EK-LM3S9B92 Firmware Development Package

USER'S GUIDE

TEXAS INSTRUMENTS

SW-EK-LM3S9B92-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-LM3S9B92 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, an Ethernet port, a USB OTG port, a push button, and a LED 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 example applications that are provided for this evaluation 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-lm3s9b92.eww`) that contains the peripheral driver 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-lm3s9b92.mpw`) that contains the peripheral driver 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-lm3s9b92` 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.3 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).

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.4 Blinky (blinky)

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

2.5 Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AutoIP will be used to obtain a link-local address. The address that is selected will be shown on the UART.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

For additional details on lwIP, refer to the lwIP web page at: `http://savannah.nongnu.org/projects/lwip/`

2.6 Ethernet with PTP (enet_ptpd)

This example application demonstrates the operation of the Stellaris Ethernet controller using the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AutoIP will be used to obtain a link-local address. The address that is selected will be output to the UART.

A default set of pages will be served up by an internal file system and the httpd server.

The IEEE 1588 (PTP) software has been enabled in this code to synchronize the internal clock to a network master clock source.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

For additional details on lwIP, refer to the lwIP web page at: `http://savannah.nongnu.org/projects/lwip/`

For additional details on the PTPd software, refer to the PTPd web page at: `http://ptpd.sourceforge.net`

2.7 Ethernet with uIP (enet_uip)

This example application demonstrates the operation of the Stellaris Ethernet controller using the uIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. A basic web site is served over the Ethernet port. The web site displays a few lines of text, and a counter that increments each time the page is sent.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

For additional details on uIP, refer to the uIP web page at: `http://www.sics.se/~adam/uip/`

2.8 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 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.9 Hello World (hello)

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

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.10 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 UART; GPIO pins B0, B1 and B2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.11 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.12 PWM (pwmgen)

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

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.13 Quick Start Game (qs-adventure)

This game consists of a Z-machine interpreter running a Z-code version of the classic Colossal Cave Adventure game originally created by William Crowther. The Ethernet interface provides a telnet server and the USB interface provides a CDC serial port. Either interface can be used to play the game, though not at the same time.

The LED on the evaluation board will be turned on when the game is being played; further connections will be refused since only one instance of the game can be played at a time. The push button on the evaluation board will restart the game from the beginning; this is equivalent to typing “restart” followed by “yes” in the game itself.

The virtual COM port provided by the ICDI board (which is connected to UART0 on the evaluation board) provides a simple status display. The most important piece of information provided is the IP address of the Ethernet interface, which is selected using AutoIP (which uses DHCP if it is present and a random link-local address otherwise).

The game is played by typing simple English sentences in order to direct the actions of the protagonist, with abbreviations being allowed. For example, “go west”, “west”, and “w” all perform the same action.

Three display modes are available; “verbose” (which displays the full description every time a location is visited), “brief” (which displays the full description the first time a location is visited and only the name every other time), and “superbrief” (which only displays the name). The default display mode is “brief”, and “look” can be used to get the full description at any time (regardless of the display mode).

For a history of the Colossal Cave Adventure game, its creation of the “interactive fiction” gaming genre, and game hints, an Internet search will turn up numerous web sites. A good starting place is `http://en.wikipedia.org/wiki/Colossal_Cave_Adventure`.

2.14 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.15 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.

2.16 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.17 uDMA with Timer (udma_timer)

This example application demonstrates the use of the timer to trigger periodic DMA transfers. A timer is configured for periodic operation. The uDMA controller channel is configured to perform a transfer when requested from the timer. For the purposes of this demonstration, the data that is transferred is the value of a separate free-running timer. However in a real application the data transferred could be to/from memory or a peripheral.

After a small number of transfers are performed, the captured timer values are compared to make sure the expected duration elapsed between transfers. The results are printed out.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.18 uDMA with Timer Edge Capture (udma_timer_ccp)

This example application demonstrates the use of the timer edge capture mode to trigger DMA transfers whenever there is an edge capture event. A timer is configured for edge capture mode using CCP1 on pin PD7. The uDMA controller is configured to transfer the captured edge value to a buffer until a certain number of edges have been captured. A PWM output is configured to generate a square wave on GPIO port bit PD0 to use as an input source for the CCP1 pin. In order to run the example, the pins PD0 and PD7 must be jumpered together.

After a certain number of edges have been captured, the DMA transfer will be complete. The example program will print out the captured values.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.19 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

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.

2.20 USB composite HID Mouse and CDC serial Device (usb_dev_chidcdc)

This example application turns the evaluation board into a composite USB mouse supporting the Human Interface Device class and a CDC serial device The mouse pointer will move in a square pattern for the duration of the time it is plugged in. The serial port is used as a command prompt to change the behavior of the board. By default the mouse will simply enumerate and not move. The serial port can then be opened and a command can be issued to start the mouse moving or stop it again.

The commands supported by the UART are the following:

? or help or h - Will display the help message.

led <on|off|toggle|activity> on - Turns on the LED. off - Turns off the LED toggle - Toggle the LED activity - Toggle the LED due to serial activity.

mouse <on|off> on - Starts the mouse moving in a square pattern. off - Stops the mouse moving.

2.21 USB HID Keyboard Device (usb_dev_keyboard)

This example application turns the evaluation board into a USB keyboard supporting the Human Interface Device class. When the push button is pressed, a sequence of key presses is simulated to type a string. Care should be taken to ensure that the active window can safely receive the text; enter is not pressed at any point so no actions are attempted by the host if a terminal window is used (for example). The status LED is used to indicate the current Caps Lock state and is updated in response to 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 push button will request a remote wakeup assuming the host has not specifically disabled such requests.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.22 USB HID Mouse Device (usb_dev_mouse)

This example application turns the evaluation board into a USB mouse supporting the Human Interface Device class. The mouse pointer will move in a square pattern for the duration of the time it is plugged in.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.23 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.

2.24 USB HID Keyboard Host (usb_host_keyboard)

This application demonstrates the handling of a USB keyboard attached to the evaluation kit. Once attached, text typed on the keyboard will appear on the UART. Any keyboard that supports the USB HID BIOS protocol is supported.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.25 USB HID Mouse Host (usb_host_mouse)

This application demonstrates the handling of a USB mouse attached to the evaluation kit. Once attached, the position of the mouse pointer and the state of the mouse buttons are output to the UART.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

2.26 USB Mass Storage Class Host (usb_host_msc)

This example application demonstrates reading a file system from a USB mass storage class device. It makes use of FatFs, a FAT file system driver. It provides a simple command console via the UART for issuing commands to view and navigate the file system on the mass storage device.

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`

2.27 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.

2.28 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.

2.29 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.

UART0, connected to the FTDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

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
```

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
```

Ethernet Flash Downloader

Usage:

`eflash [OPTION]... [INPUT FILE]`

Description:

Downloads a firmware image to a Stellaris board using an Ethernet connection to the Stellaris Boot Loader. This has the same capabilities as the Ethernet download portion of the Stellaris Flash Programmer.

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

Arguments:

--help displays usage information.

-h is an alias for `--help`.

--ip=IP specifies the IP address to be provided by the BOOTP server.

-i IP is an alias for `--ip`.

--mac=MAC specifies the MAC address

-m MAC is an alias for `--mac`.

--quiet specifies that only error information should be output.

--silent is an alias for `--quiet`.

--verbose specifies that verbose output should be output.

--version displays the version of the utility and exits.

INPUT FILE specifies the name of the firmware image file.

Example:

The following will download a firmware image to the board over Ethernet, where the target board has a MAC address of 00:11:22:33:44:55 and is given an IP address of 169.254.19.70:

```
eflash -m 00:11:22:33:44:55 -i 169.254.19.70 image.bin
```

Finder

Usage:

`finder`

Description:

This program locates Stellaris boards on the local network that are running an lwIP-based application that includes the locator service. It will display the IP address, MAC address, client address, and application description for each board that it finds. This is useful for easily finding the IP address that has been assigned to a board via DHCP or AutoIP without needing to display it from the application (which is difficult on boards that do not have a builtin display).

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

Example:

This utility can be run by clicking on the application in a filesystem browser or by invoking it from the command line as follows:

```
finder
```

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.

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.

Web Filesystem Generator

Usage:

`makefsfile [OPTION]...`

Description:

Generates a file system image for the lwIP web server. This is loosely based upon the `makefsdata` Perl script that is provided with lwIP, but does not require Perl and has several enhancements. The file system image is produced as a C source file that contains an image of all the files contained within a subtree of the development system's directory structure. This source file is then built into the application and served via HTTP by the lwIP web server.

By default, the file system image embeds the HTTP headers associated with each file in the file system image data itself. This is the default assumption of the lwIP web server implementation and is sensible if using an internal file system image containing a small number of files. If also serving files from a file system which does not embed the headers (for example the FAT file system on a microSD card) dynamic header generation must be used and internal file system images should be built using the `-h` option. In these cases, ensure that `DYNAMIC_HTTP_HEADERS` is also defined in the `lwipopts.h` file to correctly configure the web server.

The `-x` option allows an "exclude file" to be specified. This exclude file contains the names of files and directories within the input directory tree that are to be skipped in the conversion process. If this option is not present, a default set of file excludes is used. This list contains typical source code control metadata directory names (".svn" and "CVS") and system files such as "thumbs.db". To see the default exclude list, run the tool with the `-v` option and look in the output.

Each file or directory name in the exclude file must be on a separate line within the file. The exclude list must contain individual file or directory names and may not include partial paths. For example `images_old` or `.svn` would be acceptable but `images_old/.svn` would not.

In addition to generating multi-file images, the tool can also be used to dump a single file in the form of a C-style array of unsigned characters. This mode of operation is chosen using the `-f` command line option.

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

Arguments:

-b generates a position-independent binary image.

-f dumps a single file as a C-style hex character array.

-h excludes HTTP headers from files. By default, HTTP headers are added to each file in the output.

-i NAME specifies the name of the directory containing the files to be included in the image or the name of the single file to be dumped if -f is used.

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

-q enables quiet mode.

-r overwrites the the output file without prompting.

-v enables verbose output.

-x FILE specifies a file containing a list of filenames and directory names to be excluded from the generated image.

-? displays usage information.

Example:

The following will generate a file system image using all the files in the `html` directory and place the results into `fsdata.h`:

```
makefsfile -i html -o fsdata.h
```

Serial Flash Downloader

Usage:

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

Description:

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

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

Arguments:

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

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

-d disables auto-baud.

-h displays usage information.

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

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

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

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

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 Command Line Processing Module

4.1 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.

4.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

4.2.1 Data Structure Documentation

4.2.1.1 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.

4.2.2 Define Documentation

4.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

4.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

4.2.3 Function Documentation

4.2.3.1 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.

4.2.4 Variable Documentation

4.2.4.1 g_sCmdTable

Definition:

`tCmdLineEntry g_sCmdTable[]`

Description:

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

4.3 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));
}
```

5 CPU Usage Module

5.1 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.

5.2 API Functions

Functions

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

5.2.1 Function Documentation

5.2.1.1 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.

5.2.1.2 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.

5.3 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();
    }
}
```

6 Flash Parameter Block Module

6.1 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.

6.2 API Functions

Functions

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

6.2.1 Function Documentation

6.2.1.1 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.

6.2.1.2 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.

6.2.1.3 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.

6.3 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);
}
```

7 Integer Square Root Module

7.1 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.

7.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

7.2.1 Function Documentation

7.2.1.1 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.

7.3 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);
```

8 Ethernet Board Locator Module

8.1 Introduction

The locator module offers a simple way to add Ethernet board locator capability to an application which is using the lwIP TCP/IP stack. Applications running the locator service will be detected by the `finder` application which can be found in the `tools` directory of the StellarisWare installation.

APIs offered by the locator module allow an application to set various fields which are communicated to the `finder` application when it enumerates Stellaris boards on the network. These fields include an application-specified name, the MAC address of the board, the board ID and the client IP address.

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

8.2 API Functions

Functions

- void LocatorAppTitleSet (const char *pcAppTitle)
- void LocatorBoardIDSet (unsigned long ulID)
- void LocatorBoardTypeSet (unsigned long ulType)
- void LocatorClientIPSet (unsigned long ulIP)
- void LocatorInit (void)
- void LocatorMACAddrSet (unsigned char *pucMACArray)
- void LocatorVersionSet (unsigned long ulVersion)

8.2.1 Function Documentation

8.2.1.1 LocatorAppTitleSet

Sets the application title in the locator response packet.

Prototype:

```
void
LocatorAppTitleSet(const char *pcAppTitle)
```

Parameters:

pcAppTitle is a pointer to the application title string.

Description:
This function sets the application title in the locator response packet. The string is truncated at 64 characters if it is longer (without a terminating 0), and is zero-filled to 64 characters if it is shorter.

Returns:
None.

8.2.1.2 LocatorBoardIDSet

Sets the board ID in the locator response packet.

Prototype:
```
void
LocatorBoardIDSet(unsigned long ulID)
```

Parameters:
ulID is the ID of the board.

Description:
This function sets the board ID field in the locator response packet.

Returns:
None.

8.2.1.3 LocatorBoardTypeSet

Sets the board type in the locator response packet.

Prototype:
```
void
LocatorBoardTypeSet(unsigned long ulType)
```

Parameters:
ulType is the type of the board.

Description:
This function sets the board type field in the locator response packet.

Returns:
None.

8.2.1.4 LocatorClientIPSet

Sets the client IP address in the locator response packet.

Prototype:
```
void
LocatorClientIPSet(unsigned long ulIP)
```

Parameters:
ulIP is the IP address of the currently connected client.

Description:
This function sets the IP address of the currently connected client in the locator response packet. The IP should be set to 0.0.0.0 if there is no client connected. It should never be set for devices that do not have a strict one-to-one mapping of client to server (for example, a web server).

Returns:
None.

8.2.1.5 LocatorInit

Initializes the locator service.

Prototype:
```
void
LocatorInit(void)
```

Description:
This function prepares the locator service to handle device discovery requests. A UDP server is created and the locator response data is initialized to all empty.

Returns:
None.

8.2.1.6 LocatorMACAddrSet

Sets the MAC address in the locator response packet.

Prototype:
```
void
LocatorMACAddrSet(unsigned char *pucMACArray)
```

Parameters:
pucMACArray is the MAC address of the network interface.

Description:
This function sets the MAC address of the network interface in the locator response packet.

Returns:
None.

8.2.1.7 LocatorVersionSet

Sets the firmware version in the locator response packet.

Prototype:

```
void
LocatorVersionSet(unsigned long ulVersion)
```

Parameters:

ulVersion is the version number of the device firmware.

Description:

This function sets the version number of the device firmware in the locator response packet.

Returns:

None.

8.3 Programming Example

The following example shows how to set up the board locator service in an application which uses Ethernet and the lwIP TCP/IP stack.

```
//
// Initialize the lwIP TCP/IP stack.
//
lwIPInit(pucMACAddr, 0, 0, 0, IPADDR_USE_DHCP);

//
// Setup the device locator service.
//
LocatorInit();
LocatorMACAddrSet(pucMACAddr);
LocatorAppTitleSet("Your application name");
```

9 lwIP Wrapper Module

9.1 Introduction

The lwIP wrapper module provides a simple abstraction layer for the lwIP version 1.3.2 TCP/IP stack. The configuration of the TCP/IP stack is based on the options defined in the `lwipopts.h` file provided by the application.

The lwIPInit() function is used to initialize the lwIP TCP/IP stack. The lwIPEthernetIntHandler() is the interrupt handler function for use with the lwIP TCP/IP stack. This handler will process transmit and receive packets. If no RTOS is being used, the interrupt handler will also service the lwIP timers. The lwIPTimer() function is to be called periodically to support the TCP, ARP, DHCP and other timers used by the lwIP TCP/IP stack. If no RTOS is being used, this timer function will simply trigger an Ethernet interrupt to allow the interrupt handler to service the timers.

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

9.2 API Functions

Functions

- void lwIPEthernetIntHandler (void)
- void lwIPInit (const unsigned char *pucMAC, unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)
- unsigned long lwIPLocalGWAddrGet (void)
- unsigned long lwIPLocalIPAddrGet (void)
- void lwIPLocalMACGet (unsigned char *pucMAC)
- unsigned long lwIPLocalNetMaskGet (void)
- void lwIPNetworkConfigChange (unsigned long ulIPAddr, unsigned long ulNetMask, unsigned long ulGWAddr, unsigned long ulIPMode)

9.2.1 Function Documentation

9.2.1.1 lwIPEthernetIntHandler

Handles Ethernet interrupts for the lwIP TCP/IP stack.

Prototype:

```
void
lwIPEthernetIntHandler(void)
```

Description:

This function handles Ethernet interrupts for the lwIP TCP/IP stack. At the lowest level, all receive packets are placed into a packet queue for processing at a higher level. Also, the transmit packet queue is checked and packets are drained and transmitted through the Ethernet MAC as needed. If the system is configured without an RTOS, additional processing is performed at the interrupt level. The packet queues are processed by the lwIP TCP/IP code, and lwIP periodic timers are serviced (as needed).

Returns:

None.

9.2.1.2 lwIPInit

Initializes the lwIP TCP/IP stack.

Prototype:

```
void
lwIPInit(const unsigned char *pucMAC,
         unsigned long ulIPAddr,
         unsigned long ulNetMask,
         unsigned long ulGWAddr,
         unsigned long ulIPMode)
```

Parameters:

pucMAC is a pointer to a six byte array containing the MAC address to be used for the interface.

ulIPAddr is the IP address to be used (static).

ulNetMask is the network mask to be used (static).

ulGWAddr is the Gateway address to be used (static).

ulIPMode is the IP Address Mode. **IPADDR_USE_STATIC** will force static IP addressing to be used, **IPADDR_USE_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR_USE_AUTOIP** will force Link Local only.

Description:

This function performs initialization of the lwIP TCP/IP stack for the Stellaris Ethernet MAC, including DHCP and/or AutoIP, as configured.

Returns:

None.

9.2.1.3 lwIPLocalGWAddrGet

Returns the gateway address for this interface.

Prototype:

```
unsigned long
lwIPLocalGWAddrGet(void)
```

Description:

This function will read and return the currently assigned gateway address for the Stellaris Ethernet interface.

Returns:
the assigned gateway address for this interface.

9.2.1.4 lwIPLocalIPAddrGet

Returns the IP address for this interface.

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

Description:
This function will read and return the currently assigned IP address for the Stellaris Ethernet interface.

Returns:
Returns the assigned IP address for this interface.

9.2.1.5 lwIPLocalMACGet

Returns the local MAC/HW address for this interface.

Prototype:
```
void
lwIPLocalMACGet(unsigned char *pucMAC)
```

Parameters:
pucMAC is a pointer to an array of bytes used to store the MAC address.

Description:
This function will read the currently assigned MAC address into the array passed in *pucMAC*.

Returns:
None.

9.2.1.6 lwIPLocalNetMaskGet

Returns the network mask for this interface.

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

Description:
This function will read and return the currently assigned network mask for the Stellaris Ethernet interface.

Returns:
the assigned network mask for this interface.

9.2.1.7 lwIPNetworkConfigChange

Change the configuration of the lwIP network interface.

Prototype:

```
void
lwIPNetworkConfigChange(unsigned long ulIPAddr,
                        unsigned long ulNetMask,
                        unsigned long ulGWAddr,
                        unsigned long ulIPMode)
```

Parameters:

ulIPAddr is the new IP address to be used (static).

ulNetMask is the new network mask to be used (static).

ulGWAddr is the new Gateway address to be used (static).

ulIPMode is the IP Address Mode. **IPADDR_USE_STATIC** 0 will force static IP addressing to be used, **IPADDR_USE_DHCP** will force DHCP with fallback to Link Local (Auto IP), while **IPADDR_USE_AUTOIP** will force Link Local only.

Description:

This function will evaluate the new configuration data. If necessary, the interface will be brought down, reconfigured, and then brought back up with the new configuration.

Returns:

None.

9.3 Programming Example

The following example shows how to use the lwIP wrapper module to initialize the lwIP stack.

```
unsigned char pucMACArray[6];

//
// Fill in the MAC array and initialize the lwIP library using DHCP.
//
lwIPInit(pucMACArray, 0, 0, 0, IPADDR_USE_DHCP);

//
// Periodically call the lwIP timer tick.  In a real application, this
// would use a timer interrupt instead of an endless loop.
//
while(1)
{
    SysCtlDelay(1000);
    lwIPTimer(1);
}
```

10 PTPd Wrapper Module

10.1 Introduction

The PTPd wrapper module provides a simple way to include the open-source PTPd library in an application. Because the PTPd library has compile-time options that may vary from one application to the next, it is not practical to provide this library in object format. By including the `ptpdlib.c` module in your application's project and/or make file, the library can be included at compile-time with a single reference.

The PTPd library provides IEEE Precision Time Protocol (1588) ported to the Stellaris family of Ethernet-enabled devices. This port uses lwIP as the underlying TCP/IP stack. Refer to the `enet_ptpd` sample application for the EK-6965 and EK-8962 Evaluation Kits for additional details.

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

10.2 API Functions

10.3 Programming Example

```
//
// Clear out all of the run time options and protocol stack options.
//
memset(&g_sRtOpts, 0, sizeof(g_sRtOpts));
memset(&g_sPTPClock, 0, sizeof(g_sPTPClock));

//
// Initialize all PTPd Run Time and Clock Options.
// Note:  This code will be specific to your application
//
...

//
// Run the protocol engine for the first time to initialize the state
// machines.
//
protocol_first(&g_sRtOpts, &g_sPTPClock);

...

//
// Main Loop
//
while(1)
{
    ...
```

```
        //
        // Run the protocol engine for each pass through the main process loop.
        //
        protocol_loop(&g_sRtOpts, &g_sPTPClock);

        ...
    }
```

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 Ethernet Software Update Module

13.1 Introduction

The Ethernet software update module provides a convenient method of registering a callback which will be notified when a user attempts to initiate a firmware update over Ethernet using the LM Flash Programmer application. In addition to providing notification of an update request, the module also provides a function that can be called to initiate an update using the Ethernet boot loader.

To make use of this module, an application must include the lwIP TCP/IP stack and must be run on a system configured to use the Ethernet boot loader.

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

13.2 API Functions

Functions

- void SoftwareUpdateBegin (void)
- void SoftwareUpdateInit (tSoftwareUpdateRequested pfnCallback)

13.2.1 Function Documentation

13.2.1.1 SoftwareUpdateBegin

Passes control to the bootloader and initiates a remote software update over Ethernet.

Prototype:

```
void
SoftwareUpdateBegin(void)
```

Description:

This function passes control to the bootloader and initiates an update of the main application firmware image via BOOTP across Ethernet. This function may only be used on parts supporting Ethernet and in cases where the Ethernet boot loader is in use alongside the main application image. It must not be called in interrupt context.

Applications wishing to make use of this function must be built to operate with the bootloader. If this function is called on a system which does not include the bootloader, the results are unpredictable.

Note:
It is not safe to call this function from within the callback provided on the initial call to SoftwareUpdateInit(). The application must use the callback to signal a pending update (assuming the update is to be permitted) to some other code running in a non-interrupt context.

Returns:
Never returns.

13.2.1.2 SoftwareUpdateInit

Initializes the remote Ethernet software update notification feature.

Prototype:
```
void
SoftwareUpdateInit(tSoftwareUpdateRequested pfnCallback)
```

Parameters:
pfnCallback is a pointer to a function which will be called whenever a remote firmware update request is received. If the application wishes to allow the update to go ahead, it must call SoftwareUpdateBegin() from non-interrupt context after the callback is received. Note that the callback will most likely be made in interrupt context so it is not safe to call SoftwareUpdateBegin() from within the callback itself.

Description:
This function may be used on Ethernet-enabled parts to support remotely-signaled firmware updates over Ethernet. The LM Flash Programmer (LMFlash.exe) application sends a magic packet to UDP port 9 whenever the user requests an Ethernet-based firmware update. This packet consists of 6 bytes of 0xAA followed by the target MAC address repeated 4 times. This function starts listening on UDP port 9 and, if a magic packet matching the MAC address of this board is received, makes a call to the provided callback function to indicate that an update has been requested.

The callback function provided here will typically be called in the context of the lwIP Ethernet interrupt handler. It is not safe to call SoftwareUpdateBegin() in this context so the application should use the callback to signal code running in a non-interrupt context to perform the update if it is to be allowed.

UDP port 9 is chosen for this function since this is the well-known port associated with “discard” operation. In other words, any other system receiving the magic packet will simply ignore it. The actual magic packet used is modeled on Wake-On-LAN which uses a similar structure (6 bytes of 0xFF followed by 16 repetitions of the target MAC address). Some Wake-On-LAN implementations also use UDP port 9 for their signaling.

Note:
Applications using this function must initialize the lwIP stack prior to making this call and must ensure that the lwIPTimer() function is called periodically. lwIP UDP must be enabled in lwipopts.h to ensure that the magic packets can be received.

Returns:
None.

13.3 Programming Example

The following example shows how to use the software update module.

```
//*****************************************************************************
//
// A flag used to indicate that an Ethernet remote firmware update request
// has been received.
//
//*****************************************************************************
volatile tBoolean g_bFirmwareUpdate = false;

//*****************************************************************************
//
// This function is called by the software update module whenever a remote
// host requests to update the firmware on this board.  We set a flag that
// will cause the bootloader to be entered the next time the user enters a
// command on the console.
//
//*****************************************************************************
void
SoftwareUpdateRequestCallback(void)
{
    g_bFirmwareUpdate = true;
}

//*****************************************************************************
//
// The main entry point for the application.  This function contains all
// hardware initialization code and also the main loop for the application.
//
//*****************************************************************************
int
main(void)
{
    unsigned char pucMACAddr[6];

    //
    // System clock initialization and reading of the MAC address into array
    // pucMACAddr occurs here.  This code is omitted for clarity.
    //

    //
    // Initialize the lwIP TCP/IP stack.
    //
    lwIPInit(pucMACAddr, 0, 0, 0, IPADDR_USE_DHCP);

    //
    // Start the remote software update module.
    //
    SoftwareUpdateInit(SoftwareUpdateRequestCallback);

    //
    // Do whatever other setup things the application needs.
    //

    //
    // Loop until someone requests a remote firmware update.
    //
    while(!g_bFirmwareUpdate)
    {
        //
        // Perform your main loop functions here.
        //
```

```
        }

        //
        // If we drop out, a remote firmware update request has been received.
        // Transfer control to the bootloader which will perform the update.
        //
        SoftwareUpdateBegin();
    }
```

14 TFTP Server Module

14.1 Introduction

The TFTP (tiny file transfer protocol) server module provides a simple way of transfering files to and from a system over an Ethernet connection. The general-purpose server module implements all the basic TFTP protocol and interacts with applications via a number of application-provided callback functions which are called when:

- A new file transfer request is received from a client.
- Another block of file data is required to satisfy an ongoing GET (read) request.
- A new block of data is received during an ongoing PUT (write) request.
- A file transfer has completed.

To make use of this module, an application must include the lwIP TCP/IP stack with UDP enabled in the `lwipopts.h` header file.

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

14.2 Usage

The TFTP server module handles the TFTP protocol on behalf of an application but the application using it is responsible for all file system interaction - reading and writing files in response to callbacks from the TFTP server. To make use of the module, an application must provide the following callback functions to the server.

pfnRequest (type `tTFTPRequest`) This function pointer is provided to the server as a parameter to the `TFTPInit()` function. It will be called whenever a new incoming TFTP request is received by the server and allows the application to determine whether the connection should be accepted or rejected.

pfnGetData (type {`tTFTPTransfer`) This function is called to read each block of file data during an ongoing GET request. It must copy the requested number of bytes from a given position in the file into a supplied buffer. The application writes a pointer to this function into the `tTFTPConnection` instance data structure during processing of the `pfnRequest` callback if a GET request is to be accepted.

pfnPutData (type `tTFTPTransfer`) This function is called to write each block of file data during an ongoing PUT request. It must write the provided block of data into the target file. The application writes a pointer to this function into the `tTFTPConnection` instance data structure during processing of the `pfnRequest` callback if a PUT request is to be accepted.

pfnClose (type `tTFTPClose`) This function is called when a TFTP connection ends and allows the application to perform any cleanup required - freeing workspace memory and closing files, for example. The application writes a pointer to this function into the `tTFTPConnection` instance data structure during processing of the `pfnRequest` callback if the request is to be accepted.

14.2.0.3 pfnRequest

Application callback function called whenever a new TFTP request is received by the server.

Prototype:

```
tTFTPError
pfnRequest(struct _tTFTPConnection *psTFTP, tBoolean bGet, char
*pucFileName, tTFTPMode eMode)
```

Parameters:

psTFTP points to the TFTP connection instance data for the new request.

bGet is `true` if the incoming request is a GET (read) request or `false` if it is a PUT (write) request.

pucFileName points to the first character of the name of the local file which is to be read (on a GET request) or written (on a PUT request).

eMode indicates the requested transfer mode, `TFTP_MODE_NETASCII` (text) or `TFTP_MODE_OCTET` (binary).

Description:

This function, whose pointer is passed to the server as a parameter to function `TFTPInit()`, is called whenever a new TFTP request is received. It passes information about the request to the application allowing it to accept or reject it. The request type, GET or PUT, is determined from the `bGet` parameter and the target file name is provided in `pucFileName`.

If the application wishes to reject the request, it should set the `pcErrorString` field in the `psTFTP` structure and return an error code other than **TFTP_OK**.

To accept an incoming connection and start the file transfer, the application should return **TFTP_OK** after completing various fields in the `psTFTP` structure. For a GET request, fill in the `pfnGetData` and `pfnClose` function pointers and set `ulDataRemaining` to the size of the file which is being requested. For a PUT request, fill in the `pfnPutData` and `pfnClose` function pointers.

During processing of pfnRequest, the application may use the `pucUser` field as an anchor for any additional instance data required to process the request - a file handle, for example. This field will be accessible on all future callbacks related to this connection since the `psTFTP` structure is passed as a parameter in each case. Any resources allocated during `pfnRequest` can be freed during the later call to `pfnClose`.

Returns:

Returns **TFTP_OK** if the request is to be handled or any other TFTP error code if it is to be rejected.

14.2.0.4 pfnGetData

Application callback function called whenever the TFTP server needs another block of data read from the source file.

Prototype:

```
tTFTPError
pfnGetData(struct _tTFTPConnection *psTFTP)
```

Parameters:

psTFTP points to the TFTP connection instance data for the existing GET request.

Description:

This function, whose pointer was passed to the server in the `psTFTP` structure when the TFTP connection was accepted in `pfnRequest`, is called whenever the server needs a new block of file data to send back to the remote client. The application must copy a block of `psTFTP->ulDataLength` bytes of data from the source file to the buffer pointed to by `psTFTP->pucData`.

Typically, GET requests will read data sequentially from the file but, in some error recovery cases, data previously read may be requested again. The application must, therefore, ensure that the correct block of data is being returned by checking `psTFTP->ulBlockNum` and setting the source file offset correctly based on its value. The required read offset is `(psTFTP->ulBlockNum * TFTP_BLOCK_SIZE)` bytes from the start of the file.

If an error is detected while reading the file, field `psTFTP->pcErrorString` should be set and a value other than **TFTP_OK** returned.

Returns:

Returns **TFTP_OK** if the data was read successfully or any other TFTP error code if an error occurred.

14.2.0.5 pfnPutData

Application callback function called whenever the TFTP server has received data to be written to the destination file.

Prototype:

```
tTFTPError
pfnPutData(struct _tTFTPConnection *psTFTP)
```

Parameters:

psTFTP points to the TFTP connection instance data for the existing PUT request.

Description:

This function, whose pointer was passed to the server in the `psTFTP` structure when the TFTP connection was accepted in `pfnRequest`, is called whenever the server receives a block of data. The application must write a block of `psTFTP->ulDataLength` bytes of data from address `psTFTP->pucData` to the destination file.

Typically, PUT requests will write data sequentially to the file but, in some error recovery cases, data previously written may be received again. The application must, therefore, ensure that the received data is written at the correct position within the file. This position is determined from the fields `psTFTP->ulBlockNum` and `psTFTP->ulDataRemaining`. The byte offset relative to the start of the file that the data must be written to is given by `((psTFTP->ulBlockNum - 1) * TFTP_BLOCK_SIZE) + psTFTP->ulDataRemaining`.

If an error is detected while writing the file, field `psTFTP->pcErrorString` should be set and a value other than **TFTP_OK** returned.

Returns:

Returns **TFTP_OK** if the data was written successfully or any other TFTP error code if an error occurred.

14.2.0.6 pfnClose

Application callback function called whenever the TFTP connection is being closed.

Prototype:

```
void
pfnClose(struct _tTFTPConnection *psTFTP)
```

Parameters:

psTFTP points to the TFTP instance data block for the connection which is being closed.

Description:

This function, whose pointer was passed to the server in the `psTFTP` structure when the TFTP connection was accepted in `pfnRequest`, is called whenever the server is about to close the TFTP connection. An application may use it to free any resources allocated to service the connection (file handles, for example).

Returns:

None.

14.3 API Functions

Data Structures

- _tTFTPConnection

Defines

- TFTP_BLOCK_SIZE

Enumerations

- tTFTPError

Functions

- void TFTPInit (tTFTPRequest pfnRequest)

14.3.1 Data Structure Documentation

14.3.1.1 _tTFTPConnection

Definition:

```
typedef struct
{
    unsigned char *pucData;
    unsigned long ulDataLength;
    unsigned long ulDataRemaining;
    tTFTPTransfer pfnGetData;
    tTFTPTransfer pfnPutData;
    tTFTPClose pfnClose;
    unsigned char *pucUser;
    char *pcErrorString;
    udp_pcb *pPCB;
    unsigned long ulBlockNum;
}
_tTFTPConnection
```

Members:

pucData Pointer to the start of the buffer into which GET data should be copied or from which PUT data should be read.

ulDataLength The length of the data requested in response to a single pfnGetData callback or the size of the received data for a pfnPutData callback.

ulDataRemaining Count of remaining bytes to send during a GET request or the byte offset within a block during a PUT request. The application must set this field to the size of the requested file during the tTFTPRequest

pfnGetData Application function which is called whenever more data is required to satisfy a GET request. The function must copy ulDataLength bytes into the buffer pointed to by pucData.

pfnPutData Application function which is called whenever a packet of file data is received during a PUT request. The function must save the data to the target file using ulBlockNum and ulDataRemaining to indicate the position of the data in the file, and return an appropriate error code. Note that several calls to this function may be made for a given received TFTP block since the underlying networking stack may have split the TFTP packet between several packets and a callback is made for each of these. This avoids the need for a 512 byte buffer. The ulDataRemaining is used in these cases to indicate the offset of the data within the current block.

pfnClose Application function which is called when the TFTP connection is to be closed. The function should tidy up and free any resources associated with the connection prior to returning.

pucUser This field may be used by the client to store an application-specific pointer that will be accessible on all callbacks from the TFTP module relating to this connection.

pcErrorString Pointer to an error string which the client must fill in if reporting an error. This string will be sent to the TFTP client in any case where pfnPutData or pfnGetData return a value other than TFTP_OK.

pPCB A pointer to the underlying UDP connection. Applications must not modify this field.

ulBlockNum The current block number for an ongoing TFTP transfer. Applications may read this value to determine which data to return on a pfnGetData callback or where to write incoming data on a pfnPutData callback but must not modify it.

Description:
The TFTP connection control structure. This is passed to a client on all callbacks relating to a given TFTP connection. Depending upon the callback, the client may need to fill in values to various fields or use field values to determine where to transfer data from or to.

14.3.2 Define Documentation

14.3.2.1 TFTP_BLOCK_SIZE

Definition:
```
#define TFTP_BLOCK_SIZE
```

Description:
Data transfer under TFTP is performed using fixed-size blocks. This label defines the size of a block of TFTP data.

14.3.3 Typedef Documentation

14.3.3.1 tTFTPConnection

Definition:
```
typedef struct _tTFTPConnection tTFTPConnection
```

Description:
The TFTP connection control structure. This is passed to a client on all callbacks relating to a given TFTP connection. Depending upon the callback, the client may need to fill in values to various fields or use field values to determine where to transfer data from or to.

14.3.4 Enumeration Documentation

14.3.4.1 tTFTPError

Description:
TFTP error codes. Note that this enum is mapped so that all positive values match the TFTP protocol-defined error codes.

14.3.4.2 enum tTFTPMode

TFTP file transfer modes. This enum contains members defining ASCII text transfer mode (TFTP_MODE_NETASCII), binary transfer mode (TFTP_MODE_OCTET) and a marker for an invalid mode (TFTP_MODE_INVALID).

14.3.5 Function Documentation

14.3.5.1 void TFTPInit (tTFTPRequest *pfnRequest*)

Initializes the TFTP server module.

Parameters:

pfnRequest - A pointer to the function which the server will call whenever a new incoming TFTP request is received. This function must determine whether the request can be handled and return a value telling the server whether to continue processing the request or ignore it.

This function initializes the lwIP TFTP server and starts listening for incoming requests from clients. It must be called after the network stack is initialized using a call to lwIPInit().

Returns:

None.

15 Micro Standard Library Module

15.1 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.

15.2 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)

15.2.1 Data Structure Documentation

15.2.1.1 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.

15.2.2 Function Documentation

15.2.2.1 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.

15.2.2.2 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.

15.2.2.3 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.

15.2.2.4 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*.

15.2.2.5 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*.

15.2.2.6 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.

15.2.2.7 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.

15.2.2.8 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.

15.3 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);
```

16 UART Standard IO Module

16.1 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.

16.1.1 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.

16.1.2 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.

16.2 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)

16.2.1 Function Documentation

16.2.1.1 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.

16.2.1.2 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.

16.2.1.3 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.

16.2.1.4 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.

16.2.1.5 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.

16.2.1.6 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.

16.2.1.7 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.

16.2.1.8 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.

16.2.1.9 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.

16.2.1.10 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.

16.2.1.11 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.

16.2.1.12 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.

16.2.1.13 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.

16.3 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