


RDK-IDM-SBC Firmware Development Package

USER'S GUIDE



Copyright

Copyright © 2009-2011 Texas Instruments Incorporated. All rights reserved. Stellaris and StellarisWare are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

 Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

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 6852 of this document, last updated on January 11, 2011.

Table of Contents

Copyright	2
Revision Information	2
1 Introduction	7
2 Example Applications	9
2.1 Calibration for the Touch Screen (calibrate)	9
2.2 Graphics Library Demonstration (glib_demo)	9
2.3 Hello World (hello)	10
2.4 Hello using Widgets (hello_widget)	10
2.5 I2S example application using SD Card FAT file system (i2s_demo)	11
2.6 IDM-SBC Board Checkout Application (idm-checkout)	11
2.7 Graphics Library String Table Demonstration (lang_demo)	12
2.8 Blox Game (qs-blox)	12
2.9 Scribble Pad (scribble)	13
2.10 SD card using FAT file system (sd_card)	13
2.11 JPEG Image Decompression (showjpeg)	13
2.12 USB HID Keyboard Host (usb_host_keyboard)	14
2.13 USB HID Mouse Host (usb_host_mouse)	14
2.14 USB Mass Storage Class Host Example (usb_host_msc)	14
2.15 USB Stick Update Demo (usb_stick_demo)	15
2.16 USB Memory Stick Updater (usb_stick_update)	15
3 Development System Utilities	17
4 Touch Screen Driver	25
4.1 Introduction	25
4.2 API Functions	26
4.3 Programming Example	27
5 Sound Output Driver	29
5.1 Introduction	29
5.2 API Functions	29
5.3 Programming Example	33
6 SDRAM Driver	35
6.1 Introduction	35
6.2 API Functions	35
6.3 Programming Example	37
7 Pinout Driver	39
7.1 Introduction	39
7.2 API Functions	39
8 JPEG Display Widget	41
8.1 Introduction	41
8.2 API Functions	41
8.3 Programming Example	59
9 Display Driver	61
9.1 Introduction	61
9.2 API Functions	61
9.3 Programming Example	62
10 Serial Flash Driver	65
10.1 Introduction	65

10.2	API Functions	65
10.3	Programming Example	70
11	Wave Audio Driver	73
11.1	Introduction	73
11.2	API Functions	73
11.3	Programming Example	77
12	Command Line Processing Module	79
12.1	Introduction	79
12.2	API Functions	79
12.3	Programming Example	81
13	CPU Usage Module	83
13.1	Introduction	83
13.2	API Functions	83
13.3	Programming Example	84
14	CRC Module	87
14.1	Introduction	87
14.2	API Functions	87
14.3	Programming Example	87
15	Flash Parameter Block Module	89
15.1	Introduction	89
15.2	API Functions	89
15.3	Programming Example	91
16	File System Wrapper Module	93
16.1	Introduction	93
16.2	API Functions	93
16.3	Programming Example	96
17	Integer Square Root Module	99
17.1	Introduction	99
17.2	API Functions	99
17.3	Programming Example	100
18	Ethernet Board Locator Module	101
18.1	Introduction	101
18.2	API Functions	101
18.3	Programming Example	104
19	lwIP Wrapper Module	105
19.1	Introduction	105
19.2	API Functions	105
19.3	Programming Example	108
20	PTPd Wrapper Module	109
20.1	Introduction	109
20.2	API Functions	109
20.3	Programming Example	109
21	Ring Buffer Module	111
21.1	Introduction	111
21.2	API Functions	111
21.3	Programming Example	117
22	Simple Task Scheduler Module	119
22.1	Introduction	119

22.2	API Functions	119
22.3	Programming Example	124
23	Sine Calculation Module	127
23.1	Introduction	127
23.2	API Functions	127
23.3	Programming Example	128
24	Ethernet Software Update Module	129
24.1	Introduction	129
24.2	API Functions	129
24.3	Programming Example	131
25	TFTP Server Module	133
25.1	Introduction	133
25.2	Usage	133
25.3	API Functions	136
26	Micro Standard Library Module	141
26.1	Introduction	141
26.2	API Functions	141
26.3	Programming Example	147
27	UART Standard IO Module	149
27.1	Introduction	149
27.2	API Functions	150
27.3	Programming Example	156
IMPORTANT NOTICE		158

1 Introduction

The Texas Instruments® Stellaris® Intelligent Display Module (Single Board Computer) is a ready to use module that features an ARM® Cortex™-M3-based microcontroller, a TFT touch screen display, and Ethernet, USB OTG, RS232 and CAN connectivity. The module also has 8MBytes of SDRAM and 1MB of SSI-connected flash storage to augment the internal 96KB of SRAM and 256KB of flash provided by the microcontroller along with a microSD card slot supporting cards up to 2GB in size. Audio output is available via an I2S-connected audio DAC providing connections for an optional external speaker.

This document describes the board-specific drivers and example applications that are provided for this reference design 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 (`rdk-idm-sbc.eww`) that contains the peripheral driver library project, graphics library project, USB library project, and all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`rdk-idm-sbc.mpw`) that contains the peripheral driver library project, graphics library project, USB library project, and all of the board example projects, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/rdk-idm-sbc` subdirectory of the firmware development package source distribution.

2.1 Calibration for the Touch Screen (calibrate)

The raw sample interface of the touch screen driver is used to compute the calibration matrix required to convert raw samples into screen X/Y positions. The produced calibration matrix can be inserted into the touch screen driver to map the raw samples into screen coordinates.

The touch screen calibration is performed according to the algorithm described by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at <http://www.embedded.com/story/OEG20020529S0046>.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.2 Graphics Library Demonstration (gplib_demo)

This application provides a demonstration of the capabilities of the Stellaris Graphics Library. A series of panels show different features of the library. For each panel, the bottom provides a forward and back button (when appropriate), along with a brief description of the contents of the panel.

The first panel provides some introductory text and basic instructions for operation of the application.

The second panel shows the available drawing primitives: lines, circles, rectangles, strings, and images.

The third panel shows the canvas widget, which provides a general drawing surface within the widget heirarchy. A text, image, and application-drawn canvas are displayed.

The fourth panel shows the check box widget, which provides a means of toggling the state of an item. Four check boxes are provided, with each having a red “LED” to the right. The state of the LED tracks the state of the check box via an application callback.

The fifth panel shows the container widget, which provides a grouping construct typically used for

radio buttons. Containers with a title, a centered title, and no title are displayed.

The sixth panel shows the push button widget. Two columns of push buttons are provided; the appearance of each column is the same but the left column does not utilize auto-repeat while the right column does. Each push button has a red “LED” to its left, which is toggled via an application callback each time the push button is pressed.

The seventh panel shows the radio button widget. Two groups of radio buttons are displayed, the first using text and the second using images for the selection value. Each radio button has a red “LED” to its right, which tracks the selection state of the radio buttons via an application callback. Only one radio button from each group can be selected at a time, though the radio buttons in each group operate independently.

The eighth panel shows the slider widget. Six sliders constructed using the various supported style options are shown. The slider value callback is used to update two widgets to reflect the values reported by sliders. A canvas widget near the top right of the display tracks the value of the red and green image-based slider to its left and the text of the grey slider on the left side of the panel is update to show its own value. The slider on the right is configured as an indicator which tracks the state of the upper slider and ignores user input.

The ninth and final panel shows the image button widget. This operates in exactly the same way as the pushbutton widget illustrated in an earlier panel but allows the control to be defined in terms of two background images (for pressed and unpressed states) and either a text string or a keycap image. When pressed, the text or keycap image can be offset to provide a 3D effect. As in the pushbutton demonstration, red “LED” indicators toggle in response to button presses.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.3 Hello World (hello)

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

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.4 Hello using Widgets (hello_widget)

A very simple “hello world” example written using the Stellaris Graphics Library widgets. It displays a button which, when pressed, toggles display of the words “Hello World!” on the screen. This may be used as a starting point for more complex widget-based applications.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.5 I2S example application using SD Card FAT file system (i2s_demo)

This example application demonstrates playing wav files from an SD card that is formatted with a FAT file system. The application will only look in the root directory of the SD card and display all files that are found. Files can be selected to show their format and then played if the application determines that they are a valid .wav file.

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

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.6 IDM-SBC Board Checkout Application (idm-checkout)

This widget-based application exercises many of the peripherals found on the RDK-IDM-SBC reference board. It offers the following features:

- USB mouse support. The application will show the state of up to three mouse buttons and a cursor position when a USB mouse is connected to the board.
- TFTP server. This server allows the image stored in the 1MB serial flash device to be written and read from a remote Ethernet-connected system. The image in the serial flash is copied to SDRAM on startup and used as the source for the RAM-based web server file system. Suitable images can be created using the makefsfile utility with the -b and -h switches. To upload a binary image to the serial flash, use the TFTP command line `tftp -i <board IP> PUT <binary file> eeprom`. To read the current image out of serial flash, use command line `tftp -i <board IP> GET eeprom <binary file>`. When shipped, the serial flash on the board is empty. The file `ramfs_data.bin` which contains a web photo gallery is provided in the `StellarisWare/boards/rdk-idm-sbc/idm-checkout` directory and can be downloaded to the device using the TFTP command given above.
- Web server. The lwIP TCP/IP stack is used to implement a web server which can serve files from an internal file system, a FAT file system on an installed microSD card or a file system image stored in serial flash and copied to SDRAM during initialization. These file systems may be accessed from a web browser using URLs `http://<board IP>`, `http://<board IP>/sdcard/<filename>` and `http://<board ID>/ram/<filename>` respectively where `<board IP>` is the dotted decimal IP address assigned to the board and `<filename>` indicates the particular file being requested. Note that the web server does not open default filenames (such as `index.htm`) in any directory other than the root so the full path and filename must be specified.
- Touch screen. The current touch coordinates are displayed whenever the screen is pressed.
- LED control. A GUI widget allows control of one of the Ethernet LEDs on the board.
- Serial command line. A simple command line is implemented via UART0. Connect a terminal emulator set to 115200/8/N/1 and enter "help" for information on supported commands.
- JPEG image viewer. The QVGA display is used to display JPEG images from the “images” directory in the web server’s SDRAM file system image. The user can scroll the image on the display using the touchscreen. This demonstration relies upon the availability of the

ramfs_data.bin file described under “TFTP server” above. Ensure that this has been written to the serial flash and the board rebooted before trying this function.

- **Audio player.** Uncompressed WAV files stored on the microSD card may be played back via the headphone jack on the I2S daughter board. Available wave audio files are shown in a listbox on the left side of the display. Click the desired file then press “Play” to play it back. Volume control is provided via a touchscreen slider shown on the right side of the display.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.7 Graphics Library String Table Demonstration (lang_demo)

This application provides a demonstration of the capabilities of the Stellaris Graphics Library's string table functions. A series of panels show different implementations of features of the string table functions. For each panel, the bottom provides a forward and back button (when appropriate).

The first panel provides a large string with introductory text and basic instructions for operation of the application.

The second panel shows the available languages and allows them to be switched between English, German, Spanish and Italian.

The final panel provides instructions and information necessary to update the board firmware via ethernet using the LM Flash Programmer application. When the “Update” button is pressed, the application transfers control to the boot loader to allow a new application image to be downloaded. If using a recent version of LMFlash, firmware download can also be initiated automatically via a “magic packet” sent from LMFlash to the IDM-SBC. When this packet is received, the application transfers control to the boot loader without any required user interaction.

2.8 Blox Game (qs-blox)

Blox is a version of a well-known game in which you attempt to stack falling colored blocks without leaving gaps. Blocks may be moved up or down, rotated and dropped using onscreen widgets or an attached USB keyboard. When any row of the game area is filled with coloured squares, that row is removed providing more space to drop later blocks into. The game ends when there is insufficient space on the playing area for a new block to be placed on the board. Your score accumulates over time with the number of points awarded for dropping each block dependent upon the height from which the block was dropped.

When playing with the USB keyboard, the controls are as follow:

Up or Right arrow Move the current block towards the top of the game area.

Down or Left arrow Move the current block towards the bottom of the game area.

Space Drop the block.

R Rotate the block 90 degrees clockwise.

P Pause or unpause the game.

The game also offers a small web site which displays the current status and allows the difficulty level to be set. This site employs AJAX to request an XML file containing status information from the IDM and uses this to update the score, hiscore and state information once every 2.5 seconds.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.9 Scribble Pad (scribble)

The scribble pad provides a drawing area on the screen. Touching the screen will draw onto the drawing area using a selection of fundamental colors (in other words, the seven colors produced by the three color channels being either fully on or fully off). Each time the screen is touched to start a new drawing, the drawing area is erased and the next color is selected.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.10 SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based console on the display and also a UART-based command line for viewing and navigating the file system on the SD card.

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

The application may also be operated via a serial terminal attached to UART0. The RS232 communication parameters should be set to 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.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.11 JPEG Image Decompression (showjpeg)

This example application decompresses a JPEG image which is linked into the application and shows it on the 320x240 display. SDRAM is used for image storage and decompression workspace. The image may be scrolled in the display window by dragging a finger across the touchscreen.

JPEG decompression and display are handled using a custom graphics library widget, the source for which can be found in `drivers/jpgwidget.c`.

The JPEG library used by this application is release 6b of the Independent JPEG Group’s reference decoder. For more information, see the README and various text file in the `/third_party/jpeg` directory or visit <http://www.ijg.org/>.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.12 USB HID Keyboard Host (usb_host_keyboard)

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

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.13 USB HID Mouse Host (usb_host_mouse)

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

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM Flash Programmer.

2.14 USB Mass Storage Class Host Example (usb_host_msc)

This example application demonstrates reading a file system from a USB flash disk. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based console on the display and also a UART-based command line for viewing and navigating the file system on the flash disk.

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

The application may also be operated via a serial terminal attached to UART0. The RS232 communication parameters should be set to 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.

This application supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request “magic packet” from LM

Flash Programmer.

2.15 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 a "button" on the touch screen. Once the button is pressed, control is passed back to the usb_stick_update program which is still in 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.

This application also supports remote software update over Ethernet using the LM Flash Programmer application. A firmware update is initiated using the remote update request "magic packet" from LM Flash Programmer.

If the flash is updated using the Ethernet method, then the usb_stick_update program located at the beginning of flash will be erased.

2.16 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

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.

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.

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
```

FreeType Rasterizer

Usage:

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

Description:

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

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

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

Arguments:

- b** specifies that this is a bold font. This does not affect the rendering of the font, it only changes the name of the file and the name of the font structure that are produced.
- f FILENAME** specifies the base name for this font, which is used to create the output file name and the name of the font structure. The default value is "font" if not specified.
- i** specifies that this is an italic font. This does not affect the rendering of the font, it only changes the name of the file and the name of the font structure that are produced.
- m** specifies that this is a monospaced font. This causes the glyphs to be horizontally centered in a box whose width is the width of the widest glyph. For best visual results, this option should only be used for font faces that are designed to be monospaced (such as Computer Modern TeleType).
- s SIZE** specifies the size of this font, in points. The default value is 20 if not specified.
- p NUM** This specifies the index of the first character in the font that is to be encoded. If the value is not provided, it defaults to 32 which is typically the space character.
- e NUM** This specifies the index of the last character in the font that is to be encoded. If the value is not provided, it defaults to 126 which, in ISO8859-1 is tilde (). -
- w NUM** Encodes the specified character index as a space regardless of the character which may be present in the font at that location. This is helpful in allowing a space to be included in a font which only encodes a subset of the characters which would not normally include the space character (for example, numeric digits only). If absent, this value defaults to 32, ensuring that character 32 is always the space.

-n This switch overrides **-w** and causes no character to be encoded as a space unless the source font already contains a space.

INPUT FILE specifies the name of the input font file.

Example:

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

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

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

The following would render a Computer Modern small-caps font at 44 points and generate an output font containing only characters 47 through 58 (the numeric digits). Additionally, the first character in the encoded font (which is displayed if an attempt is made to render a character which is not included in the font) is forced to be a space:

```
ftrasterize -f cmscdigits -s 44 -w 47 -p 47 -e 58 cmcsc10.pfb
```

The output will be written to `fontcmscdigits44.c` and contain a definition for `g_sFontCmscdigits44` that describes the font.

GIMP Script For Texas Instruments Stellaris Button

Description:

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

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

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

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
```

String Table Generator

Usage:

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

Description:

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

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

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

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

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

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

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

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

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

This utility is contained in `tools/bin`.

Arguments:

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

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

Example:

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

```
mkstringtable str.csv str
```

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

NetPNM Converter

Usage:

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

Description:

Converts a NetPBM image file into the format that is recognized by the Stellaris Graphics Library. The input image must be in the raw PPM format (in other words, with the `P6` tag). The NetPBM image format can be produced using GIMP, NetPBM (<http://netpbm.sourceforge.net>), ImageMagick (<http://www.imagemagick.org>), or numerous other open source and proprietary image manipulation packages.

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

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

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

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

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

Arguments:

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

Example:

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

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

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

Serial Flash Downloader

Usage:

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

Description:

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

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

Arguments:

- b **BAUD** specifies the baud rate. If not specified, the default of 115,200 will be used.
- c **PORT** specifies the COM port. If not specified, the default of COM1 will be used.
- d disables auto-baud.
- h displays usage information.
- l **FILENAME** specifies the name of the boot loader image file.
- p **ADDR** specifies the address at which to program the firmware. If not specified, the default of 0 will be used.
- r **ADDR** specifies the address at which to start processor execution after the firmware has been downloaded. If not specified, the processor will be reset after the firmware has been downloaded.
- s **SIZE** specifies the size of the data packets used to download the firmware data. This must be a multiple of four between 8 and 252, inclusive. If using the Serial Flash Loader, the maximum value that can be used is 76. If using the Boot Loader, the maximum value that can be used is dependent upon the configuration of the Boot Loader. If not specified, the default of 8 will be used.

INPUT FILE specifies the name of the firmware image file.

Example:

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

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


4 Touch Screen Driver

Introduction	25
API Functions	26
Programming Example	27

4.1 Introduction

The touch screen is a pair of resistive layers on the surface of the display. One layer has connection points at the top and bottom of the screen, and the other layer has connection points at the left and right of the screen. When the screen is touched, the two layers make contact and electricity can flow between them.

The horizontal position of a touch can be found by applying positive voltage to the right side of the horizontal layer and negative voltage to the left side. When not driving the top and bottom of the vertical layer, the voltage potential on that layer will be proportional to the horizontal distance across the screen of the press, which can be measured with an ADC channel. By reversing these connections, the vertical position can also be measured. When the screen is not being touched, there will be no voltage on the non-powered layer.

By monitoring the voltage on each layer when the other layer is appropriately driven, touches and releases on the screen, as well as movements of the touch, can be detected and reported.

In order to read the current voltage on the two layers and also drive the appropriate voltages onto the layers, each side of each layer is connected to both a GPIO and an ADC channel. The GPIO is used to drive the node to a particular voltage, and when the GPIO is configured as an input, the corresponding ADC channel can be used to read the layer's voltage.

The touch screen is sampled every millisecond, with four samples required to properly read both the X and Y position. Therefore, 250 X/Y sample pairs are captured every second.

Like the display driver, the touch screen driver operates in the same four orientations (selected in the same manner). Default calibrations are provided for using the touch screen in each orientation; the calibrate application can be used to determine new calibration values if necessary.

The touch screen driver utilizes sample sequence 3 of the ADC and timer 1 subtimer A. The interrupt from the ADC sample sequence 3 is used to process the touch screen readings; the [TouchScreenIntHandler\(\)](#) function should be called when this interrupt occurs (which is typically accomplished by placing it in the vector table in the startup code for the application).

The touch screen driver makes use of calibration parameters determined using the “calibrate” example application. The theory behind these parameters is explained by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at <http://www.embedded.com/story/OEG20020529S0046>.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `touch.c` containing the source code and `touch.h` containing the API definitions for use by applications.

4.2 API Functions

Functions

- void [TouchScreenCallbackSet](#) (long (*pfnCallback)(unsigned long ulMessage, long lX, long lY))
- void [TouchScreenInit](#) (void)
- void [TouchScreenIntHandler](#) (void)

4.2.1 Function Documentation

4.2.1.1 TouchScreenCallbackSet

Sets the callback function for touch screen events.

Prototype:

```
void  
TouchScreenCallbackSet (long (*long) (unsigned ulMessage, long lX, long  
lY) pfnCallback)
```

Parameters:

pfnCallback is a pointer to the function to be called when touch screen events occur.

Description:

This function sets the address of the function to be called when touch screen events occur. The events that are recognized are the screen being touched (“pen down”), the touch position moving while the screen is touched (“pen move”), and the screen no longer being touched (“pen up”).

Returns:

None.

4.2.1.2 TouchScreenInit

Initializes the touch screen driver.

Prototype:

```
void  
TouchScreenInit (void)
```

Description:

This function initializes the touch screen driver, beginning the process of reading from the touch screen. This driver uses the following hardware resources:

- ADC sample sequence 3
- Timer 1 subtimer A

Returns:

None.

4.2.1.3 TouchScreenIntHandler

Handles the ADC interrupt for the touch screen.

Prototype:

```
void
TouchScreenIntHandler(void)
```

Description:

This function is called when the ADC sequence that samples the touch screen has completed its acquisition. The touch screen state machine is advanced and the acquired ADC sample is processed appropriately.

It is the responsibility of the application using the touch screen driver to ensure that this function is installed in the interrupt vector table for the ADC3 interrupt.

Returns:

None.

4.3 Programming Example

The following example shows how to initialize the touchscreen driver and the callback function which receives notifications of touch and release events in cases where the StellarisWare Graphics Library widget manager is not being used by the application.

```

//*****
//
// Globals used to hold the current touch position.
//
//*****
long g_lTouchX, g_lTouchY;

//*****
//
// Globals used to hold flags indicating any touchscreen event received.
//
//*****
volatile unsigned long g_ulFlags;

//*****
//
// The touch screen driver calls this function to report all state changes.
//
//*****
static long
TouchTestCallback(unsigned long ulMessage, long lX, long lY)
{
    //
    // Save the new touch position.
    //
    g_lTouchX = lX;
    g_lTouchY = lY;

    //
    // Determine what to do now. In this case, we merely set flags that the
    // application main loop can deal with later.
    //
    switch(ulMessage)

```

```
    {
        case WIDGET_MSG_PTR_UP:
            g_ulFlags |= FLAG_PTR_UP;
            break;

        case WIDGET_MSG_PTR_DOWN:
            g_ulFlags |= FLAG_PTR_DOWN;
            break;

        case WIDGET_MSG_PTR_MOVE:
            g_ulFlags |= FLAG_PTR_MOVE;
            break;

        default:
            break;
    }

    return(0);
}

int main(void)
{
    ...

    //
    // Initialize the touch screen driver.
    //
    TouchScreenInit();
    TouchScreenCallbackSet(TouchTestCallback);

    ...

    //
    // Process touch events as signalled via g_ulFlags.
    //

    ...
}
```

If using the StellarisWare Graphics Library widget manager, touchscreen initialization code is as follows. In this case, the touchscreen callback is provided within the widget manager so no additional function is required in the application code.

```
int main(void)
{
    ...

    //
    // Initialize the touch screen driver when using the graphics library
    // widget manager.
    //
    TouchScreenInit();
    TouchScreenCallbackSet(WidgetPointerMessage);

    ...
}
```

5 Sound Output Driver

Introduction	29
API Functions	29
Programming Example	33

5.1 Introduction

The sound driver allows applications to play PCM sample buffers and simple tones using the internal I2S peripheral communicating with an external audio DAC IC. Functions are provided to initialize the DAC, set the output volume, set the audio format and play a buffer of PCM audio samples. There is also a method for creating simple songs or sound effects by specifying a sequence of frequencies and the times at which they should be output.

Applications wishing to use the sound driver must ensure that they set function [SoundIntHandler\(\)](#) in the interrupt table for the I2S vector.

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

5.2 API Functions

Functions

- unsigned long [SoundBufferPlay](#) (const void *pvData, unsigned long ulLength, tBufferCallback pfnCallback)
- void [SoundDisable](#) (void)
- void [SoundInit](#) (void)
- void [SoundIntHandler](#) (void)
- void [SoundPlay](#) (const unsigned short *pusSong, unsigned long ulLength)
- void [SoundSetFormat](#) (unsigned long ulSampleRate, unsigned short usBitsPerSample, unsigned short usChannels)
- void [SoundVolumeDown](#) (unsigned long ulPercent)
- unsigned char [SoundVolumeGet](#) (void)
- void [SoundVolumeSet](#) (unsigned long ulPercent)
- void [SoundVolumeUp](#) (unsigned long ulPercent)

5.2.1 Function Documentation

5.2.1.1 SoundBufferPlay

Starts playback of a block of PCM audio samples.

Prototype:

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

Parameters:

pvData is a pointer to the audio data to play.

ulLength is the length of the data in bytes.

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

Description:

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

Returns:

0 if the buffer was accepted, returns non-zero if there was no space available for this buffer.

5.2.1.2 SoundDisable

Disables the sound output.

Prototype:

```
void  
SoundDisable(void)
```

Description:

This function disables the sound output, muting the speaker and cancelling any playback that may be in progress.

Returns:

None.

5.2.1.3 SoundInit

Initializes the sound output.

Prototype:

```
void  
SoundInit(void)
```

Description:

This function prepares the sound driver to play songs or sound effects. It must be called before any other sound function. The sound driver uses uDMA and the caller must ensure that the uDMA peripheral is enabled and its control table configured prior to making this call.

Returns:

None

5.2.1.4 SoundIntHandler

Handles the I2S sound interrupt.

Prototype:

```
void  
SoundIntHandler(void)
```

Description:

This function services the I2S interrupt and ensures that DMA buffers are correctly handled to ensure smooth flow of audio data to the DAC.

Returns:

None.

5.2.1.5 SoundPlay

Starts playback of a song.

Prototype:

```
void  
SoundPlay(const unsigned short *pusSong,  
           unsigned long ulLength)
```

Parameters:

pusSong is a pointer to the song data structure.

ulLength is the length of the song data structure in bytes.

Description:

This function starts the playback of a song or sound effect. If a song or sound effect is already being played, its playback is cancelled and the new song is started.

Returns:

None.

5.2.1.6 SoundSetFormat

Configures the I2S peripheral for the given audio data format.

Prototype:

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

Parameters:

ulSampleRate is the sample rate of the audio to be played in samples per second.

usBitsPerSample is the number of bits in each audio sample.

usChannels is the number of audio channels, 1 for mono, 2 for stereo.

Description:

This function configures the I2S peripheral in preparation for playing audio data of a particular format.

Returns:

None.

5.2.1.7 SoundVolumeDown

Decreases the volume.

Prototype:

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

Parameters:

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

Description:

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

Returns:

None.

5.2.1.8 SoundVolumeGet

Returns the current volume level.

Prototype:

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

Description:

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

Returns:

Returns the current volume.

5.2.1.9 SoundVolumeSet

Sets the volume of the music/sound effect playback.

Prototype:

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


Parameters:

ulPercent is the volume percentage, which must be between 0% (silence) and 100% (full volume), inclusive.

Description:

This function sets the volume of the sound output to a value between silence (0%) and full volume (100%).

Returns:

None.

5.2.1.10 SoundVolumeUp

Increases the volume.

Prototype:

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

Parameters:

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

Description:

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

Returns:

None.

5.3 Programming Example

The following example shows how to play a short tone at 1000 Hz.

```
//  
// Initialize the sound output.  
//  
SoundInit();  
  
//  
// Set the sound output frequency to 1000 Hz.  
//  
SoundFrequencySet(1000);  
  
//  
// Enable the sound output.  
//  
SoundEnable();  
  
//  
// Delay for a while.  
//  
SysCtlDelay(10000);
```

```
//  
// Disable the sound output.  
//  
SoundDisable();
```

6 SDRAM Driver

Introduction	35
API Functions	35
Programming Example	37

6.1 Introduction

The SDRAM driver offers a convenient API for initializing and using SDRAM attached to the External Peripheral Interface. Functions are provided to configure the EPI module and I/O pins and also to manage the SDRAM as a heap, allocating and freeing blocks.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `sdram.c` containing the source code and `sdram.h` containing the API definitions for use by applications. Applications using it must also ensure that they include source file `bget.c` which can be found in the `third_party/bget` directory.

6.2 API Functions

Functions

- void * [ExtRAMAlloc](#) (unsigned long ulSize)
- void [ExtRAMFree](#) (void *pvBlock)
- unsigned long [ExtRAMMaxFree](#) (unsigned long *pulTotalFree)
- tBoolean [SDRAMInit](#) (unsigned long ulEPIDivider, unsigned long ulConfig, unsigned long ulRefresh)

6.2.1 Function Documentation

6.2.1.1 ExtRAMAlloc

Allocates a block of memory from the SDRAM heap.

Prototype:

```
void *
ExtRAMAlloc(unsigned long ulSize)
```

Parameters:

ulSize is the size in bytes of the block of SDRAM to allocate.

Description:

This function allocates a block of SDRAM and returns its pointer. If insufficient space exists to fulfill the request, a NULL pointer is returned.

Returns:

Returns a non-zero pointer on success or NULL if it is not possible to allocate the required memory.

6.2.1.2 ExtRAMFree

Frees a block of memory in the SDRAM heap.

Prototype:

```
void  
ExtRAMFree(void *pvBlock)
```

Parameters:

pvBlock is the pointer to the block of SDRAM to free.

Description:

This function frees a block of memory that had previously been allocated using a call to [ExtRAMAlloc\(\)](#);

Returns:

None.

6.2.1.3 ExtRAMMaxFree

Reports information on the current heap usage.

Prototype:

```
unsigned long  
ExtRAMMaxFree(unsigned long *pulTotalFree)
```

Parameters:

pulTotalFree points to storage which will be written with the total number of bytes unallocated in the heap.

Description:

This function reports the total amount of memory free in the SDRAM heap and the size of the largest available block. It is included in the build only if label INCLUDE_BGET_STATS is defined.

Returns:

Returns the size of the largest available free block in the SDRAM heap.

6.2.1.4 SDRAMInit

Initializes the SDRAM.

Prototype:

```
tBoolean  
SDRAMInit(unsigned long ulEPIDivider,  
           unsigned long ulConfig,  
           unsigned long ulRefresh)
```

Parameters:

- ulEPIDivider*** is the EPI clock divider to use.
- ulConfig*** is the SDRAM interface configuration.
- ulRefresh*** is the refresh count in core clocks (0-2047).

Description:

This function must be called prior to [ExtRAMAlloc\(\)](#) or [ExtRAMFree\(\)](#). It configures the Stellaris microcontroller EPI block for SDRAM access and initializes the SDRAM heap (if SDRAM is found). The parameter *ulConfig* is the logical OR of several sets of choices:

The processor core frequency must be specified with one of the following:

- **EPI_SDRAM_CORE_FREQ_0_15** - core clock is 0 MHz < clk ≤ 15 MHz
- **EPI_SDRAM_CORE_FREQ_15_30** - core clock is 15 MHz < clk ≤ 30 MHz
- **EPI_SDRAM_CORE_FREQ_30_50** - core clock is 30 MHz < clk ≤ 50 MHz
- **EPI_SDRAM_CORE_FREQ_50_100** - core clock is 50 MHz < clk ≤ 100 MHz

The low power mode is specified with one of the following:

- **EPI_SDRAM_LOW_POWER** - enter low power, self-refresh state
- **EPI_SDRAM_FULL_POWER** - normal operating state

The SDRAM device size is specified with one of the following:

- **EPI_SDRAM_SIZE_64MBIT** - 64 Mbit device (8 MB)
- **EPI_SDRAM_SIZE_128MBIT** - 128 Mbit device (16 MB)
- **EPI_SDRAM_SIZE_256MBIT** - 256 Mbit device (32 MB)
- **EPI_SDRAM_SIZE_512MBIT** - 512 Mbit device (64 MB)

The parameter *ulRefresh* sets the refresh counter in units of core clock ticks. It is an 11-bit value with a range of 0 - 2047 counts.

Returns:

Returns **true** on success of **false** if no SDRAM is found or any other error occurs.

6.3 Programming Example

The following example shows how to initialize SDRAM and allocate a block for application use.

```
tBoolean bRetcode;
unsigned char *pucBlock;

//
// Initialize the SDRAM. This assumes that our system clock is running at
// somewhere between 50MHz and 100MHz and that the board is populated with
// a 64Mb SDRAM device. It sets the EPI clock divider to 1 (which divides
// the system clock by 2) and refreshes every 1024 cycles.
//
bRetcode = SDRAMInit(1, (EPI_SDRAM_CORE_FREQ_50_100 |
                        EPI_SDRAM_FULL_POWER | EPI_SDRAM_SIZE_64MBIT), 1024);

//
// Did we initialize the SDRAM correctly?
//
```

```
if(!bRetcode)
{
    //
    // Handle a fatal error here - the SDRAM could not be initialized.
    //
}

//
// Allocate an SDRAM buffer.
//
pucBlock = ExtRAMAlloc(MY_BLOCK_SIZE);

if(pucBlock)
{
    //
    // Go ahead and use the memory allocation.
    //

    ...

    //
    // Free the block once we are finished with it.
    //
    ExtRAMFree(pucBlock);
}
else
{
    //
    // Oops - we couldn't allocate memory. Handle the error condition here.
}
}
```

7 Pinout Driver

Introduction	39
API Functions	39

7.1 Introduction

The `set_pinout` driver offers a convenient API for configuring the device pinout appropriately for the IDM-SBC board. It encapsulates all GPIO Port Control register configuration into a single, common function that all applications running on a particular hardware platform can call to initialize the pinout.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `set_pinout.c` containing the source code and `set_pinout.h` containing the API definition for use by applications.

7.2 API Functions

Functions

- void `PinoutSet` (void)

7.2.1 Function Documentation

7.2.1.1 PinoutSet

Configures the LM3S9B92 device pinout for the RDK-IDM-SBC board.

Prototype:

```
void  
PinoutSet (void)
```

Description:

This function configures each pin of the `lm3s9b92` device to route the appropriate peripheral signal as required by the design of the `rdk-idm-sbc` development board.

Returns:

None.

8 JPEG Display Widget

Introduction	41
API Functions	41
Programming Example	59

8.1 Introduction

The `jpgwidget` module contains a custom control widget for use with the Stellaris Graphics Library. This widget supports two modes of operation, both of which display a single JPEG image within the bounds of the control.

A “JPEGButton” widget offers simple pushbutton functionality with a callback function provided by the client being called to indicate that the button has either been pressed or released. This type of control centers the provided JPEG image within the area of the widget, clipping the image if necessary to fit the available area.

A “JPEGCanvas” widget is intended for image display. It does not provide an `OnClick` callback but does offer image scrolling function using the touchscreen input to control positioning of the image within the widget area. An `OnScroll` callback is provided which will be called whenever the position of the image is changed by the user. An application may use this to pace the redraw rate for the control or, alternatively, set the widget to redraw automatically on any scroll input.

To use this widget, an application must also include the source for the JPEG decoder which can be found in `third_party/jpeg` along with the SDRAM driver from `boards/rdk-idm-sbc/drivers` and the `bget` heap manager from `third_party/bget`. The application makefile or project must also ensure that label `INCLUDE_BGET_STATS` is defined to enable optional heap manager functionality that is required by the JPEG decoder.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `jpgwidget.c` containing the source code and `jpgwidget.h` containing the API definitions for use by applications.

8.2 API Functions

Data Structures

- `tJPEGInst`
- `tJPEGWidget`

Defines

- `JPEGButton`(`sName`, `pParent`, `pNext`, `pChild`, `pDisplay`, `IX`, `IY`, `IWidth`, `IHeight`, `ulStyle`, `ulFillColor`, `ulOutlineColor`, `ulTextColor`, `pFont`, `pcText`, `puclImage`, `ullmgLen`, `ucBorderWidth`, `pfnOnClick`, `psInst`)
- `JPEGCanvas`(`sName`, `pParent`, `pNext`, `pChild`, `pDisplay`, `IX`, `IY`, `IWidth`, `IHeight`, `ulStyle`, `ulFillColor`, `ulOutlineColor`, `ulTextColor`, `pFont`, `pcText`, `puclImage`, `ullmgLen`, `ucBorderWidth`, `pfnOnScroll`, `psInst`)

- [JPEGWidgetClickCallbackSet](#)(pWidget, pfnOnClick)
- [JPEGWidgetFillColorSet](#)(pWidget, ulColor)
- [JPEGWidgetFillOff](#)(pWidget)
- [JPEGWidgetFillOn](#)(pWidget)
- [JPEGWidgetFontSet](#)(pWidget, pFnt)
- [JPEGWidgetImageOff](#)(pWidget)
- [JPEGWidgetLock](#)(pWidget)
- [JPEGWidgetOutlineColorSet](#)(pWidget, ulColor)
- [JPEGWidgetOutlineOff](#)(pWidget)
- [JPEGWidgetOutlineOn](#)(pWidget)
- [JPEGWidgetOutlineWidthSet](#)(pWidget, ucWidth)
- [JPEGWidgetScrollCallbackSet](#)(pWidget, pfnOnScrl)
- [JPEGWidgetStruct](#)(pParent, pNext, pChild, pDisplay, IX, IY, IWidth, IHeight, ulStyle, ulFillColor, ulOutlineColor, ulTextColor, pFont, pcText, puclmage, ullmgLen, ucBorderWidth, pfnOnClick, pfnOnScroll, psInst)
- [JPEGWidgetTextColorSet](#)(pWidget, ulColor)
- [JPEGWidgetTextOff](#)(pWidget)
- [JPEGWidgetTextOn](#)(pWidget)
- [JPEGWidgetTextSet](#)(pWidget, pcTtxt)
- [JPEGWidgetUnlock](#)(pWidget)
- [JW_STYLE_BUTTON](#)
- [JW_STYLE_FILL](#)
- [JW_STYLE_LOCKED](#)
- [JW_STYLE_OUTLINE](#)
- [JW_STYLE_PRESSED](#)
- [JW_STYLE_RELEASE_NOTIFY](#)
- [JW_STYLE_SCROLL](#)
- [JW_STYLE_TEXT](#)

Functions

- long [JPEGWidgetImageDecompress](#) (tWidget *pWidget)
- void [JPEGWidgetImageDiscard](#) (tWidget *pWidget)
- long [JPEGWidgetImageSet](#) (tWidget *pWidget, const unsigned char *plmg, unsigned long ullmgLen)
- void [JPEGWidgetInit](#) (tJPEGWidget *pWidget, const tDisplay *pDisplay, long IX, long IY, long IWidth, long IHeight)
- long [JPEGWidgetMsgProc](#) (tWidget *pWidget, unsigned long ulMsg, unsigned long ulParam1, unsigned long ulParam2)

8.2.1 Data Structure Documentation

8.2.1.1 tJPEGInst

Definition:

```
typedef struct
{
```

```
    unsigned short usWidth;
    unsigned short usHeight;
    short sXOffset;
    short sYOffset;
    short sXStart;
    short sYStart;
    unsigned short *pusImage;
}
tJPEGInst
```

Members:

usWidth The width of the decompressed JPEG image in pixels.

usHeight The height of the decompressed JPEG image in lines.

sXOffset The current X image display offset (pan).

sYOffset The current Y image display offset (scan).

sXStart The x coordinate of the screen position corresponding to the last scrolling calculation check for a JPEGCanvas type widget.

sYStart The y coordinate of the screen position corresponding to the last scrolling calculation check for a JPEGCanvas type widget.

pusImage A pointer to the SDRAM buffer containing the decompressed JPEG image.

Description:

The structure containing workspace fields used by the JPEG widget in decompressing and displaying the JPEG image. This structure must not be modified by the application using the widget.

8.2.1.2 tJPEGWidget

Definition:

```
typedef struct
{
    tWidget sBase;
    unsigned long ulStyle;
    unsigned long ulFillColor;
    unsigned long ulOutlineColor;
    unsigned long ulTextColor;
    const tFont *pFont;
    const char *pcText;
    const unsigned char *pucImage;
    unsigned long ulImageLen;
    unsigned char ucBorderWidth;
    void (*pfnOnClick)(tWidget *pWidget);
    void (*pfnOnScroll)(tWidget *pWidget,
                        short sX,
                        short sY);

    tJPEGInst *psJPEGInst;
}
tJPEGWidget
```

Members:

sBase The generic widget information.

ulStyle The style for this widget. This is a set of flags defined by JW_STYLE_xxx.

ulFillColor The 24-bit RGB color used to fill this JPEG widget, if JW_STYLE_FILL is selected.

ulOutlineColor The 24-bit RGB color used to outline this JPEG widget, if JW_STYLE_OUTLINE is selected.

ulTextColor The 24-bit RGB color used to draw text on this JPEG widget, if JW_STYLE_TEXT is selected.

pFont A pointer to the font used to render the JPEG widget text, if JW_STYLE_TEXT is selected.

pcText A pointer to the text to draw on this JPEG widget, if JW_STYLE_TEXT is selected.

pucImage A pointer to the compressed JPEG image to be drawn onto this widget. If NULL, the widget will be filled with the provided background color if painted.

ulImageLen The number of bytes of compressed data in the image pointed to by pucImage.

ucBorderWidth The width of the border to be drawn around the widget. This is ignored if JW_STYLE_OUTLINE is not set.

pfnOnClick A pointer to the function to be called when the button is pressed. This is ignored if JW_STYLE_BUTTON is not set.

pfnOnScroll A pointer to the function to be called if the user scrolls the displayed image. This is ignored if JW_STYLE_BUTTON is set.

psJPEGInst The following structure contains all the workspace fields required by the widget. The client must initialize this with a valid pointer to a read/write structure.

Description:

The structure that describes a JPEG widget.

8.2.2 Define Documentation

8.2.2.1 JPEGButton

Declares an initialized variable containing a JPEG button data structure.

Definition:

```
#define JPEGButton(sName,  
                  pParent,  
                  pNext,  
                  pChild,  
                  pDisplay,  
                  lX,  
                  lY,  
                  lWidth,  
                  lHeight,  
                  ulStyle,  
                  ulFillColor,  
                  ulOutlineColor,  
                  ulTextColor,  
                  pFont,  
                  pcText,  
                  pucImage,  
                  ulImgLen,  
                  ucBorderWidth,
```

```
pfnOnClick,  
psInst)
```

Parameters:

- sName** is the name of the variable to be declared.
- pParent** is a pointer to the parent widget.
- pNext** is a pointer to the sibling widget.
- pChild** is a pointer to the first child widget.
- pDisplay** is a pointer to the display on which to draw the push button.
- IX** is the X coordinate of the upper left corner of the JPEG widget.
- IY** is the Y coordinate of the upper left corner of the JPEG widget.
- IWidth** is the width of the JPEG widget.
- IHeight** is the height of the JPEG widget.
- ulStyle** is the style to be applied to the JPEG widget.
- ulFillColor** is the color used to fill in the JPEG widget.
- ulOutlineColor** is the color used to outline the JPEG widget.
- ulTextColor** is the color used to draw text on the JPEG widget.
- pFont** is a pointer to the font to be used to draw text on the push button.
- pcText** is a pointer to the text to draw on this JPEG widget.
- puclmage** is a pointer to the compressed image to draw on this JPEG widget.
- ullmgLen** is the length of the data pointed to by puclmage.
- ucBorderWidth** is the width of the border to paint if `JW_STYLE_OUTLINE` is specified.
- pfnOnClick** is a pointer to the function that is called when the JPEG button is pressed or released.
- psInst** is a pointer to a read/write `tJPEGInst` structure that the widget can use for workspace.

Description:

This macro provides an initialized JPEG button widget data structure, which can be used to construct the widget tree at compile time in global variables (as opposed to run-time via function calls).

A JPEG button displays an image centered within the widget area and sends `OnClick` messages to the client whenever a user presses or releases the touchscreen within the widget area (depending upon the state of the `JW_STYLE_RELEASE_NOTIFY` style flag). A JPEG button does not support image scrolling.

ulStyle is the logical OR of the following:

- **JW_STYLE_OUTLINE** to indicate that the JPEG widget should be outlined.
- **JW_STYLE_TEXT** to indicate that the JPEG widget should have text drawn on it (using **pFont** and **pcText**).
- **JW_STYLE_FILL** to indicate that the JPEG widget should have its background filled with color (specified by **ulFillColor**).
- **JW_STYLE_SCROLL** to indicate that the JPEG widget should be redrawn automatically each time the pointer is moved (touchscreen is dragged) rather than waiting for the gesture to end then redrawing once. A client may chose to omit this style flag and call `WidgetPaint()` from within the `pfnOnScroll` callback at a rate deemed acceptable for the application.
- **JW_STYLE_LOCKED** to indicate that the JPEG widget should ignore all user input and merely display the image. If this flag is set, **JW_STYLE_SCROLL** is ignored.
- **JW_STYLE_RELEASE_NOTIFY** to indicate that the callback should be made when the button-style widget is released. If absent, the callback is called when the widget is initially pressed. This style flag is ignored unless **JW_STYLE_BUTTON** is specified.

Returns:

Nothing; this is not a function.

8.2.2.2 JPEGCanvas

Declares an initialized variable containing a JPEG canvas data structure.

Definition:

```
#define JPEGCanvas(sName,  
                  pParent,  
                  pNext,  
                  pChild,  
                  pDisplay,  
                  lX,  
                  lY,  
                  lWidth,  
                  lHeight,  
                  ulStyle,  
                  ulFillColor,  
                  ulOutlineColor,  
                  ulTextColor,  
                  pFont,  
                  pcText,  
                  pucImage,  
                  ulImgLen,  
                  ucBorderWidth,  
                  pfnOnScroll,  
                  psInst)
```

Parameters:

sName is the name of the variable to be declared.

pParent is a pointer to the parent widget.

pNext is a pointer to the sibling widget.

pChild is a pointer to the first child widget.

pDisplay is a pointer to the display on which to draw the push button.

lX is the X coordinate of the upper left corner of the JPEG widget.

lY is the Y coordinate of the upper left corner of the JPEG widget.

lWidth is the width of the JPEG widget.

lHeight is the height of the JPEG widget.

ulStyle is the style to be applied to the JPEG widget.

ulFillColor is the color used to fill in the JPEG widget.

ulOutlineColor is the color used to outline the JPEG widget.

ulTextColor is the color used to draw text on the JPEG widget.

pFont is a pointer to the font to be used to draw text on the push button.

pcText is a pointer to the text to draw on this JPEG widget.

pucImage is a pointer to the compressed image to draw on this JPEG widget.

ulImgLen is the length of the data pointed to by pucImage.

ucBorderWidth is the width of the border to paint if JW_STYLE_OUTLINE is specified.

pfnOnScroll is a pointer to the function that is called when the user drags a finger or stylus across the widget area. The values reported as parameters to the callback indicate the number of pixels of offset from center that will be applied to the image next time it is redrawn.

psInst is a pointer to a read/write **tJPEGInst** structure that the widget can use for workspace.

Description:

This macro provides an initialized JPEG canvas widget data structure, which can be used to construct the widget tree at compile time in global variables (as opposed to run-time via function calls).

A JPEG canvas widget acts as an image display surface. User input via the touch screen controls the image positioning, allowing scrolling of a large image within a smaller area of the display. Image redraw can either be carried out automatically whenever scrolling is required or can be delegated to the application via the OnScroll callback which is called whenever the user requests an image position change.

ulStyle is the logical OR of the following:

- **JW_STYLE_OUTLINE** to indicate that the JPEG widget should be outlined.
- **JW_STYLE_TEXT** to indicate that the JPEG widget should have text drawn on it (using *pFont* and *pcText*).
- **JW_STYLE_FILL** to indicate that the JPEG widget should have its background filled with color (specified by *ulFillColor*).
- **JW_STYLE_SCROLL** to indicate that the JPEG widget should be redrawn automatically each time the pointer is moved (touchscreen is dragged) rather than waiting for the gesture to end then redrawing once. A client may chose to omit this style flag and call `WidgetPaint()` from within the *pfnOnScroll* callback at a rate deemed acceptable for the application.
- **JW_STYLE_LOCKED** to indicate that the JPEG widget should ignore all user input and merely display the image. If this flag is set, **JW_STYLE_SCROLL** is ignored.

Returns:

Nothing; this is not a function.

8.2.2.3 JPEGWidgetClickCallbackSet

Sets the function to call when the button-style widget is pressed.

Definition:

```
#define JPEGWidgetClickCallbackSet (pWidget,  
                                   pfnOnClick)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

pfnOnClick is a pointer to the function to call.

Description:

This function sets the function to be called when this widget is pressed (assuming **JW_STYLE_BUTTON** is set). The supplied function is called when the button is pressed if **JW_STYLE_RELEASE_NOTIFY** is clear or when the button is released if this style flag is set.

Returns:

None.

8.2.2.4 JPEGWidgetFillColorSet

Sets the fill color of a JPEG widget.

Definition:

```
#define JPEGWidgetFillColorSet (pWidget,  
                                ulColor)
```

Parameters:

pWidget is a pointer to the JPEG widget to be modified.

ulColor is the 24-bit RGB color to use to fill the JPEG widget.

Description:

This function changes the color used to fill the JPEG widget on the display. The display is not updated until the next paint request.

Returns:

None.

8.2.2.5 JPEGWidgetFillOff

Disables background color fill for JPEG widget.

Definition:

```
#define JPEGWidgetFillOff (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function disables background color fill for a JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.6 JPEGWidgetFillOn

Enables background color fill for a JPEG widget.

Definition:

```
#define JPEGWidgetFillOn (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function enables background color fill for JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.7 JPEGWidgetFontSet

Sets the font for a JPEG widget.

Definition:

```
#define JPEGWidgetFontSet (pWidget,  
                           pFnt)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

pFnt is a pointer to the font to use to draw text on the push button.

Description:

This function changes the font used to draw text on the JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.8 JPEGWidgetImageOff

Disables the image on a JPEG widget.

Definition:

```
#define JPEGWidgetImageOff (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function disables the drawing of an image on a JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.9 JPEGWidgetLock

Locks a JPEG widget making it ignore pointer input.

Definition:

```
#define JPEGWidgetLock (pWidget)
```

Parameters:

pWidget is a pointer to the widget to modify.

Description:

This function locks a JPEG widget and makes it ignore all pointer input. When locked, a widget acts as a passive canvas.

Returns:

None.

8.2.2.10 JPEGWidgetOutlineColorSet

Sets the outline color of a JPEG widget.

Definition:

```
#define JPEGWidgetOutlineColorSet (pWidget,  
                                   ulColor)
```

Parameters:

pWidget is a pointer to the JPEG widget to be modified.
ulColor is the 24-bit RGB color to use to outline the widget.

Description:

This function changes the color used to outline the JPEG widget on the display. The display is not updated until the next paint request.

Returns:

None.

8.2.2.11 JPEGWidgetOutlineOff

Disables outlining of a JPEG widget.

Definition:

```
#define JPEGWidgetOutlineOff (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function disables the outlining of a JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.12 JPEGWidgetOutlineOn

Enables outlining of a JPEG widget.

Definition:

```
#define JPEGWidgetOutlineOn (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function enables the outlining of a JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.13 JPEGWidgetOutlineWidthSet

Sets the outline width of a JPEG widget.

Definition:

```
#define JPEGWidgetOutlineWidthSet (pWidget,  
                                   ucWidth)
```

Parameters:

pWidget is a pointer to the JPEG widget to be modified.

ucWidth This function changes the width of the border around the JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.14 #define JPEGWidgetScrollCallbackSet(pWidget, pfnOnScrl)

Sets the function to call when the JPEG image is scrolled.

Parameters:

pWidget is a pointer to the JPEG widget to modify.

pfnOnScrl is a pointer to the function to call.

Description:

This function sets the function to be called when this widget is scrolled by dragging a finger or stylus over the image area (assuming that **JW_STYLE_BUTTON** is clear).

Returns:

None.

8.2.2.15 JPEGWidgetStruct

Declares an initialized JPEG image widget data structure.

Definition:

```
#define JPEGWidgetStruct (pParent,  
                          pNext,  
                          pChild,  
                          pDisplay,  
                          lX,  
                          lY,  
                          lWidth,  
                          lHeight,  
                          ulStyle,  
                          ulFillColor,  
                          ulOutlineColor,  
                          ulTextColor,  
                          pFont,  
                          pcText,
```

```
pucImage,  
ulImgLen,  
ucBorderWidth,  
pfnOnClick,  
pfnOnScroll,  
psInst)
```

Parameters:

pParent is a pointer to the parent widget.
pNext is a pointer to the sibling widget.
pChild is a pointer to the first child widget.
pDisplay is a pointer to the display on which to draw the push button.
IX is the X coordinate of the upper left corner of the JPEG widget.
IY is the Y coordinate of the upper left corner of the JPEG widget.
IWidth is the width of the JPEG widget.
IHeight is the height of the JPEG widget.
ulStyle is the style to be applied to the JPEG widget.
ulFillColor is the color used to fill in the JPEG widget.
ulOutlineColor is the color used to outline the JPEG widget.
ulTextColor is the color used to draw text on the JPEG widget.
pFont is a pointer to the font to be used to draw text on the push button.
pcText is a pointer to the text to draw on this JPEG widget.
pucImage is a pointer to the compressed image to draw on this JPEG widget.
ulImgLen is the length of the data pointed to by pucImage.
ucBorderWidth is the width of the border to paint if **JW_STYLE_OUTLINE** is specified.
pfnOnClick is a pointer to the function that is called when the JPEG button is pressed assuming **JW_STYLE_BUTTON** is specified.
pfnOnScroll is a pointer to the function that is called when the image is scrolled assuming **JW_STYLE_BUTTON** is not specified.
psInst is a pointer to a read/write **tJPEGInst** structure that the widget can use for workspace.

Description:

This macro provides an initialized jpeg image widget data structure, which can be used to construct the widget tree at compile time in global variables (as opposed to run-time via function calls). This must be assigned to a variable, such as:

```
tJPEGWidget g_sImageButton = JPEGWidgetStruct(...);
```

Or, in an array of variables:

```
tJPEGWidget g_psImageButtons[] =  
{  
    JPEGWidgetStruct(...),  
    JPEGWidgetStruct(...)  
};
```

ulStyle is the logical OR of the following:

- **JW_STYLE_OUTLINE** to indicate that the JPEG widget should be outlined.

- **JW_STYLE_BUTTON** to indicate that the JPEG widget should act as a button and that calls should be made to `pfnOnClick` when it is pressed or released (depending upon the state of **JW_STYLE_RELEASE_NOTIFY**). If absent, the widget acts as a canvas which allows the image, if larger than the widget display area to be scrolled by dragging a finger on the touchscreen. In this case, the `pfnOnScroll` callback will be called when any scrolling is needed.
- **JW_STYLE_TEXT** to indicate that the JPEG widget should have text drawn on it (using `pFont` and `pcText`).
- **JW_STYLE_FILL** to indicate that the JPEG widget should have its background filled with color (specified by `ulFillColor`).
- **JW_STYLE_SCROLL** to indicate that the JPEG widget should be redrawn automatically each time the pointer is moved (touchscreen is dragged) rather than waiting for the gesture to end then redrawing once. A client may chose to omit this style flag and call `WidgetPaint()` from within the `pfnOnScroll` callback at a rate deemed acceptable for the application.
- **JW_STYLE_LOCKED** to indicate that the JPEG widget should ignore all user input and merely display the image. If this flag is set, **JW_STYLE_SCROLL** is ignored.
- **JW_STYLE_RELEASE_NOTIFY** to indicate that the callback should be made when the button-style widget is released. If absent, the callback is called when the widget is initially pressed. This style flag is ignored unless **JW_STYLE_BUTTON** is specified.

Returns:

Nothing; this is not a function.

8.2.2.16 JPEGWidgetTextColorSet

Sets the text color of a JPEG widget.

Definition:

```
#define JPEGWidgetTextColorSet (pWidget,  
                               ulColor)
```

Parameters:

pWidget is a pointer to the JPEG widget to be modified.

ulColor is the 24-bit RGB color to use to draw text on the push button.

Description:

This function changes the color used to draw text on the JPEG widget on the display. The display is not updated until the next paint request.

Returns:

None.

8.2.2.17 JPEGWidgetTextOff

Disables the text on a JPEG widget.

Definition:

```
#define JPEGWidgetTextOff (pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget to modify.

Description:

This function disables the drawing of text on a JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.18 JPEGWidgetTextOn

Returns:

None.

8.2.2.19 #define JPEGWidgetTextSet(pWidget, pcTxt)

Definition:

```
#define JPEGWidgetTextOn(pWidget)
```

Description:

Changes the text drawn on a JPEG widget.

Parameters:

pWidget is a pointer to the JPEG widget to be modified.

pcTxt is a pointer to the text to draw onto the JPEG widget.

This function changes the text that is drawn onto the JPEG widget. The display is not updated until the next paint request.

Returns:

None.

8.2.2.20 JPEGWidgetUnlock

Unlocks a JPEG widget making it pay attention to pointer input.

Definition:

```
#define JPEGWidgetUnlock(pWidget)
```

Parameters:

pWidget is a pointer to the widget to modify.

Description:

This function unlocks a JPEG widget. When unlocked, a JPEG widget will respond to pointer input by scrolling or making press callbacks depending upon the style flags and the image it is currently displaying.

Returns:

None.

8.2.2.21 JW_STYLE_BUTTON

Definition:

```
#define JW_STYLE_BUTTON
```

Description:

This flag indicates that the widget should act as a button rather than as a display surface.

8.2.2.22 JW_STYLE_FILL

Definition:

```
#define JW_STYLE_FILL
```

Description:

This flag indicates that the JPEG widget's background area should be filled with color even when there is an image to display.

8.2.2.23 JW_STYLE_LOCKED

Definition:

```
#define JW_STYLE_LOCKED
```

Description:

This flag indicates that the JPEG widget should ignore all touchscreen activity.

8.2.2.24 JW_STYLE_OUTLINE

Definition:

```
#define JW_STYLE_OUTLINE
```

Description:

This flag indicates that the widget should be outlined.

8.2.2.25 JW_STYLE_PRESSED

Definition:

```
#define JW_STYLE_PRESSED
```

Description:

This flag indicates that the JPEG widget is pressed.

8.2.2.26 JW_STYLE_RELEASE_NOTIFY

Definition:

```
#define JW_STYLE_RELEASE_NOTIFY
```

Description:

This flag indicates that the JPEG widget callback should be made when the widget is released rather than when it is pressed. This style flag is ignored if JW_STYLE_BUTTON is not set.

8.2.2.27 JW_STYLE_SCROLL

Definition:

```
#define JW_STYLE_SCROLL
```

Description:

This flag indicates that the JPEG widget's image should be repainted as the user scrolls over it. This is CPU intensive but looks better than the alternative which only repaints the image when the user ends their touchscreen drag.

8.2.2.28 JW_STYLE_TEXT

Definition:

```
#define JW_STYLE_TEXT
```

Description:

This flag indicates that the JPEG widget should have text drawn on it.

8.2.3 Function Documentation

8.2.3.1 JPEGWidgetImageDecompress

Decompresses the image associated with a JPEG widget.

Prototype:

```
long  
JPEGWidgetImageDecompress (tWidget *pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget whose image is to be decompressed.

Description:

This function must be called by the client for any JPEG widget whose compressed data pointer is initialised using the JPEGCanvas, JPEGButton or JPEGWidgetStruct macros. It decompresses the image and readies it for display.

This function must NOT be used if the widget already holds a decompressed image (i.e. if this function has been called before or if a prior call has been made to [JPEGWidgetImageSet\(\)](#) without a later call to [JPEGWidgetImageDiscard\(\)](#)) since this will result in a serious memory leak.

The client is responsible for repainting the widget after this call is made.

Returns:

Returns 0 on success. Any other return code indicates failure.

8.2.3.2 JPEGWidgetImageDiscard

Frees any decompressed image held by the widget.

Prototype:

```
void  
JPEGWidgetImageDiscard(tWidget *pWidget)
```

Parameters:

pWidget is a pointer to the JPEG widget whose image is to be discarded.

Description:

This function frees any decompressed image that is currently held by the widget and returns the memory it was occupying to the SDRAM heap. After this call, [JPEGWidgetImageDecompress\(\)](#) may be called to re-decompress the same image or [JPEGWidgetImageSet\(\)](#) can be called to have the widget decompress a new image.

Returns:

None.

8.2.3.3 JPEGWidgetImageSet

Pass a new compressed image to the widget.

Prototype:

```
long  
JPEGWidgetImageSet(tWidget *pWidget,  
                   const unsigned char *pImg,  
                   unsigned long ulImgLen)
```

Parameters:

pWidget is a pointer to the JPEG widget whose image is to be set.

pImg is a pointer to the compressed JPEG data for the image.

ulImgLen is the number of bytes of data pointed to by pImg.

Description:

This function is used to change the image displayed by a JPEG widget. It is safe to call it when the widget is already displaying an image since it will free any existing image before decompressing the new one. The client is responsible for repainting the widget after this call is made.

Returns:

Returns 0 on success. Any other return code indicates failure.

8.2.3.4 JPEGWidgetInit

Initializes a JPEG widget.

Prototype:

```
void  
JPEGWidgetInit(tJPEGWidget *pWidget,
```

```
const tDisplay *pDisplay,  
long lX,  
long lY,  
long lWidth,  
long lHeight)
```

Parameters:

pWidget is a pointer to the JPEG widget to initialize.

pDisplay is a pointer to the display on which to draw the push button.

lX is the X coordinate of the upper left corner of the JPEG widget.

lY is the Y coordinate of the upper left corner of the JPEG widget.

lWidth is the width of the JPEG widget.

lHeight is the height of the JPEG widget.

Description:

This function initializes the provided JPEG widget. The widget position is set and all styles and parameters set to 0. The caller must make use of the various widget functions to set any required parameters after making this call.

Returns:

None.

8.2.3.5 JPEGWidgetMsgProc

Handles messages for a JPEG widget.

Prototype:

```
long  
JPEGWidgetMsgProc(tWidget *pWidget,  
                  unsigned long ulMsg,  
                  unsigned long ulParam1,  
                  unsigned long ulParam2)
```

Parameters:

pWidget is a pointer to the JPEG widget.

ulMsg is the message.

ulParam1 is the first parameter to the message.

ulParam2 is the second parameter to the message.

Description:

This function receives messages intended for this JPEG widget and processes them accordingly. The processing of the message varies based on the message in question.

Unrecognized messages are handled by calling WidgetDefaultMsgProc().

Returns:

Returns a value appropriate to the supplied message.

8.3 Programming Example

The following example shows how to initialize a JPEGCanvas widget. The sample application boards/rdk-idm-sbc/showjpeg provides a working example of the use of this widget.

```
...

//*****
//
// Forward references
//
//*****
extern tCanvasWidget g_sBackground;

//*****
//
// Workspace for the JPEG canvas widget.
//
//*****
tJPEGInst g_sJPEGInst;

//*****
//
// The JPEG canvas widget used to hold the decompressed JPEG image and
// display it. This is a simple JPEG canvas which will decompress and display
// the image contained in buffer g_pucJPEGImage in a 320x215 pixel control with
// a single pixel white outline. Style flag JW_STYLE_SCROLL enables automatic
// redraw based on user touchscreen scrolling. This is rather CPU intensive
// so it may be better to remove this style flag and install an OnScroll
// callback that can be used to pace the redraws. This is demonstrated in the
// showjpeg example application.
//
//*****
#define IMAGE_LEFT      0
#define IMAGE_TOP       25
#define IMAGE_WIDTH     320
#define IMAGE_HEIGHT    215
JPEGCanvas(g_sImage, &g_sBackground, 0, 0,
           &g_sKitronix320x240x16_SSD2119, IMAGE_LEFT, IMAGE_TOP, IMAGE_WIDTH,
           IMAGE_HEIGHT, (JW_STYLE_OUTLINE | JW_STYLE_SCROLL), ClrBlack,
           ClrWhite, 0, 0, 0, g_pucJPEGImage, sizeof(g_pucJPEGImage), 1, 0,
           &g_sJPEGInst);

...

int
main(void)
{
    tBoolean bRetcode;
    int iRetcode;

    //
    // Set the system clock to run at 50MHz from the PLL.
    //
    SysCtlClockSet(SYSCTL_SYSDIV_4 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                   SYSCTL_XTAL_16MHZ);

    //
    // Enable Interrupts
    //
    IntMasterEnable();

    //
    // Initialize the SDRAM controller and heap.
```

```
//
SDRAMInit(1, (EPI_SDRAM_CORE_FREQ_50_100 | EPI_SDRAM_FULL_POWER |
              EPI_SDRAM_SIZE_64MBIT), 1024);

//
// Initialize the display driver.
//
Kitronix320x240x16_SSD2119Init();

//
// Initialize the touch screen driver.
//
TouchScreenInit();

//
// Set the touch screen event handler.
//
TouchScreenCallbackSet(WidgetPointerMessage);

//
// Add the compile-time defined widgets to the widget tree.
//
WidgetAdd(WIDGET_ROOT, (tWidget *)&g_sBackground);

//
// Decompress the image we linked to the JPEG canvas widget
//
iRetcode = JPEGWidgetImageDecompress((tWidget *)&g_sImage);

//
// Was the decompression successful?
//
if(iRetcode != 0)
{
    while(1)
    {
        //
        // Something went wrong during the decompression of the JPEG
        // image. Hang here pending investigation.
        //
    }
}

//
// Issue the initial paint request to the widgets.
//
WidgetPaint(WIDGET_ROOT);

//
// Enter an infinite loop for reading and processing touchscreen input
// from the user.
//
while(1)
{
    WidgetMessageQueueProcess();
}
}
```

9 Display Driver

Introduction	61
API Functions	61
Programming Example	62

9.1 Introduction

The display driver offers a standard interface to access display functions on the Kitronix 320x240 QVGA screen and is used by the Stellaris Graphics Library and widget manager. In addition to providing the `tDisplay` structure required by the graphics library, the display driver also provides an API for initializing the display.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `kitronix320x240x16_ssd2119_idm_sbc.c` containing the source code and `kitronix320x240x16_ssd2119_idm_sbc.h` containing the API definitions for use by applications.

9.2 API Functions

Functions

- void [Kitronix320x240x16_SSD2119BacklightOff](#) (void)
- void [Kitronix320x240x16_SSD2119BacklightOn](#) (unsigned char ucBrightness)
- void [Kitronix320x240x16_SSD2119Init](#) (void)

Variables

- const tDisplay [g_sKitronix320x240x16_SSD2119](#)

9.2.1 Function Documentation

9.2.1.1 Kitronix320x240x16_SSD2119BacklightOff

Turns off the backlight.

Prototype:

```
void  
Kitronix320x240x16_SSD2119BacklightOff (void)
```

Description:

This function turns off the backlight on the display.

Returns:

None.

9.2.1.2 Kitronix320x240x16_SSD2119BacklightOn

Turns on the backlight.

Prototype:

```
void  
Kitronix320x240x16_SSD2119BacklightOn(unsigned char ucBrightness)
```

Parameters:

ucBrightness is the brightness of the backlight with 0xFF representing "on at full brightness" and 0x00 representing "off".

Description:

This function sets the brightness of the display backlight.

Returns:

None.

9.2.1.3 Kitronix320x240x16_SSD2119Init

Initializes the display driver.

Prototype:

```
void  
Kitronix320x240x16_SSD2119Init(void)
```

Description:

This function initializes the SSD2119 display controller on the panel, preparing it to display data.

Returns:

None.

9.2.2 Variable Documentation

9.2.2.1 g_sKitronix320x240x16_SSD2119

Definition:

```
const tDisplay g_sKitronix320x240x16_SSD2119
```

Description:

The display structure that describes the driver for the Kitronix K350QVG-V1-F TFT panel with an SSD2119 controller.

9.3 Programming Example

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

```
tContext sContext;

//
// Initialize the display.
//
Kitronix320x240x16_SSD2119Init();

//
// Initialize a graphics library drawing context.
//
GrContextInit(&sContext, &g_sKitronix320x240x16_SSD2119);
```


10 Serial Flash Driver

Introduction	65
API Functions	65
Programming Example	70

10.1 Introduction

This driver provides a low level interface allowing data to be written to and read from the SSI-connected 1MB serial flash device found on the board.

This device shares the SSI bus with the SDCard so care must be taken when using these two devices to ensure no contention since this driver does not contain code to arbitrate for ownership of the SSI bus.

This driver is located in `boards/rdk-idm-sbc/drivers`, with `ssiflash.c` containing the source code and `ssiflash.h` containing the API definitions for use by applications.

10.2 API Functions

Functions

- tBoolean [SSIFlashBlockErase](#) (unsigned long ulAddress, tBoolean bSync)
- unsigned long [SSIFlashBlockSizeGet](#) (void)
- tBoolean [SSIFlashChipErase](#) (tBoolean bSync)
- unsigned long [SSIFlashChipSizeGet](#) (void)
- tBoolean [SSIFlashIDGet](#) (unsigned char *pucManufacturer, unsigned char *pucDevice)
- tBoolean [SSIFlashInit](#) (void)
- tBoolean [SSIFlashIsBusy](#) (void)
- unsigned long [SSIFlashRead](#) (unsigned long ulAddress, unsigned long ulLength, unsigned char *pucDst)
- tBoolean [SSIFlashSectorErase](#) (unsigned long ulAddress, tBoolean bSync)
- unsigned long [SSIFlashSectorSizeGet](#) (void)
- unsigned long [SSIFlashWrite](#) (unsigned long ulAddress, unsigned long ulLength, unsigned char *pucSrc)

10.2.1 Function Documentation

10.2.1.1 SSIFlashBlockErase

Erases the contents of a single serial flash block.

Prototype:

```
tBoolean
```

```
SSIFlashBlockErase(unsigned long ulAddress,  
                   tBoolean bSync)
```

Parameters:

ulAddress is the start address of the block which is to be erased. This value must be an integer multiple of the block size returned by [SSIFlashBlockSizeGet\(\)](#).

bSync should be set to **true** if the function is to block until the erase operation is complete or **false** to return immediately after the operation is started.

Description:

This function erases a single block of the serial flash, returning all bytes in that block to their erased value of 0xFF. The block size and, hence, start address granularity can be determined by calling [SSIFlashBlockSizeGet\(\)](#).

The function may be synchronous (*bSync* set to **true**) or asynchronous (*bSync* set to **false**). If asynchronous, the caller is responsible for ensuring that no further serial flash operations are requested until the erase operation has completed. The state of the device may be queried by calling [SSIFlashIsBusy\(\)](#).

Three options for erasing are provided. Sectors provide the smallest erase granularity, blocks provide the option of erasing a larger section of the device in one operation and, finally, the whole device may be erased in a single operation via [SSIFlashChipErase\(\)](#).

Note:

This operation will take between 400mS and 1000mS to complete. If the *bSync* parameter is set to **true**, this function will, therefore, not return for a significant period of time.

Returns:

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

10.2.1.2 SSIFlashBlockSizeGet

Returns the size of a block for this device.

Prototype:

```
unsigned long  
SSIFlashBlockSizeGet(void)
```

Description:

This function returns the size of an eraseable block for the serial flash device. All addresses passed to [SSIFlashBlockErase\(\)](#) must be aligned on a block boundary.

Returns:

Returns the number of bytes in a block.

10.2.1.3 SSIFlashChipErase

Erases the entire serial flash device.

Prototype:

```
tBoolean  
SSIFlashChipErase(tBoolean bSync)
```

Parameters:

bSync should be set to **true** if the function is to block until the erase operation is complete or **false** to return immediately after the operation is started.

Description:

This function erases the entire serial flash device, returning all bytes in the device to their erased value of 0xFF.

The function may be synchronous (*bSync* set to **true**) or asynchronous (*bSync* set to **false**). If asynchronous, the caller is responsible for ensuring that no further serial flash operations are requested until the erase operation has completed. The state of the device may be queried by calling [SSIFlashIsBusy\(\)](#).

Note:

This operation will take between 6 and 10 seconds to complete. If the *bSync* parameter is set to **true**, this function will, therefore, not return for a significant period of time.

Returns:

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

10.2.1.4 SSIFlashChipSizeGet

Returns the total amount of storage offered by this device.

Prototype:

```
unsigned long  
SSIFlashChipSizeGet(void)
```

Description:

This function returns the size of the programmable area provided by the attached SSI flash device.

Returns:

Returns the number of bytes in the device.

10.2.1.5 SSIFlashIDGet

Returns the manufacturer and device IDs for the attached serial flash.

Prototype:

```
tBoolean  
SSIFlashIDGet(unsigned char *pucManufacturer,  
               unsigned char *pucDevice)
```

Parameters:

pucManufacturer points to storage which will be written with the manufacturer ID of the attached serial flash device.

pucDevice points to storage which will be written with the device ID of the attached serial flash device.

Description:

This function may be used to determine the manufacturer and device IDs of the attached serial flash device.

Returns:

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

10.2.1.6 SSIFlashInit

Initializes the SSI port and determines if the serial flash is available.

Prototype:

```
tBoolean  
SSIFlashInit(void)
```

Description:

This function must be called prior to any other function offered by the serial flash driver. It configures the SSI port to run in mode 0 at 10MHz and queries the ID of the serial flash device to ensure that it is available.

Note:

SSI0 is shared between the serial flash and the SDCard on the rdk-idm-sbc board. Two independent GPIOs are used to provide chip selects for these two devices but care must be taken when using both in a single application, especially during initialization of the SDCard when the SSI clock rate must initially be set to 400KHz and later increased. Since both the SSI flash and SDCard drivers initialize the SSI peripheral, application writers must be aware of the possible contention and ensure that they do not allow the possibility of two different interrupt handlers or execution threads from attempting to access both peripherals simultaneously.

This driver assumes that the application is aware of the possibility of contention and has been designed with this in mind. Other than disabling the SDCard when an attempt is made to access the serial flash, no code is included here to arbitrate for ownership of the SSI peripheral.

Returns:

Returns **true** on success or **false** if an error is reported or the expected serial flash device is not present.

10.2.1.7 SSIFlashIsBusy

Determines if the serial flash is able to accept a new instruction.

Prototype:

```
tBoolean  
SSIFlashIsBusy(void)
```

Description:

This function reads the serial flash status register and determines whether or not the device is currently busy. No new instruction may be issued to the device if it is busy.

Returns:

Returns **true** if the device is busy and unable to receive a new instruction or **false** if it is idle.

10.2.1.8 SSIFlashRead

Reads a block of serial flash into a buffer.

Prototype:

```
unsigned long
SSIFlashRead(unsigned long ulAddress,
              unsigned long ulLength,
              unsigned char *pucDst)
```

Parameters:

ulAddress is the serial flash address of the first byte to be read.

ulLength is the number of bytes of data to read.

pucDst is a pointer to storage for the data read.

Description:

This function reads a contiguous block of data from a given address in the serial flash device into a buffer supplied by the caller.

Returns:

Returns the number of bytes read.

10.2.1.9 SSIFlashSectorErase

Erases the contents of a single serial flash sector.

Prototype:

```
tBoolean
SSIFlashSectorErase(unsigned long ulAddress,
                    tBoolean bSync)
```

Parameters:

ulAddress is the start address of the sector which is to be erased. This value must be an integer multiple of the sector size returned by [SSIFlashSectorSizeGet\(\)](#).

bSync should be set to **true** if the function is to block until the erase operation is complete or **false** to return immediately after the operation is started.

Description:

This function erases a single sector of the serial flash, returning all bytes in that sector to their erased value of 0xFF. The sector size and, hence, start address granularity can be determined by calling [SSIFlashSectorSizeGet\(\)](#).

The function may be synchronous (*bSync* set to **true**) or asynchronous (*bSync* set to **false**). If asynchronous, the caller is responsible for ensuring that no further serial flash operations are requested until the erase operation has completed. The state of the device may be queried by calling [SSIFlashIsBusy\(\)](#).

Three options for erasing are provided. Sectors provide the smallest erase granularity, blocks provide the option of erasing a larger section of the device in one operation and, finally, the whole device may be erased in a single operation via [SSIFlashChipErase\(\)](#).

Note:

This operation will take between 120mS and 250mS to complete. If the *bSync* parameter is set to **true**, this function will, therefore, not return for a significant period of time.

Returns:

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

10.2.1.10 SSIFlashSectorSizeGet

Returns the size of a sector for this device.

Prototype:

```
unsigned long  
SSIFlashSectorSizeGet(void)
```

Description:

This function returns the size of an erasable sector for the serial flash device. All addresses passed to [SSIFlashSectorErase\(\)](#) must be aligned on a sector boundary.

Returns:

Returns the number of bytes in a sector.

10.2.1.11 SSIFlashWrite

Writes a block of data to the serial flash device.

Prototype:

```
unsigned long  
SSIFlashWrite(unsigned long ulAddress,  
               unsigned long ulLength,  
               unsigned char *pucSrc)
```

Parameters:

ulAddress is the first serial flash address to be written.

ulLength is the number of bytes of data to write.

pucSrc is a pointer to the data which is to be written.

Description:

This function writes a block of data into the serial flash device at the given address. It is assumed that the area to be written has previously been erased.

Returns:

Returns the number of bytes written.

10.3 Programming Example

The following example shows how to use the SSIFlash API.

```
int main(void)  
{  
    tBoolean bRetcode;  
    unsigned long ulCount;
```

```
...

//
// Initialize the serial flash device driver.
//
bRetcode = SSIFlashInit();
if(!bRetcode)
{
    //
    // An error occurred! Handle it here.
    //
    while(1);
}

//
// Erase the first block of the flash device. We call this function with
// the bSync parameter set to false so it will return immediately and
// not wait for the operation to complete.
//
bRetcode = SSIFlashBlockErase(0, false);
if(!bRetcode)
{
    //
    // An error occurred! Handle it here.
    //
    while(1);
}

//
// Wait for the erase operation to complete. In "real" code, you would,
// of course, not use an infinite loop here.
//
while(SSIFlashIsBusy());

//
// Write data to the newly erased page.
//
ulCount = SSIFlashWrite(0, DATA_LENGTH, g_pcBuffer1);
if(ulCount != DATA_LENGTH)
{
    //
    // An error occurred! Handle it here.
    //
    while(1);
}

//
// Read the data back into a different buffer.
//
ulCount = SSIFlashRead(0, DATA_LENGTH, g_pcBuffer2);
if(ulCount != DATA_LENGTH)
{
    //
    // An error occurred! Handle it here.
    //
    while(1);
}

...
}
```


11 Wave Audio Driver

Introduction	73
API Functions	73
Programming Example	77

11.1 Introduction

The wave audio driver provides applications with a simple way to play back audio clips stored in the common “.wav” format. The driver supports uncompressed mono or stereo PCM wave data stored in system memory and uses the underlying sound and wm8510 audio DAC drivers.

Functions are provided to parse the wave audio data, start playback, continue playback and query the current playback position.

Applications wishing to use this driver must ensure that they link the sound and wm8510 drivers and also set function [SoundIntHandler\(\)](#) in the interrupt table for the I2S vector.

Audio clips played by this driver may be created in any sound editor application which is capable of outputting uncompressed PCM wave files. These files may be prepared for use in your application by using the `tools/bin/makefsfile` tool with the “-f” command line switch to generate a C array from a single file.

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

11.2 API Functions

Data Structures

- [tWaveHeader](#)

Enumerations

- [tWaveReturnCode](#)

Functions

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

11.2.1 Data Structure Documentation

11.2.1.1 tWaveHeader

Definition:

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

Members:

ulSampleRate Sample rate in bytes per second.
ulAvgByteRate The average byte rate for the wav file.
ulDataSize The size of the wav data in the file.
usBitsPerSample The number of bits per sample.
usFormat The wav file format.
usNumChannels The number of audio channels.

Description:

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

11.2.2 Enumeration Documentation

11.2.2.1 tWaveReturnCode

Description:

Possible return codes from the WaveOpen function.

Enumerators:

WAVE_OK The wav data was parsed successfully.
WAVE_INVALID_RIFF The RIFF information in the wav data is not supported.
WAVE_INVALID_CHUNK The chunk size specified in the wav data is not supported.
WAVE_INVALID_FORMAT The format of the wav data is not supported.

11.2.3 Function Documentation

11.2.3.1 WaveGetTime

Formats a text string showing elapsed and total playing time.

Prototype:

```
void
WaveGetTime(tWaveHeader *pWaveHeader,
```

```
char *pcTime,  
unsigned long ulSize)
```

Parameters:

pWaveHeader is a pointer to the current wave audio clip's header information. This structure will have been populated by a previous call to [WaveOpen\(\)](#).

pcTime points to storage for the returned string.

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

Description:

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

Returns:

None.

11.2.3.2 WaveOpen

Opens a wav audio clip and parses its header information.

Prototype:

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

Parameters:

pulAddress is a pointer to the start of the wave audio clip data in memory.

pWaveHeader points to a [tWaveHeader](#) structure that will be populated based on values parsed from the wave audio clip data.

Description:

This function should be used to parse a wave audio clip in preparation for playback. If **WAVE_OK** is returned, the audio data is valid and in a format that is supported by the driver. Valid wave audio clips must contain uncompressed mono or stereo PCM samples.

Returns:

Returns **WAVE_OK** on success, **WAVE_INVALID_RIFF** if RIFF information is not supported, **WAVE_INVALID_CHUNK** if chunk size is not supported or **WAVE_INVALID_FORMAT** if data format is not supported.

11.2.3.3 WavePlaybackStatus

Returns the current playback status of the wave audio clip.

Prototype:

```
tBoolean  
WavePlaybackStatus(void)
```

Description:

This function may be called to determine whether a wave audio clip is currently playing.

Returns:

Returns **true** if a clip is currently playing or **false** if no clip is playing.

11.2.3.4 WavePlayContinue

Continues playback of an audio clip previously passed to [WavePlayStart\(\)](#).

Prototype:

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

Parameters:

pWaveHeader points to the structure containing information on the audio clip being played.

Description:

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

Returns:

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

11.2.3.5 WavePlayStart

Starts playback of an audio clip.

Prototype:

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

Parameters:

pWaveHeader points to a structure containing information on the wave audio data that is to be played.

Description:

This function will start playback of the wave audio data passed in via the psWaveHeader parameter. The [WaveOpen\(\)](#) function should previously have been called to parse the wave data and populate the pWaveHeader structure.

Returns:

None.

11.2.3.6 WaveStop

Stops playback of a wave audio clip.

Prototype:

```
void  
WaveStop(void)
```

Description:

This function may be used to stop playback of any audio clip that is currently being played.

Returns:

None.

11.3 Programming Example

The following example shows how to play a wave audio clip stored in a C array.

```
//  
// A pointer to the external array containing the wave audio data.  
//  
extern const unsigned char g_pucWavSoundEffectData[];  
  
//  
// Main application entry function.  
//  
int main(void)  
{  
    tWaveHeader sSoundEffectHeader;  
    tWaveReturnCode eRetcode;  
    tBoolean bPlaying;  
  
    //  
    // Perform hardware initialization including initializing the sound  
    // driver.  
    //  
    ...  
  
    //  
    // Parse the audio clip data and set up for playback.  
    //  
    eRetcode = WaveOpen(g_pucWavSoundEffectData, &sSoundEffectHeader);  
  
    //  
    // Did we parse the sound successfully?  
    //  
    if(eRetcode == WAVE_OK)  
    {  
        //  
        // Start playback of the wave file data.  
        //  
        WavePlayStart(&sSoundEffectHeader);  
    }  
    else  
    {  
        //  
        // Add error handling code here. This indicates that the wave audio  
        // data is invalid.  
        //  
    }  
  
    //  
    // Main application loop.
```

```
//
while(1)
{
    //
    // Do application stuff.
    //

    ...

    //
    // Process audio data if required. While playing a clip, this must be
    // done at least every 40mS. Calling the function more frequently is
    // perfectly safe.
    //
    if(bPlaying)
    {
        bPlaying = !WavePlayContinue(&sSoundEffectHeader);
    }
}
```

12 Command Line Processing Module

Introduction	79
API Functions	79
Programming Example	81

12.1 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.

12.2 API Functions

Data Structures

- `tCmdLineEntry`

Defines

- `CMDLINE_BAD_CMD`
- `CMDLINE_TOO_MANY_ARGS`

Functions

- `int CmdLineProcess (char *pcCmdLine)`

Variables

- `tCmdLineEntry g_sCmdTable[]`

12.2.1 Data Structure Documentation

12.2.1.1 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.

12.2.2 Define Documentation

12.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

12.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

12.2.3 Function Documentation

12.2.3.1 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.

12.2.4 Variable Documentation

12.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[ ]
```

Description:

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

12.3 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));
}
```

13 CPU Usage Module

Introduction	83
API Functions	83
Programming Example	84

13.1 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.

13.2 API Functions

Functions

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

13.2.1 Function Documentation

13.2.1.1 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.

13.2.1.2 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.

13.3 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();
    }
}
```


14 CRC Module

Introduction	87
API Functions	87
Programming Example	87

14.1 Introduction

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

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

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

14.2 API Functions

14.3 Programming Example

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

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

//
// Fill pucData with some data.
//
for(ulIdx = 0; ulIdx < 256; ulIdx++)
{
    pucData[ulIdx] = ulIdx;
}

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


15 Flash Parameter Block Module

Introduction	89
API Functions	89
Programming Example	91

15.1 Introduction

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

The [FlashPBlockInit\(\)](#) 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. [FlashPBlockGet\(\)](#) and [FlashPBlockSave\(\)](#) 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.

15.2 API Functions

Functions

- unsigned char * [FlashPBlockGet](#) (void)
- void [FlashPBlockInit](#) (unsigned long ulStart, unsigned long ulEnd, unsigned long ulSize)
- void [FlashPBlockSave](#) (unsigned char *pucBuffer)

15.2.1 Function Documentation

15.2.1.1 FlashPBlockGet

Gets the address of the most recent parameter block.

Prototype:

```
unsigned char *  
FlashPBlockGet (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.

15.2.1.2 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.

15.2.1.3 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.

15.3 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);  
}
```


16 File System Wrapper Module

Introduction	93
API Functions	93
Programming Example	96

16.1 Introduction

The file system wrapper module allows several binary file system images and FatFs drives to be mounted simultaneously and referenced as if part of a single file system with each mount point identified by name. This is useful in applications which make use of SDCard and USB Mass Storage Class devices, allowing these to be referenced as, for example "/sdcard" and "/usb" respectively.

Mount points are defined using an array of structures, each entry of which describes a single mount and provides its name and details of the actual file system that is to be used to support that mount.

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

16.2 API Functions

Functions

- void [fs_close](#) (struct fs_file *phFile)
- tBoolean [fs_init](#) (fs_mount_data *psMountPoints, unsigned long ulNumMountPoints)
- tBoolean [fs_map_path](#) (const char *pcPath, char *pcMapped, int iLen)
- fs_file * [fs_open](#) (char *pcName)
- int [fs_read](#) (struct fs_file *phFile, char *pcBuffer, int iCount)
- void [fs_tick](#) (unsigned long ulTickMS)

16.2.1 Function Documentation

16.2.1.1 fs_close

Closes a file.

Prototype:

```
void
fs_close(struct fs_file *phFile)
```

Parameters:

phFile is the handle of the file that is to be closed. This will have been returned by an earlier call to [fs_open\(\)](#).

Description:

This function closes the file identified by *phFile* and frees all resources associated with the file handle.

Returns:

None.

16.2.1.2 fs_init

Initializes the file system wrapper.

Prototype:

```
tBoolean
fs_init(fs_mount_data *psMountPoints,
        unsigned long ulNumMountPoints)
```

Parameters:

psMountPoints points to an array of fs_mount_data structures. Each element in the array maps a top level directory name to a particular file system image or to the FAT file system and a logical drive number.

ulNumMountPoints provides the number of populated elements in the *psMountPoints* array.

Description:

This function should be called to initialize the file system wrapper and provide it with the information required to access the files in multiple file system images via a single filename space.

Each entry in *psMountPoints* describes a top level directory in the unified namespace and indicates to fswrapper where the files for that directory can be found. Each entry can describe either a file system image in system memory or a logical disk handled via the FatFs file system driver.

For example, consider the following 3 entry mount point table:

```
{{ "internal", &g_pcFSImage, 0, NULL, NULL },
{ "sdcard", NULL, 0, SDCardEnable, SDCardDisable },
{ NULL, &g_pcFSDefault, 0, NULL, NULL }}
```

Requests to open file “/internal/index.html” will be handled by attempting to open “/index.html” in the internal file system pointed to by *g_pcFSImage*. Similarly, opening “/sdcard/images/logo.gif” will result in a call to the FAT *f_open* function requesting “0:/images/logo.gif”. If a request to open “index.htm” is received, this is handled by attempting to open “index.htm” in the default internal file system image, *g_pcFSDefault*.

Returns:

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

16.2.1.3 fs_map_path

Maps a path string containing mount point names to a path suitable for use in calls to the FatFs APIs.

Prototype:

```
tBoolean
fs_map_path(const char *pcPath,
            char *pcMapped,
            int iLen)
```

Parameters:

pcPath points to a string containing a path in the namespace defined by the mount information passed to `fs_init()`.

pcMapped points to a buffer into which the mapped path string will be written.

iLen is the size, in bytes, of the buffer pointed to by `pcMapped`.

Description:

This function may be used by applications which want to make use of FatFs functions which are not directly mapped by the fswrapper layer. A path in the namespace defined by the mount points passed to function `fs_init()` is translated to an equivalent path in the FatFs namespace and this may then be used in a direct call to functions such as `f_opendir()` or `f_getfree()`.

Returns:

Returns **true** on success or **false** if `fs_init()` has not been called, if the path provided maps to an internal file system image rather than a FatFs logical drive or if the buffer pointed to by `pcMapped` is too small to fit the output string.

16.2.1.4 fs_open

Opens a file.

Prototype:

```
struct fs_file *  
fs_open(char *pcName)
```

Parameters:

pcName points to a NULL terminated string containing the path and file name to open.

Description:

This function opens a file and returns a handle allowing it to be read.

Returns:

Returns a valid file handle on success or NULL on failure.

16.2.1.5 fs_read

Reads data from an open file.

Prototype:

```
int  
fs_read(struct fs_file *phFile,  
        char *pcBuffer,  
        int iCount)
```

Parameters:

phFile is the handle of the file which is to be read. This will have been returned by a previous call to `fs_open()`.

pcBuffer points to the first byte of the buffer into which the data read from the file will be copied. This buffer must be large enough to hold *iCount* bytes.

iCount is the maximum number of bytes of data that are to be read from the file.

Description:

This function reads the next block of data from the given file into a buffer and returns the number of bytes read or -1 if the end of the file has been reached.

Returns:

Returns the number of bytes read from the file or -1 if the end of the file has been reached and no more data is available.

16.2.1.6 fs_tick

Provides a periodic tick for the file system.

Prototype:

```
void  
fs_tick(unsigned long ulTickMS)
```

Parameters:

ulTickMS is the number of milliseconds which have elapsed since the last time this function was called.

Description:

Applications making use of the file system wrapper with underlying FatFs drives must call this function at least once every 10 milliseconds to provide a time reference for use by the file system. It is typically called in the context of the application's SysTick interrupt handler or from the handler of some other timer interrupt.

If only binary file system images are in use, this function need not be called.

Returns:

None

16.3 Programming Example

The following example shows how to set up the file system wrapper with two mount points, one for a flash-based file system image and the other for a FAT file system on an SDCard.

```
//*****  
//  
// This array describes the various file system mount points. These are passed  
// to the fswrapper module which allows us to use helpful paths to access the  
// various file systems installed via a single namespace.  
//  
// FS_ROOT is a pointer to a binary file system image (as can be generated by  
// the makefsfile utility) located in system flash.  
//  
//*****  
static fs_mount_data g_psMountData[] =  
{  
    {"sdcard", 0, 0, 0, 0}, // SDCard - FAT logical drive 0  
    {"usb", 0, 1, 0, 0}, // USB flash stick - FAT logical drive 1  
    {NULL, (unsigned char *)FS_ROOT, 0, 0, 0} // Default root directory  
};  
  
void AccessFile(void)
```



```
{
    struct fs_file *fhSDCard;
    struct fs_file *fhFlash;

    //
    // Initialize the various file systems we will be using.
    //
    fs_init(g_psMountData, 3);

    //
    // Open a file on the SDCard
    //
    fhSDCard = fs_open("/sdcard/index.htm");

    //
    // Open a file in the flash file system image. The default mount point
    // identified by the "NULL" name pointer in g_psMountData is used if the
    // first directory in the path does not match any other mount point in the
    // table.
    //
    fhFlash = fs_open("/images/logo.gif");

    //
    // Do something useful with the files here.
    //

    //
    // Close our files.
    //
    fs_close(fhFlash);
    fs_close(fhSDCard);
}
```


17 Integer Square Root Module

Introduction	99
API Functions	99
Programming Example	100

17.1 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.

17.2 API Functions

Functions

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

17.2.1 Function Documentation

17.2.1.1 `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.

17.3 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);
```

18 Ethernet Board Locator Module

Introduction	101
API Functions	101
Programming Example	104

18.1 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.

18.2 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)

18.2.1 Function Documentation

18.2.1.1 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.

18.2.1.2 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.

18.2.1.3 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.

18.2.1.4 LocatorClientIPSet

Sets the client IP address in the locator response packet.

Prototype:

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

Parameters:

uIP 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.

18.2.1.5 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.

18.2.1.6 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.

18.2.1.7 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.

18.3 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");
```


19 lwIP Wrapper Module

Introduction	105
API Functions	105
Programming Example	108

19.1 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.

19.2 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)

19.2.1 Function Documentation

19.2.1.1 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.

19.2.1.2 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.

19.2.1.3 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.

19.2.1.4 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.

19.2.1.5 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.

19.2.1.6 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.

19.2.1.7 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.

19.3 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);  
}
```

20 PTPd Wrapper Module

Introduction	109
API Functions	109
Programming Example	109

20.1 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.

20.2 API Functions

20.3 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);  
  
    ...  
}
```

21 Ring Buffer Module

Introduction	111
API Functions	111
Programming Example	117

21.1 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.

21.2 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)

21.2.1 Function Documentation

21.2.1.1 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.

21.2.1.2 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.

21.2.1.3 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.

21.2.1.4 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.

21.2.1.5 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.

21.2.1.6 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.

21.2.1.7 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.

21.2.1.8 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.

21.2.1.9 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.

21.2.1.10 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.

21.2.1.11 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.

21.2.1.12 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.

21.2.1.13 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.

21.2.1.14 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.

21.2.1.15 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.

21.3 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, pData, 11);
```

22 Simple Task Scheduler Module

Introduction	119
API Functions	119
Programming Example	124

22.1 Introduction

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

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

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

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

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

22.2 API Functions

Data Structures

- [tSchedulerTask](#)

Functions

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

- unsigned long [SchedulerTickCountGet](#) (void)

Variables

- [tSchedulerTask](#) [g_psSchedulerTable](#)[]
- unsigned long [g_ulSchedulerNumTasks](#)

22.2.1 Data Structure Documentation

22.2.1.1 tSchedulerTask

Definition:

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

Members:

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

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

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

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

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

Description:

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

22.2.2 Function Documentation

22.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

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

22.2.2.2 SchedulerElapsedTicksGet

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

Prototype:

```
unsigned long  
SchedulerElapsedTicksGet(unsigned long ulTickCount)
```

Parameters:

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

Description:

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

Returns:

The number of ticks elapsed since the provided tick count.

22.2.2.3 SchedulerInit

Initializes the task scheduler.

Prototype:

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

Parameters:

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

Description:

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

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

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

Returns:

None.

22.2.2.4 SchedulerRun

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

Prototype:

```
void  
SchedulerRun(void)
```

Description:

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

Although the scheduler makes use of the SysTick interrupt, all calls to functions configured in `g_psSchedulerTable` are made in the context of [SchedulerRun\(\)](#).

Returns:

None.

22.2.2.5 SchedulerSysTickIntHandler

Handles the SysTick interrupt on behalf of the scheduler module.

Prototype:

```
void  
SchedulerSysTickIntHandler(void)
```

Description:

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

Returns:

None.

22.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

Prototype:

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

Parameters:

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

Description:

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

Returns:

None.

22.2.2.7 SchedulerTaskEnable

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

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

None.

22.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

Prototype:

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

Description:

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

Returns:

Tick count since last boot.

22.2.3 Variable Documentation

22.2.3.1 g_psSchedulerTable

Definition:

```
tSchedulerTask g_psSchedulerTable[ ]
```

Description:

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

22.2.3.2 g_ulSchedulerNumTasks

Definition:

```
unsigned long g_ulSchedulerNumTasks
```

Description:

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

22.3 Programming Example

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

```
//*****  
//  
// Definition of the system tick rate. This results in a tick period of 10mS.  
//  
//*****  
#define TICKS_PER_SECOND 100  
  
//*****  
//  
// Prototypes of functions which will be called by the scheduler.  
//  
//*****  
static void ScrollTextBanner(void *pvParam);  
static void ToggleLED(void *pvParam);  
  
//*****  
//  
// This table defines all the tasks that the scheduler is to run, the periods  
// between calls to those tasks, and the parameter to pass to the task.  
//  
//*****  
tSchedulerTask g_psSchedulerTable[] =  
{  
    //  
    // Scroll the text banner 1 character to the left. This function is called  
    // every 20 ticks (5 times per second).  
    //  
    { ScrollTextBanner, (void *)0, 20, 0, true},  
}
```

```

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

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

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

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

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

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

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

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

```

```
//
SchedulerInit(TICKS_PER_SECOND);

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

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

23 Sine Calculation Module

Introduction	127
API Functions	127
Programming Example	128

23.1 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.

23.2 API Functions

Defines

- `cosine`(ulAngle)

Functions

- long `sine` (unsigned long ulAngle)

23.2.1 Define Documentation

23.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

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

Parameters:

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

Description:

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

Returns:

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

23.2.2 Function Documentation

23.2.2.1 sine

Computes an approximation of the sine of the input angle.

Prototype:

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

Parameters:

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

Description:

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

Returns:

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

23.3 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);  
}
```


24 Ethernet Software Update Module

Introduction	129
API Functions	129
Programming Example	131

24.1 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.

24.2 API Functions

Functions

- void [SoftwareUpdateBegin](#) (void)
- void [SoftwareUpdateInit](#) (tSoftwareUpdateRequested pfnCallback)

24.2.1 Function Documentation

24.2.1.1 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.

24.2.1.2 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.

24.3 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();  
}
```

25 TFTP Server Module

Introduction	133
Usage	133
API Functions	136

25.1 Introduction

The TFTP (tiny file transfer protocol) server module provides a simple way of transferring 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.

25.2 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.

25.2.0.3 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.

25.2.0.4 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.

25.2.0.5 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.

25.2.0.6 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.

25.3 API Functions

Data Structures

- [_tTFTPConnection](#)

Defines

- [TFTP_BLOCK_SIZE](#)

Enumerations

- [tTFTPError](#)

Functions

- void [TFTPInit](#) (tTFTPRequest pfnRequest)

25.3.1 Data Structure Documentation

25.3.1.1 `_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.

25.3.2 Define Documentation

25.3.2.1 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.

25.3.3 Typedef Documentation

25.3.3.1 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.

25.3.4 Enumeration Documentation

25.3.4.1 tTFTPError

Description:

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

25.3.4.2 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).

25.3.5 Function Documentation

25.3.5.1 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.

26 Micro Standard Library Module

Introduction	141
API Functions	141
Programming Example	147

26.1 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
<code>usprintf</code>	<code>sprintf</code>
<code>usnprintf</code>	<code>snprintf</code>
<code>uvsnprintf</code>	<code>vsnprintf</code>
<code>ustrnicmp</code>	<code>strnicmp</code>
<code>ustrtoul</code>	<code>strtoul</code>
<code>ustrstr</code>	<code>strstr</code>
<code>ulocaltime</code>	<code>localtime</code>

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

26.2 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)

26.2.1 Data Structure Documentation

26.2.1.1 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.

26.2.2 Function Documentation

26.2.2.1 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.

26.2.2.2 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 upper case letters (not lower 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.

26.2.2.3 `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 upper case letters (not lower 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.

26.2.2.4 `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*.

26.2.2.5 `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*.

26.2.2.6 `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.

26.2.2.7 `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.

26.2.2.8 `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 upper case letters (not lower 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.

26.3 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 = strtoul(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);
```

27 UART Standard IO Module

Introduction	149
API Functions	150
Programming Example	156

27.1 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.

27.1.1 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 been placed into the UART's FIFO. If it is not possible for the function to complete its operation immediately, it will busy wait.

27.1.2 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 to 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 a 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.

27.2 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)

27.2.1 Function Documentation

27.2.1.1 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.

27.2.1.2 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.

27.2.1.3 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.

27.2.1.4 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.

27.2.1.5 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.

27.2.1.6 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.

27.2.1.7 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 upper case letters (not lower 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.

27.2.1.8 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.

27.2.1.9 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.

27.2.1.10 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.

27.2.1.11 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.

27.2.1.12 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.

27.2.1.13 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.

27.3 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

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2009-2011, Texas Instruments Incorporated