RDK-IDM-L35 Firmware Development Package

USER'S GUIDE

SW-RDK-IDM-L35-UG-9453

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 9453 of this document, last updated on September 05, 2012.

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® Intelligent Display Module with 3.5” Landscape Display is a ready to use module that features an ARM® Cortex™-M3-based microcontroller and a 3.5” TFT touch screen display. The module also has four analog input channels, a small audio transducer, a microSD card slot, and an RS232-level UART.

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-L35 example applications is configured to operate alongside the boot loader to facilitate firmware update operations over the serial connection. 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.

The serial-enabled boot loader image is built in the `boot_serial` directory below the main `boards/rdk-idm-l35` 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.

The serial boot loader occupies the first 2KB of flash so example applications built to run alongside it must be linked to run from address 0x800. Care must be taken to ensure that they are written at the correct address when programming flash using a JTAG/SWD adapter.

There is an IAR workspace file (`rdk-idm-l35.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-l35.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-l35` subdirectory of the firmware development package source distribution.

2.1 Boot Loader (boot_serial)

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 UART0 to load an application.

Boot loader configuration is defined in header file bl_config.h which includes, among other parameters, the expected load address of the main application images. For the serial boot loader configuration used by all the IDM-L35 example applications, this address is 0x800. Care must be taken to ensure that main application images are written to this address rather than address 0x00 (where the boot loader resides) if using a JTAG/SWD debugger to program flash.

2.2 Calibration for the Touch Screen (calibrate)

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

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

This application supports remote software update over serial using the LM Flash Programmer application. The application transfers control to the boot loader whenever it completes to allow a new image to be downloaded if required. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

2.3 Font Viewer (fontview)

This example displays the contents of a Stellaris graphics library font on the DK board's LCD touchscreen. By default, the application shows a test font containing ASCII, the Japanese Hiragana and Katakana alphabets, and a group of Korean Hangul characters. If an SDCard is installed and the root directory contains a file named `font.bin`, this file is opened and used as the display font instead. In this case, the graphics library font wrapper feature is used to access the font from the file system rather than from internal memory.

When the “Update” button is pressed, the application transfers control to the boot loader to allow a new application image to be downloaded. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

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 heirarchy. A text, image, and application-drawn canvas are displayed.

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

The fifth panel shows the container widget, which provides a grouping construct typically used for radio buttons. Containers with a title, a centered title, and no title are displayed.

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

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

The eighth panel shows the slider widget. Six sliders constructed using the various supported style options are shown. The slider value callback is used to update two widgets to reflect the values reported by sliders. A canvas widget near the top right of the display tracks the value of the red and green image-based slider to its left and the text of the grey slider on the left side of the panel is update to show its own value. The slider second from the right 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 serial using the LM Flash Programmer application. When the “Update” button is pressed, the application transfers control to the boot loader to allow a new application image to be downloaded. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

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 serial using the LM Flash Programmer application. The application transfers control to the boot loader whenever it completes to allow a new image to be downloaded if required. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

2.6 Hello using Widgets (hello_widget)

A very simple “hello world” example written using the Stellaris Graphics Library widgets. It displays two buttons. The first, when pressed, toggles display of the words “Hello World!” on the screen. The second transfers control to the serial boot loader to perform a software update. This may be used as a starting point for more complex widget-based applications.

This application supports remote software update over serial using the LM Flash Programmer application. The application transfers control to the boot loader whenever the “Update Software” button is pressed to allow a new image to be downloaded. The LMFlash serial data rate must be set to

115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

2.7 Graphics Library String Table Demonstration (lang_demo)

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

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

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

The final panel provides instructions and information necessary to update the board firmware via serial using the LM Flash Programmer application. When the “Update” button is pressed, the application transfers control to the boot loader to allow a new application image to be downloaded. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

The string table and custom fonts used by this application can be found under /third_party/fonts/lang_demo. The original strings that the application intends displaying are found in the language.csv file (encoded in UTF8 format to allow accented characters and Asian language ideographs to be included. The mkstringtable tool is used to generate two versions of the string table, one which remains encoded in UTF8 format and the other which has been remapped to a custom codepage allowing the table to be reduced in size compared to the original UTF8 text. The tool also produces character map files listing each character used in the string table. These are then provided as input to the ftrasterize tool which generates two custom fonts for the application, one indexed using Unicode and a smaller one indexed using the custom codepage generated for this string table.

The command line parameters required for mkstringtable and ftrasterize can be found in the make-file in third_party/fonts/lang_demo.

By default, the application builds to use the custom codepage version of the string table and its matching custom font. To build using the UTF8 string table and Unicode-indexed custom font, ensure that the definition of **USE_REMAPPED_STRINGS** at the top of the lang_demo.c source file is commented out.

2.8 Quickstart Security Keypad (qs-keypad)

This application provides a security keypad to allow access to a door. Upon entry of the correct access code, a call is made to a function which could be used to set the state of a relay 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 provided 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

Input received by the application via the UART is interpreted as various commands which can be used to query the current access code, set a new code, show some statistics regarding the number of successful and unsuccessful access attempts and initiate a firmware update.

When the access code is changed, if a micro-SD card is present, the new code will be stored in a file called “key.txt” in the root directory. This file 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 type 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.

UART0, which is connected to the 3 pin header on the underside of the IDM-L35 RDK board (J3), 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.

This application supports remote software update via the serial interface using the LM Flash Programmer application. A firmware update is initiated either by entering “$*0$” on the numeric keypad or entering command “swupd” on the serial terminal command line.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

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 behavior can be modified using various commands entered via a terminal emulator connected to the IDM-L35 UART.

UART0, which is connected to the 3 pin header on the underside of the IDM-L35 RDK board (J3), 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.

This application supports remote software update over serial using the LM Flash Programmer application. Firmware updates can be initiated by entering the "swupd" command on the serial terminal. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

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`

This application supports remote software update over serial using the LM Flash Programmer application. Firmware updates can be initiated by entering the "swupd" command on the serial terminal. The LMFlash serial data rate must be set to 115200bps and the "Program Address Offset" to 0x800.

UART0, which is connected to the 6 pin header on the underside of the IDM-L35 RDK board (J8), is configured for 115200bps, and 8-n-1 mode. The USB-to-serial cable supplied with the IDM-L35 RDK may be used to connect this TTL-level UART to the host PC to allow firmware update.

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.

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:

-a specifies the index of the font character map to use in the conversion. If absent, Unicode is assumed when “-r” or “-u” is present. Without either of these switches, the Adobe Custom character map is used if such a map exists in the font, otherwise Unicode is used. This ensures backwards compatibility. To determine which character maps a font supports, call ftrasterize with the “-d” option to show font information.

-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.

-c FILENAME specifies the name of a file containing a list of character codes whose glyphs should be encoded into the output font. Each line of the file contains either a single decimal or hex character code in the chosen codepage (Unicode unless “-a” is provided), or two character codes separated by a comma and a space to indicate all characters in the inclusive range. Additionally, if the first non-comment line of the file is “REMAP”, the output font is generated to use a custom codepage with character codes starting at 1 and incrementing with every character in the character map file. This switch is only valid with “-r” and overrides “-p” and “-e” which are ignored if present.

-d displays details about the first font whose name is supplied at the end of the command line. When “-d” is used, all other switches are ignored. When used without “-v”, font header information and properties are shown along with the total number if characters encoded by the font and the number of contiguous blocks these characters are found in. With “-v”, detailed information on the character blocks is also displayed.

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

-h shows command line help information.

-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. If the size provided starts with "F", it is assumed that the following number is an index into the font's fixed size table. For example "-s F3" would select the fourth fixed size offered by the font. To determine whether a given font supports a fixed size table, use ftrasterize with the "-d" switch.

-p NUM specifies the index of the first character in the font that is to be encoded. If the value is not provided, it defaults to 32 which is typically the space character. This switch is ignored if "-c" is provided.

-e NUM specifies the index of the last character in the font that is to be encoded. If the value is not provided, it defaults to 126 which, in ISO8859-1 is tilde. This switch is ignored if "-c" is provided.

-v specifies that verbose output should be generated.

-w NUM encodes the specified character index as a space regardless of the character which may be present in the font at that location. This is helpful in allowing a space to be included in a font which only encodes a subset of the characters which would not normally include the space character (for example, numeric digits only). If absent, this value defaults to 32, ensuring that character 32 is always the space. Ignored if "-r" is specified.

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

-u causes ftrasterize to use Unicode character mapping when extracting glyphs from the source font. If absent, the Adobe Custom character map is used if it exists or Unicode otherwise.

-r specifies that the output should be a relocatable, wide character set font described using the tFontWide structure. Such fonts are suitable for encoding characters sets described using Unicode or when multiple contiguous blocks of characters are to be stored in a single font file. This switch may be used in conjunction with "-y" to create a binary font file suitable for use from a file system.

-y writes the output in binary rather than text format. This switch is only valid if used in conjunction with "-r" and is ignored otherwise. Fonts generated in binary format may be accessed by the graphics library from a file system or other indirect storage assuming that simple wrapper software is provided.

-o NUM specifies the codepoint for the first character in the source font which is to be translated to a new position in the output font. If this switch is not provided, no remapping takes place. If specified, this switch must be used in conjunction with -t which specifies where remapped characters are placed in the output font. Ignored if "-r" is specified.

-t NUM specifies the output font character index for the first character remapped from a higher codepoint in the source font. This should be used in conjunction with "-o". The default value is 0. Ignored if "-r" is specified.

-z NUM specifies the codepage identifier for the output font. This switch is only valid if used with "-r" and is primarily intended for use when performing codepage remapping and custom string tables. The number provided when performing remapping must be in the region between CODEPAGE_CUSTOM_BASE (0x8000) and 0xFFFF.

INPUT FILE specifies the name of the input font file. When used with "-r", up to four font filenames may be provided in order of priority. Characters missing from the first font are searched for in the remaining fonts. This allows the output font to contain characters from multiple different fonts and is helpful when generating multi-language string tables containing different alphabets which do not all exist in a single input font file.

Examples:

The following example produces a 24-point font called test and containing ASCII characters in the range 0x20 to 0x7F from test.ttf:

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

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

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

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

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

To generate some ISO8859 variant fonts, a block of characters from a source Unicode font must be moved downwards into the [0-255] codepoint range of the output font. This can be achieved by making use of the -t and -o switches. For example, the following will generate a font containing characters 32 to 255 of the ISO8859-5 character mapping. This contains the basic western European alphanumerics and the Cyrillic alphabet. The Cyrillic characters are found starting at Unicode character 1024 (0x400) but these must be placed starting at ISO8859-5 character number 160 (0xA0) so we encode characters 160 and above in the output from the Unicode block starting at 1024 to translate the Cyrillic glyphs into the correct position in the output:

```
ftrasterize -f cyrillic -s 18 -p 32 -e 255 -t 160 -o 1024 -u unicode.ttf
```

When encoding wide character sets for multiple alphabets (Roman, Arabic, Cyrillic, Hebrew, etc.) or to deal with ideograph-based writing systems (Hangul, Traditional or Simplified Chinese, Hiragana, Katakana, etc.), a character block map file is required to define which sections of the source font's codespace to encode into the destination font. The following example character map could be used to encode a font containing ASCII plus the Japanese Katakana alphabets:

```
################################################################################
#
# katakana.txt - Unicode block definitions for ASCII and Katakana.
#
################################################################################

# ASCII characters
0x20, 0x7E

# Katakana alphabet
0x30A0, 0x30FF
0x31F0, 0x32FF
0xFF00, 0xFFEF
```

Assuming the font "unicode.ttf" contains these glyphs and that it includes fixed size character renderings, the fifth of which uses an 8x12 character cell size, the following ftrasterize command line could then be used to generate a binary font file called fontkatakana8x12.bin containing this subset of characters:

```
ftrasterize -f katakana -s F4 -c katakana.txt -y -r -u unicode.ttf
```

In this case, the output file will be fontkatakana8x12.bin and it will contain a binary version of the font suitable for use from external memory (SDCard, a file system, serial flash memory, etc.) via a tFontWrapper and a suitable font wrapper module.

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`.

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. If encoding purely ASCII strings, the strings will also be compressed in order to reduce the space required to store them. If the CSV file contains strings encoded in other codepages, for example UTF8, the "-s" command line option must be used to specify the encoding used for the string and "-u" must also be used to ensure that the strings are stored correctly.

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 Deutsch",Prueba,Verifica
...
```

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

- GrLangEnUs – `Introduction in "English"`
- GrLangDE – `Einfuhrung, in Deutsch`
- 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 and two helper structure definitions, one a tCodepointMap array containing a single entry that is suitable for use with GrCodepageMapTableSet() and the other a tGrLibDefaults structure which can be used with GrLibInit() to initialize the graphics library to use the correct codepage for the string table.

While the contents of this .c file are readable, the string table itself may be unintelligible due to the compression or remapping 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 and external definitions for the tCodepointMap and tGrLibDefaults structures defined in the .c 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:

-u indicates that the input .csv file contains strings encoded with UTF8 or some other non-ASCII codepage. If absent, mkstringtable assumes ASCII text and uses this knowledge to apply higher compression to the string table.

-c NUM specifies the custom codepage identifier to use when remapping the string table for use with a custom font. Applications using the string table must set this value as the text codepage in use via a call to GrStringCodepageSet() and must ensure that they include an entry in their codepage mapping table specifying this value as both the source and font codepage. A macro, GRLIB_CUSTOM_MAP_xxx, containing the required tCodePointMap structure is written to the output header file to make this easier. A structure, g_GrLibDefaultxxxx, is also exported and a pointer to this may be passed to GrLibInit() to allow widgets to make use of the string table's custom codepage. Valid values to pass as NUM are in the 0x8000 to 0xFFFF range set aside by the graphics library for application-specific or custom text codepages.

-r indicates that the output string table should be constructed for use with a custom font and codepage. Character values in the string are remapped into a custom codepage intended to minimize the size of both the string table and the custom font used to display its strings. If "-r" is specified, an additional .txt output file is generated containing information that may be passed to the ftrasterize tool to create a compatible custom font containing only the characters required by the string table contents.

-s STR specifies the codepage used in the input .csv file. If this switch is absent, ASCII is assumed. Valid values of STR are "ASCII", "utf8" and "iso8859-n" where '"n" indicates the ISO8859 variant in use and can have values from 1 to 11 or 13 to 16. Care must be taken to ensure that the .csv file is correctly encoded and makes use of the encoding specified using "-s" since a mismatch will cause the output strings to be incorrect. Note, also, that support for UTF-8 and ISO8859 varies from text editor to text editor so it is important to ensure that your editor supports the desired codepage to prevent string corruption.

-t indicates that a character map file with a .txt extension should be generated in addition to the usual .c and .h output files. This output file may be passed to ftrasterize to generate a custom font containing only the glyphs required to display the strings in the table. Unlike the character map file generated when using "-r", the version generated by "-t" will not remap the codepage of the strings in the table. This has the advantage of leaving them readable in a debugger (which typically understands ASCII and common codepages) but will generate a font that is rather larger than the font that would have been generated using a remapped codepage due to the additional overhead of encoding many small blocks of discontiguous characters.

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 ASCII input file `str.csv`:

```
mkstringtable str.csv str
```

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

The following will create a string table in `widestr.c`, with prototypes in `widestr.h`, based on the UTF8 input file `widestr.csv`. This form of the call should be used to encode string tables containing accented characters or non-Western character sets:

```
mkstringtable -u widestr.csv widestr
```

In the produced `widestr.c`, there will be a string table in `g_pucTablewidestr`.

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 `P4`, `P5` or `P6` tags). The NetPBM image format can be produced using GIMP, NetPBM (`http://netpbm.sourceforge.net`), ImageMagick (`http://www.imagemagick.org`), or numerous other open source and proprietary image manipulation packages.

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

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

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

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

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

Arguments:

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

Example:

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

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

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

Serial Flash Downloader

Usage:

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

Description:

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

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

Arguments:

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

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

-d disables auto-baud.

-h displays usage information.

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

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

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

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

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-l35/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 at various brightness levels, and turning off the backlight.

The display driver can be configured for four different orientations:

- Landscape, 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. Landscape mode provides a 320x240 display and is selected by defining **LANDSCAPE**, or by not defining an orientation, when building the display driver.
- Portrait, which is the screen rotated 90 degrees clockwise (where the flex connector is on the left side of the display). Portrait mode provides a 240x320 display and is selected by defining **PORTRAIT** when building the display driver.
- Landscape flip, which is the screen rotated 180 degrees (where the flex connector is on the top side of the display). Landscape flip mode provides a 320x240 display and is selected by defining **LANDSCAPE_FLIP** when building the display driver.
- Portrait flip, which is the screen rotated 90 degrees counter-clockwise (where the flex connector is on the right of the display). Portrait flip mode provides a 240x320 display and is selected by defining **PORTRAIT_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 12 o'clock viewing angle (looking from the bottom of the display towards the top at an angle of approximately 45 degrees from the normal) so when used in landscape flip mode it has a 6 o'clock viewing angle. Similarly, it has a 3 o'clock viewing angle in portrait mode and a 9 o'clock viewing angle in portrait flip mode. Viewing the display from an angle other than the optimal viewing angle will result in reduction in the contrast ratio.

This driver is located in `boards/rdk-idm-l35/drivers`, with `kitronix320x240x16_ssd2119.c` containing the source code and `kitronix320x240x16_ssd2119.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

- void Kitronix320x240x16_SSD2119BacklightOff (void)
- void Kitronix320x240x16_SSD2119BacklightOn (unsigned char ucBrightness)
- void Kitronix320x240x16_SSD2119Init (void)

Variables

- const tDisplay g_sKitronix320x240x16_SSD2119

5.2.1 Function Documentation

5.2.1.1 Kitronix320x240x16_SSD2119BacklightOff

Turns off the backlight.

Prototype:

```
void
Kitronix320x240x16_SSD2119BacklightOff(void)
```

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

Returns:
None.

5.2.1.2 Kitronix320x240x16_SSD2119BacklightOn

Turns on the backlight.

Prototype:

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

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

Description:
This function sets the brightness of the display backlight.

Returns:
None.

5.2.1.3 Kitronix320x240x16_SSD2119Init

Initializes the display driver.

Prototype:

```
void
Kitronix320x240x16_SSD2119Init(void)
```

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

Returns:
None.

5.2.2 Variable Documentation

5.2.2.1 g_sKitronix320x240x16_SSD2119

Definition:

```
const tDisplay g_sKitronix320x240x16_SSD2119
```

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

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.
//
Kitronix240x320x16_SSD2119Init();

//
// Turn on the backlight.
//
Kitronix240x320x16_SSD2119BacklightOn(255);

//
// Initialize a graphics library drawing context.
//
GrContextInit(&sContext, &g_sKitronix240x320x16_SSD2119);
```

6 Sound Output Driver

6.1 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-l35/drivers`, with `sound.c` containing the source code and `sound.h` containing the API definitions for use by applications.

6.2 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)

6.2.1 Function Documentation

6.2.1.1 SoundDisable

Disables the sound output.

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

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

Returns:
None.

6.2.1.2 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.

6.2.1.3 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.

6.2.1.4 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.

6.2.1.5 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.

6.2.1.6 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 cancelled and the new song is started.

Returns:
None.

6.2.1.7 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.

6.2.1.8 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.

6.2.1.9 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.

6.2.1.10 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.

6.3 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();
```

7 Touch Screen Driver

7.1 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-l35/drivers`, with `touch.c` containing the source code and `touch.h` containing the API definitions for use by applications.

7.2 API Functions

Functions

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

7.2.1 Function Documentation

7.2.1.1 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.

7.2.1.2 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.

7.2.1.3 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.

7.3 Programming Example

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

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

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

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

    //
    // Determine what to do now.  In this case, we merely set flags that the
    // application main loop can deal with later.
    //
    switch(ulMessage)
    {
        case WIDGET_MSG_PTR_UP:
            g_ulFlags |= FLAG_PTR_UP;
            break;

        case WIDGET_MSG_PTR_DOWN:
            g_ulFlags |= FLAG_PTR_DOWN;
            break;

        case WIDGET_MSG_PTR_MOVE:
            g_ulFlags |= FLAG_PTR_MOVE;
            break;

        default:
            break;
    }

    return(0);
}

int main(void)
{
    ...

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

    ...

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

    ...
}
```

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

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

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

    ...

}
```

8 Command Line Processing Module

8.1 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.

8.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_INVALID_ARG
- CMDLINE_TOO_FEW_ARGS
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char ∗pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

8.2.1 Data Structure Documentation

8.2.1.1 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.

8.2.2 Define Documentation

8.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

8.2.2.2 CMDLINE_INVALID_ARG

Definition:

```
#define CMDLINE_INVALID_ARG
```

Description:

Defines the value that is returned if an argument is invalid.

8.2.2.3 CMDLINE_TOO_FEW_ARGS

Definition:

```
#define CMDLINE_TOO_FEW_ARGS
```

Description:

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

8.2.2.4 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

8.2.3 Function Documentation

8.2.3.1 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.

8.2.4 Variable Documentation

8.2.4.1 g_sCmdTable

Definition:

```
 tCmdLineEntry g_sCmdTable[]
```

Description:

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

8.3 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));
}
```

9 CPU Usage Module

9.1 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.

9.2 API Functions

Functions

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

9.2.1 Function Documentation

9.2.1.1 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.

9.2.1.2 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.

9.3 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();
    }
}
```

10 CRC Module

10.1 Introduction

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

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

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

10.2 API Functions

Functions

- unsigned short Crc16 (unsigned short usCrc, const unsigned char ∗pucData, unsigned long ulCount)
- unsigned short Crc16Array (unsigned long ulWordLen, const unsigned long ∗pulData)
- void Crc16Array3 (unsigned long ulWordLen, const unsigned long ∗pulData, unsigned short ∗pusCrc3)
- unsigned long Crc32 (unsigned long ulCrc, const unsigned char ∗pucData, unsigned long ulCount)
- unsigned char Crc8CCITT (unsigned char ucCrc, const unsigned char ∗pucData, unsigned long ulCount)

10.2.1 Function Documentation

10.2.1.1 Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

```
unsigned short
Crc16(unsigned short usCrc,
      const unsigned char *pucData,
      unsigned long ulCount)
```

Parameters:

usCrc is the starting CRC-16 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-16 of the input buffer. The CRC-16 is computed in a running fashion, meaning that the entire data block that is to have its CRC-16 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **usCrc** should be set to 0. If, however, the entire block of data is not available, then **usCrc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **usCrc** for the next portion of the data.

For example, to compute the CRC-16 of a block that has been split into three pieces, use the following:

```
usCrc = Crc16(0, pucData1, ulLen1);
usCrc = Crc16(usCrc, pucData2, ulLen2);
usCrc = Crc16(usCrc, pucData3, ulLen3);
```

Computing a CRC-16 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:

The CRC-16 of the input data.

10.2.1.2 Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

```
unsigned short
Crc16Array(unsigned long ulWordLen,
           const unsigned long *pulData)
```

Parameters:

ulWordLen is the length of the array in words (the number of bytes divided by 4).

pulData is a pointer to the data buffer.

Description:

This function is a wrapper around the running CRC-16 function, providing the CRC-16 for a single block of data.

Returns:

The CRC-16 of the input data.

10.2.1.3 Crc16Array3

Calculates three CRC-16s of an array of words.

Prototype:

```
void
Crc16Array3(unsigned long ulWordLen,
            const unsigned long *pulData,
            unsigned short *pusCrc3)
```

Parameters:

ulWordLen is the length of the array in words (the number of bytes divided by 4).

pulData is a pointer to the data buffer.

pusCrc3 is a pointer to an array in which to place the three CRC-16 values.

Description:

This function is used to calculate three CRC-16s of the input buffer; the first uses every byte from the array, the second uses only the even-index bytes from the array (in other words, bytes 0, 2, 4, etc.), and the third uses only the odd-index bytes from the array (in other words, bytes 1, 3, 5, etc.).

Returns:

None

10.2.1.4 Crc32

Calculates the CRC-32 of an array of bytes.

Prototype:

```
unsigned long
Crc32(unsigned long ulCrc,
      const unsigned char *pucData,
      unsigned long ulCount)
```

Parameters:

ulCrc is the starting CRC-32 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

This function is used to calculate the CRC-32 of the input buffer. The CRC-32 is computed in a running fashion, meaning that the entire data block that is to have its CRC-32 computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ulCrc** should be set to 0xFFFFFFFF. If, however, the entire block of data is not available, then **ulCrc** should be set to 0xFFFFFFFF for the first portion of the data, and then the returned value should be passed back in as **ulCrc** for the next portion of the data. Once all data has been passed to the function, the final CRC-32 can be obtained by inverting the last returned value.

For example, to compute the CRC-32 of a block that has been split into three pieces, use the following:

```
ulCrc = Crc32(0xFFFFFFFF, pucData1, ulLen1);
ulCrc = Crc32(ulCrc, pucData2, ulLen2);
ulCrc = Crc32(ulCrc, pucData3, ulLen3);
ulCrc ^= 0xFFFFFFFF;
```

Computing a CRC-32 in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:
The accumulated CRC-32 of the input data.

10.2.1.5 Crc8CCITT

Calculates the CRC-8-CCITT of an array of bytes.

Prototype:
```
unsigned char
Crc8CCITT(unsigned char ucCrc,
          const unsigned char *pucData,
          unsigned long ulCount)
```

Parameters:
ucCrc is the starting CRC-8-CCITT value.
pucData is a pointer to the data buffer.
ulCount is the number of bytes in the data buffer.

Description:
This function is used to calculate the CRC-8-CCITT of the input buffer. The CRC-8-CCITT is computed in a running fashion, meaning that the entire data block that is to have its CRC-8-CCITT computed does not need to be supplied all at once. If the input buffer contains the entire block of data, then **ucCrc** should be set to 0. If, however, the entire block of data is not available, then **ucCrc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **ucCrc** for the next portion of the data.

For example, to compute the CRC-8-CCITT of a block that has been split into three pieces, use the following:

```
ucCrc = Crc8CCITT(0, pucData1, ulLen1);
ucCrc = Crc8CCITT(ucCrc, pucData2, ulLen2);
ucCrc = Crc8CCITT(ucCrc, pucData3, ulLen3);
```

Computing a CRC-8-CCITT in a running fashion is useful in cases where the data is arriving via a serial link (for example) and is therefore not all available at one time.

Returns:
The CRC-8-CCITT of the input data.

10.3 Programming Example

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

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

//
// Fill pucData with some data.
```

```
//
for(ulIdx = 0; ulIdx < 256; ulIdx++)
{
    pucData[ulIdx] = ulIdx;
}

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

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 Ring Buffer Module

13.1 Introduction

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

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

13.2 API Functions

Functions

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

13.2.1 Function Documentation

13.2.1.1 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.

13.2.1.2 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.

13.2.1.3 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.

13.2.1.4 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.

13.2.1.5 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.

13.2.1.6 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.

13.2.1.7 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.

13.2.1.8 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.

13.2.1.9 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.

13.2.1.10 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.

13.2.1.11 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.

13.2.1.12 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.

13.2.1.13 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.

13.2.1.14 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.

13.2.1.15 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.

13.3 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);
```

14 Simple Task Scheduler Module

14.1 Introduction

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

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

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

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

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

14.2 API Functions

Data Structures

- tSchedulerTask

Functions

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

- unsigned long SchedulerTickCountGet (void)

Variables

- tSchedulerTask g_psSchedulerTable[ ]
- unsigned long g_ulSchedulerNumTasks

14.2.1 Data Structure Documentation

14.2.1.1 tSchedulerTask

Definition:

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

Members:

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

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

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

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

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

Description:

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

14.2.2 Function Documentation

14.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

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

14.2.2.2 SchedulerElapsedTicksGet

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

Prototype:

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

Parameters:

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

Description:

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

Returns:

The number of ticks elapsed since the provided tick count.

14.2.2.3 SchedulerInit

Initializes the task scheduler.

Prototype:

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

Parameters:

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

Description:

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

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

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

Returns:
None.

14.2.2.4 SchedulerRun

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

Prototype:
```
void
SchedulerRun(void)
```

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

Although the scheduler makes use of the SysTick interrupt, all calls to functions configured in *g_psSchedulerTable* are made in the context of SchedulerRun().

Returns:
None.

14.2.2.5 SchedulerSysTickIntHandler

Handles the SysTick interrupt on behalf of the scheduler module.

Prototype:
```
void
SchedulerSysTickIntHandler(void)
```

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

Returns:
None.

14.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

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

Parameters:

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

Description:

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

Returns:

None.

14.2.2.7 SchedulerTaskEnable

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

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

None.

14.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

Prototype:

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

Description:

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

Returns:

Tick count since last boot.

14.2.3 Variable Documentation

14.2.3.1 g_psSchedulerTable

Definition:

```
tSchedulerTask g_psSchedulerTable[]
```

Description:

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

14.2.3.2 g_ulSchedulerNumTasks

Definition:

```
unsigned long g_ulSchedulerNumTasks
```

Description:

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

14.3 Programming Example

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

```
//*****************************************************************************
//
// Definition of the system tick rate.  This results in a tick period of 10mS.
//
//*****************************************************************************
#define TICKS_PER_SECOND 100

//*****************************************************************************
//
// Prototypes of functions which will be called by the scheduler.
//
//*****************************************************************************
static void ScrollTextBanner(void *pvParam);
static void ToggleLED(void *pvParam);

//*****************************************************************************
//
// This table defines all the tasks that the scheduler is to run, the periods
// between calls to those tasks, and the parameter to pass to the task.
//
//*****************************************************************************
tSchedulerTask g_psSchedulerTable[] =
{
    //
    // Scroll the text banner 1 character to the left.  This function is called
    // every 20 ticks (5 times per second).
    //
    { ScrollTextBanner, (void *)0, 20, 0, true},
    //
    // Toggle LED number 0 every 50 ticks (twice per second).
    //
    { ToggleLED, (void *)0, 50, 0, true},

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

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

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

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

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

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

    //
    // Initialize the task scheduler and configure the SysTick to interrupt
    // 100 times per second.
    //
    SchedulerInit(TICKS_PER_SECOND);

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

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

15 Sine Calculation Module

15.1 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.

15.2 API Functions

Defines

- cosine(ulAngle)

Functions

- long sine (unsigned long ulAngle)

15.2.1 Define Documentation

15.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

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

Parameters:

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

Description:

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

Returns:

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

15.2.2 Function Documentation

15.2.2.1 sine

Computes an approximation of the sine of the input angle.

Prototype:

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

Parameters:

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

Description:

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

Returns:

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

15.3 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);
}
```

16 Micro Standard Library Module

16.1 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.

16.2 API Functions

Data Structures

- tTime

Functions

- void ulocaltime (unsigned long ulTime, tTime *psTime)
- unsigned long umktime (tTime *psTime)
- int urand (void)
- int usnprintf (char *pcBuf, unsigned long ulSize, const char *pcString,...)
- int usprintf (char *pcBuf, const char *pcString,...)
- void usrand (unsigned long ulSeed)
- int ustrcasecmp (const char *pcStr1, const char *pcStr2)
- int ustrcmp (const char *pcStr1, const char *pcStr2)
- int ustrlen (const char *pcStr)
- int ustrncmp (const char *pcStr1, const char *pcStr2, int iCount)

- char ∗ ustrncpy (char ∗pcDst, const char ∗pcSrc, int iNum)
- int ustrnicmp (const char ∗pcStr1, const char ∗pcStr2, int iCount)
- char ∗ ustrstr (const char ∗pcHaystack, const char ∗pcNeedle)
- unsigned long ustrtoul (const char ∗pcStr, const char ∗∗ppcStrRet, int iBase)
- int uvsnprintf (char ∗pcBuf, unsigned long ulSize, const char ∗pcString, va_list vaArgP)

16.2.1 Data Structure Documentation

16.2.1.1 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.

16.2.2 Function Documentation

16.2.2.1 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.

16.2.2.2 umktime

Converts calendar date and time to seconds.

Prototype:
```
unsigned long
umktime(tTime *psTime)
```

Parameters:
psTime is a pointer to the time structure that is filled in with the broken down date and time.

Description:
This function converts the date and time represented by the *psTime* structure pointer to the number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch).

Returns:
Returns the calendar time and date as seconds. If the conversion was not possible then the function returns (unsigned long)(-1).

16.2.2.3 urand

Generate a new (pseudo) random number

Prototype:
```
int
urand(void)
```

Description:
This function is very similar to the C library `rand()` function. It will generate a pseudo-random number sequence based on the seed value.

Returns:
A pseudo-random number will be returned.

16.2.2.4 usnprintf

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

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

Parameters:

pcBuf is the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

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

Description:

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

- %c to print a character
- %d or %i 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, %i, %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.

16.2.2.5 usprintf

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

Prototype:

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

Parameters:

pcBuf is the buffer where the converted string is stored.

pcString is the format string.

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

Description:

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

- %c to print a character
- %d or %i 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, %i, %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.

16.2.2.6 usrand

Set the random number generator seed.

Prototype:

```
void
usrand(unsigned long ulSeed)
```

Parameters:

ulSeed is the new seed value to use for the random number generator.

Description:

This function is very similar to the C library `srand()` function. It will set the seed value used in the `urand()` function.

Returns:

None

16.2.2.7 ustrcasecmp

Compares two strings without regard to case.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

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

Returns:

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

16.2.2.8 ustrcmp

Compares two strings.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

This function is very similar to the C library `strcmp()` function. It compares two strings, taking case into account. The comparison ends if a terminating NULL character is found in either string. In this case, the shorter string is deemed the lesser.

Returns:

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

16.2.2.9 ustrlen

Retruns the length of a null-terminated string.

Prototype:

```
int
ustrlen(const char *pcStr)
```

Parameters:

pcStr is a pointer to the string whose length is to be found.

Description:

This function is very similar to the C library `strlen()` function. It determines the length of the null-terminated string passed and returns this to the caller.

This implementation assumes that single byte character strings are passed and will return incorrect values if passed some UTF-8 strings.

Returns:

Returns the length of the string pointed to by *pcStr*.

16.2.2.10 ustrncmp

Compares two strings.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

This function is very similar to the C library `strncmp()` function. It compares at most *iCount* characters of two strings taking case into account. The comparison ends if a terminating NULL character is found in either string before *iCount* characters are compared. In this case, the shorter string is deemed the lesser.

Returns:

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

16.2.2.11 ustrncpy

Copies a certain number of characters from one string to another.

Prototype:

```
char *
ustrncpy(char *pcDst,
         const char *pcSrc,
         int iNum)
```

Parameters:

pcDst is a pointer to the destination buffer into which characters are to be copied.

pcSrc is a pointer to the string from which characters are to be copied.

iNum is the number of characters to copy to the destination buffer.

Description:

This function copies at most *iNum* characters from the string pointed to by *pcSrc* into the buffer pointed to by *pcDst*. If the end of *pcSrc* is found before *iNum* characters have been copied, remaining characters in *pcDst* will be padded with zeroes until *iNum* characters have been written. Note that the destination string will only be NULL terminated if the number of characters to be copied is greater than the length of *pcSrc*.

Returns:

Returns *pcDst*.

16.2.2.12 ustrnicmp

Compares two strings without regard to case.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

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

Returns:

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

16.2.2.13 ustrstr

Finds a substring within a string.

Prototype:

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

Parameters:

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

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

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

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

16.2.2.14 ustrtoul

Converts a string into its numeric equivalent.

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

Parameters:
pcStr is a pointer to the string containing the integer.
ppcStrRet is a pointer that will be set to the first character past the integer in the string.
iBase is the radix to use for the conversion; can be zero to auto-select the radix or between 2 and 16 to explicitly specify the radix.

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

Returns:
Returns the result of the conversion.

16.2.2.15 uvsnprintf

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

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

Parameters:
pcBuf points to the buffer where the converted string is stored.
ulSize is the size of the buffer.
pcString is the format string.
vaArgP is the list of optional arguments, which depend on the contents of the format string.

Description:

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

- %c to print a character
- %d or %i 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, %i, %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.

16.3 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);
```

17 UART Standard IO Module

17.1 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.

17.1.1 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.

17.1.2 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.

17.2 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 UARTStdioConfig (unsigned long ulPortNum, unsigned long ulBaud, unsigned long ulSrcClock)
- 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)

17.2.1 Function Documentation

17.2.1.1 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.

17.2.1.2 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.

17.2.1.3 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.

17.2.1.4 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.

17.2.1.5 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.

17.2.1.6 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.

17.2.1.7 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 or %i 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, %i, %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.

17.2.1.8 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.

17.2.1.9 UARTStdioConfig

Configures the UART console.

Prototype:

```
void
UARTStdioConfig(unsigned long ulPortNum,
                unsigned long ulBaud,
                unsigned long ulSrcClock)
```

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.

ulSrcClock is the frequency of the source clock for the UART module.

Description:

This function will configure the specified serial port to be used as a serial console. The serial parameters are set to the baud rate specified by the *ulBaud* parameter and use 8 bit, no parity, and 1 stop bit.

This function must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). This function assumes that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

17.2.1.10 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.

17.2.1.11 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, SysCtlClock-Set() 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.

17.2.1.12 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.

17.2.1.13 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.

17.2.1.14 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.

17.3 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

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

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Mobile Processors	www.ti.com/omap		
Wireless Connectivity	www.ti.com/wirelessconnectivity		
		TI E2E Community Home Page	e2e.ti.com