RDK-IDM Firmware Development Package

USER'S GUIDE

TEXAS INSTRUMENTS

Copyright

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

Revision Information

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

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® Intelligent Display Module is a ready to use module that features an ARM® Cortex™-M3-based microcontroller, a TFT touch screen display, and Ethernet connectivity. The module also has four analog input channels, a single pole relay output, a small audio transducer, a microSD card slot, and an RS232-level UART.

The Intelligent Display Module comes in two variants: the MDL-IDM which supports Power over Ethernet (PoE) and the MDL-IDM28 which does not support PoE. The two modules are otherwise identical, and the software described in this chapter works the same on both modules.

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 the capabilities of the Intelligent Display Module, the peripheral driver library, and the graphics library. These applications are intended for demonstration and as a starting point for new applications.

Each of the RDK-IDM example applications is configured to operate alongside the boot loader to facilitate firmware update operations over Ethernet. The LM Flash Programmer, available for download from `http://www.luminarymicro.com/products/software_updates.html`, may be used to replace the main application image in each case. Software is included within each application to listen for "magic packets" from LM Flash Programmer indicating that a user is requesting a firmware update and, if such a packet is received, to pass control back to the bootloader to initiate the BOOTP/TFTP firmware update process. Note that the "magic packet" functionality is available in versions of LM Flash Programmer with build numbers greater than 560 (the build number can be found in the tool's "About..." box or on the overview page of the application help file). Older versions of LM Flash Programmer supporting Ethernet operation may be used to update applications which offer a manual method of initiating the firmware update but not those which rely solely upon the "magic packet" trigger.

The Ethernet-enabled boot loader image is built from the `boot_eth` directory under the main `boards/rdk-idm` directory. Typically, it will not be necessary to flash this image unless you have made changes to the boot loader source code. Replacing the boot loader image requires the use of a hardware JTAG/SWD debugger or a Stellaris Evaluation Kit board configured as a JTAG/SWD pass-through adapter.

There is an IAR workspace file (`rdk-idm.eww`) that contains the peripheral driver library and graphics library projects, along with all of the board example projects, in a single, easy to use workspace for use with Embedded Workbench version 5.

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

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

2.1   BLDC RDK Control (bldc_ctrl)

This application provides a simple GUI for controlling a BLDC RDK board. The motor can be started and stopped, the target speed can be adjusted, and the current speed can be monitored.

The target speed up and down buttons utilize the auto-repeat capability of the push button widget. For example, pressing the up button will increase the target speed by 100 rpm. Holding it for more than 0.5 seconds will commence the auto-repeat, at which point the target speed will increase by 100 rpm every 1/10th of a second. The same behavior occurs on the down button.

Upon startup, the application will attempt to contact a DHCP server to get an IP address. If a DHCP server can not be contacted, it will instead use the IP address 169.254.19.70 without performing any ARP checks to see if it is already in use. Once the IP address is determined, it will initiate a connection to a BLDC RDK board at IP address 169.254.89.71. While attempting to contact the DHCP server and the BLDC RDK board, the target speed will display as a set of bouncing dots.

The push buttons will not operate until a connection to a BLDC RDK board has been established.

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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

2.2 Boot Loader (boot_eth)

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

2.3 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. Vidales in the June 2002 issue of Embedded Systems Design. It can be found on-line 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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

2.4 Graphics Library Demonstration (grlib_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 hierarchy. 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 in 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 rightmost slider is configured as an indicator which tracks the state of the upper slider and ignores user input.

The final panel provides instructions and information necessary to update the board firmware via ethernet using the LM Flash Programmer application. When using a version of LM Flash Programmer with a build number greater than 560, software updates will occur automatically without user intervention being required in the application. If using an earlier version of LM Flash Programmer which does not send the "magic packet" signalling an update request, the "Update" button on the final screen may be pressed to transfer control to the boot loader in preparation for a firmware download.

2.5 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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

2.6 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, shows "Hello World!" on the screen. This may be used as a starting point for more complicated 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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

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 third panel provides instructions and information necessary to update the board firmware via ethernet using the LM Flash Programmer application. When using a version of LM Flash Programmer with a build number greater than 560, software updates will occur automatically without user intervention being required in the application. If using an earlier version of LM Flash Programmer which does not send the "magic packet" signalling an update request, the "Update" button on the final screen may be pressed to transfer control to the boot loader in preparation for a firmware download.

2.8 Quickstart Security Keypad (qs-keypad)

This application provides a security keypad to allow access to a door. The relay output is momentarily toggled upon entry of the access code to activate an electric door strike, unlocking the door.

The screen is divided into three parts; the Texas Instruments banner across the top, a hint across the bottom, and the main application area in the middle (which is the only portion that should appear if this application is used for a real door access system). The hints provide an on-screen guide to what the application is expecting at any given time.

Upon startup, the screen is blank and the hint says to touch the screen. Pressing the screen will bring up the keypad, which is randomized as an added security measure (so that an observer can not "steal" the access code by simply looking at the relative positions of the button presses). The current access code is provide in the hint at the bottom of the screen (which is clearly not secure).

If an incorrect access code is entered ("#" ends the code entry), then the screen will go blank and wait for another access attempt. If the correct access code is entered, the relay will be toggled for a few seconds (as indicated by the hint at the bottom stating that the door is open) and the screen will go blank. Once the door is closed again, the screen can be touched again to repeat the process.

The UART is used to output a log of events. Each event in the log is time stamped, with the arbitrary date of February 26, 2008 at 14:00 UT (universal time) being the starting time when the application is run. The following events are logged:

- The start of the application
- The access code being changed
- Access being granted (correct access code being entered)
- Access being denied (incorrect access code being entered)
- The door being relocked after access has been granted

A simple web server is provided to allow the access code to be changed. The Ethernet interface will attempt to contact a DHCP server, and if it is unable to acquire a DHCP address it will instead use the IP address 169.254.19.70 without performing any ARP checks to see if it is already in use. The web page shows the current access code and provides a form for updating the access code.

If a micro-SD card is present, the access code will be stored in a file called "key.txt" in the root directory. This file is written whenever the access code is changed, and is read at startup to initialize the access code. If a micro-SD card is not present, or the "key.txt" file does not exist, the access code defaults to 6918.

If "∗∗" is entered on the numeric keypad, the application provides a demonstration of the Stellaris Graphics Library with various panels showing the available widget types and graphics primitives. Navigate between the panels using buttons marked "+" and "-" at the bottom of the screen and return to keypad mode by pressing the "X" buttons which appear when you are on either the first or last demonstration panel.

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

Note that remote firmware update signalling is only supported in versions of LM Flash Programmer with build numbers greater than 560. If using an earlier version of LM Flash Programmer which does not send the "magic packet" signalling an update request, an update may be initiated by entering "∗0" on the application's numeric keypad.

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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

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`

UART1, which is connected to the 3 pin header on the underside of the IDM RDK board (J2), is configured for 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type "help" for command help.

To connect the IDM RDK board's UART to a 9 pin PC serial port, use a standard male to female DB9 serial cable and connect TXD (J2 pin 1, nearest the SD card socket) to pin 2 of the male serial

cable connector, RXD (J2 pin 2, the center pin) to pin 3 of the serial connector and GND (J2 pin 3) to pin 5 of the serial connector.

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. This feature is available in versions of LM Flash Programmer with build numbers greater than 560.

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.

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.

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`.

4   Analog Input Driver

4.1   Introduction

There are four analog input channels which can sense from 0 to 3 Volts in 1024 individual, evenly spaced steps. The analog input driver will sample these channels every millisecond and will call application-supplied callback functions when the value is above or below a set value, and when it crosses a set value (subject to hysteresis) in either direction. Each channel can be independently configured, and can have individual callbacks for each event.

The analog input driver utilizes sample sequence 2 of the ADC and timer 0 subtimer A (shared with the touch screen driver). The interrupt from the ADC sample sequence 2 is used to process the analog input readings; the AnalogIntHandler() 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).

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

4.2   API Functions

Functions

- void AnalogCallbackSetAbove (unsigned long ulChannel, tAnalogCallback ∗pfnOnAbove)
- void AnalogCallbackSetBelow (unsigned long ulChannel, tAnalogCallback ∗pfnOnBelow)
- void   AnalogCallbackSetFallingEdge   (unsigned   long   ulChannel,   tAnalogCallback ∗pfnOnFallingEdge)
- void   AnalogCallbackSetRisingEdge   (unsigned   long   ulChannel,   tAnalogCallback ∗pfnOnRisingEdge)
- void AnalogInit (void)
- void AnalogIntHandler (void)
- void AnalogLevelSet (unsigned long ulChannel, unsigned short usLevel, char cHysteresis)

4.2.1   Function Documentation

4.2.1.1   AnalogCallbackSetAbove

Sets the function to be called when the analog input is above the trigger level.

Prototype:
```
void
```

```
AnalogCallbackSetAbove(unsigned long ulChannel,
                       tAnalogCallback *pfnOnAbove)
```

Parameters:

    ulChannel is the channel to modify.

    pfnOnAbove is a pointer to the function to be called whenever the analog input is above the trigger level.

Description:

    This function sets the function that should be called whenever the analog input is above the trigger level (in other words, while the analog input is above the trigger level, the callback will be called every millisecond). Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input is above the trigger level).

Returns:

    None.

4.2.1.2   AnalogCallbackSetBelow

Sets the function to be called when the analog input is below the trigger level.

Prototype:

```
void
AnalogCallbackSetBelow(unsigned long ulChannel,
                       tAnalogCallback *pfnOnBelow)
```

Parameters:

    ulChannel is the channel to modify.

    pfnOnBelow is a pointer to the function to be called whenever the analog input is below the trigger level.

Description:

    This function sets the function that should be called whenever the analog input is below the trigger level (in other words, while the analog input is below the trigger level, the callback will be called every millisecond). Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input is below the trigger level).

Returns:

    None.

4.2.1.3   AnalogCallbackSetFallingEdge

Sets the function to be called when the analog input transitions from above to below the trigger level.

Prototype:

```
void
AnalogCallbackSetFallingEdge(unsigned long ulChannel,
                             tAnalogCallback *pfnOnFallingEdge)
```

Parameters:

> ***ulChannel*** is the channel to modify.
>
> ***pfnOnFallingEdge*** is a pointer to the function to be called when the analog input transitions from above to below the trigger level.

Description:

> This function sets the function that should be called whenever the analog input transitions from above to below the trigger level. Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input transitions from above to below the trigger level).

Returns:

> None.

4.2.1.4   AnalogCallbackSetRisingEdge

Sets the function to be called when the analog input transitions from below to above the trigger level.

Prototype:

```
void
AnalogCallbackSetRisingEdge(unsigned long ulChannel,
                            tAnalogCallback *pfnOnRisingEdge)
```

Parameters:

> ***ulChannel*** is the channel to modify.
>
> ***pfnOnRisingEdge*** is a pointer to the function to be called when the analog input transitions from below to above the trigger level.

Description:

> This function sets the function that should be called whenever the analog input transitions from below to above the trigger level. Specifying a function address of 0 will cancel a previous callback function (meaning that no function will be called when the analog input transitions from below to above the trigger level).

Returns:

> None.

4.2.1.5   AnalogInit

Initializes the analog input driver.

Prototype:

```
void
AnalogInit(void)
```

Description:

> This function initializes the analog input driver, starting the sampling process and disabling all channel callbacks. Once called, the ADC2 interrupt will be asserted periodically; the AnalogIntHandler() function should be called in response to this interrupt. It is the application's responsibility to install AnalogIntHandler() in the application's vector table.

Returns:
　　None.

4.2.1.6　AnalogIntHandler

Handles the ADC sample sequence two interrupt.

Prototype:
```
void
AnalogIntHandler(void)
```

Description:
　　This function is called when the ADC sample sequence two generates an interrupt. It will read the new ADC readings, perform debouncing on the analog inputs, and call the appropriate callbacks based on the new readings.

Returns:
　　None.

4.2.1.7　AnalogLevelSet

Sets the trigger level for an analog channel.

Prototype:
```
void
AnalogLevelSet(unsigned long ulChannel,
               unsigned short usLevel,
               char cHysteresis)
```

Parameters:
　　ulChannel is the channel to modify.
　　usLevel is the trigger level for this channel.
　　cHysteresis is the hysteresis to apply to the trigger level for this channel.

Description:
　　This function sets the trigger level and hysteresis for an analog input channel. The hysteresis allows for filtering of noise on the analog input. The actual level to transition from "below" the trigger level to "above" the trigger level is the trigger level plus the hysteresis. Similarly, the actual level to transition from "above" the trigger level to "below" the trigger level is the trigger level minus the hysteresis.

Returns:
　　None.

4.3　**Programming Example**

The following examples shows how to use the analog input driver to call a function when the voltage on an input channel transitions from below 2 V to above 2 V. This assumes that the vector table

in the startup code has AnalogIntHandler() listed as the handler for the ADC sample sequence 2 interrupt.

```
extern void RisingEdgeCallback(unsigned long ulChannel);

//
// Initialize the analog input driver.
//
AnalogInit();

//
// Set the level on channel 0 to 2 V with 0.1 V of hysteresis.  Full scale
// is 3 V and provides a reading of 1023, so 2 V is 2/3 of 1023 and 0.1 V
// is 1/30 of 1023.
//
AnalogLevelSet(0, (1023 * 2) / 3, 1023 / 30);

//
// Set the rising edge callback function.
//
AnalogCallbackSetRisingEdge(0, RisingEdgeCallback);

//
// Wait for the rising edge.
//
while(1)
{
}
```

5   Display Driver

5.1   Introduction

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

The display driver can be configured for four different orientations:

- Portrait, which is the default orientation of the display driver, the preferred orientation of the display, and the orientation used by all of the example applications. Portrait mode provides a 240x320 display and is selected by defining **PORTRAIT**, or by not defining an orientation, when building the display driver.
- Landscape, which is the screen rotated 90 degrees counter-clockwise (where the flex connector is on the right side of the display). Landscape mode provides a 320x240 display and is selected by defining **LANDSCAPE** when building the display driver.
- Portrait flip, which is the screen rotated 180 degrees (where the flex connector is on the top of the display). Portrait flip mode provides a 240x320 display and is selected by defining **PORTRAIT_FLIP** when building the display driver.
- Landscape flip, which is the screen rotated 90 degrees clockwise (where the flex connector is on the left side of the display). Landscape flip mode provides a 320x240 display and is selected by defining **LANDSCAPE_FLIP** when building the display driver.

The orientation used depends on the requirements of the application and the desired viewing angle of the display. The panel itself has a 6 o'clock viewing angle, so when used in portrait flip mode it has a 12 o'clock viewing angle. Similarly, it has a 3 o'clock viewing angle in landscape mode and a 9 o'clock viewing angle in landscape flip mode. Viewing the display from an angle other than the viewing angle will result in distortion of the displayed colors.

This driver is located in `boards/rdk-idm/drivers`, with `formike240x320x16_ili9320.c` containing the source code and `formike240x320x16_ili9320.h` containing the API definitions for use by applications.

5.2   API Functions

Functions

- void Formike240x320x16_ILI9320BacklightOff (void)
- void Formike240x320x16_ILI9320BacklightOn (void)
- unsigned short Formike240x320x16_ILI9320ControllerIdGet (void)
- void Formike240x320x16_ILI9320Init (void)

Variables

- const tDisplay g_sFormike240x320x16_ILI9320

5.2.1 Function Documentation

5.2.1.1 Formike240x320x16_ILI9320BacklightOff

Turns off the backlight.

Prototype:
```
void
Formike240x320x16_ILI9320BacklightOff(void)
```

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

Returns:
None.

5.2.1.2 Formike240x320x16_ILI9320BacklightOn

Turns on the backlight.

Prototype:
```
void
Formike240x320x16_ILI9320BacklightOn(void)
```

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

Returns:
None.

5.2.1.3 Formike240x320x16_ILI9320ControllerIdGet

Determines whether an ILI9320, ILI9325 or ILI9328 controller is present.

Prototype:
```
unsigned short
Formike240x320x16_ILI9320ControllerIdGet(void)
```

Description:
This function queries the ID of the display controller in use and returns it to the caller. This driver supports both ILI9320, ILI9325 and ILI9328. These are very similar but the sense of the long display axis is reversed in the Formike KWH028Q02-F03 using an ILI9320 relative to the other two supported displays and this information is needed by the touchscreen driver to provide correct touch coordinate information.

Returns:
Returns 0x9320 if an ILI9320 controller is in use, 0x9325 if an ILI9325 is present or 0x9328 if an ILI9328 is detected.

5.2.1.4 Formike240x320x16_ILI9320Init

Initializes the display driver.

Prototype:
```
void
Formike240x320x16_ILI9320Init(void)
```

Description:
This function initializes the ILI9320, ILI9325 or ILI9328 display controller on the panel, preparing it to display data.

Returns:
None.

5.2.2   Variable Documentation

5.2.2.1 g_sFormike240x320x16_ILI9320

Definition:
```
const tDisplay g_sFormike240x320x16_ILI9320
```

Description:
The graphics library display structure that describes the driver for the F02, F03 or F05 variants of the Formike Electronic KWH028Q02 TFT panel with ILI932x controllers.

5.3   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.
//
Formike240x320x16_ILI9320Init();

//
// Turn on the backlight.
//
Formike240x320x16_ILI9320BacklightOn();

//
// Initialize a graphics library drawing context.
//
GrContextInit(&sContext, &g_sFormike240x320x16_ILI9320);
```

6   Relay Output Driver

6.1   Introduction

The relay output has three pins; a common contact, a normally closed contact, and a normally opened contact. By enabling the relay output, the normally closed contact is opened and the normally opened contact is closed. When the relay output is disabled, it reverts back to its normal state.

The relay output driver provides a means of enabling and disabling the relay. The application is responsible for any sequencing, such as enabling the relay, delaying for one second, and then disabling the relay.

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

6.2   API Functions

Functions

- void RelayDisable (void)
- void RelayEnable (void)
- void RelayInit (void)

6.2.1   Function Documentation

6.2.1.1   RelayDisable

Disable the relay output.

Prototype:
```
void
RelayDisable(void)
```

Description:
This function disables the relay output. This causes the relay to become de-energized, putting it into its default state (in other words, the normally open connect is opened and the normally closed contact is closed).

Returns:
None.

6.2.1.2   RelayEnable

Enables the relay output.

Prototype:
```
void
RelayEnable(void)
```

Description:
This function enables the relay output. This causes the relay to become energized, putting it into the non-default state (in other words, the normally open contact is closed and the normally closed contact is opened).

Returns:
None.

6.2.1.3   RelayInit

Initializes the relay output.

Prototype:
```
void
RelayInit(void)
```

Description:
This function initializes the relay output, preparing it to control the relay. The relay is started in the disabled state (in other words, de-energized).

Returns:
None.

6.3   Programming Example

The following example shows how to activate the relay output.

```
//
// Initialize the relay output.
//
RelayInit();

//
// Enable the relay.
//
RelayEnable();

//
// Delay for a while.
//
SysCtlDelay(10000);

//
// Disable the relay.
//
RelayDisable();
```

7   Sound Output Driver

7.1   Introduction

The sound output provides a means of producing simple tones using a square wave drive. The sound output driver allows the output frequency to be changed, along with the volume of the sound. 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.

When the PWM output goes high, the speaker will start to travel from its resting position to its fully deflected position. Since this takes time (on the order of 50 uS), it is possible to turn off the PWM output before it has reached the extent of its travel. Doing so will result in less air being displaced by the moving speaker cone, which results in a reduced volume. This technique is utilized to provide a simple volume control.

Since a timer PWM output is used to drive the speaker, the maximum divide that is available is 65535. When running the processor (and therefore the timer) at 50 MHz, this equates to a minimum audio frequency of approximately 762.95 Hz. Lower audio frequencies are possible if the processor clock rate is lowered, though this will degrade the performance of the entire system (this will be most noticeable in the update rate on the display).

The sound output driver utilizes timer 2 (both subtimer A and subtimer B). The interrupt from timer 2 subtimer B is used as a time base for playing songs; the SoundIntHandler() 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).

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

7.2   API Functions

Functions

- void SoundDisable (void)
- void SoundEnable (void)
- void SoundFrequencySet (unsigned long ulFrequency)
- void SoundInit (void)
- void SoundIntHandler (void)
- void SoundPlay (const unsigned short *pusSong, unsigned long ulLength)
- void SoundVolumeDown (unsigned long ulPercent)
- unsigned char SoundVolumeGet (void)
- void SoundVolumeSet (unsigned long ulPercent)
- void SoundVolumeUp (unsigned long ulPercent)

7.2.1   Function Documentation

7.2.1.1   SoundDisable

Disables the sound output.

Prototype:
```
void
SoundDisable(void)
```

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

Returns:
None.

7.2.1.2   SoundEnable

Enables the sound output.

Prototype:
```
void
SoundEnable(void)
```

Description:
This function enables the sound output, preparing it to play music or sound effects.

Returns:
None.

7.2.1.3   SoundFrequencySet

Sets the sound output frequency.

Prototype:
```
void
SoundFrequencySet(unsigned long ulFrequency)
```

Parameters:
ulFrequency is the desired sound output frequency.

Description:
This function sets the frequency of the output sound. This change will take effect immediately and will remain in effect until changed (either explicitly by another call or implicitly by the playback of a sound).

Returns:
None.

7.2.1.4   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 functions. The sound driver uses timer 2 subtimer A to produce the PWM output, and timer 2 subtimer B to be the time base for the playback of sound effects. It is the responsibility of the application to ensure that SoundIntHandler() is called when the timer 2 subtimer B interrupt occurs (typically by placing a pointer to this function in the appropriate location in the processor's vector table).

Returns:
None.

7.2.1.5   SoundIntHandler

Handles the sound timer interrupt.

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

Description:
This function provides periodic updates to the PWM output in order to produce a sound effect. It is called when the timer 2 subtimer B interrupt occurs.

Returns:
None.

7.2.1.6   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 canceled and the new song is started.

Returns:
>   None.

7.2.1.7 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.

7.2.1.8 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.

7.2.1.9 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.

7.2.1.10 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.

7.3   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();
```

8 Touch Screen Driver

8.1 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 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 0 subtimer A (shared with the analog input driver). 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/drivers`, with `touch.c` containing the source code and `touch.h` containing the API definitions for use by applications.

8.2 API Functions

Functions

- void TouchScreenCallbackSet (long (∗pfnCallback)(unsigned long ulMessage, long lX, long lY))
- void TouchScreenInit (void)
- void TouchScreenIntHandler (void)

8.2.1 Function Documentation

8.2.1.1 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.

8.2.1.2 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 0 subtimer A

Returns:
> None.

8.2.1.3   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.

8.3   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 fol-
lows. 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);

    ...

}
```

9 Command Line Processing Module

9.1 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.

9.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char ∗pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

9.2.1　Data Structure Documentation

9.2.1.1　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.

9.2.2　Define Documentation

9.2.2.1　CMDLINE_BAD_CMD

Definition:
```
#define CMDLINE_BAD_CMD
```

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

9.2.2.2　CMDLINE_TOO_MANY_ARGS

Definition:
```
#define CMDLINE_TOO_MANY_ARGS
```

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

9.2.3　Function Documentation

9.2.3.1　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.

9.2.4 Variable Documentation

9.2.4.1 g_sCmdTable

Definition:

>`tCmdLineEntry g_sCmdTable[]`

Description:

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

9.3 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));
}
```

10   CPU Usage Module

10.1   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.

10.2   API Functions

Functions

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

10.2.1 Function Documentation

10.2.1.1 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.

10.2.1.2 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.

10.3 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();
    }
}
```

11   Flash Parameter Block Module

11.1   Introduction

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

The FlashPBInit() function is used to initialize a parameter block. The primary conditions for the parameter block are that flash region used to store the parameter blocks must contain at least two erase blocks of flash to ensure fault tolerance, and the size of the parameter block must be an integral divisor of the the size of an erase block. FlashPBGet() and FlashPBSave() are used to read and write parameter block data into the parameter region. The only constraints on the content of the parameter block are that the first two bytes of the block are reserved for use by the read/write functions as a sequence number and checksum, respectively.

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

11.2   API Functions

Functions

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

11.2.1   Function Documentation

11.2.1.1   FlashPBGet

Gets the address of the most recent parameter block.

Prototype:
```
unsigned char *
FlashPBGet(void)
```

Description:
This function returns the address of the most recent parameter block that is stored in flash.

Returns:
Returns the address of the most recent parameter block, or NULL if there are no valid parameter blocks in flash.

11.2.1.2 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.

11.2.1.3 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.

11.3   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);
}
```

12   Integer Square Root Module

12.1   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.

12.2   API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

12.2.1   Function Documentation

12.2.1.1   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.

12.3   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);
```

13   Ethernet Board Locator Module

13.1   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.

13.2   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)

13.2.1   Function Documentation

13.2.1.1   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.

13.2.1.2 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.

13.2.1.3 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.

13.2.1.4 LocatorClientIPSet

Sets the client IP address in the locator response packet.

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

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

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

Returns:
>   None.

13.2.1.5 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.

13.2.1.6 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.

13.2.1.7 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.

13.3 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");
```

Standard

14   lwIP Wrapper Module

14.1   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.

14.2   API Functions

Functions

- void lwIPEthernetIntHandler (void)
- void lwIPInit (const unsigned char *pucMAC, unsigned long ulIPAddr, unsigned long ulNet-Mask, 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)

14.2.1   Function Documentation

14.2.1.1   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.

14.2.1.2 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.

14.2.1.3 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.

14.2.1.4 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.

14.2.1.5 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.

14.2.1.6 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.

14.2.1.7 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.

14.3 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);
}
```

15   PTPd Wrapper Module

15.1   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.

15.2   API Functions

15.3   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);

            ...
    }
```

16   Ring Buffer Module

16.1   Introduction

The ring buffer module provides a set of functions allowing management of a block of memory as a
ring buffer. This is typically used in buffering transmit or receive data for a communication channel
but has many other uses including implementing queues and FIFOs.

This module is contained in `utils/ringbuf.c`, with `utils/ringbuf.h` containing the API def-
initions for use by applications.

16.2   API Functions

Functions

- void RingBufAdvanceRead (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- void RingBufAdvanceWrite (tRingBufObject *ptRingBuf, unsigned long ulNumBytes)
- unsigned long RingBufContigFree (tRingBufObject *ptRingBuf)
- unsigned long RingBufContigUsed (tRingBufObject *ptRingBuf)
- tBoolean RingBufEmpty (tRingBufObject *ptRingBuf)
- void RingBufFlush (tRingBufObject *ptRingBuf)
- unsigned long RingBufFree (tRingBufObject *ptRingBuf)
- tBoolean RingBufFull (tRingBufObject *ptRingBuf)
- void RingBufInit (tRingBufObject *ptRingBuf, unsigned char *pucBuf, unsigned long ulSize)
- void RingBufRead (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ul-
  Length)
- unsigned char RingBufReadOne (tRingBufObject *ptRingBuf)
- unsigned long RingBufSize (tRingBufObject *ptRingBuf)
- unsigned long RingBufUsed (tRingBufObject *ptRingBuf)
- void RingBufWrite (tRingBufObject *ptRingBuf, unsigned char *pucData, unsigned long ul-
  Length)
- void RingBufWriteOne (tRingBufObject *ptRingBuf, unsigned char ucData)

16.2.1   Function Documentation

16.2.1.1   RingBufAdvanceRead

Remove bytes from the ring buffer by advancing the read index.

Prototype:
```
void
RingBufAdvanceRead(tRingBufObject *ptRingBuf,
                    unsigned long ulNumBytes)
```

Parameters:
>   ***ptRingBuf*** points to the ring buffer from which bytes are to be removed.
>   ***ulNumBytes*** is the number of bytes to be removed from the buffer.

Description:
>   This function advances the ring buffer read index by a given number of bytes, removing that
>   number of bytes of data from the buffer. If *ulNumBytes* is larger than the number of bytes
>   currently in the buffer, the buffer is emptied.

Returns:
>   None.

16.2.1.2 RingBufAdvanceWrite

Add bytes to the ring buffer by advancing the write index.

Prototype:
```
void
RingBufAdvanceWrite(tRingBufObject *ptRingBuf,
                     unsigned long ulNumBytes)
```

Parameters:
>   ***ptRingBuf*** points to the ring buffer to which bytes have been added.
>   ***ulNumBytes*** is the number of bytes added to the buffer.

Description:
>   This function should be used by clients who wish to add data to the buffer directly rather than via
>   calls to RingBufWrite() or RingBufWriteOne(). It advances the write index by a given number
>   of bytes. If the *ulNumBytes* parameter is larger than the amount of free space in the buffer, the
>   read pointer will be advanced to cater for the addition. Note that this will result in some of the
>   oldest data in the buffer being discarded.

Returns:
>   None.

16.2.1.3 RingBufContigFree

Returns number of contiguous free bytes available in a ring buffer.

Prototype:
```
unsigned long
RingBufContigFree(tRingBufObject *ptRingBuf)
```

Parameters:
>   ***ptRingBuf*** is the ring buffer object to check.

Description:
This function returns the number of contiguous free bytes ahead of the current write pointer in the ring buffer.

Returns:
Returns the number of contiguous bytes available in the ring buffer.

16.2.1.4 RingBufContigUsed

Returns number of contiguous bytes of data stored in ring buffer ahead of the current read pointer.

Prototype:
```
unsigned long
RingBufContigUsed(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of contiguous bytes of data available in the ring buffer ahead of the current read pointer. This represents the largest block of data which does not straddle the buffer wrap.

Returns:
Returns the number of contiguous bytes available.

16.2.1.5 RingBufEmpty

Determines whether the ring buffer whose pointers and size are provided is empty or not.

Prototype:
```
tBoolean
RingBufEmpty(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is empty. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is empty or **false** otherwise.

16.2.1.6 RingBufFlush

Empties the ring buffer.

Prototype:
```
void
RingBufFlush(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
Discards all data from the ring buffer.

Returns:
None.

16.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:
```
unsigned long
RingBufFree(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to check.

Description:
This function returns the number of bytes available in the ring buffer.

Returns:
Returns the number of bytes available in the ring buffer.

16.2.1.8 RingBufFull

Determines whether the ring buffer whose pointers and size are provided is full or not.

Prototype:
```
tBoolean
RingBufFull(tRingBufObject *ptRingBuf)
```

Parameters:
ptRingBuf is the ring buffer object to empty.

Description:
This function is used to determine whether or not a given ring buffer is full. The structure is specifically to ensure that we do not see warnings from the compiler related to the order of volatile accesses being undefined.

Returns:
Returns **true** if the buffer is full or **false** otherwise.

16.2.1.9 RingBufInit

Initialize a ring buffer object.

Prototype:
```
void
RingBufInit(tRingBufObject *ptRingBuf,
            unsigned char *pucBuf,
            unsigned long ulSize)
```

Parameters:
>   ***ptRingBuf*** points to the ring buffer to be initialized.
>   ***pucBuf*** points to the data buffer to be used for the ring buffer.
>   ***ulSize*** is the size of the buffer in bytes.

Description:
>   This function initializes a ring buffer object, preparing it to store data.

Returns:
>   None.

16.2.1.10 RingBufRead

Reads data from a ring buffer.

Prototype:
```
void
RingBufRead(tRingBufObject *ptRingBuf,
            unsigned char *pucData,
            unsigned long ulLength)
```

Parameters:
>   ***ptRingBuf*** points to the ring buffer to be read from.
>   ***pucData*** points to where the data should be stored.
>   ***ulLength*** is the number of bytes to be read.

Description:
>   This function reads a sequence of bytes from a ring buffer.

Returns:
>   None.

16.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:
```
unsigned char
RingBufReadOne(tRingBufObject *ptRingBuf)
```

Parameters:
> ***ptRingBuf*** points to the ring buffer to be written to.

Description:
> This function reads a single byte of data from a ring buffer.

Returns:
> The byte read from the ring buffer.

16.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

Prototype:
```
unsigned long
RingBufSize(tRingBufObject *ptRingBuf)
```

Parameters:
> ***ptRingBuf*** is the ring buffer object to check.

Description:
> This function returns the size of the ring buffer.

Returns:
> Returns the size in bytes of the ring buffer.

16.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

Prototype:
```
unsigned long
RingBufUsed(tRingBufObject *ptRingBuf)
```

Parameters:
> ***ptRingBuf*** is the ring buffer object to check.

Description:
> This function returns the number of bytes stored in the ring buffer.

Returns:
> Returns the number of bytes stored in the ring buffer.

16.2.1.14 RingBufWrite

Writes data to a ring buffer.

Prototype:
```
void
RingBufWrite(tRingBufObject *ptRingBuf,
             unsigned char *pucData,
             unsigned long ulLength)
```

Parameters:
  ptRingBuf points to the ring buffer to be written to.
  pucData points to the data to be written.
  ulLength is the number of bytes to be written.

Description:
  This function write a sequence of bytes into a ring buffer.

Returns:
  None.

16.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:
```
void
RingBufWriteOne(tRingBufObject *ptRingBuf,
                unsigned char ucData)
```

Parameters:
  ptRingBuf points to the ring buffer to be written to.
  ucData is the byte to be written.

Description:
  This function writes a single byte of data into a ring buffer.

Returns:
  None.

16.3 Programming Example

The following example shows how to pass data through the ring buffer.

```
char pcBuffer[128], pcData[16];
tRingBufObject sRingBuf;

//
// Initialize the ring buffer.
//
RingBufInit(&sRingBuf, pcBuffer, sizeof(pcBuffer));

//
// Write some data into the ring buffer.
//
RingBufWrite(&sRingBuf, "Hello World", 11);
```

```
//
// Read the data out of the ring buffer.
//
RingBufRead(&sRingBuf, pcData, 11);
```

17   Sine Calculation Module

17.1   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.

17.2   API Functions

Functions

- long sine (unsigned long ulAngle)

17.2.1   Function Documentation

17.2.1.1   sine

Computes an approximation of the sine of the input angle.

Prototype:
```
long
sine(unsigned long ulAngle)
```

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

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

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

17.3   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);
}
```

18   Ethernet Software Update Module

18.1   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.

18.2   API Functions

Functions

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

18.2.1   Function Documentation

18.2.1.1   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 Software-UpdateInit(). 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.

18.2.1.2   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 Software-UpdateBegin() 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.

18.3   Programming Example

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

```c
//*****************************************************************************
//
// 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();
}
```

19   TFTP Server Module

19.1   Introduction

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

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

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

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

19.2   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.

19.2.0.3 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.

19.2.0.4 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.

19.2.0.5 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.

19.2.0.6 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.

19.3 API Functions

Data Structures

- _tTFTPConnection

Defines

- TFTP_BLOCK_SIZE

Enumerations

- tTFTPError

Functions

- void TFTPInit (tTFTPRequest pfnRequest)

19.3.1 Data Structure Documentation

19.3.1.1 _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.

19.3.2   Define Documentation

19.3.2.1   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.

19.3.3   Typedef Documentation

19.3.3.1   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.

19.3.4   Enumeration Documentation

19.3.4.1   tTFTPError

Description:

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

19.3.4.2   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).

19.3.5   Function Documentation

19.3.5.1   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.

20 Micro Standard Library Module

20.1 Introduction

The micro standard library module provides a set of small implementations of functions normally found in the C library. These functions provide reduced or greatly reduced functionality in order to remain small while still being useful for most embedded applications.

The following functions are provided, along with the C library equivalent:

Function	C library equivalent
usprintf	sprintf
usnprintf	snprintf
uvsnprintf	vsnprintf
ustrnicmp	strnicmp
ustrtoul	strtoul
ustrstr	strstr
ulocaltime	localtime

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

20.2 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)

20.2.1   Data Structure Documentation

20.2.1.1   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.

20.2.2   Function Documentation

20.2.2.1   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.

20.2.2.2 usnprintf

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

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

Parameters:
> *pcBuf* is the buffer where the converted string is stored.
>
> *ulSize* is the size of the buffer.
>
> *pcString* is the format string.
>
> *...* are the optional arguments, which depend on the contents of the format string.

Description:
> This function is very similar to the C library `sprintf()` function. Only the following formatting characters are supported:
>
> - %c to print a character
> - %d to print a decimal value
> - %s to print a string
> - %u to print an unsigned decimal value
> - %x to print a hexadecimal value using lower case letters
> - %X to print a hexadecimal value using lower case letters (not upper case letters as would typically be used)
> - %p to print a pointer as a hexadecimal value
> - %% to print out a % character
>
> For %d, %p, %s, %u, %x, and %X, an optional number may reside between the % and the format character, which specifies the minimum number of characters to use for that value; if preceded by a 0 then the extra characters will be filled with zeros instead of spaces. For example, "%8d" will use eight characters to print the decimal value with spaces added to reach eight; "%08d" will use eight characters as well but will add zeros instead of spaces.
>
> The type of the arguments after *pcString* must match the requirements of the format string. For example, if an integer was passed where a string was expected, an error of some kind will most likely occur.
>
> The function will copy at most *ulSize* - 1 characters into the buffer *pcBuf*. One space is reserved in the buffer for the null termination character.
>
> The function will return the number of characters that would be converted as if there were no limit on the buffer size. Therefore it is possible for the function to return a count that is greater than the specified buffer size. If this happens, it means that the output was truncated.

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

20.2.2.3 usprintf

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

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

Parameters:
   pcBuf is the buffer where the converted string is stored.
   pcString is the format string.
   ... are the optional arguments, which depend on the contents of the format string.

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

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

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

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

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

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

20.2.2.4 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*.

20.2.2.5 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*.

20.2.2.6 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.

20.2.2.7 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.

20.2.2.8 uvsnprintf

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

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

Parameters:
> **pcBuf** points to the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

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

Description:

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

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

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

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

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

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

Returns:

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

20.3 Programming Example

The following example shows how to use some of the micro standard library functions.

```
unsigned long ulValue;
char pcBuffer[32];
tTime sTime;

//
// Convert the number in pcBuffer (previous read from somewhere) into an
// integer.  Note that this supports converting decimal values (such as
// 4583), octal values (such as 036583), and hexadecimal values (such as
// 0x3425).
```

```
//
ulValue = ustrtoul(pcBuffer, 0, 0);

//
// Convert that integer from a number of seconds into a broken down date.
//
ulocaltime(ulValue, &sTime);

//
// Print out the corresponding time of day in military format.
//
usprintf(pcBuffer, "%02d:%02d", sTime.ucHour, sTime.ucMin);
```

21 UART Standard IO Module

21.1 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.

21.1.1 Unbuffered Operation

Unbuffered operation is selected by not defining **UART_BUFFERED** when building the UART standard IO module. In unbuffered mode, calls to the module will not return until the operation has been completed. So, for example, a call to UARTprintf() will not return until the entire string has be placed into the UART's FIFO. If it is not possible for the function to complete its operation immediately, it will busy wait.

21.1.2 Buffered Operation

Buffered operation is selected by defining **UART_BUFFERED** when building the UART standard IO module. In buffered mode, there is a larger UART data FIFO in SRAM that extends the size of the hardware FIFO. Interrupts from the UART are used to transfer data between the SRAM buffer and the hardware FIFO. It is the responsibility of the application to ensure that UARTStdioIntHandler() is called when the UART interrupt occurs; typically this is accomplished by placing it in the vector table in the startup code for the application.

In addition providing a larger UART buffer, the behavior of UARTprintf() is slightly modified. If the output buffer is full, UARTprintf() will discard the remaining characters from the string instead of waiting until space becomes available in the buffer. If this behavior is not desired, UARTFlushTx() may be called to ensure that the transmit buffer is emptied prior to adding new data via UARTprintf() (though this will not work if the string to be printed is larger than the buffer).

UARTPeek() can be used to determine whether a line end is present prior to calling UARTgets() if non-blocking operation is required. In cases where the buffer supplied on UARTgets() fills before a line termination character is received, the call will return with a full buffer.

21.2 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)

21.2.1 Function Documentation

21.2.1.1 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.

21.2.1.2 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.

21.2.1.3 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.

21.2.1.4 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.

21.2.1.5 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.

21.2.1.6 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.

21.2.1.7 UARTprintf

A simple UART based printf function supporting %c, %d, %p, %s, %u, %x, and %X.

Prototype:
```
void
UARTprintf(const char *pcString,
           ...)
```

Parameters:
>   ***pcString*** is the format string.
>   ***...*** are the optional arguments, which depend on the contents of the format string.

Description:
>   This function is very similar to the C library `fprintf()` function. All of its output will be sent to the UART. Only the following formatting characters are supported:

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

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

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

Returns:
>   None.

21.2.1.8 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.

21.2.1.9 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.

21.2.1.10 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.

21.2.1.11 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.

21.2.1.12 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.

21.2.1.13 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.

21.3 Programming Example

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

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

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

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

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

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

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2008-2010, Texas Instruments Incorporated