to enable. Valid values are *WHEEL_LEFT* and *WHEEL_RIGHT*.

Description:
This function enables the wheel sensor interrupt from one EVALBOT wheel. When a sensor interrupt is enabled, callbacks will be made to the *pfnCallback* function passed to WheelSensorsInit() for that wheel. Interrupts may be disabled by calling WheelSensorIntDisable().

Returns:
None.

8.2.2.9 WheelSensorIntHandler

Handles interrupts from each of the IR sensors used to determine speed.

Prototype:
```
void
WheelSensorIntHandler(void)
```

Description:
This interrupt function is called by the processor due to an interrupt on the rising edge signals from each of the wheel sensors and is used to call a callback function to the client application informing it that the wheel has moved. The application-supplied callback function will typically be used to calculate wheel rotation speed.

Applications using the motor driver must hook this function to the interrupt vectors for each GPIO port containing a wheel sensor pin. For the existing EVALBOT hardware, this is GPIO port E.

Returns:
None.

Note:
This function is called by the interrupt system and should not be called directly from application code.

8.2.2.10 WheelSensorsInit

Initializes the infrared wheel sensors.

Prototype:
```
void
WheelSensorsInit(void (*group__sensors__api_g6c9528f10363ca78a32ded0f7d72681e)(W
eWheel) pfnCallback)
```

Parameters:
pfnCallback is a pointer to the function which will be called on each pulse from the wheel sensors. It may be null to disable callbacks.

Description:
This function must be called to initialize the infrared sensors used to calculate actual EVALBOT speed. If a non-NULL function pointer is provided in the *pfnCallback* parameters, this function will be called on each pulse from a wheel sensor when they are enabled. Note that this callback is made in interrupt context.

Returns:
None.

8.3 Programming Example

The following example shows how to use the sensors driver to read the state of the bumper switches, and to install a callback function for wheel rotation.

```
//
// Initialize the bumper sensors
//
BumpSensorsInit();

//
// Read the state of one of the bump sensors
//
bStatus = BumpSensorGetStatus(BUMP_RIGHT);

//
// Initialize the wheel sensors and provide a callback for wheel "clicks"
//
WheelSensorsInit(MyWheelClickHandler);

//
// Enable the wheel sensors for operation, and interrupt generation
//
WheelSensorEnable();
WheelSensorIntEnable(RIGHT_SIDE);
```

9 Sound Driver

9.1 Introduction

The sound driver provides functions for playing PCM sounds clips using the EVALBOT audio DAC. The application can use this driver to play clips and control the volume. This driver is used by the Wave driver and does not need to be called directly from the application if the Wave driver is used.

Note that the sound driver makes use of the uDMA controller to transfer audio samples from a buffer to the I2S peripheral. Therefore, the application or client must allocate space for the uDMA control table, like this:

```
tDMAControlTable sDMAControlTable[64]; // must be 1024-aligned
```

The uDMA control table must be aligned on a 1024 byte boundary. The method for doing this is toolchain dependent. Please see the *sound_demo* application for an example of how to use the sound driver.

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

9.2 API Functions

Functions

- void SoundBufferPlay (const void *pvData, unsigned long ulLength, void (*pfnCallback)(void *pvBuffer, unsigned long ulEvent))
- void SoundClassDDis (void)
- void SoundClassDEn (void)
- void SoundInit (void)
- void SoundIntHandler (void)
- unsigned long SoundSampleRateGet (void)
- void SoundSetFormat (unsigned long ulSampleRate, unsigned short usBitsPerSample, unsigned short usChannels)
- void SoundVolumeDown (unsigned long ulPercent)
- unsigned char SoundVolumeGet (void)
- void SoundVolumeSet (unsigned long ulPercent)
- void SoundVolumeUp (unsigned long ulPercent)

9.2.1 Function Documentation

9.2.1.1 SoundBufferPlay

Starts playback of a block of PCM audio samples.

Prototype:

```
void
SoundBufferPlay(const void *pvData,
                unsigned long ulLength,
                void (**)(void pvBuffer, unsigned long ulEvent)
pfnCallback)
```

Parameters:

pvData is a pointer to the audio data to play.

ulLength is the length of the data in bytes.

pfnCallback is a function to call when this buffer has been played.

Description:

This function starts the playback of a block of PCM audio samples. If playback of another buffer is currently ongoing, its playback is canceled and the buffer starts playing immediately.

Returns:

None.

9.2.1.2 SoundClassDDis

Disables the class D amplifier in the DAC.

Prototype:

```
void
SoundClassDDis(void)
```

Description:

This function disables the class D amplifier in the DAC. It is merely a wrapper around the DAC driver's DACClassDDis() function.

Returns:

None.

9.2.1.3 SoundClassDEn

Enables the class D amplifier in the DAC.

Prototype:

```
void
SoundClassDEn(void)
```

Description:

This function enables the class D amplifier in the DAC. It is merely a wrapper around the DAC driver's DACClassDEn() function.

Returns:
None.

9.2.1.4 SoundInit

Initialize the sound driver.

Prototype:
```
void
SoundInit(void)
```

Description:
This function initializes the audio hardware components of the EVALBOT, in preparation for playing sounds using the sound driver.

Returns:
None.

9.2.1.5 SoundIntHandler

Interrupt handler for the I2S sound driver.

Prototype:
```
void
SoundIntHandler(void)
```

Description:
This interrupt function is called by the processor due to an interrupt from the I2S peripheral, or due to the completion of an I2S/uDMA transfer. uDMA is used in ping-pong mode to keep sound buffer data flowing to the I2S audio output. As each buffer transfer is complete, the client callback function that was specified in the call to SoundBufferPlay() will be called. The client can then take action to start the next buffer playing.

Applications using the sound driver must hook this function to the interrupt vectors for the I2S0 peripheral.

Returns:
None.

Note:
This function is called by the interrupt system and should not be called directly from application code.

9.2.1.6 SoundSampleRateGet

Returns the current audio sample rate setting.

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

Description:
This function returns the sample rate that is currently set. The value returned reflects the actual rate set which may be slightly different from the value passed on the last call to SoundSetFormat() if the requested rate could not be matched exactly.

Returns:
Returns the sample rate in samples per second.

9.2.1.7 SoundSetFormat

Configures the I2S peripheral to play audio in a given format.

Prototype:
```
void
SoundSetFormat(unsigned long ulSampleRate,
               unsigned short usBitsPerSample,
               unsigned short usChannels)
```

Parameters:
ulSampleRate is the sample rate of the audio to be played in samples per second.
usBitsPerSample is the number of bits in each audio sample.
usChannels is the number of audio channels, 1 for mono, 2 for stereo.

Description:
This function configures the I2S peripheral in preparation for playing or recording audio data in a particular format.

Returns:
None.

9.2.1.8 SoundVolumeDown

Adjusts the audio volume downwards by a given amount.

Prototype:
```
void
SoundVolumeDown(unsigned long ulPercent)
```

Parameters:
ulPercent is the amount to decrease the volume, specified as a percentage relative to the full volume.

Description:
This function adjusts the audio output downwards by the specified percentage. The adjusted volume will not go below 0% (silence).

Returns:
None.

9.2.1.9 SoundVolumeGet

Returns the current sound volume.

Prototype:

```
unsigned char
SoundVolumeGet(void)
```

Description:

This function returns the current volume, specified as a percentage between 0% (silence) and 100% (full volume), inclusive.

Returns:

Returns the current volume setting expressed as a percentage.

9.2.1.10 SoundVolumeSet

Sets the audio volume to a given level.

Prototype:

```
void
SoundVolumeSet(unsigned long ulPercent)
```

Parameters:

ulPercent is the volume level to set. Valid values are between 0 and 100 inclusive.

Description:

This function sets the audio output volume. The supplied level is expressed as a percentage between 0% (silence) and 100% (full volume) inclusive.

Returns:

None.

9.2.1.11 SoundVolumeUp

Adjusts the audio volume upwards by a given amount.

Prototype:

```
void
SoundVolumeUp(unsigned long ulPercent)
```

Parameters:

ulPercent is the amount to increase the volume, specified as a percentage relative to the full volume.

Description:

This function adjusts the audio output upwards by the specified percentage. The adjusted volume will not go above 100% (full volume).

Returns:

None.

9.3 Programming Example

The following example shows how to use the sound driver to set the volume and play a clip.

```
//
// Initialize the sound driver
//
SoundInit();

//
// Set the sound clip format to 32kHz, 8-bit, mono
//
SoundSetFormat(32000, 8, 1);

//
// Set the sound clip volume to 50%
//
SoundVolumeSet(50);

//
// Play a clip from a buffer.  The application provides a callback
// function that will be called from the sound driver interrupt
// handler and can be used to provide continuous playback.
// The clip length is limited to 1024.
//
SoundBufferPlay(pClipBuffer, 512, MySoundClipCallback);
```

10 Wave Driver

10.1 Introduction

The Wave driver provides a way to play audio files in Wave format using the EVALBOT audio system. Functions are provided to parse wave file headers, and to play sound clips using an easy interface. The driver uses interrupts and uDMA to keep the sound playing while the application performs other processing. The application only needs to periodically call the WavePlayContinue() function to keep the Wave driver running.

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

10.2 API Functions

Data Structures

- tWaveHeader

Enumerations

- tWaveReturnCode

Functions

- void WaveGetTime (tWaveHeader *pWaveHeader, char *pcTime, unsigned long ulSize)
- tWaveReturnCode WaveOpen (unsigned long *pulAddress, tWaveHeader *pWaveHeader)
- tBoolean WavePlaybackStatus (void)
- tBoolean WavePlayContinue (tWaveHeader *pWaveHeader)
- void WavePlayStart (tWaveHeader *pWaveHeader)
- void WaveStop (void)

10.2.1 Data Structure Documentation

10.2.1.1 tWaveHeader

Definition:

```
typedef struct
{
```

```
    unsigned long ulSampleRate;
    unsigned long ulAvgByteRate;
    unsigned long ulDataSize;
    unsigned short usBitsPerSample;
    unsigned short usFormat;
    unsigned short usNumChannels;
}
tWaveHeader
```

Members:

ulSampleRate Sample rate in bytes per second.

ulAvgByteRate The average byte rate for the wav file.

ulDataSize The size of the wav data in the file.

usBitsPerSample The number of bits per sample.

usFormat The wav file format.

usNumChannels The number of audio channels.

Description:

The header information parsed from a ".wav" file during a call to the function WaveOpen.

10.2.2 Enumeration Documentation

10.2.2.1 tWaveReturnCode

Description:

Possible return codes from the WaveOpen function.

Enumerators:

WAVE_OK The wav data was parsed successfully.

WAVE_INVALID_RIFF The RIFF information in the wav data is not supported.

WAVE_INVALID_CHUNK The chunk size specified in the wav data is not supported.

WAVE_INVALID_FORMAT The format of the wav data is not supported.

10.2.3 Function Documentation

10.2.3.1 WaveGetTime

Formats a text string showing elapsed and total playing time.

Prototype:

```
void
WaveGetTime(tWaveHeader *pWaveHeader,
            char *pcTime,
            unsigned long ulSize)
```

Parameters:

pWaveHeader is a pointer to the current wave file's header information. This structure will have been populated by a previous call to WaveOpen().

pcTime points to storage for the returned string.

ulSize is the size of the buffer pointed to by *pcTime*.

Description:
This function may be called periodically by an application during the time that a wave file is playing. It formats a text string containing the current playback time and the total length of the audio clip. The formatted string may be up to 12 bytes in length containing the terminating NULL so, to prevent truncation, *ulSize* must be 12 or larger.

Returns:
None.

10.2.3.2 WaveOpen

Opens a wav file and parses its header information.

Prototype:
```
tWaveReturnCode
WaveOpen(unsigned long *pulAddress,
         tWaveHeader *pWaveHeader)
```

Parameters:
pulAddress is a pointer to the start of the wav format data in memory.
pWaveHeader is a pointer to a caller-supplied tWaveHeader structure that will be populated by the function.

Description:
This function is used to parse a wav header and populate a caller-supplied header structure in preparation for playback. It can also be used to test if a clip is in wav format or not.

Returns:
WAVE_OK if the file was parsed successfully, **WAVE_INVALID_RIFF** if RIFF information is not supported, **WAVE_INVALID_CHUNK** if the chunk size is not supported, **WAVE_INVALID_FORMAT** if the format is not supported.

10.2.3.3 WavePlaybackStatus

Returns the current playback status of the wave file.

Prototype:
```
tBoolean
WavePlaybackStatus(void)
```

Description:
This function may be called to determine whether a wave file is currently playing.

Returns:
Returns **true** if a wave file us currently playing or **false** if no file is playing.

10.2.3.4 WavePlayContinue

Continues playback of a wave file previously passed to WavePlayStart().

Prototype:

```
tBoolean
WavePlayContinue(tWaveHeader *pWaveHeader)
```

Parameters:

pWaveHeader points to the structure containing information on the wave file being played.

Description:

This function must be called periodically (at least every 40mS) after WavePlayStart() to continue playback of an audio wav file. It does any necessary housekeeping to keep the DAC supplied with audio data and returns a value indicating when the playback has completed.

Returns:

Returns **true** when playback is complete or **false** if more audio data still remains to be played.

10.2.3.5 WavePlayStart

Initialize and start playing a wav file.

Prototype:

```
void
WavePlayStart(tWaveHeader *pWaveHeader)
```

Parameters:

pWaveHeader is the wave header structure containing the clip format information.

Description:

This function will prepare a wave audio clip for playing, using the wave format information passed in the structure pWaveHeader, which was previously populated by a call to WaveOpen(). Once this function is used to prepare the clip for playing, then WavePlayContinue() should be used to cause the clip to actually play on the audio output.

Returns:

None.

10.2.3.6 WaveStop

Change the playback state to stop the playback.

Prototype:

```
void
WaveStop(void)
```

Description:

This function changes the state of playback to “stopped” so that the audio clip will stop playing. It does not stop the clip immediately but instead changes an internal flag so that the audio clip will stop at the next buffer update. This allows this function to be called from the context of an interrupt handler.

Returns:
None.

10.3 Programming Example

The following example shows how to use the Wave driver to parse a wave audio clip and start playing.

```
//
// Open a wave file and parse the format.  A pointer to the beginning
// of the clip is provided, and the function will populate the caller-
// supplied wave header structure.
//
WaveOpen(pWaveBuffer, &MyWaveHeader);

//
// Start playing the wave clip
//
WavePlayStart(&MyWaveHeader);

//
// Periodically call the "continue" function to keep the wave playing
// This will return true when the clip is ended.
//
... (from a loop or periodic call)
WavePlayContinue(&MyWaveHeader);
```

11 Command Line Processing Module

11.1 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.

11.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

11.2.1 Data Structure Documentation

11.2.1.1 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.

11.2.2 Define Documentation

11.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

11.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

11.2.3 Function Documentation

11.2.3.1 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.

11.2.4 Variable Documentation

11.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

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

11.3 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));
}
```

12 CPU Usage Module

12.1 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.

12.2 API Functions

Functions

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

12.2.1 Function Documentation

12.2.1.1 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.

12.2.1.2 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.

12.3 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();
    }
}
```

13 CRC Module

13.1 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.

13.2 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)

13.2.1 Function Documentation

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 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

13.2.1.4 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.

13.2.1.5 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.

13.3 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);
```

14 Flash Parameter Block Module

14.1 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.

14.2 API Functions

Functions

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

14.2.1 Function Documentation

14.2.1.1 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.

14.2.1.2 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.

14.2.1.3 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.

14.3 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);
}
```

15 Integer Square Root Module

15.1 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.

15.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

15.2.1 Function Documentation

15.2.1.1 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.

15.3 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);
```

16 Ring Buffer Module

16.1 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.

16.2 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)

16.2.1 Function Documentation

16.2.1.1 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.

16.2.1.2 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.

16.2.1.3 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.

16.2.1.4 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.

16.2.1.5 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.

16.2.1.6 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.

16.2.1.7 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.

16.2.1.8 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.

16.2.1.9 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.

16.2.1.10 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.

16.2.1.11 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.

16.2.1.12 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.

16.2.1.13 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.

16.2.1.14 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.

16.2.1.15 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.

16.3 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);
```

17 Simple Task Scheduler Module

17.1 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.

17.2 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

17.2.1 Data Structure Documentation

17.2.1.1 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.

17.2.2 Function Documentation

17.2.2.1 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.

17.2.2.2 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.

17.2.2.3 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.

17.2.2.4 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.

17.2.2.5 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.

17.2.2.6 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.

17.2.2.7 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.

17.2.2.8 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.

17.2.3 Variable Documentation

17.2.3.1 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.

17.2.3.2 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.

17.3 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();
    }
}
```

18 Sine Calculation Module

18.1 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.

18.2 API Functions

Defines

- cosine(ulAngle)

Functions

- long sine (unsigned long ulAngle)

18.2.1 Define Documentation

18.2.1.1 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.

18.2.2 Function Documentation

18.2.2.1 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.

18.3 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);
}
```

19 Micro Standard Library Module

19.1 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.

19.2 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)

19.2.1 Data Structure Documentation

19.2.1.1 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.

19.2.2 Function Documentation

19.2.2.1 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.

19.2.2.2 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).

19.2.2.3 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.

19.2.2.4 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 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.

19.2.2.5 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 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.

19.2.2.6 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

19.2.2.7 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*.

19.2.2.8 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*.

19.2.2.9 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*.

19.2.2.10 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*.

19.2.2.11 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*.

19.2.2.12 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*.

19.2.2.13 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.

19.2.2.14 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.

19.2.2.15 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 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.

19.3 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);
```

20 UART Standard IO Module

20.1 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.

20.1.1 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.

20.1.2 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.

20.2 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)

20.2.1 Function Documentation

20.2.1.1 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.

20.2.1.2 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.

20.2.1.3 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.

20.2.1.4 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.

20.2.1.5 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.

20.2.1.6 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.

20.2.1.7 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.

20.2.1.8 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.

20.2.1.9 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.

20.2.1.10 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.

20.2.1.11 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.

20.2.1.12 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.

20.2.1.13 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.

20.3 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