EK-LM4F232 Firmware Development Package

USER'S GUIDE

SW-EK-LM4F232-UG-1.0

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 1.0 of this document, last updated on April 11, 2013.

Table of Contents

1 Introduction

The Texas Instruments® Stellaris® EK-LM4F232 evaluation board is a platform that can be used for software development and prototyping a hardware design. It can also be used as a guide for custom board design using a Stellaris microcontroller.

The EK-LM4F232 includes a Stellaris ARM® Cortex™-M3-based microcontroller and the following features:

- Stellaris® LM4F232H5QD microcontroller
- Four 20V analog inputs
- 3-axis analog accelerometer
- On-board temperature sensor
- Bright 96 x 64 16-bit color OLED display
- 5 user buttons
- User LED
- Shunt for microcontroller current consumption measurement
- MicroSD card connector
- USB OTG connector
- On-board In-Circuit Debug Interface (ICDI)
- Coin cell backup battery for Hibernate feature
- Power supply option from USB ICDI connection, or OTG connection

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

2 Example Applications

The example applications show how to utilize features of the EK-LM4F232 evaluation board. Examples are included to show how to use many of the general features of the Stellaris microcontroller, as well as the feature that are unique to this evaluation board.

A number of drivers are provided to make it easier to use the features of the EK-LM4F232. These drivers also contain low-level code that make use of the Stellaris peripheral driver library and utilities.

There is an IAR workspace file (`ek-lm4f232.eww`) that contains the peripheral driver library project, 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 (`ek-lm4f232.mpw`) that contains the peripheral driver library project, 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/ek-lm4f232` subdirectory of the firmware development package source distribution.

2.1 Bit-Banding (bitband)

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

2.2 Blinky (blinky)

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

2.3 Boot Loader Demo 1 (boot_demo1)

An example to demonstrate the use of a flash-based boot loader. At startup, the application will configure the UART and USB peripherals, and then branch to the boot loader to await the start of an update. If using the serial boot loader (boot_serial), the UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

This application is intended for use with any of the three flash-based boot loader flavors (boot_serial or boot_usb) included in the software release. To accommodate the largest of these (boot_usb), the link address is set to 0x2800. If you are using serial, you may change this address to a 1KB boundary higher than the last address occupied by the boot loader binary as long as you also rebuild the boot loader itself after modifying its bl_config.h file to set APP_START_ADDRESS to the same value.

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

Note that the LM4F232 and other Blizzard-class devices also support the serial and USB boot loaders in ROM. To make use of this function, link your application to run at address 0x0000 in flash and enter the bootloader using the ROM_UpdateSerial and ROM_UpdateUSB functions (defined in rom.h). This mechanism is used in the utils/swupdate.c module when built specifically targeting a suitable Blizzard-class device.

2.4 Boot Loader Demo 2 (boot_demo2)

An example to demonstrate the use of a flash-based boot loader. At startup, the application will configure the UART, USB and Ethernet peripherals, wait for a widget on the screen to be pressed, and then branch to the boot loader to await the start of an update. If using the serial boot loader (boot_serial), the UART will always be configured at 115,200 baud and does not require the use of auto-bauding.

This application is intended for use with any of the three flash-based boot loader flavors (boot_eth, boot_serial or boot_usb) included in the software release. To accommodate the largest of these (boot_usb), the link address is set to 0x2800. If you are using serial or Ethernet boot loader, you may change this address to a 1KB boundary higher than the last address occupied by the boot loader binary as long as you also rebuild the boot loader itself after modifying its bl_config.h file to set APP_START_ADDRESS to the same value.

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

Note that the LM4F232 and other Blizzard-class devices also the support serial and USB boot loaders in ROM. To make use of this function, link your application to run at address 0x0000 in flash and enter the bootloader using either the ROM_UpdateUSB or ROM_UpdateSerial functions (defined in rom.h). This mechanism is used in the utils/swupdate.c module when built specifically targeting a suitable Blizzard-class device.

2.5 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 Tiva C Series microcontroller, utilizing either UART0, I2C0, SSI0, or USB. 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.

The configuration is set to boot applications which are linked to run from address 0x2800 in flash. This is higher than strictly necessary but is intended to allow the example boot loader-aware applications provided in the release to be used with any of the three boot loader example configurations supplied (serial or USB) without having to adjust their link addresses.

Note that the LM4F232 and other Blizzard-class devices also support serial boot loaders in ROM.

2.6 USB Boot Loader (boot_usb)

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

Applications intended for use with this version of the boot loader should be linked to run from address 0x2800 in flash (rather than the default run address of 0). This address is chosen to ensure that boot loader images built with all supported compilers may be used without modifying the application start address. Depending upon the compiler and optimization level you are using, however, you may find that you can reclaim some space by lowering this address and rebuilding both the application and boot loader. To do this, modify the makefile or project you use to build the application to show the new run address and also change the APP_START_ADDRESS value defined in bl_config.h before rebuilding the boot loader.

The USB boot loader may be demonstrated using the boot_demo1 and boot_demo2 example applications in addition to the boot_usb boot loader binary itself. Note that these are the only two example applications currently configured to run alongside the USB boot loader but making any of the other applications boot loader compatible is simply a matter of relinking them with the new start address and adding a mechanism to transfer control to the boot loader when required.

The Windows device driver required to communicate with the USB boot loader can be found on the software and documentation CD from the development kit package. It can also be found in the Windows driver package which can be downloaded via a link from `http://www.ti.com/tiva`.

A Windows command-line application, dfuprog, is also provided which illustrates how to perform uploads and downloads via the USB DFU protocol. The source for this application can be found in the /ti/TivaWare-for-C-Series/tools directory and the prebuilt executable is available in the package “Windows-side examples for USB kits” available for download via a link from `http://www.ti.com/tivaware`.

2.7 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 example uses calls to the TivaWare Graphics Library graphics primitives functions to update the display. For a similar example using widgets, please see “hello_widget”.

2.8 Hibernate Example (hibernate)

An example to demonstrate the use of the Hibernation module. The user can put the microcontroller in hibernation by pressing the select button. The microcontroller will then wake on its own after 5 seconds, or immediately if the user presses the select button again. The program keeps a count of the number of times it has entered hibernation. The value of the counter is stored in the battery backed memory of the Hibernation module so that it can be retrieved when the microcontroller wakes.

2.9 Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M4 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the display; GPIO pins D0, D1 and D2 will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

2.10 MPU (mpu_fault)

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

2.11 Data Logger (qs-logger)

This example application is a data logger. It can be configured to collect data from up to 10 data sources. The possible data sources are:

- 4 analog inputs, 0-20V
- 3-axis accelerometer
- internal and external temperature sensors
- processor current consumption

The data logger provides a menu navigation that is operated by the buttons on the EK-LM4F232 board (up, down, left, right, select). The data logger can be configured by using the menus. The following items can be configured:

- data sources to be logged
- sample rate
- storage location
- sleep modes
- clock

Using the data logger:

Use the CONFIG menu to configure the data logger. The following choices are provided:

- CHANNELS - enable specific channels of data that will be logged
- PERIOD - select the sample period
- STORAGE - select where the collected data will be stored:
  - FLASH - stored in the internal flash memory
  - USB - stored on a connected USB memory stick

  - HOST PC - transmitted to a host PC via USB OTG virtual serial port
  - NONE - the data will only be displayed and not stored
- SLEEP - select whether or not the board sleeps between samples. Sleep mode is allowed when storing to flash at with a period of 1 second or longer.
- CLOCK - allows setting of internal time-of-day clock that is used for time stamping of the sampled data

Use the START menu to start the data logger running. It will begin collecting and storing the data. It will continue to collect data until stopped by pressing the left button or select button.

While the data logger is collecting data and it is not configured to sleep, a simple strip chart showing the collected data will appear on the display. If the data logger is configured to sleep, then no strip chart will be shown.

If the data logger is storing to internal flash memory, it will overwrite the oldest data. If storing to a USB memory device it will store data until the device is full.

The VIEW menu allows viewing the values of the data sources in numerical format. When viewed this way the data is not stored.

The SAVE menu allows saving data that was stored in internal flash memory to a USB stick. The data will be saved in a text file in CSV format.

The ERASE menu is used to erase the internal memory so more data can be saved.

When the EK-LM4F232 board running qs-logger is connected to a host PC via the USB OTG connection for the first time, Windows will prompt for a device driver for the board. This can be found in /ti/TivaWare-for-C-Series/windows_drivers assuming you installed the software in the default folder.

A companion Windows application, logger, can be found in the /ti/TivaWare-for-C-Series/tools/bin directory. When the data logger's STORAGE option is set to "HOST PC" and the board is connected to a PC via the USB OTG connection, captured data will be transfered back to the PC using the virtual serial port that the EK board offers. When the logger application is run, it will search for the first connected EK-LM4F232 board and display any sample data received. The application also offers the option to log the data to a file on the PC.

2.12 SD card using FAT file system (sd_card)

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

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

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

2.13 Sine Demo (sine_demo)

This example uses the floating point capabilities of the Tiva C Series processor to compute a sine wave and show it on the display.

2.14 SoftUART Echo (softuart_echo)

This example application utilizes the SoftUART to echo text. The SoftUART is configured to use the same pins as the first UART (connected to the FTDI virtual serial port on the evaluation board), at 115,200 baud, 8-n-1 mode. All characters received on the SoftUART are transmitted back to the SoftUART.

2.15 Timer (timers)

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

2.16 UART Echo (uart_echo)

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

2.17 uDMA (udma_demo)

This example application demonstrates the use of the uDMA controller to transfer data between memory buffers, and to transfer data to and from a UART. The test runs for 10 seconds before exiting.

2.18 USB Generic Bulk Device (usb_dev_bulk)

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

A Windows INF file for the device is provided on the installation CD and in the C:/ti/TivaWare-for-C-Series/windows_drivers directory of TivaWare releases. This INF contains information required to install the WinUSB subsystem on WindowsXP and Vista PCs. WinUSB is a Windows subsystem

allowing user mode applications to access the USB device without the need for a vendor-specific kernel mode driver.

A sample Windows command-line application, usb_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided. The application binary is installed as part of the "Windows-side examples for USB kits" package (SW-USB-win) on the installation CD or via download from `http://www.ti.com/tivaware` . Project files are included to allow the examples to be built using Microsoft Visual Studio 2008. Source code for this application can be found in directory ti/TivaWare-for-C-Series/tools/usb_bulk_example.

2.19 USB HID Keyboard Device (usb_dev_keyboard)

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

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

2.20 USB MSC Device (usb_dev_msc)

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

2.21 USB Serial Device (usb_dev_serial)

This example application turns the evaluation kit into a virtual serial port when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect UART0 traffic to and from the USB host system.

Assuming you installed TivaWare in the default directory, a driver information (INF) file for use with Windows XP, Windows Vista and Windows7 can be found in C:/ti/TivaWare-for-C-Series/windows_drivers. For Windows 2000, the required INF file is in C:/ti/TivaWare-for-C-Series/windows_drivers/win2K.

2.22 USB host audio example application using SD Card FAT file system (usb_host_audio)

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

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

2.23 USB HID Keyboard Host (usb_host_keyboard)

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

2.24 USB HID Mouse Host (usb_host_mouse)

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

2.25 USB Mass Storage Class Host Example (usb_host_msc)

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

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

2.26 USB Stick Update Demo (usb_stick_demo)

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

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

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

2.27 USB Memory Stick Updater (usb_stick_update)

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

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

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

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

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

2.28 Watchdog (watchdog)

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

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 TI Tiva microcontrollers.

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

USB DFU Programmer

Usage:

```
dfuprog [OPTION]...
```

Description:

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

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

Arguments:

-e specifies the address of the binary.

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

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

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

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

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

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

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

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

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

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

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

-v displays verbose output during the requested operation.

-h displays this help information.

-? displays this help information.

Example:

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

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

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

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

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

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

USB DFU Wrapper

Usage:

`dfuwrap [OPTION]...`

Description:

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

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

Arguments:

-a ADDR specifies the address of the binary.

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

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

-e enables verbose output.

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

-h displays usage information.

-i FILE specifies the name of the input file.

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

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

-q specifies that only error information should be output.

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

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

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

Example:

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

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

FreeType Rasterizer

Usage:

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

Description:

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

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

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

Arguments:

-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.

USB DFU Library

Description:

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

The source code for this DLL is contained in `tools/lmdfu`. The DLL binary is installed as part of the "TivaWare Embedded USB drivers" package (SW-USB-windrivers) shipped on the release CD and downloadable from http://www.ti.com/tivaware. A Microsoft Visual Studio 2008 project file is provided to allow the application to be built.

USB Dynamic Link Library

Description:

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

The source code for this DLL is contained in `tools/lmusbdll`. The DLL binary is installed as part of the "Tiva Embedded USB drivers" package (SW-USB-windrivers) shipped on the release CD and downloadable from http://www.ti.com/tivaware. A Microsoft Visual Studio 2008 project file is provided to allow the DLL to be built on a PC which has the Windows Device Driver Kit installed.

Data Logger

Usage:

```
logger
```

Description:

Provides a Windows front-end for the ek-lm4f232 data logger application (qs-logger) and allows captured data to be logged to file and displayed on the screen in several strip charts.

The qs-logger application provides a virtual COM port via USB and the logger application opens this and parses data received from the board as it is captured. All control, other than setting up the file and deciding which captured channels' data to display, is performed using the menus provided by the qs-logger application on the ek-lm4f232 board.

The device driver required to support the qs-logger application's virtual COM port on Windows can be found in `windows_drivers`.

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

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 TivaWare 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.

-i indicates that the string table should be written into a binary file rather than included within the .c output file. The binary output name will use the same filename as the .c and .h

files but will have a .bin file extension. When a binary file is generated, the .c file will contain all structures usually generated but will not contain the array containing the string table data. A binary string table is position-independent and may be stored anywhere in memory on condition that it is aligned on a 32-bit word boundary. The binary string table is used in exactly the same way as the linked .c version except that the parameter passed to GrStringTableSet() must be the address at which the application located the first word of the string table data rather than a label exported from the string table's .c file.

-I FILENAME specifies an additional header file which is parsed for additional language IDs. Although the tool recognizes all the GrLangXxx labels found in grlib.h, applications may define their own language IDs in a header file and pass this to mkstringtable. This header will be included in the output C files allowing custom language labels to be used without the need to edit the mkstringtable output files later. Header files passed with the "-I" parameter must contain only comments, blank lines and definitions of the form "#define LangIDLabel 0x0000" with the label's value given in hex and in the range 0x0001 to 0xFFFF.

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 TivaWare 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 Tiva board using a UART connection to the TivaWare Serial Flash Loader or the TivaWare Boot Loader. This has the same capabilities as the serial download portion of the LM Flash Programmer tool.

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

Arguments:

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

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

-d disables auto-baud.

-h displays usage information.

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

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

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

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

USB Bulk Data Transfer Example

Description:

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

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

4 Buttons Driver

4.1 Introduction

The buttons driver provides functions to make it easy to use the push buttons on the EK-LM4F232 evaluation board. The driver provides a function to initialize all the hardware required for the buttons, and features for debouncing and querying the button state.

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

4.2 API Functions

Functions

- void ButtonsInit (void)
- uint8_t ButtonsPoll (uint8_t *pui8Delta, uint8_t *pui8RawState)

4.2.1 Function Documentation

4.2.1.1 ButtonsInit

Initializes the GPIO pins used by the board pushbuttons.

Prototype:

```
void
ButtonsInit(void)
```

Description:

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

Returns:

None.

4.2.1.2 ButtonsPoll

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

Prototype:

```
uint8_t
ButtonsPoll(uint8_t *pui8Delta,
            uint8_t *pui8RawState)
```

Parameters:

pui8Delta points to a character that will be written to indicate which button states changed since the last time this function was called. This value is derived from the debounced state of the buttons.

pui8RawState points to a location where the raw button state will be stored.

Description:

This function should be called periodically by the application to poll the pushbuttons. It determines both the current debounced state of the buttons and also which buttons have changed state since the last time the function was called.

In order for button debouncing to work properly, this function should be caled at a regular interval, even if the state of the buttons is not needed that often.

If button debouncing is not required, the the caller can pass a pointer for the *pui8RawState* parameter in order to get the raw state of the buttons. The value returned in *pui8RawState* will be a bit mask where a 1 indicates the buttons is pressed.

Returns:

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

4.3 Programming Example

The following example shows how to use the buttons driver to initialize the buttons, debounce and read the buttons state.

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

//
// From timed processing loop (for example every 10 ms)
//
...
{
    //
    // Poll the buttons.  When called periodically this function will
    // run the button debouncing algorithm.
    //
    ucState = ButtonsPoll(&ucDelta, 0);

    //
    // Test to see if the SELECT button was pressed and do something
    //
    if(BUTTON_PRESSED(SELECT_BUTTON, ucState, ucDelta))
    {
        ...
        // SELECT button action
    }
}
```

5 Display Driver

5.1 Introduction

The display driver offers a standard interface to access display functions on the CrystalFontz 96x64 16-bit color OLED display and is used by the Stellaris Graphics Library and widget manager. In addition to providing the `tDisplay` structure required by the graphics library, the display driver also provides an API for initializing the display.

This driver is located in `boards/armadillo/drivers`, with `cfal96x64x16.c` containing the source code and `cfal96x64x16.h` containing the API definitions for use by applications.

5.2 API Functions

Functions

- void CFAL96x64x16Init (void)

Variables

- const tDisplay g_sCFAL96x64x16

5.2.1 Function Documentation

5.2.1.1 CFAL96x64x16Init

Initializes the display driver.

Prototype:

```
void
CFAL96x64x16Init(void)
```

Description:

This function initializes the SSD1332 display controller on the panel, preparing it to display data.

Returns:

None.

5.2.2 Variable Documentation

5.2.2.1 g_sCFAL96x64x16

Definition:

```
const tDisplay g_sCFAL96x64x16
```

Description:

The display structure that describes the driver for the Crystalfontz CFAL9664-F-B1 OLED panel with SSD 1332 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.
//
CFAL96x64x16Init();

//
// Initialize a graphics library drawing context.
//
GrContextInit(&sContext, &g_sCFAL96x64x16);
```

6 Command Line Processing Module

6.1 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.

6.2 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_psCmdTable[ ]

6.2.1 Data Structure Documentation

6.2.1.1 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.

6.2.2 Define Documentation

6.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

6.2.2.2 CMDLINE_INVALID_ARG

Definition:

```
#define CMDLINE_INVALID_ARG
```

Description:

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

6.2.2.3 CMDLINE_TOO_FEW_ARGS

Definition:

```
#define CMDLINE_TOO_FEW_ARGS
```

Description:

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

6.2.2.4 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

6.2.3 Function Documentation

6.2.3.1 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_psCmdTable` containing `tCmdLineEntry` structures which must be provided by the application. The array must be terminated with an entry whose **pcCmd** field contains a NULL pointer.

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.

6.2.4 Variable Documentation

6.2.4.1 g_psCmdTable

Definition:

```
tCmdLineEntry g_psCmdTable[]
```

Description:

This is the command table that must be provided by the application. The last element of the array must be a structure whose pcCmd field contains a NULL pointer.

6.3 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." },
    { 0, 0, 0 }
};

//
// 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));
}
```

7 CPU Usage Module

7.1 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.

7.2 API Functions

Functions

- void CPUUsageInit (uint32_t ui32ClockRate, uint32_t ui32Rate, uint32_t ui32Timer)
- uint32_t CPUUsageTick (void)

7.2.1 Function Documentation

7.2.1.1 CPUUsageInit

Initializes the CPU usage measurement module.

Prototype:

```
void
CPUUsageInit(uint32_t ui32ClockRate,
             uint32_t ui32Rate,
             uint32_t ui32Timer)
```

Parameters:

ui32ClockRate is the rate of the clock supplied to the timer module.

ui32Rate is the number of times per second that CPUUsageTick() is called.

ui32Timer 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.

7.2.1.2 CPUUsageTick

Updates the CPU usage for the new timing period.

Prototype:

```
uint32_t
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.

7.3 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();
    }
}
```

8 CRC Module

8.1 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.

8.2 API Functions

Functions

- uint16_t Crc16 (uint16_t ui16Crc, const uint8_t *pui8Data, uint32_t ui32Count)
- uint16_t Crc16Array (uint32_t ui32WordLen, const uint32_t *pui32Data)
- void Crc16Array3 (uint32_t ui32WordLen, const uint32_t *pui32Data, uint16_t *pui16Crc3)
- uint32_t Crc32 (uint32_t ui32Crc, const uint8_t *pui8Data, uint32_t ui32Count)
- uint8_t Crc8CCITT (uint8_t ui8Crc, const uint8_t *pui8Data, uint32_t ui32Count)

8.2.1 Function Documentation

8.2.1.1 Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

```
uint16_t
Crc16(uint16_t ui16Crc,
      const uint8_t *pui8Data,
      uint32_t ui32Count)
```

Parameters:

ui16Crc is the starting CRC-16 value.

pui8Data is a pointer to the data buffer.

ui32Count 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 **ui16Crc** should be set to 0. If, however, the entire block of data is not available, then **ui16Crc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **ui16Crc** 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:

```
ui16Crc = Crc16(0, pui8Data1, ui32Len1);
ui16Crc = Crc16(ui16Crc, pui8Data2, ui32Len2);
ui16Crc = Crc16(ui16Crc, pui8Data3, ui32Len3);
```

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.

8.2.1.2 Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

```
uint16_t
Crc16Array(uint32_t ui32WordLen,
           const uint32_t *pui32Data)
```

Parameters:

ui32WordLen is the length of the array in words (the number of bytes divided by 4).

pui32Data 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.

8.2.1.3 Crc16Array3

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

Prototype:

```
void
Crc16Array3(uint32_t ui32WordLen,
            const uint32_t *pui32Data,
            uint16_t *pui16Crc3)
```

Parameters:

ui32WordLen is the length of the array in words (the number of bytes divided by 4).

pui32Data is a pointer to the data buffer.

pui16Crc3 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

8.2.1.4 Crc32

Calculates the CRC-32 of an array of bytes.

Prototype:

```
uint32_t
Crc32(uint32_t ui32Crc,
      const uint8_t *pui8Data,
      uint32_t ui32Count)
```

Parameters:

ui32Crc is the starting CRC-32 value.

pui8Data is a pointer to the data buffer.

ui32Count 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 **ui32Crc** should be set to 0xFFFFFFFF. If, however, the entire block of data is not available, then **ui32Crc** should be set to 0xFFFFFFFF for the first portion of the data, and then the returned value should be passed back in as **ui32Crc** 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:

```
ui32Crc = Crc32(0xFFFFFFFF, pui8Data1, ui32Len1);
ui32Crc = Crc32(ui32Crc, pui8Data2, ui32Len2);
ui32Crc = Crc32(ui32Crc, pui8Data3, ui32Len3);
ui32Crc ^= 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.

8.2.1.5 Crc8CCITT

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

Prototype:

```
uint8_t
Crc8CCITT(uint8_t ui8Crc,
          const uint8_t *pui8Data,
          uint32_t ui32Count)
```

Parameters:

ui8Crc is the starting CRC-8-CCITT value.

pui8Data is a pointer to the data buffer.

ui32Count 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 **ui8Crc** should be set to 0. If, however, the entire block of data is not available, then **ui8Crc** should be set to 0 for the first portion of the data, and then the returned value should be passed back in as **ui8Crc** 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:

```
ui8Crc = Crc8CCITT(0, pui8Data1, ui32Len1);
ui8Crc = Crc8CCITT(ui8Crc, pui8Data2, ui32Len2);
ui8Crc = Crc8CCITT(ui8Crc, pui8Data3, ui32Len3);
```

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.

8.3 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);
```

9 Flash Parameter Block Module

9.1 Introduction

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

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

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

9.2 API Functions

Functions

- uint8_t * FlashPBGet (void)
- void FlashPBInit (uint32_t ui32Start, uint32_t ui32End, uint32_t ui32Size)
- void FlashPBSave (uint8_t *pui8Buffer)

9.2.1 Function Documentation

9.2.1.1 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

```
uint8_t *
FlashPBGet(void)
```

Description:

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

Returns:

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

9.2.1.2 FlashPBInit

Initializes the flash parameter block.

Prototype:

```
void
FlashPBInit(uint32_t ui32Start,
            uint32_t ui32End,
            uint32_t ui32Size)
```

Parameters:

ui32Start 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.

ui32End 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.

ui32Size 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 *ui32Start* and *ui32End* 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 (*ui32Size*). 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 *ui32Start* and *ui32End* 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 (*ui32Size*) 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 (*ui32End* - *ui32Start*) divided by the parameter block size (*ui32Size*) must be less than or equal to 128. If not, it will not be possible in all cases to determine which parameter block is the most recent (specifically when dealing with the sequence number wrapping back to zero).

When the microcontroller is initially programmed, the flash blocks used for parameter block storage are left in an erased state.

This function must be called before any other flash parameter block functions are called.

Returns:

None.

9.2.1.3 FlashPBSave

Writes a new parameter block to flash.

Prototype:

```
void
FlashPBSave(uint8_t *pui8Buffer)
```

Parameters:

pui8Buffer is the address of the parameter block to be written to flash.

Description:

This function will write a parameter block to flash. Saving the new parameter blocks involves three steps:

- Setting the sequence number such that it is one greater than the sequence number of the latest parameter block in flash.
- Computing the checksum of the parameter block.
- Writing the parameter block into the storage immediately following the latest parameter block in flash; if that storage is at the start of an erase block, that block is erased first.

By this process, there is always a valid parameter block in flash. If power is lost while writing a new parameter block, the checksum will not match and the partially written parameter block will be ignored. This is what makes this fault-tolerant.

Another benefit of this scheme is that it provides wear leveling on the flash. Since multiple parameter blocks fit into each erase block of flash, and multiple erase blocks are used for parameter block storage, it takes quite a few parameter block saves before flash is re-written.

Returns:

None.

9.3 Programming Example

The following example shows how to use the flash parameter block module to read the contents of a flash parameter block.

```
unsigned char pucBuffer[16], *pucPB;

//
// Initialize the flash parameter block module, using the last two pages of
// a 64 KB device as the parameter block.
//
FlashPBInit(0xf800, 0x10000, 16);

//
// Read the current parameter block.
//
pucPB = FlashPBGet();
if(pucPB)
{
    memcpy(pucBuffer, pucPB);
}
```

10 Integer Square Root Module

10.1 Introduction

The integer square root module provides an integer version of the square root operation that can be used instead of the floating point version provided in the C library. The algorithm used is a derivative of the manual pencil-and-paper method that used to be taught in school, and is closely related to the pencil-and-paper division method that is likely still taught in school.

For full details of the algorithm, see the article by Jack W. Crenshaw in the February 1998 issue of Embedded System Programming. It can be found online at `http://www.embedded.com/98/9802fe2.htm`.

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

10.2 API Functions

Functions

- uint32_t isqrt (uint32_t ui32Value)

10.2.1 Function Documentation

10.2.1.1 isqrt

Compute the integer square root of an integer.

Prototype:

```
uint32_t
isqrt(uint32_t ui32Value)
```

Parameters:

ui32Value is the value whose square root is desired.

Description:

This function will compute the integer square root of the given input value. Since the value returned is also an integer, it is actually better defined as the largest integer whose square is less than or equal to the input value.

Returns:

Returns the square root of the input value.

10.3 Programming Example

The following example shows how to compute the square root of a number.

```
unsigned long ulValue;

//
// Get the square root of 52378.  The result returned will be 228, which is
// the largest integer less than or equal to the square root of 52378.
//
ulValue = isqrt(52378);
```

11 random Utility Module

11.1 Introduction

This module implements an entropy-based random number generator (RNG). The API only consists of two functions: RandomAddEntropy() and RandomSeed(). Software uses RandomAddEntropy() to add random data (entropy) to the pool used by the RNG. Entropy can originate from various sources including ADC measurements, sensor readings, time between interrupts, timing user inputs (such as keyboard or mouse), etc. Essentially anything that can create random data.

This module can be used in different ways. The simplest method is to feed the entropy pool and call RandomSeed() to obtain a new random number. Assuming that the entropy pool has changed since the last call to RandomSeed(), a new random number is returned by RandomSeed(). If the entropy pool has not changed since the last call to RandomSeed(), the same number is returned.

Software can also use this module in combination with other RNG or pseudo-random number generator (PRNG) software. For example, software can feed the entropy pool and generate seed data that is fed into another, more sophisticated RNG or PRNG. A simple example of this is to use RandomSeed() to obtain the seed used by the PRNG functions located in `utils/ustdlib.c`.

How this module is used should be dictated by the requirements of the end user application.

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

11.2 API Functions

Functions

- void RandomAddEntropy (uint32_t ui32Entropy)
- uint32_t RandomSeed (void)

11.2.1 Function Documentation

11.2.1.1 RandomAddEntropy

Add entropy to the pool.

Prototype:

```
void
RandomAddEntropy(uint32_t ui32Entropy)
```

Parameters:

ui32Entropy is an 8-bit value that is added to the entropy pool

Description:
This function allows the user application code to add entropy (random data) to the pool at any time.

Returns:
None

11.2.1.2 RandomSeed

Set the random number generator seed.

Prototype:
```
uint32_t
RandomSeed(void)
```

Description:
Seed the random number generator by running a MD4 hash on the entropy pool. Note that the entropy pool may change from beneath us, but for the purposes of generating random numbers that is not a concern. Also, the MD4 hash was broken long ago, but since it is being used to generate random numbers instead of providing security this is not a concern.

Returns:
New seed value.

11.3 Programming Example

The following example shows how to produce a random number using the SysTick timer and an ADC reading to feed the entropy pool.

```
unsigned long ulRandomNumber1, ulRandomNumber2;

//
// Add entropy to the pool and generate a new random number.
//
RandomAddEntropy(SysTickValueGet());
ulRandomNumber1 = RandomSeed();

//
// Add entropy to the pool and generate a new random number.
//
RandomAddEntropy(ulADCValue);
ulRandomNumber2 = RandomSeed();
```

The following example shows how to use seed data from the RNG module with another piece of RNG software. This example adds entropy from various sources.

```
unsigned long ulRandomNumber1, ulRandomNumber2;

//
// Initialize the entropy pool.
//
RandomAddEntropy(SysTickValueGet());
RandomAddEntropy(ulSensorValue);
```

```
RandomAddEntropy(ulADCValue);

//
// Seed the random number generator.
//
usrand(RandomSeed());

//
// Generate random numbers.
//
ulRandomNumber1 = urand();
ulRandomNumber2 = urand();
```

12 Ring Buffer Module

12.1 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.

12.2 API Functions

Functions

- void RingBufAdvanceRead (tRingBufObject *psRingBuf, uint32_t ui32NumBytes)
- void RingBufAdvanceWrite (tRingBufObject *psRingBuf, uint32_t ui32NumBytes)
- uint32_t RingBufContigFree (tRingBufObject *psRingBuf)
- uint32_t RingBufContigUsed (tRingBufObject *psRingBuf)
- bool RingBufEmpty (tRingBufObject *psRingBuf)
- void RingBufFlush (tRingBufObject *psRingBuf)
- uint32_t RingBufFree (tRingBufObject *psRingBuf)
- bool RingBufFull (tRingBufObject *psRingBuf)
- void RingBufInit (tRingBufObject *psRingBuf, uint8_t *pui8Buf, uint32_t ui32Size)
- void RingBufRead (tRingBufObject *psRingBuf, uint8_t *pui8Data, uint32_t ui32Length)
- uint8_t RingBufReadOne (tRingBufObject *psRingBuf)
- uint32_t RingBufSize (tRingBufObject *psRingBuf)
- uint32_t RingBufUsed (tRingBufObject *psRingBuf)
- void RingBufWrite (tRingBufObject *psRingBuf, uint8_t *pui8Data, uint32_t ui32Length)
- void RingBufWriteOne (tRingBufObject *psRingBuf, uint8_t ui8Data)

12.2.1 Function Documentation

12.2.1.1 RingBufAdvanceRead

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

Prototype:

```
void
RingBufAdvanceRead(tRingBufObject *psRingBuf,
                   uint32_t ui32NumBytes)
```

Parameters:

psRingBuf points to the ring buffer from which bytes are to be removed.

ui32NumBytes 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 *ui32NumBytes* is larger than the number of bytes currently in the buffer, the buffer is emptied.

Returns:

None.

12.2.1.2 RingBufAdvanceWrite

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

Prototype:

```
void
RingBufAdvanceWrite(tRingBufObject *psRingBuf,
                    uint32_t ui32NumBytes)
```

Parameters:

psRingBuf points to the ring buffer to which bytes have been added.

ui32NumBytes 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 *ui32NumBytes* 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.

12.2.1.3 RingBufContigFree

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

Prototype:

```
uint32_t
RingBufContigFree(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.4 RingBufContigUsed

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

Prototype:

```
uint32_t
RingBufContigUsed(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.5 RingBufEmpty

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

Prototype:

```
bool
RingBufEmpty(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.6 RingBufFlush

Empties the ring buffer.

Prototype:

```
void
RingBufFlush(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf is the ring buffer object to empty.

Description:

Discards all data from the ring buffer.

Returns:
None.

12.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:
```
uint32_t
RingBufFree(tRingBufObject *psRingBuf)
```

Parameters:
psRingBuf 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.

12.2.1.8 RingBufFull

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

Prototype:
```
bool
RingBufFull(tRingBufObject *psRingBuf)
```

Parameters:
psRingBuf 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.

12.2.1.9 RingBufInit

Initialize a ring buffer object.

Prototype:
```
void
RingBufInit(tRingBufObject *psRingBuf,
            uint8_t *pui8Buf,
            uint32_t ui32Size)
```

Parameters:

psRingBuf points to the ring buffer to be initialized.

pui8Buf points to the data buffer to be used for the ring buffer.

ui32Size is the size of the buffer in bytes.

Description:

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

Returns:

None.

12.2.1.10 RingBufRead

Reads data from a ring buffer.

Prototype:

```
void
RingBufRead(tRingBufObject *psRingBuf,
            uint8_t *pui8Data,
            uint32_t ui32Length)
```

Parameters:

psRingBuf points to the ring buffer to be read from.

pui8Data points to where the data should be stored.

ui32Length is the number of bytes to be read.

Description:

This function reads a sequence of bytes from a ring buffer.

Returns:

None.

12.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:

```
uint8_t
RingBufReadOne(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

Prototype:

```
uint32_t
RingBufSize(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

Prototype:

```
uint32_t
RingBufUsed(tRingBufObject *psRingBuf)
```

Parameters:

psRingBuf 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.

12.2.1.14 RingBufWrite

Writes data to a ring buffer.

Prototype:

```
void
RingBufWrite(tRingBufObject *psRingBuf,
             uint8_t *pui8Data,
             uint32_t ui32Length)
```

Parameters:

psRingBuf points to the ring buffer to be written to.

pui8Data points to the data to be written.

ui32Length is the number of bytes to be written.

Description:

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

Returns:
None.

12.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:

```
void
RingBufWriteOne(tRingBufObject *psRingBuf,
                uint8_t ui8Data)
```

Parameters:

psRingBuf points to the ring buffer to be written to.

ui8Data is the byte to be written.

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

Returns:
None.

12.3 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);
```

13 Simple Task Scheduler Module

13.1 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.

13.2 API Functions

Data Structures

- tSchedulerTask

Functions

- uint32_t SchedulerElapsedTicksCalc (uint32_t ui32TickStart, uint32_t ui32TickEnd)
- uint32_t SchedulerElapsedTicksGet (uint32_t ui32TickCount)
- void SchedulerInit (uint32_t ui32TicksPerSecond)
- void SchedulerRun (void)
- void SchedulerSysTickIntHandler (void)
- void SchedulerTaskDisable (uint32_t ui32Index)
- void SchedulerTaskEnable (uint32_t ui32Index, bool bRunNow)
- uint32_t SchedulerTickCountGet (void)

Variables

- tSchedulerTask g_psSchedulerTable[]
- uint32_t g_ui32SchedulerNumTasks

13.2.1 Data Structure Documentation

13.2.1.1 tSchedulerTask

Definition:

```
typedef struct
{
    void (*pfnFunction)(void *);
    void *pvParam;
    uint32_t ui32FrequencyTicks;
    uint32_t ui32LastCall;
    bool 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.

ui32FrequencyTicks 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.

ui32LastCall 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.

13.2.2 Function Documentation

13.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

```
uint32_t
SchedulerElapsedTicksCalc(uint32_t ui32TickStart,
                          uint32_t ui32TickEnd)
```

Parameters:

ui32TickStart is the system tick count for the start of the period.

ui32TickEnd 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.

13.2.2.2 SchedulerElapsedTicksGet

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

Prototype:

```
uint32_t
SchedulerElapsedTicksGet(uint32_t ui32TickCount)
```

Parameters:

ui32TickCount 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 *ui32TickCount* 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 *ui32TickCount*. 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.

13.2.2.3 SchedulerInit

Initializes the task scheduler.

Prototype:

```
void
SchedulerInit(uint32_t ui32TicksPerSecond)
```

Parameters:

ui32TicksPerSecond 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.

13.2.2.4 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.

13.2.2.5 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.

13.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

Prototype:
```
void
SchedulerTaskDisable(uint32_t ui32Index)
```

Parameters:

ui32Index 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.

13.2.2.7 SchedulerTaskEnable

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

Prototype:

```
void
SchedulerTaskEnable(uint32_t ui32Index,
                    bool bRunNow)
```

Parameters:

ui32Index 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.

13.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

Prototype:

```
uint32_t
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.

13.2.3 Variable Documentation

13.2.3.1 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.

13.2.3.2 g_ui32SchedulerNumTasks

Definition:

```
uint32_t g_ui32SchedulerNumTasks
```

Description:

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

13.3 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();
    }
}
```

14 Sine Calculation Module

14.1 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.

14.2 API Functions

Defines

- cosine(ui32Angle)

Functions

- int32_t sine (uint32_t ui32Angle)

14.2.1 Define Documentation

14.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

```
#define cosine(ui32Angle)
```

Parameters:

ui32Angle 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.

14.2.2 Function Documentation

14.2.2.1 sine

Computes an approximation of the sine of the input angle.

Prototype:

```
int32_t
sine(uint32_t ui32Angle)
```

Parameters:

ui32Angle 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.

14.3 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);
}
```

15 SMBus Stack

15.1 Introduction

The SMBus stack takes advantage of the SMBus extensions present on the I2C module. All standard SMBus protocols are supported in the SMBus stack, including Packet Error Checking (PEC) and Address Resolution Protocol (ARP). PEC can be enabled or disabled on a per transfer basis by using the SMBusPECEnable() and SMBusPECDisable() functions.

The stack uses a per instance configuration data structure to define various settings for each bus. The data structure has both public and private members, and software should take care not to modify members that it does not need to. For example, the interrupt state machine is tracked via the configuration structure and can be adversely affected by modifications made by the user application. Public members include things such as the base address of the I2C peripheral being used, transmit/receive buffer pointers, transfer sizes, etc.

User application software is responsible for doing basic configuration of each I2C peripheral being used for SMBus before attempting to do any bus transactions. For example, user code must enable the GPIO ports, configure the pins, set up the functional IO mux, and enable the clock to the I2C peripheral. Everything else, including initialization of the specific I2C peripheral and interrupts, is handled via SMBus stack calls such as SMBusMasterInit(), SMBusSlaveInit(), SMBusMasterIntEnable() and SMBusSlaveIntEnable(). When using ARP, software can optionally define a Unique Device Identification (UDID) structure to be used by the slave during the configuration phase.

As mentioned above, the SMBus stack is based on an interrupt-driven state machine. When performing master operations, an application can choose to either poll the status of a transaction using SMBusStatusGet() or look at the return code from any of the SMBusMaster functions that initiate new transactions. If the SMBus instance is busy, it will return without impacting the ongoing transfer. Slave operations can also use SMBusStatuGet() to query the status of an ongoing transfer. This implementation is RTOS-friendly.

On the master side, things are very straightforward, with user code needing only a call to SMBusMasterIntProcess() in the simplest case. Return codes can be tracked for events such as slave NACK or other error conditions if desired. Once the stack is configured at initialization time, the user code makes calls to the various SMBusMaster functions to initiate transfers using specific SMBus protocols.

The SMBus slave requires much more interaction from the user application. Since the slave is “dumb”, meaning that it doesn’t know which protocol to use until software tells it, the slave interrupt service routine requires much more code than the master case. The typical flow would be a call to SMBusSlaveIntProcess() followed by code that analyses the return code and the first data byte received. The typical SMBus flow is to have the master send a command byte first. Once the ISR analyzes the first data byte, it must set stack-specific flags for things such as process call or block transfers so that the state machine functions correctly.

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

15.2 API Functions

Data Structures

- tSMBus
- tSMBusUDID

Functions

- void SMBusARPDisable (tSMBus *psSMBus)
- void SMBusARPEnable (tSMBus *psSMBus)
- void SMBusARPUDIDPacketDecode (tSMBusUDID *pUDID, uint8_t *pui8Address, uint8_t *pui8Data)
- void SMBusARPUDIDPacketEncode (tSMBusUDID *pUDID, uint8_t ui8Address, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterARPAssignAddress (tSMBus *psSMBus, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterARPNotifyMaster (tSMBus *psSMBus, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterARPPrepareToARP (tSMBus *psSMBus)
- tSMBusStatus SMBusMasterBlockProcessCall (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8TxData, uint8_t ui8TxSize, uint8_t *pui8RxData)
- tSMBusStatus SMBusMasterBlockRead (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterBlockWrite (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8Data, uint8_t ui8Size)
- tSMBusStatus SMBusMasterByteReceive (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterByteSend (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Data)
- tSMBusStatus SMBusMasterByteWordRead (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8Data, uint8_t ui8Size)
- tSMBusStatus SMBusMasterByteWordWrite (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8Data, uint8_t ui8Size)
- tSMBusStatus SMBusMasterHostNotify (tSMBus *psSMBus, uint8_t ui8OwnSlaveAddress, uint8_t *pui8Data)
- tSMBusStatus SMBusMasterI2CRead (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t *pui8Data, uint8_t ui8Size)
- tSMBusStatus SMBusMasterI2CWrite (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t *pui8Data, uint8_t ui8Size)
- tSMBusStatus SMBusMasterI2CWriteRead (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t *pui8TxData, uint8_t ui8TxSize, uint8_t *pui8RxData, uint8_t ui8RxSize)
- void SMBusMasterInit (tSMBus *psSMBus, uint32_t ui32I2CBase, uint32_t ui32SMBusClock)
- void SMBusMasterIntEnable (tSMBus *psSMBus)
- tSMBusStatus SMBusMasterIntProcess (tSMBus *psSMBus)
- tSMBusStatus SMBusMasterProcessCall (tSMBus *psSMBus, uint8_t ui8TargetAddress, uint8_t ui8Command, uint8_t *pui8TxData, uint8_t *pui8RxData)

- tSMBusStatus SMBusMasterQuickCommand (tSMBus *psSMBus, uint8_t ui8TargetAddress, bool bData)
- void SMBusPECDisable (tSMBus *psSMBus)
- void SMBusPECEnable (tSMBus *psSMBus)
- uint8_t SMBusRxPacketSizeGet (tSMBus *psSMBus)
- void SMBusSlaveACKSend (tSMBus *psSMBus, bool bACK)
- void SMBusSlaveAddressSet (tSMBus *psSMBus, uint8_t ui8AddressNum, uint8_t ui8SlaveAddress)
- bool SMBusSlaveARPFlagARGet (tSMBus *psSMBus)
- void SMBusSlaveARPFlagARSet (tSMBus *psSMBus, bool bValue)
- bool SMBusSlaveARPFlagAVGet (tSMBus *psSMBus)
- void SMBusSlaveARPFlagAVSet (tSMBus *psSMBus, bool bValue)
- void SMBusSlaveBlockTransferDisable (tSMBus *psSMBus)
- void SMBusSlaveBlockTransferEnable (tSMBus *psSMBus)
- uint8_t SMBusSlaveCommandGet (tSMBus *psSMBus)
- tSMBusStatus SMBusSlaveDataSend (tSMBus *psSMBus)
- void SMBusSlaveI2CDisable (tSMBus *psSMBus)
- void SMBusSlaveI2CEnable (tSMBus *psSMBus)
- void SMBusSlaveInit (tSMBus *psSMBus, uint32_t ui32I2CBase)
- tSMBusStatus SMBusSlaveIntAddressGet (tSMBus *psSMBus)
- void SMBusSlaveIntEnable (tSMBus *psSMBus)
- tSMBusStatus SMBusSlaveIntProcess (tSMBus *psSMBus)
- void SMBusSlaveManualACKDisable (tSMBus *psSMBus)
- void SMBusSlaveManualACKEnable (tSMBus *psSMBus)
- bool SMBusSlaveManualACKStatusGet (tSMBus *psSMBus)
- void SMBusSlaveProcessCallDisable (tSMBus *psSMBus)
- void SMBusSlaveProcessCallEnable (tSMBus *psSMBus)
- void SMBusSlaveRxBufferSet (tSMBus *psSMBus, uint8_t *pui8Data, uint8_t ui8Size)
- void SMBusSlaveTransferInit (tSMBus *psSMBus)
- void SMBusSlaveTxBufferSet (tSMBus *psSMBus, uint8_t *pui8Data, uint8_t ui8Size)
- void SMBusSlaveUDIDSet (tSMBus *psSMBus, tSMBusUDID *pUDID)
- tSMBusStatus SMBusStatusGet (tSMBus *psSMBus)

15.2.1 Data Structure Documentation

15.2.1.1 tSMBus

Definition:

```
typedef struct
{
    tSMBusUDID *pUDID;
    uint32_t ui32I2CBase;
    uint8_t *pui8TxBuffer;
    uint8_t *pui8RxBuffer;
    uint8_t ui8TxSize;
    uint8_t ui8TxIndex;
```

```
    uint8_t ui8RxSize;
    uint8_t ui8RxIndex;
    uint8_t ui8OwnSlaveAddress;
    uint8_t ui8TargetSlaveAddress;
    uint8_t ui8CurrentCommand;
    uint8_t ui8CalculatedCRC;
    uint8_t ui8ReceivedCRC;
    uint8_t ui8MasterState;
    uint8_t ui8SlaveState;
    uint16_t ui16Flags;
}
tSMBus
```

Members:

pUDID The SMBus Unique Device ID (UDID) for this SMBus instance. If operating as a host, master-only, or on a bus that does not use Address Resolution Protocol (ARP), this is not required. This member can be set via a direct structure access or using the SMBusSlaveInit function. For detailed information about the UDID, refer to the SMBus spec.

ui32I2CBase The base address of the I2C master peripheral. This member can be set via a direct structure access or using the SMBusMasterInit or SMBusSlaveInit functions.

pui8TxBuffer The address of the data buffer used for transmit operations. For master operations, this member is set by the SMBusMasterxxxx functions that pass a buffer pointer (for example, SMBusMasterBlockWrite). For slave operations, this member can be set via direct structure access or using the SMBusSlaveTxBufferSet function.

pui8RxBuffer The address of the data buffer used for receive operations. For master operations, this member is set by the SMBusMasterxxxx functions that pass a buffer pointer (for example, SMBusMasterBlockRead). For slave operations, this member can be set via direct structure access or using the SMBusSlaveRxBufferSet function.

ui8TxSize The amount of data to transmit from pui8TxBuffer. For master operations this member is set by the SMBusMasterxxxx functions either via an input argument (example SMBusMasterByteWordWrite) or explicitly (example SMBusMasterSendByte). In master mode, this member should not be accessed or modified by the application. For slave operations, this member can be set via direct structure access of using the SMBusSlaveTxBufferSet function.

ui8TxIndex The current index in the transmit buffer. This member should not be accessed or modified by the application.

ui8RxSize The amount of data to receive into pui8RxBuffer. For master operations, this member is set by the SMBusMasterxxxx functions either via an input argument (example SMBusMasterByteWordRead), explicitly (example SMBusMasterReceiveByte), or by the slave (example SMBusMasterBlockRead). In master mode, this member should not be accessed or modified by the application. For slave operations, this member can be set via direct structure access of using the SMBusSlaveRxBufferSet function.

ui8RxIndex The current index in the receive buffer. This member should not be accessed or modified by the application.

ui8OwnSlaveAddress The active slave address of the I2C peripheral on the device. When using dual address in slave mode, the active address is store here. In master mode, this member is not used. This member is updated as requests come in from the master.

ui8TargetSlaveAddress The address of the targeted slave device. In master mode, this member is set by the ui8TargetSlaveAddress argument in the SMBusMasterxxxx transfer functions. In slave mode, it is not used. This member should not be modified by the application.

ui8CurrentCommand The last used command. In master mode, this member is set by the ui8Command argument in the SMBusMasterxxxx transfer functions. In slave mode, the

first received byte will always be considered the command. This member should not be modified by the application.

ui8CalculatedCRC The running CRC calculation used for transfers that require Packet Error Checking (PEC). This member is updated by the SMBus software and should not be modified by the application.

ui8ReceivedCRC The received CRC calculation used for transfers that require Packet Error Checking (PEC). This member is updated by the SMBus software and should not be modified by the application.

ui8MasterState The current state of the SMBusMasterISRProcess state machine. This member should not be accessed or modified by the application.

ui8SlaveState The current state of the SMBusSlaveISRProcess state machine. This member should not be accessed or modified by the application.

ui16Flags Flags used for various items in the SMBus state machines for different transaction types and status.
FLAG_PEC can be modified via the SMBusPECEnable or SMBusPECDisable functions or via direct structure access.
FLAG_BLOCK_TRANSFER can be set via the SMBusSlaveBlockTransferEnable function and is cleared automatically by the SMBusSlaveTransferInit function or manually using the SMBusSlaveBlockTransferDisable function.
FLAG_RAW_I2C can be modified via the SMBusSlaveI2CEnable or SMBusSlaveI2CDisable functions or via direct structure access.
FLAG_TRANSFER_IN_PROGRESS should not be modified by the application, but can be read via the SMBusStatusGet function.
FLAG_PROCESS_CALL can be set via the SMBusSlaveProcessCallEnable function and is cleared automatically by the SMBusSlaveTransferInit function or manually using the SMBusSlaveProcessCallDisable function.
FLAG_ADDRESS_RESOLVED is only used by an SMBus Slave that supports ARP. This flag can be modified via the SMBusSlaveARPFlagARSet function and read via SMBusSlaveARPFlagARGet. It can also be modified by direct structure access.
FLAG_ADDRESS_VALID is only used by an SMBus Slave that supports ARP. This flag can be modified via the SMBusSlaveARPFlagAVSet function and read via SMBusSlaveARPFlagAVGet. It can also be modified by direct structure access.
FLAG_ARP is used to indicate that ARP is currently active. This flag is not used by the SMBus stack and can (optionally) be used by the application to keep track of the ARP session.

Description:

This structure contains the state of a single instance of an SMBus module. Master and slave instances require unique configuration structures.

15.2.1.2 tSMBusUDID

Definition:

```
typedef struct
{
    uint8_t ui8DeviceCapabilities;
    uint8_t ui8Version;
    uint16_t ui16VendorID;
    uint16_t ui16DeviceID;
    uint16_t ui16Interface;
    uint16_t ui16SubSystemVendorID;
```

```
    uint16_t ui16SubSystemDeviceID;
    uint32_t ui32VendorSpecificID;
}
tSMBusUDID
```

Members:

ui8DeviceCapabilities Device capabilities field. This 8-bit field reports generic SMBus capabilities such as address type for ARP.

ui8Version Version Revision field. This 8-bit field reports UDID revision information as well as some vendor-specific things such as silicon revision.

ui16VendorID Vendor ID. This 16-bit field contains the manufacturer's ID as assigned by the SBS Implementers' Forum of the PCI SIG.

ui16DeviceID Device ID. This 16-bit field contains the device ID assigned by the device manufacturer.

ui16Interface Interface. This 16-bit field identifies the protocol layer interfaces supported over the SMBus connection.

ui16SubSystemVendorID Subsystem Vendor ID. This 16-bit field holds additional information that may be derived from the vendor ID or other information.

ui16SubSystemDeviceID Subsystem Device ID. This 16-bit field holds additional information that may be derived from the device ID or other information.

ui32VendorSpecificID Vendor-specific ID. This 32-bit field contains a unique number that can be assigned per device by the manufacturer.

Description:

This structure holds the SMBus Unique Device ID (UDID). For detailed information, please refer to the SMBus Specification.

15.2.2 Function Documentation

15.2.2.1 SMBusARPDisable

Clears the ARP flag in the configuration structure.

Prototype:

```
void
SMBusARPDisable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function clears the Address Resolution Protocol (ARP) flag in the configuration structure. This flag can be used to track the state of a device during the ARP process.

Returns:

None.

15.2.2.2 SMBusARPEnable

Sets the ARP flag in the configuration structure.

Prototype:

```
void
SMBusARPEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function sets the Address Resolution Protocol (ARP) flag in the configuration structure. This flag can be used to track the state of a device during the ARP process.

Returns:

None.

15.2.2.3 SMBusARPUDIDPacketDecode

Decodes an SMBus packet into a UDID structure and address.

Prototype:

```
void
SMBusARPUDIDPacketDecode(tSMBusUDID *pUDID,
                         uint8_t *pui8Address,
                         uint8_t *pui8Data)
```

Parameters:

pUDID specifies the structure that is updated with new data.

pui8Address specifies the location of the variable that holds the the address sent with the UDID (byte 17).

pui8Data specifies the location of the source data.

Description:

This function takes a data buffer and decodes it into a tSMBusUDID structure and an address variable. It is assumed that there are 17 bytes in the data buffer.

Returns:

None.

15.2.2.4 SMBusARPUDIDPacketEncode

Encodes a UDID structure and address into SMBus-transferable byte order.

Prototype:

```
void
SMBusARPUDIDPacketEncode(tSMBusUDID *pUDID,
                         uint8_t ui8Address,
                         uint8_t *pui8Data)
```

Parameters:

pUDID specifies the structure to encode.

ui8Address specifies the address to send with the UDID (byte 17).

pui8Data specifies the location of the destination data buffer.

Description:
This function takes a tSMBusUDID structure and re-orders the bytes so that it can be transferred on the bus. The destination data buffer must contain at least 17 bytes.

Returns:
None.

15.2.2.5 SMBusMasterARPAssignAddress

Sends an ARP Assign Address packet.

Prototype:
```
tSMBusStatus
SMBusMasterARPAssignAddress(tSMBus *psSMBus,
                            uint8_t *pui8Data)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
pui8Data is a pointer to the transmit data buffer. This buffer should be correctly formatted using SMBusARPUDIDPacketEncode() and should contain the UDID data and the address for the slave.

Description:
This function sends an Assign Address packet, used during Address Resolution Protocol (ARP). Because SMBus requires data bytes be sent out MSB first, the UDID and target address should be formatted correctly by the application or using SMBusARPUDIDPacketEncode() and placed into a data buffer pointed to by pui8Data.

Returns:
Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.6 SMBusMasterARPNotifyMaster

Sends a Notify ARP Master packet.

Prototype:
```
tSMBusStatus
SMBusMasterARPNotifyMaster(tSMBus *psSMBus,
                           uint8_t *pui8Data)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
pui8Data is a pointer to the transmit data buffer. The data payload should be 0x0000 for this packet.

Description:
This function sends a Notify ARP Master packet, used during Address Resolution Protocol (ARP). This packet is used by a slave to indicate to the ARP Master that it needs attention.

Returns:
Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.7 SMBusMasterARPPrepareToARP

Sends a Prepare to ARP packet.

Prototype:
```
tSMBusStatus
SMBusMasterARPPrepareToARP(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
This function sends a Prepare to ARP packet, used during Address Resolution Protocol (ARP). This packet is used by an ARP Master to alert devices on the bus that ARP is about to begin. All ARP-capable devices must acknowledge all bytes in this packet and clear their Address Resolved (AR) flag.

Returns:
Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.8 SMBusMasterBlockProcessCall

Initiates a master Block Process Call transfer to an SMBus slave.

Prototype:
```
tSMBusStatus
SMBusMasterBlockProcessCall(tSMBus *psSMBus,
                            uint8_t ui8TargetAddress,
                            uint8_t ui8Command,
                            uint8_t *pui8TxData,
                            uint8_t ui8TxSize,
                            uint8_t *pui8RxData)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
ui8TargetAddress specifies the slave address of the target device.
ui8Command is the command byte sent before the data is requested.
pui8TxData is a pointer to the transmit data buffer.
ui8TxSize is the number of bytes to send to the slave.

pui8RxData is a pointer to the receive data buffer.

Description:

This function supports the Block Write/Block Read Process Call protocol. The amount of data sent to the slave is user defined but limited to 32 data bytes. The amount of data read is defined by the slave device, but should never exceed 32 bytes per the SMBus spec. The receive size is the first data byte returned by the slave, so the actual size is populated in SMBusMasterISRProcess(). In the application interrupt handler, SMBusRxPacketSizeGet() can be used to obtain the amount of data sent by the slave.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, **SMBUS_DATA_SIZE_ERROR** if ui8TxSize is greater than 32, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.9 SMBusMasterBlockRead

Initiates a master Block Read transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterBlockRead(tSMBus *psSMBus,
                     uint8_t ui8TargetAddress,
                     uint8_t ui8Command,
                     uint8_t *pui8Data)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

ui8Command is the command byte sent before the data is requested.

pui8Data is a pointer to the receive data buffer.

Description:

This function supports the Block Read protocol. The amount of data read is defined by the slave device, but should never exceed 32 bytes per the SMBus spec. The receive size is the first data byte returned by the slave, so this function assumes a size of 3 until the actual number is sent by the slave. In the application interrupt handler, SMBusRxPacketSizeGet() can be used to obtain the amount of data sent by the slave.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.10 SMBusMasterBlockWrite

Initiates a master Block Write transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterBlockWrite(tSMBus *psSMBus,
                      uint8_t ui8TargetAddress,
                      uint8_t ui8Command,
                      uint8_t *pui8Data,
                      uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

ui8Command is the command byte sent before the data is requested.

pui8Data is a pointer to the transmit data buffer.

ui8Size is the number of bytes to send to the slave.

Description:

This function supports the Block Write protocol. The amount of data sent to the slave is user defined, but limited to 32 bytes per the SMBus spec.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, **SMBUS_DATA_SIZE_ERROR** if ui8Size is greater than 32, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.11 SMBusMasterByteReceive

Initiates a master Receive Byte transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterByteReceive(tSMBus *psSMBus,
                       uint8_t ui8TargetAddress,
                       uint8_t *pui8Data)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

pui8Data is a pointer to the location to store the received data byte.

Description:

The Receive Byte protocol is a basic SMBus protocol that receives a single data byte from the slave. Unlike most of the other SMBus protocols, Receive Byte does not send a “command” byte before the data payload and is intended for basic communication.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.12 SMBusMasterByteSend

Initiates a master Send Byte transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterByteSend(tSMBus *psSMBus,
                    uint8_t ui8TargetAddress,
                    uint8_t ui8Data)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

ui8Data is the data byte to send to the slave.

Description:

The Send Byte protocol is a basic SMBus protocol that sends a single data byte to the slave. Unlike most of the other SMBus protocols, Send Byte does not send a “command” byte before the data payload and is intended for basic communication.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.13 SMBusMasterByteWordRead

Initiates a master Read Byte or Read Word transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterByteWordRead(tSMBus *psSMBus,
                        uint8_t ui8TargetAddress,
                        uint8_t ui8Command,
                        uint8_t *pui8Data,
                        uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

ui8Command is the command byte sent before the data is requested.

pui8Data is a pointer to the receive data buffer.

ui8Size is the number of bytes to receive from the slave.

Description:

This function supports both the Read Byte and Read Word protocols. The amount of data to receive is user defined, but limited to 1 or 2 bytes.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, **SMBUS_DATA_SIZE_ERROR** if ui8Size is greater than 2, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.14 SMBusMasterByteWordWrite

Initiates a master Write Byte or Write Word transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterByteWordWrite(tSMBus *psSMBus,
                         uint8_t ui8TargetAddress,
                         uint8_t ui8Command,
                         uint8_t *pui8Data,
                         uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

ui8Command is the command byte sent before the data payload.

pui8Data is a pointer to the transmit data buffer.

ui8Size is the number of bytes to send to the slave.

Description:

This function supports both the Write Byte and Write Word protocols. The amount of data to send is user defined, but limited to 1 or 2 bytes.

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, **SMBUS_DATA_SIZE_ERROR** if ui8Size is greater than 2, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.15 SMBusMasterHostNotify

Initiates a master Host Notify transfer to the SMBus Host.

Prototype:

```
tSMBusStatus
SMBusMasterHostNotify(tSMBus *psSMBus,
                      uint8_t ui8OwnSlaveAddress,
                      uint8_t *pui8Data)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8OwnSlaveAddress specifies the peripheral's own slave address.

pui8Data is a pointer to the two byte data payload.

Description:

The Host Notify protocol is used by SMBus slaves to alert the bus Host about an event. Most slave devices that operate in this environment only become a bus master when this packet type is used. Host Notify always sends two data bytes to the host along with the peripheral's own slave address so that the Host knows which peripheral requested the Host's attention.

This protocol does not support PEC. The PEC flag is explicitly cleared within this function, so if PEC is enabled prior to calling it, it must be re-enabled afterwards.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.16 SMBusMasterI2CRead

Initiates a "raw" I2C read transfer to a slave device.

Prototype:

```
tSMBusStatus
SMBusMasterI2CRead(tSMBus *psSMBus,
                   uint8_t ui8TargetAddress,
                   uint8_t *pui8Data,
                   uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

pui8Data is a pointer to the receive data buffer.

ui8Size is the number of bytes to send to the slave.

Description:

This function receives a user-defined number of bytes from an I2C slave without using an SMBus protocol. The data size is only limited to the size of the ui8Size variable, which is an unsigned character (8 bits, value of 255).

Because this function uses "raw" I2C, PEC is not supported.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.17 SMBusMasterI2CWrite

Initiates a “raw” I2C write transfer to a slave device.

Prototype:

```
tSMBusStatus
SMBusMasterI2CWrite(tSMBus *psSMBus,
                    uint8_t ui8TargetAddress,
                    uint8_t *pui8Data,
                    uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

pui8Data is a pointer to the transmit data buffer.

ui8Size is the number of bytes to send to the slave.

Description:

This function sends a user-defined number of bytes to an I2C slave without using an SMBus protocol. The data size is only limited to the size of the ui8Size variable, which is an unsigned character (8 bits, value of 255).

Because this function uses “raw” I2C, PEC is not supported.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.18 SMBusMasterI2CWriteRead

Initiates a “raw” I2C write-read transfer to a slave device.

Prototype:

```
tSMBusStatus
SMBusMasterI2CWriteRead(tSMBus *psSMBus,
                        uint8_t ui8TargetAddress,
                        uint8_t *pui8TxData,
                        uint8_t ui8TxSize,
                        uint8_t *pui8RxData,
                        uint8_t ui8RxSize)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

pui8TxData is a pointer to the transmit data buffer.

ui8TxSize is the number of bytes to send to the slave.

pui8RxData is a pointer to the receive data buffer.

ui8RxSize is the number of bytes to receive from the slave.

Description:

This function initiates a write-read transfer to an I2C slave without using an SMBus protocol. The user-defined number of bytes is written to the slave first, followed by the reception of the user-defined number of bytes. The transmit and receive data sizes are only limited to the size of the ui8TxSize and ui8RxSize variables, which are unsigned characters (8 bits, value of 255).

Because this function uses “raw” I2C, PEC is not supported.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.19 SMBusMasterInit

Initializes an I2C master peripheral for SMBus functionality.

Prototype:

```
void
SMBusMasterInit(tSMBus *psSMBus,
                uint32_t ui32I2CBase,
                uint32_t ui32SMBusClock)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui32I2CBase specifies the base address of the I2C master peripheral.

ui32SMBusClock specifies the system clock speed of the MCU.

Description:

This function initializes an I2C peripheral for SMBus master use. The instance-specific configuration structure is initialized to a set of known values and the I2C peripheral is configured for 100kHz use, which is required by the SMBus specification.

Returns:

None.

15.2.2.20 SMBusMasterIntEnable

Enables the appropriate master interrupts for stack processing.

Prototype:

```
void
SMBusMasterIntEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function enables the I2C interrupts used by the SMBus master. Both the peripheral-level and NVIC-level interrupts are enabled. SMBusMasterInit() must be called before this function because this function relies on the I2C base address being defined.

Returns:
None.

15.2.2.21 SMBusMasterIntProcess

Master ISR processing function for the SMBus application.

Prototype:
```
tSMBusStatus
SMBusMasterIntProcess(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
This function must be called in the application interrupt service routine (ISR) to process SMBus master interrupts.

Returns:
Returns **SMBUS_TIMEOUT** if a bus timeout is detected, **SMBUS_ARB_LOST** if I2C bus arbitration lost is detected, **SMBUS_ADDR_ACK_ERROR** if the address phase of a transfer results in a NACK, **SMBUS_DATA_ACK_ERROR** if the data phase of a transfer results in a NACK, **SMBUS_DATA_SIZE_ERROR** if a receive buffer overrun is detected or if a transmit operation tries to write more data than is allowed, **SMBUS_MASTER_ERROR** if an unknown error occurs, **SMBUS_PEC_ERROR** if the received PEC byte does not match the locally calculated value, or **SMBUS_OK** if processing finished successfully.

15.2.2.22 SMBusMasterProcessCall

Initiates a master Process Call transfer to an SMBus slave.

Prototype:
```
tSMBusStatus
SMBusMasterProcessCall(tSMBus *psSMBus,
                       uint8_t ui8TargetAddress,
                       uint8_t ui8Command,
                       uint8_t *pui8TxData,
                       uint8_t *pui8RxData)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
ui8TargetAddress specifies the slave address of the target device.
ui8Command is the command byte sent before the data is requested.
pui8TxData is a pointer to the transmit data buffer.
pui8RxData is a pointer to the receive data buffer.

Description:
This function supports the Process Call protocol. The amount of data sent to and received from the slave is fixed to 2 bytes per direction (2 sent, 2 received).

This protocol supports the optional PEC byte for error checking. To use PEC, SMBusPECEnable() must be called before this function.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.23 SMBusMasterQuickCommand

Initiates a master Quick Command transfer to an SMBus slave.

Prototype:

```
tSMBusStatus
SMBusMasterQuickCommand(tSMBus *psSMBus,
                        uint8_t ui8TargetAddress,
                        bool bData)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui8TargetAddress specifies the slave address of the target device.

bData is the value of the single data bit sent to the slave.

Description:

Quick Command is an SMBus protocol that sends a single data bit using the I2C R/S bit. This function issues a single I2C transfer with the slave address and data bit.

This protocol does not support PEC. The PEC flag is explicitly cleared within this function, so if PEC is enabled prior to calling it, it must be re-enabled afterwards.

Returns:

Returns **SMBUS_PERIPHERAL_BUSY** if the I2C peripheral is currently active, **SMBUS_BUS_BUSY** if the bus is already in use, or **SMBUS_OK** if the transfer has successfully been initiated.

15.2.2.24 SMBusPECDisable

Disables Packet Error Checking (PEC).

Prototype:

```
void
SMBusPECDisable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function disables the transmission and checking of a PEC byte in SMBus transactions.

Returns:

None.

15.2.2.25 SMBusPECEnable

Enables Packet Error Checking (PEC).

Prototype:

```
void
SMBusPECEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function enables the transmission and checking of a PEC byte in SMBus transactions.

Returns:

None.

15.2.2.26 SMBusRxPacketSizeGet

Returns the number of bytes in the receive buffer.

Prototype:

```
uint8_t
SMBusRxPacketSizeGet(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function returns the number of bytes in the active receive buffer. It can be used to determine how many bytes have been received in the slave receive or master block read configurations.

Returns:

Number of bytes in the buffer.

15.2.2.27 SMBusSlaveACKSend

Sets the value of the ACK bit when using manual acknowledgement.

Prototype:

```
void
SMBusSlaveACKSend(tSMBus *psSMBus,
                  bool bACK)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

bACK specifies whether to ACK (**true**) or NACK (**false**).

Description:

This function sets the value of the ACK bit. In order for the ACK bit to take effect, manual acknowledgement must be enabled on the slave using SMBusSlaveManualACKEnable().

Returns:
None.

15.2.2.28 SMBusSlaveAddressSet

Sets the slave address for an SMBus slave peripheral.

Prototype:
```
void
SMBusSlaveAddressSet(tSMBus *psSMBus,
                     uint8_t ui8AddressNum,
                     uint8_t ui8SlaveAddress)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
ui8AddressNum specifies which address (primary or secondary)
ui8SlaveAddress is the address of the slave.

Description:
This function sets the slave address. Both the primary and secondary addresses can be set using this function. To set the primary address (stored in I2CSOAR), ui8AddressNum should be '0'. To set the secondary address (stored in I2CSOAR2), ui8AddressNum should be '1'.

Returns:
None.

15.2.2.29 SMBusSlaveARPFlagARGet

Returns the current value of the AR (Address Resolved) flag.

Prototype:
```
bool
SMBusSlaveARPFlagARGet(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
This returns the value of the AR (Address Resolved) flag.

Returns:
Returns **true** if set, **false** if cleared.

15.2.2.30 SMBusSlaveARPFlagARSet

Sets the value of the AR (Address Resolved) flag.

Prototype:

```
void
SMBusSlaveARPFlagARSet(tSMBus *psSMBus,
                       bool bValue)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

bValue is the value to set the flag.

Description:

This function allows the application to set the value of the AR flag. All SMBus slaves must support the AR and AV flags. On POR, the AR flag is cleared. It is also cleared when a slave receives the ARP Reset Device command.

Returns:

None.

15.2.2.31 SMBusSlaveARPFlagAVGet

Returns the current value of the AV (Address Valid) flag.

Prototype:

```
bool
SMBusSlaveARPFlagAVGet(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This returns the value of the AV (Address Valid) flag.

Returns:

Returns **true** if set, or **false** if cleared.

15.2.2.32 SMBusSlaveARPFlagAVSet

Sets the value of the AV (Address Valid) flag.

Prototype:

```
void
SMBusSlaveARPFlagAVSet(tSMBus *psSMBus,
                       bool bValue)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

bValue is the value to set the flag.

Description:

This function allows the application to set the value of the AV flag. All SMBus slaves must support the AR and AV flags. On POR, the AV flag is cleared. It is also cleared when a slave receives the ARP Reset Device command.

Returns:
None.

15.2.2.33 SMBusSlaveBlockTransferDisable

Clears the block transfer flag for an SMBus slave transfer.

Prototype:
```
void
SMBusSlaveBlockTransferDisable(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
Clears the block transfer flag in the configuration structure. The user application can either call this function to clear the flag, or use SMBusSlaveTransferInit() to clear out all transfer-specific flags.

Returns:
None.

15.2.2.34 SMBusSlaveBlockTransferEnable

Sets the block transfer flag for an SMBus slave transfer.

Prototype:
```
void
SMBusSlaveBlockTransferEnable(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
Sets the block transfer flag in the configuration structure so that the SMBus slave can respond correctly to a Block Write or Block Read request. This flag must be set prior to the data portion of the packet.

Returns:
None.

15.2.2.35 SMBusSlaveCommandGet

Get the current command byte.

Prototype:
```
uint8_t
SMBusSlaveCommandGet(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

Returns the current value of the ui8CurrentCommand variable in the SMBus configuration structure. This can be used to help the user application set up the SMBus slave transmit and receive buffers.

Returns:

None.

15.2.2.36 SMBusSlaveDataSend

Sends data outside of the interrupt processing function.

Prototype:

```
tSMBusStatus
SMBusSlaveDataSend(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function sends data outside the interrupt processing function, and should only be used when SMBusSlaveIntProcess() returns **SMBUS_SLAVE_NOT_READY**. At this point, the application should set up the transfer and call this function (it assumes that the transmit buffer has already been populated when called). When called, this function updates the slave state machine as if SMBusSlaveIntProcess() were called.

Returns:

Returns **SMBUS_SLAVE_NOT_READY** if the slave's transmit buffer is not yet initialized (ui8TxSize is 0), or **SMBUS_OK** if processing finished successfully.

15.2.2.37 SMBusSlaveI2CDisable

Clears the "raw" I2C flag for an SMBus slave transfer.

Prototype:

```
void
SMBusSlaveI2CDisable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

Clears the raw I2C flag in the configuration structure. This flag is a global setting similar to the PEC flag and cannot be cleared using SMBusSlaveTransferInit().

Returns:

None.

15.2.2.38 SMBusSlaveI2CEnable

Sets the “raw” I2C flag for an SMBus slave transfer.

Prototype:

```
void
SMBusSlaveI2CEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

Sets the raw I2C flag in the configuration structure so that the SMBus slave can respond correctly to raw I2C (non-SMBus protocol) requests. This flag must be set prior to the transfer, and is a global setting.

Returns:

None.

15.2.2.39 SMBusSlaveInit

Initializes an I2C slave peripheral for SMBus functionality.

Prototype:

```
void
SMBusSlaveInit(tSMBus *psSMBus,
               uint32_t ui32I2CBase)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

ui32I2CBase specifies the base address of the I2C slave peripheral.

Description:

This function initializes an I2C peripheral for SMBus slave use. The instance-specific configuration structure is initialized to a set of known values and the I2C peripheral is configured based on the input arguments.

The default configuration of the SMBus slave uses automatic acknowledgement. If manual acknowledgement is required, call SMBusSlaveManualACKEnable().

Returns:

None.

15.2.2.40 SMBusSlaveIntAddressGet

Determine whether primary or secondary slave address has been requested by the master.

Prototype:

```
tSMBusStatus
SMBusSlaveIntAddressGet(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
Tells the caller whether the I2C slave address requested by the master or SMBus Host is the primary or secondary I2C slave address of the peripheral. The primary is defined as the address programmed into I2CSOAR, and the secondary as the address programmed into I2CSOAR2.

Returns:
Returns **SMBUS_SLAVE_ADDR_PRIMARY** if the primary address is called out or **SMBUS_SLAVE_ADDR_SECONDARY** if the secondary address is called out.

15.2.2.41 SMBusSlaveIntEnable

Enables the appropriate slave interrupts for stack processing.

Prototype:
```
void
SMBusSlaveIntEnable(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
This function enables the I2C interrupts used by the SMBus slave. Both the peripheral-level and NVIC-level interrupts are enabled. SMBusSlaveInit() must be called before this function because this function relies on the I2C base address being defined.

Returns:
None.

15.2.2.42 SMBusSlaveIntProcess

Slave ISR processing function for the SMBus application.

Prototype:
```
tSMBusStatus
SMBusSlaveIntProcess(tSMBus *psSMBus)
```

Parameters:
psSMBus specifies the SMBus configuration structure.

Description:
This function must be called in the application interrupt service routine (ISR) to process SMBus slave interrupts.

If manual acknowledge is enabled using SMBusSlaveManualACKEnable(), this function processes the data byte, but does not send the ACK/NACK value. In this case, the user application is responsible for sending the acknowledge bit based on the return code of this function.

When receiving a Quick Command from the master, the slave has some set-up requirements. When the master sends the R/S (data) bit as '0', nothing additional needs to be done in the

slave and SMBusSlaveIntProcess() returns **SMBUS_SLAVE_QCMD_0**. However, when the master sends the R/S (data) bit as '1', the slave must write the data register with data containing a '1' in bit 7. This means that when receiving a Quick Command, the slave must set up the TX buffer to either have 1 data byte with bit 7 set to '1' or set up the TX buffer to be zero length. In the case where 1 data byte is put in the TX buffer, SMBusSlaveIntProcess() returns **SMBUS_OK** the first time its called and **SMBUS_SLAVE_QCMD_0** the second. In the case where the TX buffer has no data, SMBusSlaveIntProcess() will return **SMBUS_SLAVE_ERROR** the first time its called, and **SMBUS_SLAVE_QCMD_1** the second time.

Returns:

Returns **SMBUS_SLAVE_FIRST_BYTE** if the first byte (typically the SMBus command) has been received; **SMBUS_SLAVE_NOT_READY** if the slave's transmit buffer is not yet initialized when the master requests data from the slave; **SMBUS_DATA_SIZE_ERROR** if during a master block write, the size sent by the master is greater than the amount of available space in the receive buffer; **SMBUS_SLAVE_ERROR** if a buffer overrun is detected during a slave receive operation or if data is sent and was not expected; **SMBUS_SLAVE_QCMD_0** if a Quick Command was received with data '0'; **SMBUS_SLAVE_QCMD_1** if a Quick Command was received with data '1'; **SMBUS_TRANSFER_COMPLETE** if a STOP is detected on the bus, marking the end of a transfer; **SMBUS_PEC_ERROR** if the received PEC byte does not match the locally calculated value; or **SMBUS_OK** if processing finished successfully.

15.2.2.43 SMBusSlaveManualACKDisable

Disables manual acknowledgement for the SMBus slave.

Prototype:

```
void
SMBusSlaveManualACKDisable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function disables manual acknowledge capability in the slave. When manual acknowledgement is disabled, the slave automatically ACKs every byte sent by the master.

Returns:

None.

15.2.2.44 SMBusSlaveManualACKEnable

Enables manual acknowledgement for the SMBus slave.

Prototype:

```
void
SMBusSlaveManualACKEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function enables manual acknowledge capability in the slave. If the application requires that the slave NACK on a bad command or a bad PEC calculation, manual acknowledgement allows this to happen.

In the case of responding to a bad command with a NACK, the application should use SMBusSlaveACKSend() to ACK/NACK the command. The slave ISR should check for the SMBUS_SLAVE_FIRST_BYTE return code from SMBusSlaveISRProcess() and ACK/NACK accordingly. All other cases should be handled in the application based on the return code of SMBusSlaveISRProcess().

Returns:

None.

15.2.2.45 SMBusSlaveManualACKStatusGet

Returns the manual acknowledgement status of the SMBus slave.

Prototype:

```
bool
SMBusSlaveManualACKStatusGet(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function returns the state of the I2C ACKOEN bit in the I2CSACKCTL register. This feature is disabled out of reset and must be enabled using SMBusSlaveManualACKEnable().

Returns:

Returns **true** if manual acknowledge is enabled, or **false** if manual acknowledge is disabled.

15.2.2.46 SMBusSlaveProcessCallDisable

Clears the process call flag for an SMBus slave transfer.

Prototype:

```
void
SMBusSlaveProcessCallDisable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

Clears the process call flag in the configuration structure. The user application can either call this function to clear the flag, or use SMBusSlaveTransferInit() to clear out all transfer-specific flags.

Returns:

None.

15.2.2.47 SMBusSlaveProcessCallEnable

Sets the process call flag for an SMBus slave transfer.

Prototype:

```
void
SMBusSlaveProcessCallEnable(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

Sets the process call flag in the configuration structure so that the SMBus slave can respond correctly to a Process Call request. This flag must be set prior to the data portion of the packet.

Returns:

None.

15.2.2.48 SMBusSlaveRxBufferSet

Set the address and size of the slave receive buffer.

Prototype:

```
void
SMBusSlaveRxBufferSet(tSMBus *psSMBus,
                      uint8_t *pui8Data,
                      uint8_t ui8Size)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

pui8Data is a pointer to the receive data buffer.

ui8Size is the number of bytes in the buffer.

Description:

This function sets the address and size of the slave receive buffer.

Returns:

None.

15.2.2.49 SMBusSlaveTransferInit

Sets up the SMBus slave for a new transfer.

Prototype:

```
void
SMBusSlaveTransferInit(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:
This function is used to re-initialize the configuration structure for a new transfer. Once a transfer is complete and the data has been processed, unused flags, states, the data buffers and buffer indexes should be reset to a known state before a new transfer.

Returns:
None.

15.2.2.50 SMBusSlaveTxBufferSet

Set the address and size of the slave transmit buffer.

Prototype:
```
void
SMBusSlaveTxBufferSet(tSMBus *psSMBus,
                      uint8_t *pui8Data,
                      uint8_t ui8Size)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
pui8Data is a pointer to the transmit data buffer.
ui8Size is the number of bytes in the buffer.

Description:
This function sets the address and size of the slave transmit buffer.

Returns:
None.

15.2.2.51 SMBusSlaveUDIDSet

Sets a slave's UDID structure.

Prototype:
```
void
SMBusSlaveUDIDSet(tSMBus *psSMBus,
                  tSMBusUDID *pUDID)
```

Parameters:
psSMBus specifies the SMBus configuration structure.
pUDID is a pointer to the UDID configuration for the slave. This is only needed if the slave is on a bus that uses ARP.

Description:
This function sets the UDID for a slave instance.

Returns:
None.

15.2.2.52 SMBusStatusGet

Returns the state of an SMBus transfer.

Prototype:

```
tSMBusStatus
SMBusStatusGet(tSMBus *psSMBus)
```

Parameters:

psSMBus specifies the SMBus configuration structure.

Description:

This function returns the status of an SMBus transaction. It can be used to determine whether a transfer is ongoing or complete.

Returns:

Returns **SMBUS_TRANSFER_IN_PROGRESS** if transfer is ongoing, or **SMBUS_TRANSFER_COMPLETE** if transfer has completed.

15.3 Programming Example

The following example shows how to initialize both a master and slave SMBus instance. In this example, it is assumed the I2C0 is the master and resides on pin PB2 and PB3. I2C1 is the slave and resides on pins PA6 and PA7.

```
tSMBus g_sMaster;
tSMBus g_sSlave;
unsigned char g_pucSlaveTxBuffer[MAX_SMB_BLOCK_SIZE] = {0};
unsigned char g_pucSlaveRxBuffer[MAX_SMB_BLOCK_SIZE] = {0};

void
MyInit(void)
{
    //
    // Enable the peripherals for the SMBus master.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

    //
    // Configure the required pins for I2C0.
    //
    GPIOPinTypeI2CSCL(GPIO_PORTB_BASE, GPIO_PIN_2);
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_3);

    //
    // Configure the IO mux so that the I2C pins for I2C0 are on PB2/3.
    //
    GPIOPinConfigure(GPIO_PB2_I2C0SCL);
    GPIOPinConfigure(GPIO_PB3_I2C0SDA);

    //
    // Initialize the master SMBus port.
    //
    SMBusMasterInit(&g_sMaster, I2C0_MASTER_BASE, SysCtlClockGet());

    //
```

```
    // Enable master interrupts.
    //
    SMBusMasterIntEnable(&g_sMaster);

    //
    // Enable the peripherals for the SMBus slave.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C1);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);

    //
    // Configure the required pins for I2C1.
    //
    GPIOPinTypeI2CSCL(GPIO_PORTA_BASE, GPIO_PIN_6);
    GPIOPinTypeI2C(GPIO_PORTA_BASE, GPIO_PIN_7);

    //
    // Configure the IO mux so that the I2C pins for I2C1 are on PA6/7.
    //
    GPIOPinConfigure(GPIO_PA6_I2C1SCL);
    GPIOPinConfigure(GPIO_PA7_I2C1SDA);

    //
    // Initialize the Slave SMBus port.
    //
    SMBusSlaveInit(&g_sSlave, I2C1_SLAVE_BASE);

    //
    // Enable slave interrupts.
    //
    SMBusSlaveIntEnable(&g_sSlave);

    //
    // Set the slave addresses.
    //
    SMBusSlaveAddressSet(&g_sSlave, 0, SMBUS_ADR_SMART_BATTERY);

    //
    // Set up the slave idle transfer.  It is good practice to populate the
    // slave buffers before attaching to the bus so that they're in a known
    // state.
    //
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, MAX_SMB_BLOCK_SIZE);
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, MAX_SMB_BLOCK_SIZE);
}
```

The following example shows how an application might implement a SMBus master ISR.

```
void
SMBusMasterIntHandler(void)
{
    tSMBusStatus eStatus;

    //
    // Process the interrupt.
    //
    eStatus = SMBusMasterIntProcess(&g_sMaster);

    //
    // Check for errors.
    //
    switch(eStatus)
    {
        case SMBUS_PEC_ERROR:
        {
```

```
            //
            // Handle error.
            //

            break;
        }

        case SMBUS_TIMEOUT:
        {
            //
            // Handle error.
            //

            break;
        }

        case SMBUS_ADDR_ACK_ERROR:
        case SMBUS_DATA_ACK_ERROR:
        {
            //
            // Handle error.
            //

            break;
        }
    }
}
```

The following example shows how an application might implement a SMBus slave ISR. Recall that the slave requires much more user application code than the master case. This example shows how to handle various protocols using arbitrary command bytes. Software can either check for the **SMBUS_TRANSFER_COMPLETE** return code or use SMBusStatusGet() to determine when a transfer is complete.

```
void
SMBusSlaveIntProcess(void)
{
    tSMBusStatus eStatus;

    //
    // Process the interrupt.
    //
    eStatus = SMBusSlaveIntProcess(&g_sSlave);

    //
    // See if the first byte/command was received.
    //
    if(eStatus == SMBUS_SLAVE_FIRST_BYTE)
    {
        //
        // Figure out which command was sent and set up the transfer
        // accordingly.
        //
        switch(SMBusSlaveCommandGet(&g_sSlave))
        {
            //
            // Write byte protocol.
            //
            case 0xf1:
            {
                SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 1);
                break;
            }
```

```
//
// Write word protocol.
//
case 0xf2:
{
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 2);
    break;
}

//
// Read byte protocol.  TX buffer can be populated in ISR context
// or in another function.
//
case 0xf3:
{
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 1);
    break;
}

//
// Read word protocol.  TX buffer can be populated in ISR context
// or in another function.
//
case 0xf4:
{
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 2);
    break;
}

//
// Process call protocol.  TX buffer can be populated in ISR
// context or in another function.
//
case 0xf5:
{
    SMBusSlaveProcessCallEnable(&g_sSlave);
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, 2);
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer, 2);
    break;
}

//
// Block write protocol.  Set the size to the max to accomodate up
// to the maximum transfer size.  When the bus sees the STOP signal
// it marks the end of transfer and the actual size of the RX
// operation is obtained from the configuration structure by using
// SMBusRxPacketSizeGet().
//
case 0xf6:
{
    SMBusSlaveBlockTransferEnable(&g_sSlave);
    SMBusSlaveRxBufferSet(&g_sSlave, g_pucSlaveRxBuffer,
                          MAX_SMB_BLOCK_SIZE);
    break;
}

//
// Block read protocol.  The size for the slave TX operation is
// application-specific, so a global variable is used for example
// purpose.  TX buffer can be populated in ISR context or in
// another function.
//
case 0xf7:
{
    SMBusSlaveBlockTransferEnable(&g_sSlave);
    SMBusSlaveTxBufferSet(&g_sSlave, g_pucSlaveTxBuffer, g_ucSize);
```

```
                        break;
                    }
                }
            }
        }
```

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
ulocaltime	localtime
umktime	mktime
urand	rand
usnprintf	snprintf
usprintf	sprintf
usrand	srand
ustrcasecmp	strcasecmp
ustrcmp	strcmp
ustrlen	strlen
ustrncmp	strncmp
ustrncpy	strncpy
ustrnicmp	strnicmp
ustrstr	strstr
ustrtof	strtof
ustrtoul	strtoul
uvsnprintf	vsnprintf

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

16.2 API Functions

Functions

- void ulocaltime (time_t timer, struct tm *tm)
- time_t umktime (struct tm *timeptr)
- int urand (void)
- int usnprintf (char *restrict s, size_t n, const char *restrict format,...)
- int usprintf (char *restrict s, const char *format,...)
- void usrand (unsigned int seed)
- int ustrcasecmp (const char *s1, const char *s2)

- int ustrcmp (const char *s1, const char *s2)
- size_t ustrlen (const char *s)
- int ustrncasecmp (const char *s1, const char *s2, size_t n)
- int ustrncmp (const char *s1, const char *s2, size_t n)
- char * ustrncpy (char *restrict s1, const char *restrict s2, size_t n)
- char * ustrstr (const char *s1, const char *s2)
- float ustrtof (const char *nptr, const char **endptr)
- unsigned long ustrtoul (const char *restrict nptr, const char **restrict endptr, int base)
- int uvsnprintf (char *restrict s, size_t n, const char *restrict format, va_list arg)

16.2.1 Function Documentation

16.2.1.1 ulocaltime

Converts from seconds to calendar date and time.

Prototype:

```
void
ulocaltime(time_t timer,
           struct tm *tm)
```

Parameters:

timer is the number of seconds.

tm 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.

16.2.1.2 umktime

Converts calendar date and time to seconds.

Prototype:

```
time_t
umktime(struct tm *timeptr)
```

Parameters:

timeptr 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 *timeptr* 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 (uint32_t)(-1).

16.2.1.3 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.

16.2.1.4 usnprintf

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

Prototype:

```
int
usnprintf(char *restrict s,
          size_t n,
          const char *restrict format,
          ...)
```

Parameters:

s is the buffer where the converted string is stored.

n is the size of the buffer.

format 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 *format* 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 *n* - 1 characters into the buffer *s*. 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.

16.2.1.5 usprintf

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

Prototype:

```
int
usprintf(char *restrict s,
         const char *format,
         ...)
```

Parameters:

s is the buffer where the converted string is stored.

format 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 *format* 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 *s* 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.

16.2.1.6 usrand

Set the random number generator seed.

Prototype:
```
void
usrand(unsigned int seed)
```

Parameters:
seed 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

16.2.1.7 ustrcasecmp

Compares two strings without regard to case.

Prototype:
```
int
ustrcasecmp(const char *s1,
            const char *s2)
```

Parameters:
s1 points to the first string to be compared.
s2 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 int16_ter string is deemed the lesser.

Returns:
Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

16.2.1.8 ustrcmp

Compares two strings.

Prototype:

```
int
ustrcmp(const char *s1,
        const char *s2)
```

Parameters:

s1 points to the first string to be compared.

s2 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 int16_ter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

16.2.1.9 ustrlen

Returns the length of a null-terminated string.

Prototype:

```
size_t
ustrlen(const char *s)
```

Parameters:

s 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 *s*.

16.2.1.10 ustrncasecmp

Compares two strings without regard to case.

Prototype:

```
int
ustrncasecmp(const char *s1,
             const char *s2,
             size_t n)
```

Parameters:

s1 points to the first string to be compared.

s2 points to the second string to be compared.

n is the maximum number of characters to compare.

Description:

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

Returns:

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

16.2.1.11 ustrncmp

Compares two strings.

Prototype:

```
int
ustrncmp(const char *s1,
         const char *s2,
         size_t n)
```

Parameters:

s1 points to the first string to be compared.

s2 points to the second string to be compared.

n is the maximum number of characters to compare.

Description:

This function is very similar to the C library `strncmp()` function. It compares at most *n* characters of two strings taking case into account. The comparison ends if a terminating NULL character is found in either string before *n* characters are compared. In this case, the int16_ter string is deemed the lesser.

Returns:

Returns 0 if the two strings are equal, -1 if *s1* is less than *s2* and 1 if *s1* is greater than *s2*.

16.2.1.12 ustrncpy

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

Prototype:

```
char *
ustrncpy(char *restrict s1,
         const char *restrict s2,
         size_t n)
```

Parameters:

s1 is a pointer to the destination buffer into which characters are to be copied.

s2 is a pointer to the string from which characters are to be copied.
n is the number of characters to copy to the destination buffer.

Description:
This function copies at most *n* characters from the string pointed to by *s2* into the buffer pointed to by *s1*. If the end of *s2* is found before *n* characters have been copied, remaining characters in *s1* will be padded with zeroes until *n* 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 *s2*.

Returns:
Returns *s1*.

16.2.1.13 ustrstr

Finds a substring within a string.

Prototype:
```
char *
ustrstr(const char *s1,
        const char *s2)
```

Parameters:
s1 is a pointer to the string that will be searched.
s2 is a pointer to the substring that is to be found within *s1*.

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 *s2* within *s1* or NULL if no match is found.

16.2.1.14 ustrtof

Converts a string into its floating-point equivalent.

Prototype:
```
float
ustrtof(const char *nptr,
        const char **endptr)
```

Parameters:
nptr is a pointer to the string containing the floating-point value.
endptr is a pointer that will be set to the first character past the floating-point value in the string.

Description:
This function is very similar to the C library `strtof()` 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 a floating-point value.

Returns:
Returns the result of the conversion.

16.2.1.15 ustrtoul

Converts a string into its numeric equivalent.

Prototype:
```
unsigned long
ustrtoul(const char *restrict nptr,
         const char **restrict endptr,
         int base)
```

Parameters:
nptr is a pointer to the string containing the integer.
endptr is a pointer that will be set to the first character past the integer in the string.
base 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.

16.2.1.16 uvsnprintf

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

Prototype:
```
int
uvsnprintf(char *restrict s,
           size_t n,
           const char *restrict format,
           va_list arg)
```

Parameters:
s points to the buffer where the converted string is stored.
n is the size of the buffer.
format is the format string.
arg 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 *format* 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 *n* parameter limits the number of characters that will be stored in the buffer pointed to by *s* 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 (bool bEnable)
- void UARTFlushRx (void)
- void UARTFlushTx (bool bDiscard)
- unsigned char UARTgetc (void)
- int UARTgets (char *pcBuf, uint32_t ui32Len)
- int UARTPeek (unsigned char ucChar)
- void UARTprintf (const char *pcString,...)
- int UARTRxBytesAvail (void)
- void UARTStdioConfig (uint32_t ui32PortNum, uint32_t ui32Baud, uint32_t ui32SrcClock)
- void UARTStdioIntHandler (void)
- int UARTTxBytesFree (void)
- void UARTvprintf (const char *pcString, va_list vaArgP)
- int UARTwrite (const char *pcBuf, uint32_t ui32Len)

17.2.1 Function Documentation

17.2.1.1 UARTEchoSet

Enables or disables echoing of received characters to the transmitter.

Prototype:

```
void
UARTEchoSet(bool 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(bool 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,
         uint32_t ui32Len)
```

Parameters:

pcBuf points to a buffer for the incoming string from the UART.

ui32Len 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(uint32_t ui32PortNum,
                uint32_t ui32Baud,
                uint32_t ui32SrcClock)
```

Parameters:
ui32PortNum is the number of UART port to use for the serial console (0-2)
ui32Baud is the bit rate that the UART is to be configured to use.
ui32SrcClock 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 *ui32Baud* 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 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.

17.2.1.11 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.

17.2.1.12 UARTvprintf

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

Prototype:

```
void
UARTvprintf(const char *pcString,
            va_list vaArgP)
```

Parameters:

pcString is the format string.

vaArgP is a variable argument list pointer whose content will depend upon the format string passed in *pcString*.

Description:

This function is very similar to the C library `vprintf()` 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 in the variable arguments list 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.

17.2.1.13 UARTwrite

Writes a string of characters to the UART output.

Prototype:

```
int
UARTwrite(const char *pcBuf,
          uint32_t ui32Len)
```

Parameters:

pcBuf points to a buffer containing the string to transmit.

ui32Len 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 *ui32Len* 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 *ui32Len* 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

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

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

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

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

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 Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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