EK-LM3S3748 Firmware Development Package

USER'S GUIDE

SW-EK-LM3S3748-UG-6075

Copyright

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

Revision Information

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

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® EK-LM3S3748 evaluation board is a platform that can be used for software development and to prototype a hardware design. It contains a Stellaris ARM® Cortex™-M3-based microcontroller, along with an color STN display, a navigation switch, a small speaker, and USB host and device ports (shared via a USB mux) that can be used to exercise the peripherals on the microcontroller. Additionally, all of the microcontroller's pins are brought to unpopulated stake headers, allowing for easy connection to other hardware for the purposes of prototyping (after the stake headers have been populated by the customer).

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

2 Example Applications

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

There is an IAR workspace file (`ek-lm3s3748.eww`) that contains the peripheral driver library project, graphics library project, USB library project, and 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-lm3s3748.mpw`) that contains the peripheral driver library project, graphics library project, USB library project, and 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-lm3s3748` subdirectory of the firmware development package source distribution.

2.1 AES Pre-expanded Key (aes_expanded_key)

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

2.2 AES Normal Key (aes_set_key)

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

2.3 Audio Playback (audio)

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

2.4 Bit-Banding (bitband)

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

2.5 Blinky (blinky)

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

2.6 Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of the ROM-based boot loader. At startup, the application will configure the UART and branch to the boot loader to await the start of an update. The UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

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

2.7 Boot Loader Demo 2 (boot_demo2)

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

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

2.8 USB Boot Loader (boot_usb)

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

Applications intended for use with this version of the boot loader should be linked to run from address 0x1800 in flash (rather than the default run address of 0). This address is chosen to ensure that boot loader images built with all supported compilers may be used without modifying the application start address. Depending upon the compiler and optimization level you are using, however, you may find that you can reclaim some space by lowering this address and rebuilding both the application and boot loader. To do this, modify the makefile or project you use to build the application to show the new run address and also change the APP_START_ADDRESS value defined in bl_config.h before rebuilding the boot loader.

The USB boot loader may be demonstrated using the usb_boot_demo1 and usb_boot_demo2 example applications in addition to the boot_usb boot loader binary itself. Note that these are the only two example applications currently configured to run alongside the USB boot loader but making any of the other applications boot loader compatible is simply a matter of relinking them with the new start address.

The boot loader configuration used here enables a GPIO check on boot. If the "Navigate" button

on the EK board is pressed down when power is applied, the USB boot loader will run and prepare to receive download or upload commands from the USB host. If the button is not pressed, control will immediately transfer to the main application if one is present at 0x1800.

The Windows device driver required to communicate with the USB boot loader can be found on the software and documentation CD from the evaluation kit package. It can also be found in the Windows driver package which can be downloaded from `http://www.luminarymicro.com/products/software_updates.html`.

A Windows command-line application, dfuprog, is also provided which illustrates how to perform uploads and downloads via the USB DFU protocol. The source for this application can be found in the "windows" subdirectory for your target board and the prebuilt executable is available in the package "Windows-side examples for USB kits" available for download from `http://www.luminarymicro.com/products/software_updates.html`.

2.9 GPIO JTAG Recovery (gpio_jtag)

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

In this example, four pins (PC0, PC1, PC2, and PC3) are switched.

2.10 Graphics Library Demonstration (grlib_demo)

This application provides a demonstration of the capabilities of the Stellaris Graphics Library. The display will be configured to demonstrate the available drawing primitives: lines, circles, rectangles, strings, and images.

2.11 Hello World (hello)

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

2.12 Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M3 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the display; GPIO pins D0, D1 and D2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the

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

2.13 MPU (mpu_fault)

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

2.14 PWM (pwmgen)

This example application utilizes the PWM peripheral to output a 20% duty cycle PWM signal and a 80% duty cycle PWM signal, both at 8000 Hz. Once configured, the application enters an infinite loop, doing nothing while the PWM peripheral continues to output its signals.

2.15 Quickstart Oscilloscope (qs-scope)

A two channel oscilloscope implemented using the Stellaris microcontroller's analog-to-digital converter (ADC). The oscilloscope supports sample rates of up to 1M sample per second and will show the captured waveforms on the color STN display. On-screen menus provide user control over timebase, channel voltage scale and position, trigger type, trigger level and trigger position. Other features include the ability to save captured data as comma-separated-value files suitable for use with spreadsheet applications or bitmap images on either an installed microSD card or a USB flash drive. The board may also be connected to a WindowsXP or Vista host system and controlled remotely using a Windows application.

Oscilloscope User Interface

All oscilloscope controls and settings are accessed using the navigation control on the board. This control offers up, down, left, right and select functions in a single unit. Rocking the control in the desired direction sends "up", "down", "left" or "right" messages to the application and pressing on the center sends a "select" message.

Controls and settings are arranged into groups by function such as display settings, trigger settings, file operations and setup choices. These groups are accessed by pressing "select" to display the main menu. With the menu displayed, use "up" and "down" to select between the available groups. When the desired group is highlighted, press "select" once again to dismiss the menu.

Controls from the currently selected group are shown in the bottom portion of the application display. Use "up" and "down" to cycle through the controls in the group and "left" and "right" to change the value of, or select the action associated with the control which is currently displayed.

The control groups and the individual controls offered by each are outlined below:

```
Group    Control       Setting
-----    -------       -------

Display
         Channel 2     ON or OFF.
         Timebase      Select values from 2uS to 50mS per division.
```

```
        Ch1 Scale     Select values from 100mV to 10V per division.
        Ch2 Scale     Select values from 100mV to 10V per division.
        Ch1 Offset    Press and hold "left" or "right" to move the
                      waveform up or down in 100mV increments.
        Ch2 Offset    Press and hold "left" or "right" to move the
                      waveform up or down in 100mV increments.
Trigger
        Trigger       The trigger type - Always, Rising, Falling or
                      Level.
        Trig Channel  1 or 2 to select the channel used for triggering.
        Trig Level    Press and hold "left" or "right" to change the
                      trigger level in 100mV increments.
        Trig Pos      Press and hold "left" or "right" to move the
                      trigger position on the display.
        Mode          Running or Stopped.
        One Shot      If the current mode is "Stopped", pressing
                      "left" or "right" initiates capture and display
                      of a single waveform.
Setup
        Captions      Select ON to show the timebase and scale
                      captions or OFF to remove them from the display.
        Voltages      Select ON to show the measured voltages for each
                      channel or OFF to remove them from the display.
        Grid          Select ON to show the graticule lines or OFF to
                      remove them from the display.
        Ground        Select ON to show dotted lines corresponding
                      to the ground levels for each channel or OFF
                      to remove them from the display.
        Trig Level    Select ON to show a solid horizontal line
                      corresponding to the trigger level for the
                      trigger channel or OFF to remove this line from
                      the display.
        Trig Pos      Select ON to show a solid vertical line at the
                      trigger position or OFF to remove this line from
                      the display.
        Clicks        Select ON to enable sounds on button presses or
                      OFF to disable them.
        USB Mode      Select Host to operate in USB host mode and
                      allow use of flash memory sticks, or Device to
                      operate as a USB device and allow connection to
                      a host PC system.
File
        CSV on SD     Save the current waveform data as a text file
                      on an installed microSD card.
        CSV on USB    Save the current waveform data as a text file
                      on an installed USB flash stick (if in USB host
                      mode - see the Setup group above).
        BMP on SD     Save the current waveform display as a bitmap
                      on an installed microSD card.
        BMP on USB    Save the current waveform display as a bitmap
                      on an installed USB flash stick (if in USB host
                      mode - see the Setup group above).
Help
        Help          Pressing "left" or "right" will show or hide the
                      screen showing oscilloscope connection help.
        Channel 1     Pressing "left" or "right" will cause the scale
                      and position for the channel 1 waveform to be set
                      such that the waveform is visible on the display.
        Channel 2     Pressing "left" or "right" will cause the scale
                      and position for the channel 2 waveform to be set
                      such that the waveform is visible on the display.
```

Oscilloscope Connections

The 8 pins immediately above the color STN display panel offer connections for both channels of

the oscilloscope and also two test signals that can be used to provide input in the absence of any other suitable signals. Each channel input must lie in the range -16.5V to +16.5V relative to the board ground allowing differences of up to 33V to be measured.

The connections are as follow where pin 1 is the leftmost pin, nearest the microSD card socket:

1	Test 1	A test signal connected to one side of the speaker on the board.
2	Channel 1+	This is the positive connection for channel 1 of the oscilloscope.
3	Channel 1-	This is the negative connection for channel 1 of the oscilloscope.
4	Ground	This is connected to board ground.
5	Test 2	A test signal connected to the board Status LED which is driven from PWM0. This signal is configured to provide a 1KHz square wave.
6	Channel 2+	This is the positive connection for channel 2 of the oscilloscope.
7	Channel 2-	This is the negative connection for channel 2 of the oscilloscope.
8	Ground	This is connected to board ground.

Triggering and Sample Rate Notes

The oscilloscope can sample at a maximum combined rate of 1M samples per second. When both channels are enabled, therefore, the maximum sample rate on each channel is 500K samples per second. For maximum resolution at the lowest timebases (maximum samples rates), disable channel 2 if it is not required. These sample rates give usable waveform capture for signals up to around 100KHz.

Trigger detection is performed in software during ADC interrupt handling. At the highest sampling rates, this interrupt service routine consumes almost all the available CPU cycles when searching for trigger conditions. At these sample rates, if a trigger level is set which does not correspond to a voltage that is ever seen in the trigger channel signal, the user interface response can become sluggish. To combat this, the oscilloscope will abort any pending waveform capture operation if a key is pressed before the capture cycle as completed. This prevents the user interface from being locked out and allows the trigger level or type to be changed to values more appropriate for the signal being measured.

File Operations

Comma-separated-value or bitmap files representing the last waveform captured may be saved to either a microSD card or a USB flash drive. In each case, the files are written to the root directory of the microSD card or flash drive with file names of the form “scopeXXX.csv” or “scopeXXX.bmp” where “XXX” represents the lowest, three digit, decimal number which offers a file name which does not already exist on the device.

Companion Application

A companion application, LMScope, which runs on WindowsXP and Vista PCs and the required device driver installer are available on the software CD and via download from the TI Stellaris web site at `http://focus.ti.com/mcu/docs/mcuorphan.tsp?contentId=87903`. This application offers full control of the oscilloscope from the PC and allows waveform display and save to local hard disk.

Note that the USB device drivers for the oscilloscope device must be installed prior to running the LMScope application. If the drivers are not installed when LMScope is run, the application will report that various required DLLs including lmusbdll.dll and winusb.dll are missing. To install the drivers, make sure that the USB mode is set to “Device” in the Setup menu then connect the

ek-lm3s3748 board to a PC via the “USB Device” connector. This will cause Windows to prompt for device driver installation. The required driver can be found on the kit installation CD or in the “SW-USB-windrivers-xxxx” package downloadable from the software update web site.

2.16 SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple command console via a serial port for issuing commands to view and navigate the file system on the SD card.

The first UART, which is connected to the FTDI virtual serial port on the evaluation board, is configured for 115,200 bits per second, and 8-N-1 mode. When the program is started a message will be printed to the terminal. Type “help” for command help.

For additional details about FatFs, see the following site: `http://elm-chan.org/fsw/ff/00index_e.html`

2.17 Timer (timers)

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

2.18 UART (uart_echo)

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

2.19 uDMA (udma_demo)

This example application demonstrates the use of the uDMA controller to transfer data between memory buffers, and to transfer data to and from a UART.

2.20 USB Boot Loader Demo 1 (usb_boot_demo1)

An example to demonstrate the use of the flash-based USB boot loader. At startup, the application displays a message then branches to the USB boot loader to await the start of an update. The boot loader presents a Device Firmware Upgrade interface to the host allowing new applications to be downloaded to flash via USB.

The usb_boot_demo2 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

The application dfuwrap, found in the boards directory, can be used to prepare binary images for download to a particular position in device flash. This application adds a Stellairs-specific prefix and a DFU standard suffix to the binary. A sample Windows command line application, dfuprog, is also provided which allows either binary images or DFU-wrapped files to be downloaded to the board or uploaded from it.

The usb_boot_demo1 and usb_boot_demo2 applications are essentially identical to boot_demo1 and boot_demo2 with the exception that they are linked to run at address 0x1800 rather than 0x0. This is due to the fact that the USB boot loader is not currently included in the Stellaris ROM and therefore has to be stored in the bottom few KB of flash with the main application stored above it.

2.21 USB Boot Loader Demo 2 (usb_boot_demo2)

An example to demonstrate the use of the flash-based USB boot loader. At startup, the application displays a message then waits for the user to press the select button before branching to the USB boot loader to await the start of an update. The boot loader presents a Device Firmware Upgrade interface to the host allowing new applications to be downloaded to flash via USB.

The usb_boot_demo1 application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

The application dfuwrap, found in the boards directory, can be used to prepare binary images for download to a particular position in device flash. This application adds a Stellaris-specific prefix and a DFU standard suffix to the binary. A sample Windows command line application, dfuprog, is also provided which allows either binary images or DFU-wrapped files to be downloaded to the board or uploaded from it.

The usb_boot_demo1 and usb_boot_demo2 applications are essentially identical to boot_demo1 and boot_demo2 with the exception that they are linked to run at address 0x1800 rather than 0x0. This is due to the fact that the USB boot loader is not currently included in the Stellaris ROM and therefore has to be stored in the bottom few KB of flash with the main application stored above it.

2.22 USB Generic Bulk Device (usb_dev_bulk)

This example provides a generic USB device offering simple bulk data transfer to and from the host. The device uses a vendor-specific class ID and supports a single bulk IN endpoint and a single bulk OUT endpoint. Data received from the host is assumed to be ASCII text and it is echoed back with the case of all alphabetic characters swapped.

A Windows INF file for the device is provided on the installation CD. This INF contains information required to install the WinUSB subsystem on WindowsXP and Vista PCs. WinUSB is a Windows subsystem allowing user mode applications to access the USB device without the need for a vendor-specific kernel mode driver. The device driver may also be downloaded from `http://www.luminarymicro.com/products/software_updates.html` as part of the “Stellaris embedded USB drivers” package (SW-USB-windrivers).

A sample Windows command-line application, usb_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided. The application binary is installed as part of

the “Windows-side examples for USB kits” package (SW-USB-win) on the installation CD or via download from `http://www.luminarymicro.com/products/software_updates.html` . Project files are included to allow the examples to be built using Microsoft VisualStudio. Source code for this application can be found in directory StellarisWare/tools/usb_bulk_example.

2.23 USB Composite Serial Device (usb_dev_cserial)

This example application turns the evaluation kit into a multiple virtual serial ports when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect UART0 traffic to and from the USB host system. The first virtual serial port will echo data to the physical UART0 port on the device which is connected to the virtual serial port on the FTDI device on this board. The physical UART0 will also echo onto the first virtual serial device provided by the Stellaris controller. The second Stellaris virtual serial port will provide a console that can echo data to both the FTDI virtual serial port and the first Stellaris virtual serial port. It will also allow turning on, off or toggling the boards led status. Typing a "?" and pressing return should echo a list of commands to the terminal, since this board can show up as possibly three individual virtual serial devices.

The usb_dev_cserial_win2k.inf may be used to install the example as a virtual COM port on a Windows2000 system. For WindowsXP or Vista, usb_dev_cserial.inf should be used.

2.24 USB HID Keyboard Device (usb_dev_keyboard)

This example application turns the evaluation board into a USB keyboard supporting the Human Interface Device class. The color STN display shows a virtual keyboard which can be navigated using the direction control button on the board. Pressing down on the button presses the highlighted key, sending its usage code and, if necessary, a shift modifier, to the USB host. The board status LED is used to indicate the current Caps Lock state and is updated in response to pressing the “Caps” key on the virtual keyboard or any other keyboard attached to the same USB host system.

The device implemented by this application also supports USB remote wakeup allowing it to request the host to reactivate a suspended bus. If the bus is suspended (as indicated on the application display), pressing the Select key will request a remote wakeup assuming the host has not specifically disabled such requests.

2.25 USB HID Mouse Device (usb_dev_mouse)

This example application turns the evaluation board into a USB mouse supporting the Human Interface Device class. Presses on the navigation control on the evaluation board are translated into mouse movement and button press messages in HID reports sent to the USB host allowing the evaluation board to control the mouse pointer on the host system.

2.26 USB MSC Device (usb_dev_msc)

This example application turns the evaluation board into a USB mass storage class device. The application will use the microSD card for the storage media for the mass storage device. The screen will display the current action occuring on the device ranging from disconnected, no media, reading, writing and idle.

2.27 USB Serial Device (usb_dev_serial)

This example application turns the evaluation kit into a virtual serial port when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect UART0 traffic to and from the USB host system. File usb_dev_serial_win2k.inf may be used to install the example as a virtual COM port on a Windows2000 system. For WindowsXP or Vista, usb_dev_serial.inf should be used.

2.28 USB HID Keyboard Host (usb_host_keyboard)

This example application demonstrates how to support a USB keyboard attached to the evaluation kit board. The display will show if a keyboard is currently connected and the current state of the Caps Lock key on the keyboard that is connected on the bottom status area of the screen. Pressing any keys on the keyboard will cause them to be printed on the screen and to be sent out the UART at 115200 baud with no parity, 8 bits and 1 stop bit. Any keyboard that supports the USB HID bios protocol should work with this demo application.

2.29 USB HID Mouse Host (usb_host_mouse)

This example application demonstrates how to support a USB mouse attached to the evaluation kit board. The display will show if a mouse is currently connected and the current state of the buttons on the on the bottom status area of the screen. The main drawing area will show a mouse cursor that can be moved around in the main area of the screen. If the left mouse button is held while moving the mouse, the cursor will draw on the screen. A side effect of the application not being able to read the current state of the screen is that the cursor will erase anything it moves over while the left mouse button is not pressed.

2.30 USB Mass Storage Class Host (usb_host_msc)

This example application demonstrates how to connect a USB mass storage class device to the evaluation kit. When a device is detected, the application displays the contents of the file system and allows browsing using the buttons.

2.31 USB Stick Update Demo (usb_stick_demo)

An example to demonstrate the use of the flash-based USB stick update program. This example is meant to be loaded into flash memory from a USB memory stick, using the USB stick update program (usb_stick_update), running on the microcontroller.

After this program is built, the binary file (usb_stick_demo.bin), should be renamed to the filename expected by usb_stick_update ("FIRMWARE.BIN" by default) and copied to the root directory of a USB memory stick. Then, when the memory stick is plugged into the eval board that is running the usb_stick_update program, this example program will be loaded into flash and then run on the microcontroller.

This program simply displays a message on the screen and prompts the user to press the select button. Once the button is pressed, control is passed back to the usb_stick_update program which is still is flash, and it will attempt to load another program from the memory stick. This shows how a user application can force a new firmware update from the memory stick.

2.32 USB Memory Stick Updater (usb_stick_update)

This example application behaves the same way as a boot loader. It resides at the beginning of flash, and will read a binary file from a USB memory stick and program it into another location in flash. Once the user application has been programmed into flash, this program will always start the user application until requested to load a new application.

When this application starts, if there is a user application already in flash (at **APP_START_ADDRESS**), then it will just run the user application. It will attempt to load a new application from a USB memory stick under the following conditions:

- no user application is present at **APP_START_ADDRESS**
- the user application has requested an update by transferring control to the updater
- the user holds down the eval board push button when the board is reset

When this application is attempting to perform an update, it will wait forever for a USB memory stick to be plugged in. Once a USB memory stick is found, it will search the root directory for a specific file name, which is *FIRMWARE.BIN* by default. This file must be a binary image of the program you want to load (the .bin file), linked to run from the correct address, at **APP_START_ADDRESS**.

The USB memory stick must be formatted as a FAT16 or FAT32 file system (the normal case), and the binary file must be located in the root directory. Other files can exist on the memory stick but they will be ignored.

2.33 Watchdog (watchdog)

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

3 Development System Utilities

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

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

AES Key Expansion Utility

Usage:

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

Description:

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

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

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

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

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

Arguments:

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

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

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

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

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

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

-v, --version show program version.

-h, --help display usage information.

The **--keysize** and **--key** arguments are mandatory. Only one each of **--data** or **--code**, and **--encrypt** or **--decrypt** should be used. If not specified otherwise then the default is **--data --encrypt**.
FILE is the name of the file that will be created containing the expanded key. This file will be in the form of a C header file and should be included in your application.

Example:

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

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

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

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

Audio Converter

Usage:

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

Description:

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

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

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

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

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

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

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

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

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

Arguments:

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

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

-h displays usage information.

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

-o FILENAME specifies the name of the output file.

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

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

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

Example:

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

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

USB DFU Programmer

Usage:

`dfuprog [OPTION]...`

Description:

Downloads images to a Texas Instruments Stellaris microcontroller running the USB Device Firmware Upgrade boot loader. Additionally, this utility may be used to read back the existing application image or a subsection of flash and store it either as raw binary data or as a DFU-downloadable image file.

The source code for this utility is contained in `tools/dfuprog`. The binary for this utility is installed as part of the "Windows-side examples for USB kits" package (SW-USB-win) shipped on the release CD and downloadable from http://www.luminarymicro.com/products/software_updates.html. A Microsoft Visual Studio project file is provided to allow the application to be built.

Arguments:

-e specifies the address of the binary.

-u specifies that an image is to be uploaded from the board into the target file. If absent, the file will be downloaded to the board.

-c specifies that a section of flash memory is to be cleared. The address and size of the block may be specified using the -a and -l parameters. If these are absent, the entire writable area of flash is erased.

-f FILE specifies the name of the file to download or, if -u is given, to upload.

-b specifies that an uploaded file is to be stored as raw binary data without the DFU file wrapper. This option is only valid if used alongside -u.

-d specifies that the VID and PID in the DFU file wrapper should be ignored for a download operation.

-s specifies that image verification should be skipped following a download operation.

-a ADDR specifies the address at which the binary file will be downloaded or from which an uploaded file will be read. If a download operation is taking place and the source file provided is DFU-wrapped, this parameter will be ignored.

-l SIZE specifies the number of bytes to be uploaded when used in conjunction with -i or the number of bytes of flash to erase if used in conjunction with -c.

-i NUM specifies the zero-based index of the USB DFU device to access if more than one is currently attached to the system. If absent, the first device found is used.

-x specifies that destination file for an upload operation should be overwritten without prompting if it already exists.

-w specifies that the utility should wait for the user to press a key before it exits.

-v displays verbose output during the requested operation.

-h displays this help information.

-? displays this help information.

Example:

The following example writes binary file program.bin to the device flash memory at address 0x1800:

```
dfuprog -f program.bin -a 0x1800
```

The following example writes DFU-wrapped file program.dfu to the flash memory of the second connected USB DFU device at the address found in the DFU file prefix:

```
dfuprog -i 1 -f program.dfu
```

The following example uploads (reads) the current application image into a DFU-formatted file appimage.dfu:

```
dfuprog -u -f appimage.dfu
```

USB DFU Wrapper

Usage:

```
dfuwrap [OPTION]...
```

Description:

Prepares binary images for download to a particular position in device flash via the USB device firmware upgrade protocol. A Stellaris-specific prefix and a DFU standard suffix are added to the binary.

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

Arguments:

-a ADDR specifies the address of the binary.

-c specifies that the validity of the DFU wrapper on the input file should be checked.

-d ID specifies the USB device ID to place into the DFU wrapper. If not specified, the default of 0x0000 will be used.

-e enables verbose output.

-f specifies that a DFU wrapper should be added to the file even if one already exists.

-h displays usage information.

-i FILE specifies the name of the input file.

-o FILE specifies the name of the output file. If not specified, the default of image.dfu will be used.

-p ID specifies the USB product ID to place into the DFU wrapper. If not specified, the default of 0x00ff will be used.

-q specifies that only error information should be output.

-r specifies that the DFU header should be removed from the input file.

-v ID specifies the USB vendor ID to place into the DFU wrapper. if not specified, the default of 0x1cbe will be used.

-x specifies that the output file should be overwritten without prompting.

Example:

The following example adds a DFU wrapper which will cause the image to be programmed to address 0x1800:

```
dfuwrap -i program.bin -o program.dfu -a 0x1800
```

FreeType Rasterizer

Usage:

```
ftrasterize [OPTION]... [INPUT FILE]
```

Description:

Uses the FreeType font rendering package to convert a font into the format that is recognized by the graphics library. Any font that is recognized by FreeType can be used, which includes TrueType®, OpenType®, PostScript® Type 1, and Windows® FNT fonts. A complete list of supported font formats can be found on the FreeType web site at `http://www.freetype.org`.

FreeType is used to render the glyphs of a font at a specific size in monochrome, using the result as the bitmap images for the font. These bitmaps are compressed and the results are written as a C source file that provides a tFont structure describing the font.

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

Arguments:

-b specifies that this is a bold font. This does not affect the rendering of the font, it only changes the name of the file and the name of the font structure that are produced.

-f FILENAME specifies the base name for this font, which is used to create the output file name and the name of the font structure. The default value is “font” if not specified.

-i specifies that this is an italic font. This does not affect the rendering of the font, it only changes the name of the file and the name of the font structure that are produced.

-m specifies that this is a monospaced font. This causes the glyphs to be horizontally centered in a box whose width is the width of the widest glyph. For best visual results, this option should only be used for font faces that are designed to be monospaced (such as Computer Modern TeleType).

-s SIZE specifies the size of this font, in points. The default value is 20 if not specified.

INPUT FILE specifies the name of the input font file.

Example:

The following example produces a 24-point font called test from test.ttf:

```
ftrasterize -f test -s 24 test.ttf
```

The result will be written to `fonttest24.c`, and will contain a structure called `g_sFontTest24` that describes the font.

USB DFU Library

Description:

LMDFU is a Windows dynamic link library offering a high level interface to the USB Device Firmware Upgrade functionality provided by the Stellaris USB boot loader (boot_usb). This DLL is used by the dfuprog utility and also by the LMFlash application to allow download and upload of application images to or from a Stellaris-based board via USB.

The source code for this DLL is contained in `tools/lmdfu`. The DLL binary is installed as part of the "Stellaris embedded USB drivers" package (SW-USB-windrivers) shipped on the release CD and downloadable from http://www.ti.com/software_updates.html. A Microsoft Visual Studio project file is provided to allow the application to be built.

GIMP Script For Texas Instruments Stellaris Button

Description:

This is a script-fu plugin for GIMP (`http://www.gimp.org`) that produces push button images that can be used by the push button widget. When installed into `${HOME}/.gimp-2.4/scripts`, this will be available under Xtns->Buttons->LMI Button. When run, a dialog will be displayed allowing the width and height of the button, the radius of the corners, the thickness of the 3D effect, the color of the button, and the pressed state of the button to be selected. Once the desired configuration is selected, pressing OK will create the push button image in a new GIMP image. The image should be saved as a raw PPM file so that it can be converted to a C array by `pnmtoc`.

This script is provided as a convenience to easily produce a particular push button appearance; the push button images can be of any desired appearance.

This script is located in `tools/lmi-button/lmi-button.scm`.

Oscilloscope Application

Description:

LMScope is a Windows front end for the qs-scope example shipped with ek-lm3s3748 kits. The application communicates with the board via USB and allows control of various oscilloscope parameters and display of captured data.

Detailed documentation for this example is provided in a help file which is installed alongside the example binary.

The source code for this application is contained in `tools/lmscope`. The binary is installed as part of the "Windows-side examples for USB kits" package (SW-EK-USB-win) shipped on the release CD and downloadable from http://www.luminarymicro.com/products/software_updates.html. A Microsoft Visual Studio project file is provided to allow the application to be built.

USB Dynamic Link Library

Description:

LMUSBDLL is a simple Windows dynamic link library offering low level packet read and write functions for some USB-connected Stellaris example applications. The DLL is written above the Microsoft WinUSB interface and is intended solely to ensure that various Windows-side example applications can be built without having to use WinUSB header files. These header files are not included in the Visual Studio tools and are only shipped in the Windows Device Driver Kit (DDK). By providing this simple mapping DLL which links to WinUSB, the user avoids the need for a multi-gigabyte download to build the examples.

The source code for this DLL is contained in `tools/lmdfu`. The DLL binary is installed as part of the "Stellaris embedded USB drivers" package (SW-USB-windrivers) shipped on the release CD and downloadable from http://www.ti.com/software_updates.html. A Microsoft Visual Studio project file is provided to allow the DLL to be built on a PC which has the Windows Device Driver Kit installed.

String Table Generator

Usage:

```
mkstringtable [INPUT FILE] [OUTPUT FILE]
```

Description:

Converts a comma separated file (.csv) to a table of strings that can be used by the Stellaris Graphics Library. The source .csv file has a simple fixed format that supports multiple strings in multiple languages. A .c and .h file will be created that can be compiled in with an application and used with the graphics library's string table handling functions. The strings will also be compressed in order to reduce the space required to store them.

The format of the input .csv file is simple and easily edited in any plain text editor or a spreadsheet editor capable of reading and editing a .csv file. The .csv file format has a header row where the first entry in the row can be any string as it is ignored. The remaining entries in the row must be one of the GrLang* language definitions defined by the graphics library in `grlib.h` or they must have a `#define` definition that is valid for the application as this text is used directly in the C output file that is produced. Adding additional languages only requires that the value is unique in the table and that the name used is defined by the application.

The strings are specified one per line in the .csv file. The first entry in any line is the value that is used as the actual text for the definition for the given string. The remaining entries should be the strings for each language specified in the header. Single words with no special characters do not require quotations, however any strings with a "," character must be quoted as the ","

character is the delimiter for each item in the line. If the string has a quote character “"” it must be preceded by another quote character.

The following is an example .csv file containing string in English (US), German, Spanish (SP), and Italian:

```
LanguageIDs,GrLangEnUS,GrLangDE,GrLangEsSP,GrLangIt
STR_CONFIG,Configuration,Konfigurieren,Configuracion,Configurazione
STR_INTRO,Introduction,Einfuhrung,Introduccion,Introduzione
STR_QUOTE,Introduction in ""English"","Einfuhrung, in Deutch",Prueba,Verifica
...
```

In this example, `STR_QUOTE` would result in the following strings in the various languages:

- GrLangEnUs – `Introduction in "English"`
- GrLangDE – `Einfuhrung, in Deutch`
- GrLangEsSP – `Prueba`
- GrLangIt – `Verifica`

The resulting .c file contains the string table that must be included with the application that is using the string table. While the contents of this .c file are readable, the string table itself may be unintelligible due to the compression used on the strings themselves. The .h file that is created has the definition for the string table as well as an enumerated type `enum SCOMP_STR_INDEX` that contains all of the string indexes that were present in the original .csv file.

The code that uses the string table produced by this utility must refer to the strings by their identifier in the original .csv file. In the example above, this means that the value `STR_CONFIG` would refer to the “Configuration” string in English (GrLangEnUS) or “Konfigurieren” in German (GrLangDE).

This utility is contained in `tools/bin`.

Arguments:

INPUT FILE specifies the input .csv file to use to create a string table.

OUTPUT FILE specifies the root name of the output files as `<OUTPUT FILE>.c` and `<OUTPUT FILE>.h`. The value is also used in the naming of the string table variable.

Example:

The following will create a string table in `str.c`, with prototypes in `str.h`, based on the input file `str.csv`:

```
mkstringtable str.csv str
```

In the produced `str.c`, there will be a string table in `g_pucTablestr`.

NetPNM Converter

Usage:

`pnmtoc [OPTION]... [INPUT FILE]`

Description:

Converts a NetPBM image file into the format that is recognized by the Stellaris Graphics Library. The input image must be in the raw PPM format (in other words, with the `P6` tag). The NetPBM image format can be produced using GIMP, NetPBM

(http://netpbm.sourceforge.net), ImageMagick (http://www.imagemagick.org), or numerous other open source and proprietary image manipulation packages.

The resulting C image array definition is written to standard output; this follows the convention of the NetPBM toolkit after which the application was modeled (both in behavior and naming). The output should be redirected into a file so that it can then be used by the application.

To take a JPEG and convert it for use by the graphics library (using GIMP; a similar technique would be used in other graphics programs):

1. Load the file (File->Open).
2. Convert the image to indexed mode (Image->Mode->Indexed). Select "Generate optimum palette" and select either 2, 16, or 256 as the maximum number of colors (for a 1 BPP, 4 BPP, or 8 BPP image respectively). If the image is already in indexed mode, it can be converted to RGB mode (Image->Mode->RGB) and then back to indexed mode.
3. Save the file as a PNM image (File->Save As). Select raw format when prompted.
4. Use pnmtoc to convert the PNM image into a C array.

This sequence will be the same for any source image type (GIF, BMP, TIFF, and so on); once loaded into GIMP, it will treat all image types equally. For some source images, such as a GIF which is naturally an indexed format with 256 colors, the second step could be skipped if an 8 BPP image is desired in the application.

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

Arguments:

-c specifies that the image should be compressed. Compression is bypassed if it would result in a larger C array.

Example:

The following will produce a compressed image in foo.c from foo.ppm:

```
pnmtoc -c foo.ppm > foo.c
```

This will result in an array called g_pucImage that contains the image data from foo.ppm.

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
```

USB Bulk Data Transfer Example

Description:

usb_bulk_example is a Windows command line application which communicates with the StellarisWare usb_dev_bulk example. The application finds the Stellaris device on the USB bus then, if found, prompts the user to enter strings which are sent to the application running on the Stellaris board. This application then inverts the case of the alphabetic characters in the string and returns the data back to the USB host where it is displayed.

The source code for this application is contained in `tools/usb_bulk_example`. The binary is installed as part of the "Windows-side examples for USB kits" package (SW-USB-win) shipped on the release CD and downloadable from http://www.luminarymicro.com/products/software_updates.html. A Microsoft Visual Studio project file is provided to allow the application to be built.

4 Buttons Driver

4.1 Introduction

The board has a navigation switch that has a button for up, down, left, right, and select. The button driver handles sampling those buttons, performing software debouncing, and implementing an auto-repeat feature.

Debouncing the buttons is performed via the periodic sampling of the button states. If a button is in the opposite state from its debounced state for three consecutive samples, the debounced state is toggled. By choosing an appropriate button sampling rate, any button bounce will appear as a single sampled state in the opposite state and will therefore not affect the debounced state.

The choice of button sampling rate is a trade-off between the quality of debouncing and the latency introduced by the debouncing of the buttons (in other words, the amount of time between the button being pressed and the recognition of the button press). If the sample rate is too high, button bounce may not be properly filtered, and an excessive amount of processor time will be dedicated to button debouncing. If the sample rate is too low, it will take too long to recognize a button press, which will be noticeable to the user. A sampling rate between 1 ms and 10 ms typically provide a good balance of these two factors.

After the buttons have been debounced, the amount of time they are held is tracked in order to provide an auto-repeat function (if desired). There are two parameters that affect the auto-repeat; *ucInitialTicks* and *ucRepeatTicks*. *ucInitialTicks* is the number of samples that the button must be pressed before auto-repeat commences. After auto-repeat has started, *ucRepeatTicks* is the number of samples between repeated “presses” of the button.

Macros are provided to simplify the detection of button press, release, and auto-repeat events.

This driver is located in `boards/ek-lm3s3748/drivers`, with `buttons.c` containing the source code and `buttons.h` containing the API definitions for use by applications.

4.2 API Functions

Functions

- void ButtonsInit (void)
- unsigned char ButtonsPoll (unsigned char *pucDelta, unsigned char *pucRepeat)
- void ButtonsSetAutoRepeat (unsigned char ucButtonIDs, unsigned char ucInitialTicks, unsigned char ucRepeatTicks)

4.2.1 Function Documentation

4.2.1.1 ButtonsInit

Initializes the GPIO pins used by the board pushbuttons.

Prototype:

```
void
ButtonsInit(void)
```

Description:

This function must be called during application initialization to configure the GPIO pins to which the pushbuttons are attached. It enables the port used by the buttons and configures each button GPIO as an input with a weak pull-up.

Returns:

None.

4.2.1.2 ButtonsPoll

Polls the current state of the buttons and determines which have changed.

Prototype:

```
unsigned char
ButtonsPoll(unsigned char *pucDelta,
            unsigned char *pucRepeat)
```

Parameters:

pucDelta points to a character that will be written to indicate which button states changed since the last time this function was called. This value is derived from the debounced state of the buttons.

pucRepeat points to a character that will be written to indicate which buttons are signaling an auto-repeat as a result of this call.

Description:

This function should be called periodically by the application to poll the pushbuttons. It determines both which buttons have changed state since the last call and also signals auto-repeats based on the button state and the number of ButtonsPoll() calls made since the button was last pressed.

Auto-repeats are signaled at an application-specified rate if a key has been held down for longer than an initial delay period. To ensure that auto-repeats are generated at the desired rate, the application should ensure that this function is called at a regular period since all auto-repeat timing is calculated in terms of calls to ButtonsPoll().

Returns:

Returns the current debounced state of the buttons where a 1 in the button ID's position indicates that the button is released and a 0 indicates that it is pressed.

4.2.1.3 ButtonsSetAutoRepeat

Sets the auto-repeat parameters for one or more buttons.

Prototype:

```
void
ButtonsSetAutoRepeat(unsigned char ucButtonIDs,
                     unsigned char ucInitialTicks,
                     unsigned char ucRepeatTicks)
```

Parameters:

ucButtonIDs is a bitmask containing the OR-ed IDs of the buttons whose auto-repeat parameters are to be set.

ucInitialTicks is the number of ticks (calls to ButtonsPoll()) before the first auto-repeat is reported for the key if it is pressed for an extended period.

ucRepeatTicks is the number of ticks that must elapse after the initial period (*ucInitialTicks*) has expired between each subsequent auto-repeat is reported for the key.

Description:

This function may be called to change the auto-repeat delay and repeat period for one or more keys. Auto-Repeat allows an application to be signaled periodically if any key is held down for an extended period of time. After an initial delay following the original button press, a repeat signal flag is generated at a period determined by *ucRepeatTicks* and the interval between calls to ButtonsPoll().

For example, to configure a button such that it starts auto-repeating 500mS after it is initially pressed and signals an auto-repeat every 100mS until it is released, and assuming that ButtonsPoll() is called every 50mS, the following parameters would be used:

```
ucInitialTicks = 10
```

```
ucRepeatTicks = 2
```

Returns:

None.

4.3 Programming Example

The following example shows how to use the button API.

```
unsigned char ucButtons, ucChanged, ucRepeat;

//
// Initialize the buttons.
//
ButtonsInit();

//
// Loop forever sampling the buttons.
//
while(1)
{
    //
    // Sleep for some amount of time.
    //
```

```
    SysCtlDelay(1000);

    //
    // Sample the buttons.
    //
    ucButtons = ButtonsPoll(&ucChanged, &ucRepeat);

    //
    // See if the select button was pressed.
    //
    if(BUTTON_PRESSED(SELECT_BUTTON, ucButtons, ucChanged))
    {
        //
        // Do something in response to the select button press.
        //
    }
}
```

5 Class-D Audio Driver

5.1 Introduction

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

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

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

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

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

This driver is located in `boards/ek-lm3s3748/drivers`, with `class-d.c` containing the source code and `class-d.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

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

5.2.1 Function Documentation

5.2.1.1 ClassDBusy

Determines if the Class-D audio driver is busy.

Prototype:

```
tBoolean
ClassDBusy(void)
```

Description:

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

Returns:

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

5.2.1.2 ClassDInit

Initializes the Class-D audio driver.

Prototype:

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

Parameters:

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

Description:

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

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

Note:

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

Returns:

None.

5.2.1.3 ClassDPlayADPCM

Plays a buffer of 8 KHz IMA ADPCM data.

Prototype:

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

Parameters:

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

ulLength is the number of bytes in the buffer.

Description:

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

Returns:

None.

5.2.1.4 ClassDPlayPCM

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

Prototype:

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

Parameters:

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

ulLength is the number of bytes in the buffer.

Description:

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

Returns:

None.

5.2.1.5 ClassDPWMHandler

Handles the PWM1 interrupt.

Prototype:

```
void
ClassDPWMHandler(void)
```

Description:

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

Returns:

None.

5.2.1.6 ClassDStop

Stops playback of the current audio stream.

Prototype:

```
void
ClassDStop(void)
```

Description:

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

Returns:

None.

5.2.1.7 ClassDVolumeDown

Decreases the volume of the audio playback.

Prototype:

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

Parameters:

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

Description:

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

Returns:

None.

5.2.1.8 ClassDVolumeSet

Sets the volume of the audio playback.

Prototype:

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

Parameters:

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

Description:

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

Returns:
None.

5.2.1.9 ClassDVolumeUp

Increases the volume of the audio playback.

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

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

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

Returns:
None.

5.3 Programming Example

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

```
unsigned char pucBuffer[256];

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

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

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

6 Display Driver

6.1 Introduction

In addition to providing the `tDisplay` structure required by the graphics library, the display driver also provides APIs for initializing the display, turning on the backlight, and turning off the backlight.

This driver is located in `boards/ek-lm3s3748/drivers`, with `formike128x128x16.c` containing the source code and `formike128x128x16.h` containing the API definitions for use by applications.

6.2 API Functions

Functions

- void Formike128x128x16BacklightOff (void)
- void Formike128x128x16BacklightOn (void)
- void Formike128x128x16Init (void)

Variables

- const tDisplay g_sFormike128x128x16

6.2.1 Function Documentation

6.2.1.1 Formike128x128x16BacklightOff

Turns off the backlight.

Prototype:

```
void
Formike128x128x16BacklightOff(void)
```

Description:

This function turns off the backlight on the display.

Returns:

None.

6.2.1.2 Formike128x128x16BacklightOn

Turns on the backlight.

Prototype:

```
void
Formike128x128x16BacklightOn(void)
```

Description:
This function turns on the backlight on the display.

Returns:
None.

6.2.1.3 Formike128x128x16Init

Initializes the display driver.

Prototype:

```
void
Formike128x128x16Init(void)
```

Description:
This function initializes the ST7637 display controller on the panel, preparing it to display data.

Returns:
None.

6.2.2 Variable Documentation

6.2.2.1 g_sFormike128x128x16

Definition:

```
const tDisplay g_sFormike128x128x16
```

Description:
The graphics library display structure that describes the driver for the Formike Electronic KWH015C04-F01 color STN panel with an ST7637 controller.

6.3 Programming Example

The following example shows how to initialize the display and prepare to draw on it using the graphics library.

```
tContext sContext;

//
// Initialize the display.
//
```

```
    Formike128x128x16Init();

    //
    // Turn on the backlight.
    //
    Formike128x128x16BacklightOn();

    //
    // Initialize a graphics library drawing context.
    //
    GrContextInit(&sContext, &g_sFormike128x128x16);
```

7 Command Line Processing Module

7.1 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.

7.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

7.2.1 Data Structure Documentation

7.2.1.1 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.

7.2.2 Define Documentation

7.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

7.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

7.2.3 Function Documentation

7.2.3.1 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.

7.2.4 Variable Documentation

7.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

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

7.3 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));
}
```

8 CPU Usage Module

8.1 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.

8.2 API Functions

Functions

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

8.2.1 Function Documentation

8.2.1.1 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.

8.2.1.2 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.

8.3 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();
    }
}
```

9 Flash Parameter Block Module

9.1 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.

9.2 API Functions

Functions

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

9.2.1 Function Documentation

9.2.1.1 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.

9.2.1.2 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.

9.2.1.3 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.

9.3 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);
}
```

10 Integer Square Root Module

10.1 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.

10.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

10.2.1 Function Documentation

10.2.1.1 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.

10.3 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);
```

11 Ring Buffer Module

11.1 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.

11.2 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)

11.2.1 Function Documentation

11.2.1.1 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.

11.2.1.2 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.

11.2.1.3 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.

11.2.1.4 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.

11.2.1.5 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.

11.2.1.6 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.

11.2.1.7 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.

11.2.1.8 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.

11.2.1.9 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.

11.2.1.10 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.

11.2.1.11 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.

11.2.1.12 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.

11.2.1.13 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.

11.2.1.14 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.

11.2.1.15 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.

11.3 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);
```

12 Sine Calculation Module

12.1 Introduction

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

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

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

12.2 API Functions

Functions

- long sine (unsigned long ulAngle)

12.2.1 Function Documentation

12.2.1.1 sine

Computes an approximation of the sine of the input angle.

Prototype:

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

Parameters:

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

Description:

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

Returns:

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

12.3 Programming Example

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

```
unsigned long ulValue;

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

13 Micro Standard Library Module

13.1 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.

13.2 API Functions

Data Structures

- tTime

Functions

- void ulocaltime (unsigned long ulTime, tTime *psTime)
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)
- int usprintf (char *pcBuf, const char *pcString,...)
- int ustrcasecmp (const char *pcStr1, const char *pcStr2)
- int ustrnicmp (const char *pcStr1, const char *pcStr2, int iCount)
- char * ustrstr (const char *pcHaystack, const char *pcNeedle)
- unsigned long ustrtoul (const char *pcStr, const char **ppcStrRet, int iBase)
- int uvsnprintf (char *pcBuf, unsigned long ulSize, const char *pcString, va_list vaArgP)

13.2.1 Data Structure Documentation

13.2.1.1 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.

13.2.2 Function Documentation

13.2.2.1 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.

13.2.2.2 usnprintf

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

Prototype:

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

Parameters:

pcBuf is the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

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

Description:

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

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

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

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

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

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

Returns:

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

13.2.2.3 usprintf

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

Prototype:

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

Parameters:

pcBuf is the buffer where the converted string is stored.

pcString is the format string.

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

Description:

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

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

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

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

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

Returns:

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

13.2.2.4 ustrcasecmp

Compares two strings without regard to case.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

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

Returns:

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

13.2.2.5 ustrnicmp

Compares two strings without regard to case.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

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

Returns:

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

13.2.2.6 ustrstr

Finds a substring within a string.

Prototype:

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

Parameters:

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

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

Description:

This function is very similar to the C library `strstr()` function. It scans a string for the first instance of a given substring and returns a pointer to that substring. If the substring cannot be found, a NULL pointer is returned.

Returns:

Returns a pointer to the first occurrence of *pcNeedle* within *pcHaystack* or NULL if no match is found.

13.2.2.7 ustrtoul

Converts a string into its numeric equivalent.

Prototype:

```
unsigned long
ustrtoul(const char *pcStr,
         const char **ppcStrRet,
         int iBase)
```

Parameters:

pcStr is a pointer to the string containing the integer.

ppcStrRet is a pointer that will be set to the first character past the integer in the string.

iBase is the radix to use for the conversion; can be zero to auto-select the radix or between 2 and 16 to explicitly specify the radix.

Description:

This function is very similar to the C library `strtoul()` function. It scans a string for the first token (that is, non-white space) and converts the value at that location in the string into an integer value.

Returns:

Returns the result of the conversion.

13.2.2.8 uvsnprintf

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

Prototype:

```
int
uvsnprintf(char *pcBuf,
           unsigned long ulSize,
           const char *pcString,
           va_list vaArgP)
```

Parameters:

pcBuf points to the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

vaArgP is the list of optional arguments, which depend on the contents of the format string.

Description:

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

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

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

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

The *ulSize* parameter limits the number of characters that will be stored in the buffer pointed to by *pcBuf* to prevent the possibility of a buffer overflow. The buffer size should be large enough to hold the expected converted output string, including the null termination character.

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

Returns:

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

13.3 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);
```

14 UART Standard IO Module

14.1 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.

14.1.1 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.

14.1.2 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.

14.2 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)

14.2.1 Function Documentation

14.2.1.1 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.

14.2.1.2 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.

14.2.1.3 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.

14.2.1.4 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.

14.2.1.5 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.

14.2.1.6 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.

14.2.1.7 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.

14.2.1.8 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.

14.2.1.9 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.

14.2.1.10 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.

14.2.1.11 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.

14.2.1.12 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.

14.2.1.13 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.

14.3 Programming Example

The following example shows how to use the UART standard IO module to write a string to the UART “console”.

```
//
// Configure the appropriate pins as UART pins; in this case, PA0/PA1 are
// used for UART0.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

//
// Initialize the UART standard IO module.
//
UARTStdioInit(0);

//
// Print a string.
//
UARTprintf("Hello world!\n");
```

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

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

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