DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI Firmware Development Package

USER'S GUIDE

SW-DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI-UG-9453

Copyright

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

Revision Information

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

Table of Contents

1 Introduction

The Texas Instruments Stellaris® DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI evaluation board is a platform that can be used for developing software and prototyping a hardware design. It contains a Stellaris ARM® Cortex ™ -M3-based microcontroller, along with a QVGA color touchscreen display, a user button, a small speaker, a potentiometer thumbwheel, a microSD card slot, a stereo audio codec, a USB OTG connector and an Ethernet connector. Additionally, many of the microcontroller's pins are connected to jumpers, allowing for easy connection to other hardware for the purposes of prototyping (after the stake headers have been populated by the customer).

The board also contains 1 MB of SSI-connected flash memory and a daughter board with 8 MB of SDRAM.

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

2 Introduction

The Texas Instruments' Stellaris® DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI development board is a platform that can be used for developing software and prototyping a hardware design. It contains a Stellaris ARM® Cortex ™ -M3-based microcontroller, along with a QVGA color touch-screen display, a user button, a small speaker, a potentiometer thumbwheel, a microSD card slot, a stereo audio codec, a USB OTG connector, and an Ethernet connector. Additionally, many of the microcontroller's pins are connected to jumpers, allowing for easy connection to other hardware for the purposes of prototyping (after the stake headers have been populated by the customer). Coupled with an EM2 daughter board and one of several supported low-power radio transceivers (LPRF) evaluation modules, the DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI also provides a powerful platform to evaluate TI's range of LPRF software and hardware solutions.

This document provides a brief overview of the SimpliciTI software stack and describes the example applications that are provided specifically for this stack. It should be read in conjunction with the main "DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI Firmware Development Package User's Guide" which documents other examples applications provided for the board and describes board-specific drivers and utility function modules.

The software provided in this firmware development package must be installed in the same root directory as the "DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI Firmware Development Package" (typically C:\StellarisWare) since LPRF examples require various source files from the main board software installation. Examples provided here will not build if installed in isolation.

3 Expansion Boards

3.1 Introduction

The DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI development kit board contains a header populated with the microcontroller's Extended Peripheral Interface (EPI) and I2C0 signals. The EPI is a flexible, high-speed parallel bus allowing connection of external peripherals or memories and several expansion boards are available which illustrate how EPI may be used in different operating modes. Expansion boards not requiring EPI may also make use of DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI's ability to mux other signals to these pins to gain access to peripherals such as UART and SSI or use them as simple GPIOs.

3.2 EPI Configuration for Expansion Boards

The EPI must be configured differently depending upon the mode in which it is to be used and the peripherals to which it is connected. There are several steps required to configure the pins used to connect your peripherals via EPI and the EPI peripheral itself:

1. Enable all GPIO ports containing pins which are used by EPI using calls to the SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOx) function.
2. Enable the EPI peripheral using a call to the SysCtlPeripheralEnable(SYSCTL_PERIPH_EPI0) function.
3. Ensure that the required EPI signals are connected to the required pins by writing the GPIO Port Control Register (GPIO_O_PCTL) for each GPIO port containing EPI signals.
4. Configure the operating mode of the EPI by calling the EPIModeSet() function and passing the required mode.
5. Configure the clock rate of the EPI module by calling the EPIDividerSet() function.
6. Configure parameters specific to the EPI operating mode by calling one of the EPIConfigGPModeSet(), EPIConfigHB8Set(), EPIConfigHB16Set() or EPIConfigSDRAMSet() functions depending upon the mode you selected via EPIModeSet().
7. Select the address mapping for the peripherals or memory attached using a call to the EPIAddressMapSet() function.

This configuration is handled in the PinoutSet() function which can be found in the file `set_pinout.c` in directory `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`. Two different implementations of this function are provided. The default implementation, used in all the example applications for the board, makes use of information from an I2C-connected EEPROM device found on some expansion boards to determine the EPI configuration at runtime. If no EEPROM device is found, the configuration is set assuming that the SDRAM expansion board is connected (since this simple expansion board does not have the identification EEPROM).

In a typical end-user application, this dynamic determination of the EPI configuration is unlikely to be used and, for this reason, a secondary implementation of the PinoutSet() function is provided which sets a static EPI configuration. By default, the example provided sets the EPI into HostBus-8 mode with timings appropriate for the Flash and SRAM expansion board. To compile this implementation of the function into your project, ensure that the `SIMPLE_PINOUT_SET` label is defined in your makefile or toolchain project and passed to the compiler. To set a static initialization for the FPGA expansion board, set `SIMPLE_PINOUT_SET` and also `EPI_CONFIG_FPGA` before compiling your application.

For further information on the functions mentioned here, see the *Stellaris Peripheral Driver Library User's Guide*.

3.3 Stellaris SDRAM Expansion Board

A expansion board containing 8 MB of SDRAM is included in the basic DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI package and is used by the `qs-checkout` and `showjpeg` example applications which make use of the memory for image decompression workspace and, in the `qs-checkout` case, to store a file system image that is used by the application's embedded web server.

This expansion board does not have an EEPROM containing EPI configuration information so PinoutSet() defaults to configuring EPI for this board if no I2C EEPROM is found. The SDRAMInit() function may also be called after PinoutSet(), if SDRAM timing parameters other than the defaults are required since this function reconfigures the EPI peripheral based on parameters defining the required clock rate and refresh interval for the SDRAM.

After configuration, the SDRAM is managed as a heap via the ExtRAMAlloc() and ExtRAM-Free() functions which, along with SDRAMInit(), can be found in the file `extram.c` under the `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers` directory.

3.4 Stellaris Flash and SRAM Memory Expansion Board

This complex expansion board, available separately from the basic DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board, connects three devices via EPI:

- 8 MB of x8 Flash,
- 1 MB of x8 SRAM and
- the QVGA touchscreen (via a Customer Programmable Logic Devic (CPLD)).

An I2C EEPROM on this board contains configuration and timing information allowing the Pinout-Set() function to configure the EPI correctly and allow all the devices to be accessed.

The expansion board connects to the jumper block near the QVGA touchscreen in addition to the EPI connector and intercepts control and data signals to the panel. The touchscreen display, which is normally accessed via GPIOs when this expansion board is absent, is driven via a CPLD which allows control and data access to the device via particular EPI addresses.

When this expansion board is detected in PinoutSet(), the `g_eDaughterType` global variable is set allowing the display driver (`kitronix320x240x16_ssd2119_8bit.c`) and touchscreen driver (`touch.c`) to dynamically reconfigure themselves to operate with the new expansion board. This variable is also used in the `extram.c` driver to configure the external SRAM as the heap which can be managed using the ExtRAMAlloc() and ExtRAMFree() calls (following a call to ExtRAMHeapInit()).

The `qs-checkout` example application detects this board and serves a web site from the external Flash if one is found. The external Flash is factory-programmed with the same "photo gallery" web site image that is found in the serial Flash on the base DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board. Both the `qs-checkout` and `showjpeg` examples make use of the SRAM on this expansion board for image decompression workspace.

3.5 Execution from External Memories

External, EPI-connected Flash memory is intended primarily for data storage but it is possible to execute code from Flash, SRAM, or SDRAM attached to the external peripheral interface. Note that execution from external Flash or RAM results in lower performance than executing from internal Flash. A program running from off-chip memory will typically run at approximately 5-10% of the speed of the same program in internal Flash. This results from the combined effects of the 8-bit or 16-bit wide external interface compared to the fast 32-bit wide access to internal Flash and the multiple clock cycles required for an access to the external Flash. SDRAM is recommended for the best performance using external memory.

Three example applications are provided illustrating execution from external Flash when using the optional Flash and SRAM Memory Expansion Board:

boot_eth_ext This boot loader variant is configured to download applications sent from the LM-Flash utility to external Flash and boot them directly from external memory. To force a download of a new binary while this boot loader is present in internal Flash, reboot the board with the “User Switch” pressed. Applications used with this boot loader must be compiled and linked to run from address 0x60000000 and must not program the system clock or reconfigure the EPI.

ext_demo_1 This simple application prints some status to UART0 then transfers control back to the boot loader in preparation for the download of a new firmware image. It must be used with the `boot_eth_ext` example boot loader and will not run in isolation.

ext_demo_2 This example is similar to `uart_echo` with the exception that it runs from external Flash memory and is configured to look for a particular string (“swupd”) in the user input and transfer control back to the boot loader when this is detected. Like `ext_demo_1`, it must be used with the `boot_eth_ext` example boot loader and will not run in isolation.

3.5.1 Building and Running from External Memory

Building an application to run either partially or wholly from external memory requires a few changes compared to an application intended for internal Flash execution.

- A new linker script file is required which defines the external memory area and determines which sections of your code and data are to be placed there. The scripts used for the `ext_demo_1` and `ext_demo_2` applications may be used as references to develop your own versions.
- The exception vector table must reside in either internal Flash or SRAM since the Cortex M3 Nested Vectored Interrupt Controller (NVIC) is unable to access the EPI memory region.
- All EPI configuration must be performed before any code or data in external memory is accessed. Similarly, no system clock changes which would require EPI reconfiguration may be made in the code executing from external memory.

The `boot_eth_ext` example boot loader configures the system clock and EPI on behalf of the application it will ultimately boot. It also copies the application vector table from the start of the binary in external Flash to the bottom of internal SRAM and sets the NVIC Vector Table Offset Register to point to the new table. The application linker scripts are set up to reserve this area of SRAM for the vector table.

3.5.2 Downloading an Image to External Flash

A binary image may be downloaded to external Flash using one of two methods. With the `boot_eth_ext` boot loader image in internal Flash, the LM Flash Programmer utility (LMFlash) may be used to write a binary file to the base of the Flash found on the Flash and SRAM Memory Expansion Board via Ethernet.

Other applications running in internal memory may also support download to external Flash by making use of the general purpose TFTP server found in `utils/tftp.c` along with a suitable support file which provides Flash write support. An example of such a file is `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/qs-checkout/tftp_qs.c` which supports reading and writing files either from or to the external Flash device on an Flash and SRAM Memory Expansion Board or the serial Flash device found on the base DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board. With this server running as part of the Stellaris application, files can be written to the external Flash device using command-line TFTP from an Ethernet-connected host system.

3.5.3 Debugging Code in External Memory

Third-party debugger support for the EPI address space and external Flash is not currently included in the StellarisWare release, however, you can still debug code or view data in this memory region. Once the system clock and EPI configuration has been set, debuggers can read and display the content of EPI memory as usual.

Debugging code running in external memories can pose problems, however. The Cortex-M3 Flash Patch and Breakpoint (FPB) unit is unable to access the EPI address range so hardware breakpoints cannot be set in this region. This does not cause problems when debugging code from external SRAM or SDRAM since software breakpoints can be used as normal, but when debugging from external Flash, this is problematic and requires a different approach to debugging in this situation.

The Cortex-M3 Data Watchpoint and Trace (DWT) unit has visibility into EPI address space and, depending upon the debugger, this can allow read watchpoints to be used as code breakpoints in external Flash. If this is not possible on a particular debugger, code can be recompiled with an explicit breakpoint (BKPT) instruction or a "while(1)" at the desired position to force the processor to stop at that point. Execution can be resumed by editing the PC register value to skip to the next instruction.

Note: Assembler-level (but not source-level) single stepping is supported from external Flash since this does not require the use of a hardware breakpoint unit.

3.6 Stellaris FPGA Expansion Board

The FPGA Expansion Board is available separately from the basic DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board and illustrates connection of an FPGA device to the Stellaris microcontroller using the General Purpose (or Machine-to-Machine) mode of the External Peripheral Interface (EPI). The expansion board contains following major components:

- Xilinx XC3S100E FGPA
- 1 MB of SRAM dedicated to the FPGA
- Omnivision OV7690/OV7191 VGA CMOS camera module

An I2C EEPROM on this board contains configuration and timing information allowing the PinoutSet() function to configure the EPI correctly and allow all the devices to be accessed. I2C is also used to initialize and control the VGA camera on the board.

The expansion board connects to the jumper block near the QVGA touchscreen in addition to the EPI connector and intercepts control and data signals to the panel. The touchscreen display, which is normally accessed via GPIOs when this expansion board is absent, is driven directly from the FPGA which mixes graphics and video to generate the display. Note that the thumbwheel potentiometer must be disabled by removing the “PB4/POT” jumper when the FPGA Expansion Board is in use since this board requires the EPI signal which is found on pin PB4.

When this expansion board is detected during a call to PinoutSet(), the `g_eDaughterType` global variable is set allowing the display driver (`kitronix320x240x16_ssd2119_8bit.c`) and touchscreen driver (`touch.c`) to dynamically reconfigure themselves to operate with the new expansion board.

After a reset, the FPGA operates in a legacy mode where the LCD display controller’s control and data registers are mapped to locations in the FPGA address space. This mode is used by the `kitronix320x240x16_ssd2119_8bit.c` display driver and allows existing applications to continue working with the new expansion board present. To allow use of the motion video features of the FPGA, however, a second display handling mode is supported. This mode stores a copy of the current graphics frame buffer in FPGA SRAM and allows mixing of this image with video from the camera using chromakeying. To access this mode, a new display driver, `kitronix320x240x16_fpga.c` must be used. This driver will work only when the FPGA Expansion Board is present and requires the use of a different `tDisplay` pointer when initializing the graphics library and widgets. Otherwise it is functionally equivalent to the previous driver from an application’s perspective.

In addition to the new display driver, two other software modules are provided to offer support for the video features of the expansion board. These are both found in the `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers` directory. File `camera.c` and its associated header `camera.h` provide a high level API to initialize and control the camera on the expansion board. Functions allow capture to be started or stopped, picture attributes such as mirror, flip, brightness and contrast to be set, and capture and display position and size to be controlled.

To allow motion video to be added easily to widget-based applications, another module, `vidwidget.c` may be used. This is a graphics library widge class which supports video display. A single widget of this class may be created and the widget's area will be filled with motion video according to the widget's current style and parameters passed to its various APIs. This widget class acts as a wrapper over the API provided by `camera.c`. Any application using it should not make calls to the camera API in addition to the video widget API since all camera control functions are supported at the widget interface level. Example application `videocap` illustrates the use of the video widget.

3.7 Stellaris EM2 Expansion Board

The EM2 Expansion Board is available separately from the basic DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board and offers a simple interface between the DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI EPI connector and up to two Texas Instruments RF Evaluation Modules (EMs). The board provides no function other than routing SSI, UART, I2C, and GPIO signals to the appropriate module connectors. It contains a 1Kb I2C EEPROM which is used for board identification purposes. The content of the EEPROM causes the `PinoutSet` function to set the `g_eDaughterType` global variable to the `DAUGHTER_EM2` value and prevents the function from configuring the EPI peripheral which is not used by the EM2 Expansion Board.

This expansion board enables wireless application development using Low Power RF and RFID evaluation modules on the DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI platform. Supported protocols include ZigBee and Texas Instruments' SimpliciTI.

4 SimpliciTI Low Power RF Stack

4.1 SimpliciTI Overview

SimpliciTI is a simple, connection-based, peer-to-peer communication protocol intended to allow radio communication to be implemented quickly and cheaply in an application making use of one of Texas Instrument's low power radio transcievers.

The protocol supports three types of device - End Devices, Access Points and Range Extenders - allowing direct peer-to-peer and star network topologies to be implemented. Simple example applications are provided illustrating networks using each of these types of device. Each of these examples is implemented using one or more application binaries, one for each device type in the example network.

For a detailed description of each of these examples, please see the document "SimpliciTI Sample Application User's Guide" which can be found in `C:/StellarisWare/SimpliciTI-1.1.1/Documents` assuming you installed StellarisWare in the default directory.

To build these sample applications for non-Stellaris SimpliciTI platforms you will need to download and install the generic SimpliciTI-1.1.1 software release. This can be found via the SimpliciTI web page at `http://www.ti.com/simpliciti`.

4.1.1 Simple Peer-to-Peer

The most basic SimpliciTI example shows communication between two peer devices, one acting as talker ("LinkTo") and the other as listener ("LinkFrom"). Once a link is established between the two devices, they exchange small packets and toggle their LEDs according to one of the bytes in the packet.

This example requires two boards to run.

4.1.2 Polling with Access Point

This example expands upon "Simple Peer-to-Peer" and illustrates communication between two peer devices one of which, the Receiver, operates in polling mode. To support polling, an Access Point supporting store-and-forward capability must be present. Messages from the Sender device are stored by the Access Point and passed on to the Receiver the next time it polls for data.

This example requires three boards to run - one access point, one talker end device and one listener end device.

4.1.3 Access Point as Data Hub

This example illustrates the case where an Access Point also acts as an End Device for one or more other devices. Devices create links with the access point then transmit packets telling it to toggle its LEDs based on end device button presses.

This example requires at least two boards to run - one access point and at least one end device.

4.1.4 Cascading End Devices

This example application mimics a sensor or alarm network and illustrates broadcast communication without the need to form explicit links between devices. Each device sleeps for a whole then wakes up and checks its “sensor”. If the sensor indicates that an alarm should be notified, it starts transmitting the alarm message repeatedly. If no alarm is signalled, it listens for a while for any incoming alarm messages from other devices. If not are heard, it goes back to sleep, otherwise it starts repeatedly retransmitting the alarm.

This example requires a minimum of two boards to run. More boards may be used to illustrate a larger sensor network.

4.2 Supported Transceivers

The Stellaris implementation of SimpliciTI-1.1.1 supports the following radio transceivers and frequency bands. A separate code tree is provided for each transceiver/frequency band combination. In most cases, the contents of each tree are identical except for the radio configuration information but this arrangement allows users to start from a known configuration as close as possible to their own without the need to modify or rebuild any examples initially.

The SimpliciTI source code included in the StellarisWare release includes support for various System-on-Chip radios. These are not, however, supported by the Stellaris example applications or board support package which supports only EM boards equipped with the transceivers listed here. Stellaris example applications will, however, communicate with other TI evaluation and reference boards running an equivalent example on a compatible System-on-Chip radio device.

Transceiver	**Frequency Band**
CC1100E	470MHz
CC1100E	950MHz
CC1101	433MHz
CC1101	868MHz
CC1101	915MHz
CC2420	2.4GHz (proprietary)
CC2500	2.4GHz (802.15.4)

4.3 SimpliciTI on Stellaris

The SimpliciTI stack source code can be found in the `StellarisWare/SimpliciTI-1.1.1` directory. The source code is essentially unchanged from the standard SimpliciTI 1.1.1 release which can be downloaded from the Texas Instruments SimpliciTI web site at http://www.ti.com/simpliciti. A small number of cosmetic changes have been made to allow the code to be built on all Stellarisware-supported toolchains but these amount to no more than variable renaming and some slight restructuring to remove warnings only seen on a subset of compilers. A couple of slightly more significant changes have been made and these are detailed below.

4.3.1 Important Differences

The Stellaris port of SimpliciTI includes the following differences compared to the generic release of the stack software. None of these changes alter the API or function offered by SimpliciTI but they do make it somewhat easier to add SimpliciTI support to your application and to support all toolchains supported by StellarisWare.

4.3.1.1 SimpliciTI Configuration Parameters

The original SimpliciTI release supports IAR and Code Composer Studio toolchains and passes stack configuration parameters by means of toolchain-specific text files which contain compiler command line parameters defining various labels. To ensure portability without the need to have multiple copies of configuration files, the Stellaris port allows the configuration parameters to be placed in a header file instead. This header, `simpliciti_config.h`, can be found in each example application directory. When building the stack, if the single label `SIMPLICITI_CONFIG_HEADER` is defined, the build will pull configuration parameters from `simpliciti_config.h`. If this label is undefined, the parameters may be passed via the command line as in the non-Stellaris case.

4.3.1.2 Device Addressing

The SimpliciTI protocol requires that all devices in a network are identified by a unique address. In the example applications shipped with SimpliciTI 1.1.1 addresses are defined in the project configuration parameters with different addresses used for different device types. This does, however, pose problems in cases where multiple end devices are required - the user needs to modify the configuration and build different binaries for each end device to use.

In the Stellaris implementation, although hardcoded addresses are still supported via the `simpliciti_config.h` header file, the default option is to derive a 4 byte SimpliciTI network address from the least significant 4 bytes of the Ethernet MAC address stored in the flash user registers. This ensures that every device has a unique ID and allows the same end device binary to be used on multiple boards without running the risk of an address clash.

4.3.1.3 Wrapper Files

To make it easier to include the SimpliciTI stack in your application, two wrapper files are provided. `simplicitilib.c` includes all the source files from the SimpliciTI stack and `simplicitilib.h` includes all the public headers from the stack. Include `simplicitilib.c` in your project and include `simplicitilib.h` in any source file that is to call SimpliciTI API functions and you will be ready to start using the stack. To ensure that all the required header files are found, also make sure that you have the following paths in your toolchain's include path for the project. Note that these are all relative to the StellarisWare installation directory.

- SimpliciTI-1.1.1
- SimpliciTI-1.1.1/Components/bsp
- SimpliciTI-1.1.1/Components/bsp/boards/dk-lm3s9d96
- SimpliciTI-1.1.1/Components/bsp/drivers
- SimpliciTI-1.1.1/Components/mrfi
- SimpliciTI-1.1.1/Components/simpliciti/nwk
- SimpliciTI-1.1.1/Components/simpliciti/nwk_applications

4.3.1.4 Example Application Code

The original SimpliciTI-1.1.1 example application source and project files are not included in the StellarisWare release. Instead, the SimpliciTI examples have been rewritten for use with the development board. These mimic the operation of the original examples using graphical widgets on the display to show the LED states and provide the necessary buttons. These examples can be found under `C:/StellarisWare/boards/dk-lm3s9d96-em2-cc1101_433-simpliciti`.

4.3.1.5 Support for eZ340-Chronos Interoperability

To allow interoperation with the default software application found on the eZ340-Chronos sports watch, the radio configuration parameters for the CC1101 transceiver have been updated. These parameters are to be found in the file `smartrf_CC1101.h` in directory `SimpliciTI-1.1.1/Components/mrfi/smartrf/CC1101`.

The parameter set used with this radio is controlled by several labels which can be defined in `simpliciti_config.h`. These are as follow:

CHRONOS_RADIO_CONFIG If this label is set, the CC1101 transceiver is configured with parameters allowing communication with the eZ430-Chronos sport watch application. If undefined, the parameters allow communication with other SimpliciTI-enabled platforms using a compatible radio and running one of the standard SimpliciTI example applications.

ISM_US This label selects the 915MHz frequency band typically used by devices in the United States.

ISM_EU This label selects the 868MHz frequency band typically used by devices in Europe.

ISM_LF This label selects the 433MHz frequency band.

4.3.2 SimpliciTI Documentation

Documentation for the SimpliciTI stack can be found in the `SimpliciTI-1.1.1/Documents` directory beneath the StellarisWare installation directory. The API is documented in `SimpliciTI API.pdf` and `SimpliciTI Sample Application User's Guide.pdf` also provides valuable information on building and running the standard example applications on different SimpliciTI-supported hardware platforms.

The generic SimpliciTI-1.1.1 software release supporting various boards other than Stellaris platforms can be downloaded via the SimpliciTI web page at `http://www.ti.com/simpliciti`.

4.3.3 The Stellaris SimpliciTI BSP

The SimpliciTI software stack makes use of a simple board support package. This software can be found under `C:/StellarisWare/SimpliciTI-1.1.1/Components/bsp` and comprises three main directories:

1. boards
2. drivers
3. mcus

The `boards` directory contains header files and source defining board- specific, low level operations relating to system initialization and the use of GPIO-based buttons and LEDs. The Stellaris BSP implementation can be found in the `dk-lm3s9b96`subdirectory.

Header files `bsp_board_defs.h`, `bsp_button_defs.h` and `bsp_driver_defs.h` contain macro definitions that are used by the common code in the `drivers` directory to provide button and LED support to applications. Files `bsp_drivers.c` and `bsp_board.c` contain board-specific initialization and timing functions.

The `boards/dk-lm3s9b96` directory contains one more subdirectory, `bsp_external`, which contains important files used to support SSI communication to and signalling from the various supported radio transceivers. The file `mrfi_board.c` in this directory contains a simple wrapper function allowing the radio transceiver's signal GPIOs to be connected to the MCU interrupt and have the low level interrupt handler call into a common handler in the SimpliciTI radio support code. Header file `mrfi_board_defs.h` contains a collection of macros to support SSI communication and GPIO pin configuration. These are used by the common radio support code found in `C:/StellarisWare/SimpliciTI-1.1.1/Components/mrfi` .

The `drivers` directory contains code common to all BSPs. Files here make use of macros defined in the `boards` directory header files to provide functions to control LEDs and read the states of buttons. Although the Stellaris implementation includes definitions for the single user LED and button on the DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board, these are not used in the Stellaris versions of the SimpliciTI example applications which, instead, mimic the operation of LEDs and buttons using widgets shown on the QVGA display.

The `mcu` directory contains processor-specific macro definitions to control interrupts and definitions of unambiguously-sized data types.

5 Example Applications

The example applications show how to use features of the Cortex-M3 microprocessor, the peripherals on the Stellaris microcontroller, and the various libraries provided in the StellarisWare software release - the peripheral driver library, graphics library, and USB library. These applications are intended for demonstration and as a starting point for new applications.

There is an IAR workspace file (`dk-lm3s9d96-em2-cc1101_433-simpliciti.eww`) that contains the peripheral driver library project, graphics library project, USB library project, and all of the board example projects, in a single, easy-to-use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`dk-lm3s9d96-em2-cc1101_433-simpliciti.mpw`) that contains the peripheral driver library project, graphics library project, USB library project, and all of the board example projects, in a single, easy-to-use workspace for use with uVision.

All of these examples are located in the `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti` subdirectory of the firmware development package source distribution.

5.1 “Cascading End Devices” Example (simpliciti_cascade)

This application offers the functionality of the generic SimpliciTI “Cascading End Devices” example and simulates a network of alarm devices. When an alarm is raised on any one device, the signal is cascaded through the network and retransmitted by all the other devices receiving it.

The application can communicate with other SimpliciTI-enabled devices running with compatible radios and running their own version of the ”Cascading End Devices” example or with other Stellaris development boards running this example.

To run this binary correctly, the Stellaris development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

The main loop of this application wakes every 5 seconds or so and checks its “sensor”, in this case a button on the display labelled “Sound Alarm”. If its own alarm has not been raised, it listens for alarm messages from other devices. If nothing is heard, the application toggles LED1 then goes back to sleep again. If, however, an alarm is signalled by the local “sensor” or an alarm message from another device is received, the application continually retransmits the alarm message and toggles LED2. In the Stellaris implementation of the application the “LEDs” are shown using onscreen widgets.

For additional information on running this example and an explanation of the communication between talker and listener, see section 3.3 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming you installed StellarisWare in its default directory.

5.2 Access Point for use with eZ430-Chronos-433 (simpliciti_chronos)

This application provides a SimpliciTI Low Power RF access point that is capable of receiving and displaying data from an eZ430-Chronos-433 development tool running the default Sports Watch firmware. The development board must be equipped with an EM2 expansion board and a CC1101EM 433Mhz radio transceiver and antenna for this application to run correctly. The CC1101EM module must be installed in the “mod1” connector (the connector nearest the oscillator) on the EM2 expansion board.

The eZ430-Chronos-433 development tool uses the SimpliciTI protocol to support three of the Sport Watch features - “RF Tilt Control”, “PPT Control” and “RF Sync”. The simpliciti_chronos example application detects which of these modes is in operation and configures the display appropriately to show the information that the watch is transmitting. If no packet is! received from the watch within 5 seconds, the display reverts to the initial opening screen and the application waits for new data.

To select each of the operating modes, cycle through the various options offered by pressing the lower left “#” button on the sports watch until either “ACC”, “PPt” or “SYNC” is displayed. To activate data transmission press the lower right, down arrow key. When transmission is active, a wireless icon will flash on the watch’s display. Pressing the down arrow key once again will disable the transmitter.

RF Tilt Control Mode (ACC) In RF Tilt Control mode, the watch sends packets containing information on button presses and also the output from its integrated 3-axis accelerometer. The simpliciti_chronos application displays button presses by highlighting graphics of the buttons on the display. Accelerometer data is plotted in an area of the screen with (x,y) data used to determine the position of lines drawn and z data controlling the color. Buttons are provided on the display to set the origin point for the accelerometer data (“Calibrate”) and clear the accelerometer display (“Clear”).

PowerPoint Control Mode (PPt) In PowerPoint Control mode, only button press information is transmitted by the watch. When these packets are detected by the access point, it displays an image of the watch face and highlights which buttons have been pressed.

Sync Mode (SYNC) Sync mode is used to set various watch parameters and allows an access point to send commands to the watch and retrieve status from it. Every 500mS or so, the watch sends a “ready to receive” packet indicating that it is able to process a new command. The access point application responds to this with a status request causing the watch to send back a 19 byte status packet containing the current time, date, alarm time, altitude and temperature. This data is then formatted and displayed on the development board LCD screen. A button on the display allows the watch display format to be toggled between metric and imperial. When this button is pressed, a command is sent to the watch to set the new format and this display format change takes effect with the next incoming status message.

Note that the “ready to receive” packets sent by the watch are not synchronized with the watch second display. This introduces an interference effect that results in the development board seconds display on the LCD not update at regular 1 second intervals.

5.3 Access Point for “Access Point as Data Hub” example (simpliciti_hub_ap)

This application offers the access point functionality of the generic SimpliciTI Ap_as_Data_Hub. To run this example, two additional SimpliciTI-enabled boards using compatible radios must also be present, each running the EndDevice configuration of the application. If using the Stellaris development board, this function is found in the simpliciti_hub_dev example application. On other hardware it is the EndDevice configuration of the Ap_as_Data_Hub example as supplied with SimpliciTI 1.1.1.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

When the access point application is started, both “LEDs” on the display are lit indicating that the access point is waiting for connections from end devices. The LEDs may start flashing, indicating that the frequency agility feature has caused an automatic chanel change. This flashing will continue until a message is received from an end device. When an end device connects to the access point, pressing buttons on the end device will send a message to the access point which will toggle one of its LEDs depending upon the content of the message.

The access point also offers the option to force a channel change. Pressing the “Change Channel” button cycles to the next available radio channel. As in the automatic channel change case, this will cause the LEDs to flash until a new message is received from the end device indicating that it has also changes to the new channel.

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.4 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.4 End Device for “Access Point as Data Hub” example (simpliciti_hub_dev)

This application offers the end device functionality of the generic SimpliciTI “Access Point as Data Hub” example. Pressing buttons on the display will toggle the corresponding LEDs on the access point board to which this end device is linked.

The application can communicate with another SimpliciTI-enabled device equipped with a compatible radio and running its own version of the access point from the ”Access Point as Data Hub” example or with other development boards running the simpliciti_hub_ap example.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

Start the board running the access point example first then start the end devices. The LEDs on the end device will flash once to indicate that they have joined the network. After this point, pressing one of the buttons on the display will send a message to the access point causing it to toggle either LED1 or LED2 depending upon which button was pressed.

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.4 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.5 Channel Sniffer for use with “Access Point as a Data Hub” example (simpliciti_hub_sniffer)

This application provides channel sniffer functionality in a SimpliciTI low power RF network. It may be run on a development board equipped with an EM expansion board and SimpliciTI-compatible radio board to determine which channel the network is currently on. Use this application alongside the “Access Point as a Data Hub” example (simpliciti_hub_ap/dev) which supports frequency agility.

The functionality provided here is equivalent to the “Channel Sniffer” configuration included in the generic SimpliciTI “AP_as_Data_Hub” example application.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle

- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

At least one additional board is required to run this application and two additional boards are needed to show all features of the “Access Point as a Data Hub” example. If running the sniffer application with only one other board, make sure that that board is running the access point binary, simpliciti_hub_ap from StellarisWare or the relevant access point configuration of the generic SimpliciTI “Ap_as_Data_Hub” example if running the access point on a different hardware platform.

Start the board running the sniffer application and both LEDs on the screen will flash until the board joins the access point’s network. After the network is joined the LEDs will indicate the current SimpliciTI channel number in binary. The channel number is also displayed in text at the bottom of the screen. To change the network channel, press button 1 on the board running the access point software.

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.4.2 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.6 Access Point for “Polling with Access Point” example (simpliciti_polling_ap)

This application offers the access point functionality of the generic SimpliciTI Polling_with_AP example. To run this example, two additional SimpliciTI-enabled boards using compatible radios must also be present, one running the sender application and the other running the receiver. If using the Stellaris development board, these functions are found in the simpliciti_polling_dev example application. On other hardware, these are the Sender and Receiver configurations of the “Polling_with_AP” example as supplied with SimpliciTI 1.1.1.

The functionality provided here is equivalent to the “Access Point” configuration included in the generic SimpliciTI “Polling with AP” example application.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

To run this example, power up the access point board and both its LEDs should light indicating that it is active. Next, power up the receiver board and press button 2 (or, on single button boards, press the button for less than 3 seconds). At this point, only LED1 on the receiver board should be lit. Finally power up the sender and press its button 1 (or, on single button boards, press the button for more than 3 seconds). Both LEDs on the sender will blink until it successfully links with the receiver. After successful linking, the sender will transmit a message to the receiver every 3 to 6 seconds. This message will be stored by the access point and passed to the receiver the next time it polls the access point. While the example is running, LEDs on both the sender and receiver will blink. No user interaction is required on the access point.

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.2 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.7 End Device for “Polling with Access Point” example (simpliciti_polling_dev)

This application offers the end device functionality of the generic SimpliciTI Polling_with_AP example. The application can communicate with other SimpliciTI-enabled devices with compatible radios running the “Polling_with_AP” Sender or Receiver configuration. To run this example, a third SimpliciTI-enabled board must also be present running the access point binary, simpliciti_polling_ap for Stellaris development board, or the relevant Access Point configuration of the Polling_with_AP example if using another hardware platform.

The functionality provided here is equivalent to the “Sender” and ”Receiver” configurations included in the generic SimpliciTI “Polling with AP” example application.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

Power up the access point board and both its LEDs should light indicating that it is active. Next, power up the receiver board and select the correct mode by pressing the “Receiver” button on the development board display, pressing button 2 on dual button boards, or pressing the single button for less than 3 seconds on boards with only one button. At this point, only LED1 on the receiver board should be lit. Finally power up the sender select the mode using the onscreen button, by pressing button 2 on dual button boards or by pressing an holding the single button for more than 3 seconds. Both LEDs on the sender will blink until it successfully links with the receiver. After successful linking, the sender will transmit a message to the receiver every 3 to 6 seconds. This message will be stored by the access point and passed to the receiver the next time it polls the access point. While the example is running, LEDs on both the sender and receiver will blink. No user interaction is required on the access point.

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.2 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.8 Range Extender for “Polling with Access Point” example (simpliciti_range_extender)

This application provides range extender functionality in a SimpliciTI low power RF network. It may be run on a development board equipped with an EM expansion board and SimpliciTI-compatible radio board to extend the distance between end devices and the access point when running the “Polling with Access Point” example (simpliciti_polling_ap and simpliciti_polling_dev).

The functionality provided here is equivalent to the “Range Extender” configuration included in the generic SimpliciTI “Polling_with_AP” example application. The application merely receives all SimpliciTI packets and retransmits them. No user interaction is required.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

For additional information on running this example and an explanation of the communication between the two devices and access point, see section 3.2 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming that StellarisWare is installed in its default directory.

5.9 “Simple Peer-to-Peer” example (simpliciti_simple_p2p)

This application offers the functionality of the generic SimpliciTI Simple_Peer_to_Peer example. Whereas the original example builds two independent executables, this version implements both the talker (LinkTo) and listener (LinkListen) functionality, the choice being made via two buttons shown on the display. The application can communicate with another SimpliciTI-enabled device with a compatible radio running one of the Simple_Peer_to_Peer example binaries or another copy of itself running on a second development board.

To run this binary correctly, the development board must be equipped with an EM2 expansion board with a CC1101:433 EM module installed in the “MOD1” position (the connectors nearest the oscillator on the EM2). Hardware platforms supporting SimpliciTI 1.1.1 with which this application may communicate are the following:

- SmartRF04EB + CC1110EM
- EM430F6137RF900
- FET430F6137RF900
- CC1111EM USB Dongle
- EXP430FG4618 + CC1101:433 + USB Debug Interface
- EXP430FG4618 + CC1100:433 + USB Debug Interface
- Stellaris Development Board + EM2 expansion board + CC1101:433

On starting the application, you are presented with two choices on the LCD display. If your companion board is running the “LinkTo” configuration of the application, press the “LinkListen” button. If the companion board is running “LinkListen”, press the “LinkTo” button. Once one of these buttons is pressed, the application attempts to start communication with its peer, either listening for an incoming link request or sending a request. After a link is established the talker board (running in “LinkTo” mode) sends packets to the listener which echos them back after toggling an LED. In the Stellaris implementation of the application the “LEDs” are shown using onscreen widgets.

For additional information on this example and an explanation of the communication between talker and listener, see section 3.1 of the “SimpliciTI Sample Application User’s Guide” which can be found under C:/StellarisWare/SimpliciTI-1.1.1/Documents assuming you installed StellarisWare in its default directory.

6 Development System Utilities

These are tools that run on the development system, not on the embedded target. They are provided to assist in the development of firmware for Stellaris microcontrollers.

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

USB DFU Programmer

Usage:

`dfuprog [OPTION]...`

Description:

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

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

Arguments:

-e specifies the address of the binary.

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

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

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

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

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

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

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

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

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

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

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

-v displays verbose output during the requested operation.

-h displays this help information.

-? displays this help information.

Example:

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

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

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

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

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

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

USB DFU Wrapper

Usage:

`dfuwrap [OPTION]...`

Description:

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

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

Arguments:

-a ADDR specifies the address of the binary.

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

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

-e enables verbose output.

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

-h displays usage information.

-i FILE specifies the name of the input file.

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

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

-q specifies that only error information should be output.

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

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

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

Example:

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

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

Ethernet Flash Downloader

Usage:

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

Description:

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

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

Arguments:

--help displays usage information.

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

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

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

--mac=MAC specifies the MAC address

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

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

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

--verbose specifies that verbose output should be output.

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

Finder

Usage:

`finder`

Description:

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

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

Example:

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

```
finder
```

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 Stellaris USB boot loader (boot_usb). This DLL is used by the dfuprog utility and also by the LMFlash application to allow download and upload of application images to or from a Stellaris-based board via USB.

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

GIMP Script For Texas Instruments Stellaris Button

Description:

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

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

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

USB Dynamic Link Library

Description:

LMUSBDLL is a simple Windows dynamic link library offering low level packet read and write functions for some USB-connected Stellaris example applications. The DLL is written above the Microsoft WinUSB interface and is intended solely to ensure that various Windows-side example applications can be built without having to use WinUSB header files. These header files are not included in the Visual Studio tools and are only shipped in the Windows Device

Driver Kit (DDK). By providing this simple mapping DLL which links to WinUSB, the user avoids the need for a multi-gigabyte download to build the examples.

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

Web Filesystem Generator

Usage:

`makefsfile [OPTION]...`

Description:

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

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

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

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

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

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

Arguments:

-b generates a position-independent binary image.

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

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

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

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

-q enables quiet mode.

-r overwrites the the output file without prompting.

-v enables verbose output.

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

-? displays usage information.

Example:

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

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

String Table Generator

Usage:

`mkstringtable [INPUT FILE] [OUTPUT FILE]`

Description:

Converts a comma separated file (.csv) to a table of strings that can be used by the Stellaris Graphics Library. The source .csv file has a simple fixed format that supports multiple strings in multiple languages. A .c and .h file will be created that can be compiled in with an application and used with the graphics library's string table handling functions. If encoding purely ASCII strings, the strings will also be compressed in order to reduce the space required to store them. If the CSV file contains strings encoded in other codepages, for example UTF8, the "-s" command line option must be used to specify the encoding used for the string and "-u" must also be used to ensure that the strings are stored correctly.

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

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

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

```
LanguageIDs,GrLangEnUS,GrLangDE,GrLangEsSP,GrLangIt
STR_CONFIG,Configuration,Konfigurieren,Configuracion,Configurazione
```

```
STR_INTRO,Introduction,Einfuhrung,Introduccion,Introduzione
STR_QUOTE,Introduction in ""English"","Einfuhrung, in Deutsch",Prueba,Verifica
...
```

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

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

The resulting .c file contains the string table that must be included with the application that is using the string table and two helper structure definitions, one a tCodepointMap array containing a single entry that is suitable for use with GrCodepageMapTableSet() and the other a tGrLibDefaults structure which can be used with GrLibInit() to initialize the graphics library to use the correct codepage for the string table.

While the contents of this .c file are readable, the string table itself may be unintelligible due to the compression or remapping used on the strings themselves. The .h file that is created has the definition for the string table as well as an enumerated type `enum SCOMP_STR_INDEX` that contains all of the string indexes that were present in the original .csv file and external definitions for the tCodepointMap and tGrLibDefaults structures defined in the .c file.

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

This utility is contained in `tools/bin`.

Arguments:

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

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

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

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

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

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

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

Example:

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

```
mkstringtable str.csv str
```

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

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

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

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

NetPNM Converter

Usage:

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

Description:

Converts a NetPBM image file into the format that is recognized by the Stellaris Graphics Library. The input image must be in the raw PPM format (in other words, with the `P4`, `P5` or `P6` tags). The NetPBM image format can be produced using GIMP, NetPBM (`http://netpbm.sourceforge.net`), ImageMagick (`http://www.imagemagick.org`), or numerous other open source and proprietary image manipulation packages.

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

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

1. Load the file (File->Open).

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

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

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

Arguments:

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

Example:

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

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

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

Serial Flash Downloader

Usage:

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

Description:

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

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

Arguments:

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

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

-d disables auto-baud.

-h displays usage information.

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

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

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

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

USB Bulk Data Transfer Example

Description:

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

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

7 Display Driver

7.1 Introduction

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `kitronix320x240x16_ssd2119_8bit.c` containing the source code and `kitronix320x240x16_ssd2119_8bit.h` containing the API definitions for use by applications.

7.2 API Functions

7.3 Programming Example

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

```
tContext sContext;

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

//
// Turn the backlight on at full brightness.
//
Kitronix320x240x16_SSD2119BacklightOn(0xFF);

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

8 External Flash Driver

8.1 Introduction

The external flash driver offers functions to query, erase and program the AMD-compatible flash found on the Flash/SRAM/LCD daughter board.

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `extflash.c` containing the source code and `extflash.h` containing the API definitions for use by applications.

8.2 API Functions

8.3 Programming Example

The following example shows how to check for the presence of the external flash, erase the first block and program some data to it.

```
tBoolean bRetcode;

//
// Make sure that the flash is accessible.
//
if(!ExtFlashPresent())
{
    //
    // Handle a fatal error here - the flash could not be accessed.
    //
}

//
// Erase the first block of the flash device.  This call is synchronous
// and only returns once the erase operation has completed.
//
bRetcode = ExtFlashBlockErase(EXT_FLASH_BASE, true);

if(bRetcode)
{
    //
    // Now write some data to the first block of the flash.  We assume that
    // the array g_pucData contains at least MY_DATA_SIZE bytes and that
    // this contains the data to be written.  This call is also synchronous
    // and only returns once the write has completed.
    //
    bRetcode = ExtFlashWrite(EXT_FLASH_BASE, MY_DATA_SIZE, g_pucData);

    //
    // Did we write the data successfully?
```

```
    //
    if(!bRetcode)
    {
        //
        // An error occurred while trying to write the data.  Handle this
        // here.
        //
    }
}
else
{
    //
    // Oops - we couldn't erase the flash block!. Handle the error
    // condition here.
    //
}
```

9 External RAM Driver

9.1 Introduction

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `extram.c` containing the source code and `extram.h` containing the API definitions for use by applications. Applications using it must also ensure that they include source file `bget.c` which can be found in the `third_party/bget` directory.

9.2 API Functions

9.3 Programming Example

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

```
tBoolean bRetcode;
unsigned char *pucBlock;

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

//
// Did we initialize the SDRAM correctly?
//
if(!bRetcode)
{
    //
    // Handle a fatal error here - the SDRAM could not be initialized.
    //
}

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

if(pucBlock)
```

```
{
    //
    // Go ahead and use the memory allocation.
    //

    ...

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

10 JPEG Display Widget

10.1 Introduction

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

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

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

To use this widget, an application must also include the source for the JPEG decoder which can be found in `third_party/jpeg` along with the SDRAM driver from `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers` and the bget heap manager from `third_party/bget`. The application makefile or project must also ensure that label `INCLUDE_BGET_STATS` is defined to enable optional heap manager functionality that is required by the JPEG decoder.

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `jpgwidget.c` containing the source code and `jpgwidget.h` containing the API definitions for use by applications.

10.2 API Functions

10.3 Programming Example

The following example shows how to initialize a JPEGCanvas widget. The sample application `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/showjpeg` provides a working example of the use of this widget.

```
...

//*****************************************************************************
//
// Forward references
//
//*****************************************************************************
```

```
extern tCanvasWidget g_sBackground;

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

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

...

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

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

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

    //
    // Initialize the SDRAM controller and heap.
    //
    SDRAMInit(1, (EPI_SDRAM_CORE_FREQ_50_100 | EPI_SDRAM_FULL_POWER |
              EPI_SDRAM_SIZE_64MBIT), 1024);

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

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

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

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

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

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

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

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

11 Pinout Driver

11.1 Introduction

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `set_pinout.c` containing the source code and `set_pinout.h` containing the API definition for use by applications.

11.2 API Functions

12 Serial Flash Driver

12.1 Introduction

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

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `ssiflash.c` containing the source code and `ssiflash.h` containing the API definitions for use by applications.

12.2 API Functions

12.3 Programming Example

The following example shows how to use the SSIFlash API.

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

    ...

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

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

```
            //
            // An error occurred!  Handle it here.
            //
            while(1);
        }

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

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

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

        ...
    }
```

13 Sound Driver

13.1 Introduction

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

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `sound.c` containing the source code and `sound.h` containing the API definitions for use by applications.

13.2 API Functions

13.3 Programming Example

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

```
//
// Initialize the sound output.
//
SoundInit(0);

//
// Set the sound output frequency to 1000 Hz.
//
SoundFrequencySet(1000);

//
// Enable the sound output.
//
SoundEnable();

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

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

14 Thumbwheel Driver

14.1 Introduction

The board contains a thumbwheel potentiometer attached to one of the analog-to-digital converter input channels. This driver offers a simple way to configure the ADC to sample the potentiometer and to request that samples be captured.

An application wishing to make use of the thumbwheel initializes the driver and provides a callback function which will be used to notify it when samples have been captured. Another function is provided to allow sample capture to be triggered.

The potentiometer sample captured is read from the ADC, converted from a raw value to millivolts them passed to the calling application via its callback function.

This driver makes use of ADC sequence 2.

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `thumbwheel.c` containing the source code and `thumbwheel.h` containing the API definitions for use by applications.

14.2 API Functions

14.3 Programming Example

The following example shows how to initialize the thumbwheel driver and capture one sample from the potentiometer.

```
volatile tBoolean g_bSampleRead = false;
unsigned short g_usThumbwheelmV = 0;

//
// Application callback function which will be informed on the thumbwheel
// potentiometer voltage when a new sample is available.  The parameter
// passed is scaled in millivolts.
//
void AppThumbwheelCallback(unsigned short usVoltagemV)
{
    //
    // Remember the voltage we were passed and signal the main loop that
    // the thumbween has been read.
    //
    g_usThumbwheelmV = usVoltagemV;
    g_bSampleRead = true;
}

int main(void)
{
```

```
    ...

    //
    // Initialize the thumbwheel driver and provide it our callback
    //
    ThumbwheelInit();
    ThumbwheelCallbackSet(AppThumbwheelCallback);

    //
    // Request a sample from the thumbwheel.
    //
    ThumbwheelTrigger();

    //
    // Wait until the sample is available.
    //
    while(!g_bSampleRead);

    //
    // Do something with the result.
    //

    ...
}
```

15 Touch Screen Driver

15.1 Introduction

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

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

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

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

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

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

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

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

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `touch.c` containing the source code and `thumbwheel.h` containing the API definitions for use by applications.

15.2 API Functions

15.3 Programming Example

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

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

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

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

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

        case WIDGET_MSG_PTR_DOWN:
            g_ulFlags |= FLAG_PTR_DOWN;
            break;

        case WIDGET_MSG_PTR_MOVE:
            g_ulFlags |= FLAG_PTR_MOVE;
            break;

        default:
            break;
    }

    return(0);
}

int main(void)
```

```
{
    ...

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

    ...

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

    ...
}
```

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

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

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

    ...

}
```

16 Video Camera API

16.1 Introduction

The camera module is a low level driver supporting the motion video capture and display functions provided by the optional FPGA/Camera/LCD daughter board. Applications making use of the graphics library may access these features via the “vidwidget” widget which wraps the camera API and offers a higher level interface. Applications must not mix calls to the camera and vidwidget APIs.

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `camera.c` containing the source code and `camera.h` containing the API definitions for use by applications.

16.2 API Functions

16.3 Programming Example

The following example shows how to initialize the video camera on the daughter board and start capturing VGA video into a buffer in the daughter board’s SRAM.

```
#include "inc/hw_types.h"
#include "driverlib/sysctl.h"
#include "drivers/set_pinout.h"
#include "drivers/camera.h"

int
main(void)
{

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

    //
    // Set the device pinout appropriately for this board.
    //
    PinoutSet();

    //
    // Make sure we detected the FPGA daughter board since this application
    // requires it.
    //
    if(g_eDaughterType != DAUGHTER_FPGA)
    {
        //
```

```
        // We can’t run - the FPGA daughter board doesn’t seem to be there.
        //
        while(1)
        {
            //
            // Hang here on error.
            //
        }
    }

    //
    // Initialize the camera module and configure for VGA-sized capture.
    //
    CameraInit((CAMERA_SIZE_VGA | CAMERA_FORMAT_RGB565), VIDEO_BUFF_BASE, 0);

    //
    // Start capturing motion video.
    //
    CameraCaptureStart();

    //
    // Set the video display buffer address.
    //
    CameraDisplayBufferSet(VIDEO_BUFF_BASE, VIDEO_BUFF_STRIDE, true);

    //
    // Downscale the 640x480 video to fit the 320x240 display.
    //
    CameraDisplayDownscaleSet(true);

    //
    // Start display of the video.
    //
    CameraDisplayStart(true);

    //
    // Drop into our main loop (which, in this case, doesn’t do anything).
    //
    while(1)
    {
    }
}
```

17 Video Display Widget

17.1 Introduction

The vidwidget module contains a custom control widget for use with the Stellaris Graphics Library and DK-LM3S9D96-EM2-CC1101_433-SIMPLICITI board equipped with the optional FPGA/Camera/LCD daughter board. This widget supports display of motion or still video images on the LCD display. A “VidWidget” class object can be thought of as a video canvas.

When using the FPGA/Camera/LCD daughter board, motion video and graphics occupy two different display planes with graphics logically “on top” of video. Although the video widget may occupy any subrectangle of the display, the video image itself is always positioned at the top left of the display and covers the whole screen area. Areas of the graphics display which are painted with a chromakey color become transparent when video is enabled and allow the video image to shine through the graphics plane. The video widget automatically paints itself with the user’s chromakey color but note that any other areas of the display containing this color will also become transparent and show video. Applications should, therefore, chose a unique color that does not appear elsewhere in their user interface for the areas of the screen which are to show video. Typically a color such as bright magenta (ClrMagenta) is used.

Video may be captured in VGA (640x480) or QVGA (320x240) resolution and may be displayed full size or downscaled by a factor of 2 in both dimensions. In cases where VGA video is being capture and no downscaling is selected, dragging a finger or stylus over the video widget area will cause the video displayed to scroll within the widget.

To use this widget, an application must also include the source for the camera driver, `camera.c`, and must be using the special display driver intended for the FPGA daughter board, `kitronix_320x240x16_fpga.c`. Both of these files can be found in the same directory as the video widget source file. The underlying camera API is wrapped by vidwidget so applications using vidwidget must not call any camera APIs directly.

This driver is located in `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/drivers`, with `vidwidget.c` containing the source code and `vidwidget.h` containing the API definitions for use by applications.

17.2 API Functions

17.3 Programming Example

The following example shows how to initialize a video widget. The sample application `boards/dk-lm3s9d96-em2-cc1101_433-simpliciti/videocap` provides a working example of the use of this widget.

```
#include "inc/hw_types.h"
```

```
#include "driverlib/interrupt.h"
#include "driverlib/sysctl.h"
#include "grlib/grlib.h"
#include "grlib/widget.h"
#include "drivers/vidwidget.h"
#include "drivers/kitronix320x240x16_fpga.h"
#include "drivers/touch.h"
#include "drivers/set_pinout.h"
#include "drivers/camera.h"

//*****************************************************************************
//
// Workspace for the video widget.
//
//*****************************************************************************
tVideoInst g_sVideoInst;

//*****************************************************************************
//
// The video widget used to hold the decompressed JPEG image and
// display it.  This is a simple JPEG canvas which will decompress and display
// the image contained in buffer g_pucJPEGImage in a 320x215 pixel control with
// a single pixel white outline.  Style flag JW_STYLE_SCROLL enables automatic
// redraw based on user touchscreen scrolling.  This is rather CPU intensive
// so it may be better to remove this style flag and install an OnScroll
// callback that can be used to pace the redraws.  This is demonstrated in the
// showjpeg example application.
//
//*****************************************************************************
#define VID_LEFT      0
#define VID_TOP       0
#define VID_WIDTH     320
#define VID_HEIGHT    240
VideoWidget(g_sBackground, WIDGET_ROOT, 0, 0,
       &g_sKitronix320x240x16_FPGA, VID_LEFT, VID_TOP, VID_WIDTH, VID_HEIGHT,
       VW_STYLE_VGA, ClrMagenta, 0, 0, 0, &g_sVideoInst);
...

int
main(void)
{

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

    //
    // Set the device pinout appropriately for this board.
    //
    PinoutSet();

    //
    // Make sure we detected the FPGA daughter board since this application
    // requires it.
    //
    if(g_eDaughterType != DAUGHTER_FPGA)
    {
        //
        // We can't run - the FPGA daughter board doesn't seem to be there.
        //
        while(1)
        {
            //
            // Hang here on error.
```

```
            //
        }
    }

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

    //
    // Initialize the video capture widget.  This must be done before the
    // display driver is initialized.
    //
    VideoWidgetCameraInit(&g_sBackground, VIDEO_BUFF_BASE);

    //
    // Initialize the display driver.
    //
    Kitronix320x240x16_FPGAInit(GRAPHICS_BUFF_BASE);

    //
    // Enable the display backlight.
    //
    Kitronix320x240x16_FPGABacklight(true);

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

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

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

    //
    // Paint the widget tree to make sure they all appear on the display.
    //
    WidgetPaint(WIDGET_ROOT);

    //
    // Now that everything is set up, turn on the video display.
    //
    VideoWidgetBlankSet((tWidget *)&g_sBackground, false);

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

18 Command Line Processing Module

18.1 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.

18.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_INVALID_ARG
- CMDLINE_TOO_FEW_ARGS
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

18.2.1 Data Structure Documentation

18.2.1.1 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.

18.2.2 Define Documentation

18.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

18.2.2.2 CMDLINE_INVALID_ARG

Definition:

```
#define CMDLINE_INVALID_ARG
```

Description:

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

18.2.2.3 CMDLINE_TOO_FEW_ARGS

Definition:

```
#define CMDLINE_TOO_FEW_ARGS
```

Description:

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

18.2.2.4 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

18.2.3 Function Documentation

18.2.3.1 CmdLineProcess

Process a command line string into arguments and execute the command.

Prototype:

```
int
CmdLineProcess(char *pcCmdLine)
```

Parameters:

pcCmdLine points to a string that contains a command line that was obtained by an application by some means.

Description:

This function will take the supplied command line string and break it up into individual arguments. The first argument is treated as a command and is searched for in the command table. If the command is found, then the command function is called and all of the command line arguments are passed in the normal argc, argv form.

The command table is contained in an array named `g_sCmdTable` which must be provided by the application.

Returns:

Returns **CMDLINE_BAD_CMD** if the command is not found, **CMDLINE_TOO_MANY_ARGS** if there are more arguments than can be parsed. Otherwise it returns the code that was returned by the command function.

18.2.4 Variable Documentation

18.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

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

18.3 Programming Example

The following example shows how to process a command line.

```
//
// Code for the "foo" command.
//
int
ProcessFoo(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "bar" command.
//
int
ProcessBar(int argc, char *argv[])
{
    //
    // Do something, using argc and argv if the command takes arguments.
    //
}

//
// Code for the "help" command.
//
int
ProcessHelp(int argc, char *argv[])
{
    //
    // Provide help.
    //
}

//
// The table of commands supported by this application.
//
tCmdLineEntry g_sCmdTable[] =
{
    { "foo", ProcessFoo, "The first command." },
    { "bar", ProcessBar, "The second command." },
    { "help", ProcessHelp, "Application help." }
};

//
// Read a process a command.
//
int
Test(void)
{
    unsigned char pucCmd[256];

    //
    // Retrieve a command from the user into pucCmd.
    //
    ...

    //
    // Process the command line.
    //
    return(CmdLineProcess(pucCmd));
}
```

19 CPU Usage Module

19.1 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.

19.2 API Functions

Functions

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

19.2.1 Function Documentation

19.2.1.1 CPUUsageInit

Initializes the CPU usage measurement module.

Prototype:

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

Parameters:

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

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

ulTimer is the index of the timer module to use.

Description:

This function prepares the CPU usage measurement module for measuring the CPU usage of the application.

Returns:

None.

19.2.1.2 CPUUsageTick

Updates the CPU usage for the new timing period.

Prototype:

```
unsigned long
CPUUsageTick(void)
```

Description:

This function, when called at the end of a timing period, will update the CPU usage.

Returns:

Returns the CPU usage percentage as a 16.16 fixed-point value.

19.3 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();
    }
}
```

20 CRC Module

20.1 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.

20.2 API Functions

Functions

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

20.2.1 Function Documentation

20.2.1.1 Crc16

Calculates the CRC-16 of an array of bytes.

Prototype:

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

Parameters:

usCrc is the starting CRC-16 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

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

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

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

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

Returns:

The CRC-16 of the input data.

20.2.1.2 Crc16Array

Calculates the CRC-16 of an array of words.

Prototype:

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

Parameters:

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

pulData is a pointer to the data buffer.

Description:

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

Returns:

The CRC-16 of the input data.

20.2.1.3 Crc16Array3

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

Prototype:

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

Parameters:

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

pulData is a pointer to the data buffer.

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

Description:

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

Returns:

None

20.2.1.4 Crc32

Calculates the CRC-32 of an array of bytes.

Prototype:

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

Parameters:

ulCrc is the starting CRC-32 value.

pucData is a pointer to the data buffer.

ulCount is the number of bytes in the data buffer.

Description:

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

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

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

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

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

20.2.1.5 Crc8CCITT

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

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

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

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

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

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

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

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

20.3 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);
```

21 Flash Parameter Block Module

21.1 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.

21.2 API Functions

Functions

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

21.2.1 Function Documentation

21.2.1.1 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

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

Description:

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

Returns:

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

21.2.1.2 FlashPBInit

Initializes the flash parameter block.

Prototype:

```
void
FlashPBInit(unsigned long ulStart,
            unsigned long ulEnd,
            unsigned long ulSize)
```

Parameters:

ulStart is the address of the flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash.

ulEnd is the address of the end of flash memory to be used for storing flash parameter blocks; this must be the start of an erase block in the flash (the first block that is NOT part of the flash memory to be used), or the address of the first word after the flash array if the last block of flash is to be used.

ulSize is the size of the parameter block when stored in flash; this must be a power of two less than or equal to the flash erase block size (typically 1024).

Description:

This function initializes a fault-tolerant, persistent storage mechanism for a parameter block for an application. The last several erase blocks of flash (as specified by *ulStart* and *ulEnd* are used for the storage; more than one erase block is required in order to be fault-tolerant.

A parameter block is an array of bytes that contain the persistent parameters for the application. The only special requirement for the parameter block is that the first byte is a sequence number (explained in FlashPBSave()) and the second byte is a checksum used to validate the correctness of the data (the checksum byte is the byte such that the sum of all bytes in the parameter block is zero).

The portion of flash for parameter block storage is split into N equal-sized regions, where each region is the size of a parameter block (*ulSize*). Each region is scanned to find the most recent valid parameter block. The region that has a valid checksum and has the highest sequence number (with special consideration given to wrapping back to zero) is considered to be the current parameter block.

In order to make this efficient and effective, three conditions must be met. The first is *ulStart* and *ulEnd* must be specified such that at least two erase blocks of flash are dedicated to parameter block storage. If not, fault tolerance can not be guaranteed since an erase of a single block will leave a window where there are no valid parameter blocks in flash. The second condition is that the size (*ulSize*) of the parameter block must be an integral divisor of the size of an erase block of flash. If not, a parameter block will end up spanning between two erase blocks of flash, making it more difficult to manage. The final condition is that the size of the flash dedicated to parameter blocks (*ulEnd* - *ulStart*) divided by the parameter block size (*ulSize*) must be less than or equal to 128. If not, it will not be possible in all cases to determine which parameter block is the most recent (specifically when dealing with the sequence number wrapping back to zero).

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

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

Returns:

None.

21.2.1.3 FlashPBSave

Writes a new parameter block to flash.

Prototype:

```
void
FlashPBSave(unsigned char *pucBuffer)
```

Parameters:

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

Description:

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

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

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

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

Returns:

None.

21.3 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);
}
```

22 File System Wrapper Module

22.1 Introduction

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

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

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

22.2 API Functions

Functions

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

22.2.1 Function Documentation

22.2.1.1 fs_close

Closes a file.

Prototype:

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

Parameters:

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

Description:

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

Returns:
None.

22.2.1.2 fs_init

Initializes the file system wrapper.

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

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

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

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

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

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

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

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

22.2.1.3 fs_map_path

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

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

Parameters:

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

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

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

Description:

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

Returns:

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

22.2.1.4 fs_open

Opens a file.

Prototype:

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

Parameters:

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

Description:

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

Returns:

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

22.2.1.5 fs_read

Reads data from an open file.

Prototype:

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

Parameters:

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

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

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

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

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

22.2.1.6 fs_tick

Provides a periodic tick for the file system.

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

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

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

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

Returns:
None

22.3 Programming Example

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

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

void AccessFile(void)
```

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

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


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

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

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

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

23 Integer Square Root Module

23.1 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.

23.2 API Functions

Functions

- unsigned long isqrt (unsigned long ulValue)

23.2.1 Function Documentation

23.2.1.1 isqrt

Compute the integer square root of an integer.

Prototype:

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

Parameters:

ulValue is the value whose square root is desired.

Description:

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

Returns:

Returns the square root of the input value.

23.3 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);
```

24 Ethernet Board Locator Module

24.1 Introduction

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

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

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

24.2 API Functions

Functions

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

24.2.1 Function Documentation

24.2.1.1 LocatorAppTitleSet

Sets the application title in the locator response packet.

Prototype:

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

Parameters:

pcAppTitle is a pointer to the application title string.

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

Returns:
None.

24.2.1.2 LocatorBoardIDSet

Sets the board ID in the locator response packet.

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

Parameters:
ulID is the ID of the board.

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

Returns:
None.

24.2.1.3 LocatorBoardTypeSet

Sets the board type in the locator response packet.

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

Parameters:
ulType is the type of the board.

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

Returns:
None.

24.2.1.4 LocatorClientIPSet

Sets the client IP address in the locator response packet.

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

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

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

Returns:
None.

24.2.1.5 LocatorInit

Initializes the locator service.

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

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

Returns:
None.

24.2.1.6 LocatorMACAddrSet

Sets the MAC address in the locator response packet.

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

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

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

Returns:
None.

24.2.1.7 LocatorVersionSet

Sets the firmware version in the locator response packet.

Prototype:

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

Parameters:

ulVersion is the version number of the device firmware.

Description:

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

Returns:

None.

24.3 Programming Example

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

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

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

25 lwIP Wrapper Module

25.1 Introduction

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

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

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

25.2 API Functions

Functions

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

25.2.1 Function Documentation

25.2.1.1 lwIPEthernetIntHandler

Handles Ethernet interrupts for the lwIP TCP/IP stack.

Prototype:

```
void
lwIPEthernetIntHandler(void)
```

Description:

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

Returns:

None.

25.2.1.2 lwIPInit

Initializes the lwIP TCP/IP stack.

Prototype:

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

Parameters:

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

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

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

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

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

Description:

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

Returns:

None.

25.2.1.3 lwIPLocalGWAddrGet

Returns the gateway address for this interface.

Prototype:

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

Description:

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

Returns:
the assigned gateway address for this interface.

25.2.1.4 lwIPLocalIPAddrGet

Returns the IP address for this interface.

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

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

Returns:
Returns the assigned IP address for this interface.

25.2.1.5 lwIPLocalMACGet

Returns the local MAC/HW address for this interface.

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

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

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

Returns:
None.

25.2.1.6 lwIPLocalNetMaskGet

Returns the network mask for this interface.

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

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

Returns:
the assigned network mask for this interface.

25.2.1.7 lwIPNetworkConfigChange

Change the configuration of the lwIP network interface.

Prototype:

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

Parameters:

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

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

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

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

Description:

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

Returns:

None.

25.3 Programming Example

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

```
unsigned char pucMACArray[6];

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

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

26 PTPd Wrapper Module

26.1 Introduction

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

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

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

26.2 API Functions

26.3 Programming Example

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

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

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

...

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

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

    ...
}
```

27 Ring Buffer Module

27.1 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.

27.2 API Functions

Functions

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

27.2.1 Function Documentation

27.2.1.1 RingBufAdvanceRead

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

Prototype:

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

Parameters:

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

ulNumBytes is the number of bytes to be removed from the buffer.

Description:

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

Returns:

None.

27.2.1.2 RingBufAdvanceWrite

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

Prototype:

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

Parameters:

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

ulNumBytes is the number of bytes added to the buffer.

Description:

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

Returns:

None.

27.2.1.3 RingBufContigFree

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

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to check.

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

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

27.2.1.4 RingBufContigUsed

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

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

Returns:
Returns the number of contiguous bytes available.

27.2.1.5 RingBufEmpty

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

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

Parameters:
ptRingBuf is the ring buffer object to empty.

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

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

27.2.1.6 RingBufFlush

Empties the ring buffer.

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

Discards all data from the ring buffer.

Returns:

None.

27.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to check.

Description:

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

Returns:

Returns the number of bytes available in the ring buffer.

27.2.1.8 RingBufFull

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

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

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

Returns:

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

27.2.1.9 RingBufInit

Initialize a ring buffer object.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be initialized.

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

ulSize is the size of the buffer in bytes.

Description:

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

Returns:

None.

27.2.1.10 RingBufRead

Reads data from a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be read from.

pucData points to where the data should be stored.

ulLength is the number of bytes to be read.

Description:

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

Returns:

None.

27.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:

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

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

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

Returns:
The byte read from the ring buffer.

27.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

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

27.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

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

27.2.1.14 RingBufWrite

Writes data to a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be written to.

pucData points to the data to be written.

ulLength is the number of bytes to be written.

Description:

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

Returns:

None.

27.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be written to.

ucData is the byte to be written.

Description:

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

Returns:

None.

27.3 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);
```

28 Simple Task Scheduler Module

28.1 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.

28.2 API Functions

Data Structures

- tSchedulerTask

Functions

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

- unsigned long SchedulerTickCountGet (void)

Variables

- tSchedulerTask g_psSchedulerTable[ ]
- unsigned long g_ulSchedulerNumTasks

28.2.1 Data Structure Documentation

28.2.1.1 tSchedulerTask

Definition:

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

Members:

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

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

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

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

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

Description:

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

28.2.2 Function Documentation

28.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

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

Parameters:

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

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

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

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

28.2.2.2 SchedulerElapsedTicksGet

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

Prototype:
```
unsigned long
SchedulerElapsedTicksGet(unsigned long ulTickCount)
```

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

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

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

28.2.2.3 SchedulerInit

Initializes the task scheduler.

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

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

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

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

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

Returns:
None.

28.2.2.4 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.

28.2.2.5 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.

28.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

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

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

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

Returns:
None.

28.2.2.7 SchedulerTaskEnable

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

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

Parameters:
ulIndex is the index of the task which is to be enabled in the global *g_psSchedulerTable* array.
bRunNow is **true** if the task is to be run on the next call to SchedulerRun() or **false** if one whole period is to elapse before the task is run.

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

Returns:
None.

28.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

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

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

Returns:
Tick count since last boot.

28.2.3 Variable Documentation

28.2.3.1 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.

28.2.3.2 g_ulSchedulerNumTasks

Definition:

```
unsigned long g_ulSchedulerNumTasks
```

Description:

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

28.3 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();
    }
}
```

29 Sine Calculation Module

29.1 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.

29.2 API Functions

Defines

- cosine(ulAngle)

Functions

- long sine (unsigned long ulAngle)

29.2.1 Define Documentation

29.2.1.1 cosine

Computes an approximation of the cosine of the input angle.

Definition:

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

Parameters:

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

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

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

29.2.2 Function Documentation

29.2.2.1 sine

Computes an approximation of the sine of the input angle.

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

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

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

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

29.3 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);
}
```

30 Software I2C Module

30.1 Introduction

The software I2C module uses a timer and two GPIO pins to create a slow-speed software I2C peripheral. Multiple software I2C modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software I2C module supports master mode only; multi-master support is not provided. A callback mechanism is used to simulate the interrupts that would be provided by a hardware I2C module.

The API for the software I2C module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware I2C module. The two notable differences are the function prefix being “SoftI2C” instead of “I2CMaster”, and the first argument of each API is a pointer to the tSoftI2C data structure instead of the base address of the hardware module.

Timing for the software I2C module is provided by the application. The SoftI2CTimerTick() function must be called on a periodic basis to provide the timing for the software I2C module. The timer tick function must be called at four times the desired I2C clock rate; for example, to operate the software I2C interface at 10 KHz, the tick function must be called at a 40 KHz rate. By having the application providing the timing, the timer resource can be flexible and multiple software I2C modules can be driven from a single timer resource. Alternatively, if the software I2C module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire I2C transaction has completed (maximizing both I2C clock speed and processor usage, but not requiring a timer).

The software I2C module requires two GPIO pins; one for SCL and one for SDA. The per-instance data structure is approximately 20 bytes in length (the actual length depends on how the structure is packed by the compiler).

As a point of reference, the following are some rough measurements of the processor usage of the software I2C module at various I2C clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software I2C module, processor clock speed, interrupt priority, and compiler.

I2C Clock	% Of Processor	Million Cycles Per Second
5 KHz	4.53	2.26
10 KHz	9.05	4.52
15 KHz	13.53	6.76
20 KHz	18.03	9.01
25 KHz	22.51	11.25
30 KHz	27.05	13.52
35 KHz	31.52	15.76
40 KHz	36.06	18.03
45 KHz	40.54	20.27
50 KHz	44.96	22.48

This module is contained in `utils/softi2c.c`, with `utils/softi2c.h` containing the API definitions for use by applications.

30.2 API Functions

Data Structures

- tSoftI2C

Functions

- tBoolean SoftI2CBusy (tSoftI2C ∗pI2C)
- void SoftI2CCallbackSet (tSoftI2C ∗pI2C, void (∗pfnCallback)(void))
- void SoftI2CControl (tSoftI2C ∗pI2C, unsigned long ulCmd)
- unsigned long SoftI2CDataGet (tSoftI2C ∗pI2C)
- void SoftI2CDataPut (tSoftI2C ∗pI2C, unsigned char ucData)
- unsigned long SoftI2CErr (tSoftI2C ∗pI2C)
- void SoftI2CInit (tSoftI2C ∗pI2C)
- void SoftI2CIntClear (tSoftI2C ∗pI2C)
- void SoftI2CIntDisable (tSoftI2C ∗pI2C)
- void SoftI2CIntEnable (tSoftI2C ∗pI2C)
- tBoolean SoftI2CIntStatus (tSoftI2C ∗pI2C, tBoolean bMasked)
- void SoftI2CSCLGPIOSet (tSoftI2C ∗pI2C, unsigned long ulBase, unsigned char ucPin)
- void SoftI2CSDAGPIOSet (tSoftI2C ∗pI2C, unsigned long ulBase, unsigned char ucPin)
- void SoftI2CSlaveAddrSet (tSoftI2C ∗pI2C, unsigned char ucSlaveAddr, tBoolean bReceive)
- void SoftI2CTimerTick (tSoftI2C ∗pI2C)

30.2.1 Data Structure Documentation

30.2.1.1 tSoftI2C

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulSCLGPIO;
    unsigned long ulSDAGPIO;
    unsigned char ucFlags;
    unsigned char ucSlaveAddr;
    unsigned char ucData;
    unsigned char ucState;
    unsigned char ucCurrentBit;
    unsigned char ucIntMask;
    unsigned char ucIntStatus;
}
tSoftI2C
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware I2C implementation. This address can be set via a direct structure access or using the SoftI2CCallbackSet function.

ulSCLGPIO The address of the GPIO pin to be used for the SCL signal. This member can be set via a direct structure access or using the SoftI2CSCLGPIOSet function.

ulSDAGPIO The address of the GPIO pin to be used for the SDA signal. This member can be set via a direct structure access or using the SoftI2CSDAGPIOSet function.

ucFlags The flags that control the operation of the SoftI2C module. This member should not be accessed or modified by the application.

ucSlaveAddr The slave address that is currently being accessed. This member should not be accessed or modified by the application.

ucData The data that is currently being transmitted or received. This member should not be accessed or modified by the application.

ucState The current state of the SoftI2C state machine. This member should not be accessed or modified by the application.

ucCurrentBit The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.

ucIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

ucIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

Description:

This structure contains the state of a single instance of a SoftI2C module.

30.2.2 Function Documentation

30.2.2.1 SoftI2CBusy

Indicates whether or not the SoftI2C module is busy.

Prototype:

```
tBoolean
SoftI2CBusy(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function returns an indication of whether or not the SoftI2C module is busy transmitting or receiving data.

Returns:

Returns **true** if the SoftI2C module is busy; otherwise, returns **false**.

30.2.2.2 SoftI2CCallbackSet

Sets the callback used by the SoftI2C module.

Prototype:

```
void
SoftI2CCallbackSet(tSoftI2C *pI2C,
                   void (*pfnCallback)(void))
```

Parameters:

pI2C specifies the SoftI2C data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftI2C module.

Returns:

None.

30.2.2.3 SoftI2CControl

Controls the state of the SoftI2C module.

Prototype:

```
void
SoftI2CControl(tSoftI2C *pI2C,
               unsigned long ulCmd)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulCmd command to be issued to the SoftI2C module.

Description:

This function is used to control the state of the SoftI2C module send and receive operations. The *ucCmd* parameter can be one of the following values:

- **SOFTI2C_CMD_SINGLE_SEND**
- **SOFTI2C_CMD_SINGLE_RECEIVE**
- **SOFTI2C_CMD_BURST_SEND_START**
- **SOFTI2C_CMD_BURST_SEND_CONT**
- **SOFTI2C_CMD_BURST_SEND_FINISH**
- **SOFTI2C_CMD_BURST_SEND_ERROR_STOP**
- **SOFTI2C_CMD_BURST_RECEIVE_START**
- **SOFTI2C_CMD_BURST_RECEIVE_CONT**
- **SOFTI2C_CMD_BURST_RECEIVE_FINISH**
- **SOFTI2C_CMD_BURST_RECEIVE_ERROR_STOP**

Returns:

None.

30.2.2.4 SoftI2CDataGet

Receives a byte that has been sent to the SoftI2C module.

Prototype:

```
unsigned long
SoftI2CDataGet(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function reads a byte of data from the SoftI2C module that was received as a result of an appropriate call to SoftI2CControl().

Returns:

Returns the byte received by the SoftI2C module, cast as an unsigned long.

30.2.2.5 SoftI2CDataPut

Transmits a byte from the SoftI2C module.

Prototype:

```
void
SoftI2CDataPut(tSoftI2C *pI2C,
               unsigned char ucData)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ucData data to be transmitted from the SoftI2C module.

Description:

This function places the supplied data into SoftI2C module in preparation for being transmitted via an appropriate call to SoftI2CControl().

Returns:

None.

30.2.2.6 SoftI2CErr

Gets the error status of the SoftI2C module.

Prototype:

```
unsigned long
SoftI2CErr(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function is used to obtain the error status of the SoftI2C module send and receive operations.

Returns:
Returns the error status, as one of **SOFTI2C_ERR_NONE**, **SOFTI2C_ERR_ADDR_ACK**, or **SOFTI2C_ERR_DATA_ACK**.

30.2.2.7 SoftI2CInit

Initializes the SoftI2C module.

Prototype:
```
void
SoftI2CInit(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
This function initializes operation of the SoftI2C module. After successful initialization of the SoftI2C module, the software I2C bus is in the idle state.

Returns:
None.

30.2.2.8 SoftI2CIntClear

Clears the SoftI2C “interrupt”.

Prototype:
```
void
SoftI2CIntClear(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
The SoftI2C “interrupt” source is cleared, so that it no longer asserts. This function must be called in the “interrupt” handler to keep it from being called again immediately on exit.

Returns:
None.

30.2.2.9 SoftI2CIntDisable

Disables the SoftI2C “interrupt”.

Prototype:
```
void
SoftI2CIntDisable(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
Disables the SoftI2C "interrupt" source.

Returns:
None.

30.2.2.10 SoftI2CIntEnable

Enables the SoftI2C "interrupt".

Prototype:
```
void
SoftI2CIntEnable(tSoftI2C *pI2C)
```

Parameters:
pI2C specifies the SoftI2C data structure.

Description:
Enables the SoftI2C "interrupt" source.

Returns:
None.

30.2.2.11 SoftI2CIntStatus

Gets the current SoftI2C "interrupt" status.

Prototype:
```
tBoolean
SoftI2CIntStatus(tSoftI2C *pI2C,
                 tBoolean bMasked)
```

Parameters:
pI2C specifies the SoftI2C data structure.
bMasked is **false** if the raw "interrupt" status is requested and **true** if the masked "interrupt" status is requested.

Description:
This returns the "interrupt" status for the SoftI2C module. Either the raw "interrupt" status or the status of "interrupts" that are allowed to reflect to the processor can be returned.

Returns:
The current interrupt status, returned as **true** if active or **false** if not active.

30.2.2.12 SoftI2CSCLGPIOSet

Sets the GPIO pin to be used as the SoftI2C SCL signal.

Prototype:

```
void
SoftI2CSCLGPIOSet(tSoftI2C *pI2C,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftI2C SCL signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

30.2.2.13 SoftI2CSDAGPIOSet

Sets the GPIO pin to be used as the SoftI2C SDA signal.

Prototype:

```
void
SoftI2CSDAGPIOSet(tSoftI2C *pI2C,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftI2C SDA signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

30.2.2.14 SoftI2CSlaveAddrSet

Sets the address that the SoftI2C module places on the bus.

Prototype:

```
void
SoftI2CSlaveAddrSet(tSoftI2C *pI2C,
```

```
unsigned char ucSlaveAddr,
tBoolean bReceive)
```

Parameters:

pI2C specifies the SoftI2C data structure.

ucSlaveAddr 7-bit slave address

bReceive flag indicating the type of communication with the slave.

Description:

This function sets the address that the SoftI2C module places on the bus when initiating a transaction. When the *bReceive* parameter is set to **true**, the address indicates that the SoftI2C moudle is initiating a read from the slave; otherwise the address indicates that the SoftI2C module is initiating a write to the slave.

Returns:

None.

30.2.2.15 SoftI2CTimerTick

Performs the periodic update of the SoftI2C module.

Prototype:

```
void
SoftI2CTimerTick(tSoftI2C *pI2C)
```

Parameters:

pI2C specifies the SoftI2C data structure.

Description:

This function performs the periodic, time-based updates to the SoftI2C module. The transmission and reception of data over the SoftI2C link is performed by the state machine in this function.

This function must be called at four times the desired SoftI2C clock rate. For example, to run the SoftI2C clock at 10 KHz, this function must be called at a 40 KHz rate.

Returns:

None.

30.3 Programming Example

The following example shows how to configure the software I2C module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software I2C.
//
tSoftI2C g_sI2C;

//
// The timer tick function.
```

```
//
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software I2C timer tick function.
    //
    SoftI2CTimerTick(&g_sI2C);
}

//
// The callback function for the software I2C.  This function is equivalent
// to the interrupt handler for a hardware I2C.
//
void
I2CCallback(void)
{
    //
    // Clear the interrupt.
    //
    SoftI2CIntClear(&g_sI2C);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the software I2C and send some data.
//
void
TestSoftI2C(void)
{
    //
    // Clear the software I2C instance data.
    //
    memset(&g_sI2C, 0, sizeof(g_sI2C));

    //
    // Set the callback function used for this software I2C.
    //
    SoftI2CCallbackSet(&g_sI2C, I2CCallback);

    //
    // Configure the pins used for the software I2C.  This example uses
    // pins PD0 and PE1.
    //
    SoftI2CSCLGPIOSet(&g_sI2C, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftI2CSDAGPIOSet(&g_sI2C, GPIO_PORTE_BASE, GPIO_PIN_1);

    //
    // Enable the GPIO modules that contains the GPIO pins to be used by
    // the software I2C.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

    //
    // Initalize the software I2C module.
    //
```

```
    SoftI2CInit(&g_sI2C);

    //
    // Configure the timer used to generate the timing for the software
    // I2C.  The interface will be run at 10 KHz, requiring a timer tick
    // at 40 KHz.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 40000);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntEnable(INT_TIMER0A);
    TimerEnable(TIMER0_BASE, TIMER_A);

    //
    // Enable the software I2C interrupt.
    //
    SoftI2CIntEnable(&g_sI2C);

    //
    // Send a single byte to the slave device.
    //
    SoftI2CSlaveAddrSet(&g_sI2C, 0x55, 0);
    SoftI2CDataPut(&g_sI2C, 0xaa);
    SoftI2CControl(&g_sI2C, SOFTI2C_CMD_SINGLE_SEND);

    //
    // Wait until the software I2C is idle.  The completion interrupt will
    // be sent to the callback function prior to exiting this loop.
    //
    while(SoftI2CBusy(&g_sI2C))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware I2C module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware I2C.
//
void
I2C0IntHandler(void)
{
    //
    // Clear the asserted interrupt sources.
    //
    I2CMasterIntClear(I2C0_MASTER_BASE);

    //
    // Handle the interrupt.
    //
    ...
}

//
// Setup the hardware I2C and send some data.
//
void
TestI2C(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the I2C, as well as the I2C module.
    //
```

```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_I2C0);

    //
    // Configure the GPIO pins for use by the I2C module.
    //
    GPIOPinTypeI2C(GPIO_PORTB_BASE, GPIO_PIN_2 | GPIO_PIN_3);

    //
    // Initalize the hardware I2C module.
    //
    I2CMasterInitExpClk(I2C0_MASTER_BASE, SysCtlClockGet(), false);

    //
    // Enable the hardware I2C.
    //
    I2CMasterEnable(I2C0_MASTER_BASE);

    //
    // Enable the interrupt in the hardware I2C.
    //
    I2CMasterIntEnable(I2C0_MASTER_BASE);
    IntEnable(INT_I2C0);

    //
    // Write some data into the hardware I2C transmit FIFO.
    //
    I2CMasterSlaveAddrSet(I2C0_MASTER_BASE, 0x55, 0);
    I2CMasterDataPut(I2C0_MASTER_BASE, 0xaa);
    I2CMasterControl(I2C0_MASTER_BASE, I2C_MASTER_CMD_SINGLE_SEND);

    //
    // Wait until the hardware I2C is idle.  The interrupt will be sent to
    // the interrupt handler prior to exiting this loop.
    //
    while(I2CBusy(I2C0_MASTER_BASE))
    {
    }
}
```

31 Software SSI Module

31.1 Introduction

The software SSI module uses a timer and a few GPIO pins to create a slow-speed software SSI peripheral. Multiple software SSI modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers if unique clock rates are required, and processor cycles to execute the code). The software SSI module supports the Motorola® SPI™ formats with 4 to 16 data bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware SSI module.

The API for the software SSI module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware SSI module. The two notable difference are the function prefix being “SoftSSI” instead of “SSI”, and the first argument of each API is a pointer to the tSoftSSI data structure instead of the base address of the hardware module.

Timing for the software SSI module is provided by the application. The SoftSSITimerTick() function must be called on a periodic basis to provide the timing for the software SSI module. The timer tick function must be called at twice the desired SSI clock rate; for example, to operate the software SSI interface at 10 KHz, the tick function must be called at a 20 KHz rate. By having the application providing the timing, the timer resource to be used is flexible and multiple software SSI modules can be driven from a single timer resource. Alternatively, if the software SSI module is only needed for brief periods of time and processor usage is not a concern, the timer tick function can simply be called in a loop until the entire SSI transaction has completed (maximizing both SSI clock speed and processor usage, but not requiring a timer).

The software SSI module requires a few as two and as many as four GPIO pins. The following table shows the possible pin usages for the software SSI module:

Fss	Clk	Tx	Rx	Pins	Description
	yes	yes		2	transmit only
yes	yes	yes		3	
	yes		yes	2	receive only
yes	yes		yes	3	
	yes	yes	yes	3	transmit and receive
yes	yes	yes	yes	4	

For the cases where Fss is not used, it is up to the application to control that signal (either via a separately-controlled GPIO, or by being tied to ground in the hardware).

The per-instance data structure is approximately 52 bytes in length (the actual length will depend upon how the structure is packed by the compiler in use).

As a point of reference, the following are some rough measurements of the processor usage of the software SSI module at various SSI clock speeds with the processor running at 50 MHz. Actual processor usage may vary, depending upon how the application uses the software SSI module, processor clock speed, interrupt priority, and compiler in use.

SSI Clock	% Of Processor	Million Cycles Per Second
10 KHz	5.26	2.63
20 KHz	10.48	5.24
30 KHz	15.68	7.84
40 KHz	20.90	10.45
50 KHz	26.10	13.05
60 KHz	31.38	15.69
70 KHz	36.54	18.27
80 KHz	41.79	20.89
90 KHz	47.06	23.53
100 KHz	52.17	26.08

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

31.2 API Functions

Data Structures

- tSoftSSI

Functions

- tBoolean SoftSSIBusy (tSoftSSI ∗pSSI)
- void SoftSSICallbackSet (tSoftSSI ∗pSSI, void (∗pfnCallback)(void))
- void SoftSSIClkGPIOSet (tSoftSSI ∗pSSI, unsigned long ulBase, unsigned char ucPin)
- void SoftSSIConfigSet (tSoftSSI ∗pSSI, unsigned char ucProtocol, unsigned char ucBits)
- tBoolean SoftSSIDataAvail (tSoftSSI ∗pSSI)
- void SoftSSIDataGet (tSoftSSI ∗pSSI, unsigned long ∗pulData)
- long SoftSSIDataGetNonBlocking (tSoftSSI ∗pSSI, unsigned long ∗pulData)
- void SoftSSIDataPut (tSoftSSI ∗pSSI, unsigned long ulData)
- long SoftSSIDataPutNonBlocking (tSoftSSI ∗pSSI, unsigned long ulData)
- void SoftSSIDisable (tSoftSSI ∗pSSI)
- void SoftSSIEnable (tSoftSSI ∗pSSI)
- void SoftSSIFssGPIOSet (tSoftSSI ∗pSSI, unsigned long ulBase, unsigned char ucPin)
- void SoftSSIIntClear (tSoftSSI ∗pSSI, unsigned long ulIntFlags)
- void SoftSSIIntDisable (tSoftSSI ∗pSSI, unsigned long ulIntFlags)
- void SoftSSIIntEnable (tSoftSSI ∗pSSI, unsigned long ulIntFlags)
- unsigned long SoftSSIIntStatus (tSoftSSI ∗pSSI, tBoolean bMasked)
- void SoftSSIRxBufferSet (tSoftSSI ∗pSSI, unsigned short ∗pusRxBuffer, unsigned short usLen)
- void SoftSSIRxGPIOSet (tSoftSSI ∗pSSI, unsigned long ulBase, unsigned char ucPin)
- tBoolean SoftSSISpaceAvail (tSoftSSI ∗pSSI)
- void SoftSSITimerTick (tSoftSSI ∗pSSI)
- void SoftSSITxBufferSet (tSoftSSI ∗pSSI, unsigned short ∗pusTxBuffer, unsigned short usLen)
- void SoftSSITxGPIOSet (tSoftSSI ∗pSSI, unsigned long ulBase, unsigned char ucPin)

31.2.1 Data Structure Documentation

31.2.1.1 tSoftSSI

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulFssGPIO;
    unsigned long ulClkGPIO;
    unsigned long ulTxGPIO;
    unsigned long ulRxGPIO;
    unsigned short *pusTxBuffer;
    unsigned short *pusRxBuffer;
    unsigned short usTxBufferLen;
    unsigned short usTxBufferRead;
    unsigned short usTxBufferWrite;
    unsigned short usRxBufferLen;
    unsigned short usRxBufferRead;
    unsigned short usRxBufferWrite;
    unsigned short usTxData;
    unsigned short usRxData;
    unsigned char ucFlags;
    unsigned char ucBits;
    unsigned char ucState;
    unsigned char ucCurrentBit;
    unsigned char ucIntMask;
    unsigned char ucIntStatus;
    unsigned char ucIdleCount;
}
tSoftSSI
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware SSI implementation. This address can be set via a direct structure access or using the SoftSSICallbackSet function.

ulFssGPIO The address of the GPIO pin to be used for the Fss signal. If this member is zero, the Fss signal is not generated. This member can be set via a direct structure access or using the SoftSSIFssGPIOSet function.

ulClkGPIO The address of the GPIO pin to be used for the Clk signal. This member can be set via a direct structure access or using the SoftSSIClkGPIOSet function.

ulTxGPIO The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the SoftSSITxGPIOSet function.

ulRxGPIO The address of the GPIO pin to be used for the Rx signal. If this member is zero, the Rx signal is not read. This member can be set via a direct structure access or using the SoftSSIRxGPIOSet function.

pusTxBuffer The address of the data buffer used for the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

pusRxBuffer The address of the data buffer used for the receive FIFO. This member can be set via a direct structure access or using the SoftSSIRxBufferSet function.

usTxBufferLen The length of the transmit FIFO. This member can be set via a direct structure access or using the SoftSSITxBufferSet function.

usTxBufferRead The index into the transmit FIFO of the next word to be transmitted. This member should be initialized to zero, but should not be accessed or modified by the application.

usTxBufferWrite The index into the transmit FIFO of the next location to store data into the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.

usRxBufferLen The length of the receive FIFO. This member can be set via a direct structure access or using the SoftSSIRxBufferSet function.

usRxBufferRead The index into the receive FIFO of the next word to be read from the FIFO. This member should be initialized to zero, but should not be accessed or modified by the application.

usRxBufferWrite The index into the receive FIFO of the location to store the next word received. This member should be initialized to zero, but should not be accessed or modified by the application.

usTxData The word that is currently being transmitted. This member should not be accessed or modified by the application.

usRxData The word that is currently being received. This member should not be accessed or modified by the application.

ucFlags The flags that control the operation of the SoftSSI module. This member should not be accessed or modified by the application.

ucBits The number of data bits in each SoftSSI frame, which also specifies the width of each data item in the transmit and receive FIFOs. This member can be set via a direct structure access or using the SoftSSIConfigSet function.

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

ucCurrentBit The number of bits that have been transmitted and received in the current frame. This member should not be accessed or modified by the application.

ucIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

ucIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

ucIdleCount The number of tick counts that the SoftSSI module has been idle with data stored in the receive FIFO, which is used to generate the receive timeout interrupt. This member should not be accessed or modified by the application.

Description:

This structure contains the state of a single instance of a SoftSSI module.

31.2.2 Function Documentation

31.2.2.1 SoftSSIBusy

Determines whether the SoftSSI transmitter is busy or not.

Prototype:

```
tBoolean
SoftSSIBusy(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

Allows the caller to determine whether all transmitted bytes have cleared the transmitter. If **false** is returned, then the transmit FIFO is empty and all bits of the last transmitted word have left the shift register.

Returns:

Returns **true** if the SoftSSI is transmitting or **false** if all transmissions are complete.

31.2.2.2 SoftSSICallbackSet

Sets the callback used by the SoftSSI module.

Prototype:

```
void
SoftSSICallbackSet(tSoftSSI *pSSI,
                   void (*pfnCallback)(void))
```

Parameters:

pSSI specifies the SoftSSI data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an "interrupt" produced by the SoftSSI module.

Returns:

None.

31.2.2.3 SoftSSIClkGPIOSet

Sets the GPIO pin to be used as the SoftSSI Clk signal.

Prototype:

```
void
SoftSSIClkGPIOSet(tSoftSSI *pSSI,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Clk signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

31.2.2.4 SoftSSIConfigSet

Sets the configuration of a SoftSSI module.

Prototype:

```
void
SoftSSIConfigSet(tSoftSSI *pSSI,
                 unsigned char ucProtocol,
                 unsigned char ucBits)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ucProtocol specifes the data transfer protocol.

ucBits specifies the number of bits transferred per frame.

Description:

This function configures the data format of a SoftSSI module. The *ucProtocol* parameter can be one of the following values: **SOFTSSI_FRF_MOTO_MODE_0**, **SOFTSSI_FRF_MOTO_MODE_1**, **SOFTSSI_FRF_MOTO_MODE_2**, or **SOFTSSI_FRF_MOTO_MODE_3**. These frame formats imply the following polarity and phase configurations:

```
Polarity Phase         Mode
  0       0    SOFTSSI_FRF_MOTO_MODE_0
  0       1    SOFTSSI_FRF_MOTO_MODE_1
  1       0    SOFTSSI_FRF_MOTO_MODE_2
  1       1    SOFTSSI_FRF_MOTO_MODE_3
```

The *ucBits* parameter defines the width of the data transfers, and can be a value between 4 and 16, inclusive.

Returns:

None.

31.2.2.5 SoftSSIDataAvail

Determines if there is any data in the receive FIFO.

Prototype:

```
tBoolean
SoftSSIDataAvail(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function determines if there is any data available to be read from the receive FIFO.

Returns:

Returns **true** if there is data in the receive FIFO or **false** if there is no data in the receive FIFO.

31.2.2.6 SoftSSIDataGet

Gets a data element from the SoftSSI receive FIFO.

Prototype:

```
void
SoftSSIDataGet(tSoftSSI *pSSI,
               unsigned long *pulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pulData is a pointer to a storage location for data that was received over the SoftSSI interface.

Description:

This function gets received data from the receive FIFO of the specified SoftSSI module and places that data into the location specified by the *pulData* parameter.

Note:

Only the lower N bits of the value written to *pulData* contain valid data, where N is the data width as configured by SoftSSIConfigSet(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pulData* contain valid data.

Returns:

None.

31.2.2.7 SoftSSIDataGetNonBlocking

Gets a data element from the SoftSSI receive FIFO.

Prototype:

```
long
SoftSSIDataGetNonBlocking(tSoftSSI *pSSI,
                          unsigned long *pulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pulData is a pointer to a storage location for data that was received over the SoftSSI interface.

Description:

This function gets received data from the receive FIFO of the specified SoftSSI module and places that data into the location specified by the *ulData* parameter. If there is no data in the FIFO, then this function returns a zero.

Note:

Only the lower N bits of the value written to *pulData* contain valid data, where N is the data width as configured by SoftSSIConfigSet(). For example, if the interface is configured for 8-bit data width, only the lower 8 bits of the value written to *pulData* contain valid data.

Returns:

Returns the number of elements read from the SoftSSI receive FIFO.

31.2.2.8 SoftSSIDataPut

Puts a data element into the SoftSSI transmit FIFO.

Prototype:

```
void
SoftSSIDataPut(tSoftSSI *pSSI,
               unsigned long ulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulData is the data to be transmitted over the SoftSSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SoftSSI module.

Note:

The upper 32 - N bits of the *ulData* are discarded, where N is the data width as configured by SoftSSIConfigSet(). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ulData* are discarded.

Returns:

None.

31.2.2.9 SoftSSIDataPutNonBlocking

Puts a data element into the SoftSSI transmit FIFO.

Prototype:

```
long
SoftSSIDataPutNonBlocking(tSoftSSI *pSSI,
                          unsigned long ulData)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulData is the data to be transmitted over the SoftSSI interface.

Description:

This function places the supplied data into the transmit FIFO of the specified SoftSSI module. If there is no space in the FIFO, then this function returns a zero.

Note:

The upper 32 - N bits of the *ulData* are discarded, where N is the data width as configured by SoftSSIConfigSet(). For example, if the interface is configured for 8-bit data width, the upper 24 bits of *ulData* are discarded.

Returns:

Returns the number of elements written to the SSI transmit FIFO.

31.2.2.10 SoftSSIDisable

Disables the SoftSSI module.

Prototype:

```
void
SoftSSIDisable(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function disables operation of the SoftSSI module. If a data transfer is in progress, it is finished before the module is fully disabled.

Returns:

None.

31.2.2.11 SoftSSIEnable

Enables the SoftSSI module.

Prototype:

```
void
SoftSSIEnable(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function enables operation of the SoftSSI module. The SoftSSI module must be configured before it is enabled.

Returns:

None.

31.2.2.12 SoftSSIFssGPIOSet

Sets the GPIO pin to be used as the SoftSSI Fss signal.

Prototype:

```
void
SoftSSIFssGPIOSet(tSoftSSI *pSSI,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:
This function sets the GPIO pin that is used for the SoftSSI Fss signal. If there is not a GPIO pin allocated for Fss, the SoftSSI module does not assert/deassert the Fss signal, leaving it to the application either to do manually or to not do at all if the slave device has Fss tied to ground.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:
None.

31.2.2.13 SoftSSIIntClear

Clears SoftSSI "interrupt" sources.

Prototype:
```
void
SoftSSIIntClear(tSoftSSI *pSSI,
                unsigned long ulIntFlags)
```

Parameters:
pSSI specifies the SoftSSI data structure.
ulIntFlags is a bit mask of the "interrupt" sources to be cleared.

Description:
The specified SoftSSI "interrupt" sources are cleared so that they no longer assert. This function must be called in the "interrupt" handler to keep the "interrupt" from being recognized again immediately upon exit. The *ulIntFlags* parameter is the logical OR of any of the **SOFTSSI_TXEOT**, **SOFTSSI_RXTO**, and **SOFTSSI_RXOR** values.

Returns:
None.

31.2.2.14 SoftSSIIntDisable

Disables individual SoftSSI "interrupt" sources.

Prototype:
```
void
SoftSSIIntDisable(tSoftSSI *pSSI,
                  unsigned long ulIntFlags)
```

Parameters:
pSSI specifies the SoftSSI data structure.
ulIntFlags is a bit mask of the "interrupt" sources to be disabled.

Description:
Disables the indicated SoftSSI "interrupt" sources. The *ulIntFlags* parameter can be any of the **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, or **SOFTSSI_RXOR** values.

Returns:
None.

31.2.2.15 SoftSSIIntEnable

Enables individual SoftSSI "interrupt" sources.

Prototype:
```
void
SoftSSIIntEnable(tSoftSSI *pSSI,
                 unsigned long ulIntFlags)
```

Parameters:
pSSI specifies the SoftSSI data structure.
ulIntFlags is a bit mask of the "interrupt" sources to be enabled.

Description:
Enables the indicated SoftSSI "interrupt" sources. Only the sources that are enabled can be reflected to the callback function; disabled sources do not result in a callback. The *ulIntFlags* parameter can be any of the **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, or **SOFTSSI_RXOR** values.

Returns:
None.

31.2.2.16 SoftSSIIntStatus

Gets the current "interrupt" status.

Prototype:
```
unsigned long
SoftSSIIntStatus(tSoftSSI *pSSI,
                 tBoolean bMasked)
```

Parameters:
pSSI specifies the SoftSSI data structure.
bMasked is **false** if the raw "interrupt" status is required or **true** if the masked "interrupt" status is required.

Description:
This function returns the "interrupt" status for the SoftSSI module. Either the raw "interrupt" status or the status of "interrupts" that are allowed to reflect to the callback can be returned.

Returns:
The current "interrupt" status, enumerated as a bit field of **SOFTSSI_TXEOT**, **SOFTSSI_TXFF**, **SOFTSSI_RXFF**, **SOFTSSI_RXTO**, and **SOFTSSI_RXOR**.

31.2.2.17 SoftSSIRxBufferSet

Sets the receive FIFO buffer for a SoftSSI module.

Prototype:

```
void
SoftSSIRxBufferSet(tSoftSSI *pSSI,
                   unsigned short *pusRxBuffer,
                   unsigned short usLen)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pusRxBuffer is the address of the receive FIFO buffer.

usLen is the size, in 16-bit half-words, of the receive FIFO buffer.

Description:

This function sets the address and size of the receive FIFO buffer and also resets the read and write pointers, marking the receive FIFO as empty. When the buffer pointer and length are configured as zero, all data received from the slave device is discarded. This capability is useful when there is no GPIO pin allocated for the Rx signal.

Returns:

None.

31.2.2.18 SoftSSIRxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Rx signal.

Prototype:

```
void
SoftSSIRxGPIOSet(tSoftSSI *pSSI,
                 unsigned long ulBase,
                 unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Rx signal. If there is not a GPIO pin allocated for Rx, the SoftSSI module does not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

31.2.2.19 SoftSSISpaceAvail

Determines if there is any space in the transmit FIFO.

Prototype:

```
tBoolean
SoftSSISpaceAvail(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function determines if there is space available in the transmit FIFO.

Returns:

Returns **true** if there is space available in the transmit FIFO or **false** if there is no space available in the transmit FIFO.

31.2.2.20 SoftSSITimerTick

Performs the periodic update of the SoftSSI module.

Prototype:

```
void
SoftSSITimerTick(tSoftSSI *pSSI)
```

Parameters:

pSSI specifies the SoftSSI data structure.

Description:

This function performs the periodic, time-based updates to the SoftSSI module. The transmission and reception of data over the SoftSSI link is performed by the state machine in this function.

This function must be called at twice the desired SoftSSI clock rate. For example, to run the SoftSSI clock at 10 KHz, this function must be called at a 20 KHz rate.

Returns:

None.

31.2.2.21 SoftSSITxBufferSet

Sets the transmit FIFO buffer for a SoftSSI module.

Prototype:

```
void
SoftSSITxBufferSet(tSoftSSI *pSSI,
                   unsigned short *pusTxBuffer,
                   unsigned short usLen)
```

Parameters:

pSSI specifies the SoftSSI data structure.

pusTxBuffer is the address of the transmit FIFO buffer.

usLen is the size, in 16-bit half-words, of the transmit FIFO buffer.

Description:

This function sets the address and size of the transmit FIFO buffer and also resets the read and write pointers, marking the transmit FIFO as empty.

Returns:

None.

31.2.2.22 SoftSSITxGPIOSet

Sets the GPIO pin to be used as the SoftSSI Tx signal.

Prototype:

```
void
SoftSSITxGPIOSet(tSoftSSI *pSSI,
                 unsigned long ulBase,
                 unsigned char ucPin)
```

Parameters:

pSSI specifies the SoftSSI data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used for the SoftSSI Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

31.3 Programming Example

The following example shows how to configure the software SSI module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software SSI.
//
tSoftSSI g_sSSI;

//
// The buffer used to hold the transmit data.
//
unsigned short g_pusTxBuffer[8];

//
// The timer tick function.
//
```

```
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software SSI timer tick function.
    //
    SoftSSITimerTick(&g_sSSI);
}

//
// The callback function for the software SSI.  This function is equivalent
// to the interrupt handler for a hardware SSI.
//
void
SSICallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftSSIIntStatus(&g_sSSI, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftSSIIntClear(&g_sSSI, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software SSI and send some data.
//
void
TestSoftSSI(void)
{
    //
    // Clear the software SSI instance data.
    //
    memset(&g_sSSI, 0, sizeof(g_sSSI));

    //
    // Set the callback function used for this software SSI.
    //
    SoftSSICallbackSet(&g_sSSI, SSICallback);

    //
    // Configure the pins used for the software SSI.  This example uses
    // pins PD0, PE1, and PF2.
    //
    SoftSSIFssGPIOSet(&g_sSSI, GPIO_PORTD_BASE, GPIO_PIN_0);
    SoftSSIClkGPIOSet(&g_sSSI, GPIO_PORTE_BASE, GPIO_PIN_1);
    SoftSSITxGPIOSet(&g_sSSI, GPIO_PORTF_BASE, GPIO_PIN_2);

    //
    // Configure the data buffer used as the transmit FIFO.
    //
```

```
    SoftSSITxBufferSet(&g_sSSI, g_pusTxBuffer, 8);

    //
    // Enable the GPIO modules that contains the GPIO pins to be used by
    // the software SSI.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);

    //
    // Initalize the software SSI module, using mode 3 and 8 data bits.
    //
    SoftSSIConfigSet(&g_sSSI, SOFTSSI_FRF_MOTO_MODE_3, 8);

    //
    // Enable the software SSI.
    //
    SoftSSIEnable(&g_sSSI);

    //
    // Configure the timer used to generate the timing for the software
    // SSI.  The interface will be run at 10 KHz, requiring a timer tick
    // at 20 KHz.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    TimerConfigure(TIMER0_BASE, TIMER_CFG_32_BIT_PER);
    TimerLoadSet(TIMER0_BASE, TIMER_A, SysCtlClockGet() / 20000);
    TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
    IntEnable(INT_TIMER0A);
    TimerEnable(TIMER0_BASE, TIMER_A);

    //
    // Enable the transmit FIFO half full interrupt in the software SSI.
    //
    SoftSSIIntEnable(&g_sSSI, SOFTSSI_TXFF);

    //
    // Write some data into the software SSI transmit FIFO.
    //
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);
    SoftSSIDataPut(&g_sSSI, 0x55);
    SoftSSIDataPut(&g_sSSI, 0xaa);

    //
    // Wait until the software SSI is idle.  The transmit FIFO half full
    // interrupt will be sent to the callback function prior to exiting
    // this loop.
    //
    while(SoftSSIBusy(&g_sSSI))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware SSI module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware SSI.
//
void
SSI0IntHandler(void)
```

```
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SSIIntStatus(SSI0_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    SSIIntClear(SSI0_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware SSI and send some data.
//
void
TestSSI(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the SSI, as well as the SSI module.
    //
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_SSI0);

    //
    // Configure the GPIO pins for use by the SSI module.
    //
    GPIOPinTypeSSI(GPIO_PORTA_BASE, (GPIO_PIN_2 | GPIO_PIN_3 |
                                     GPIO_PIN_4 | GPIO_PIN_5));

    //
    // Initalize the hardware SSI module, using mode 3 and 8 data bits.
    //
    SSIConfigSetExpClk(SSI0_BASE, SysCtlClockGet(), SSI_FRF_MOTO_MODE_3,
                       SSI_MODE_MASTER, 10000, 8);

    //
    // Enable the hardware SSI.
    //
    SSIEnable(SSI0_BASE);

    //
    // Enable the transmit FIFO half full interrupt in the hardware SSI.
    //
    SSIIntEnable(SSI0_BASE, SSI_TXFF);
    IntEnable(INT_SSI0);

    //
    // Write some data into the hardware SSI transmit FIFO.
    //
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);
    SSIDataPut(SSI0_BASE, 0x55);
    SSIDataPut(SSI0_BASE, 0xaa);

    //
```

```
        // Wait until the hardware SSI is idle.  The transmit FIFO half full
        // interrupt will be sent to the interrupt handler prior to exiting
        // this loop.
        //
        while(SSIBusy(SSI0_BASE))
        {
        }
    }
```

32 Software UART Module

32.1 Introduction

The software UART module uses two timers and a two GPIO pins to create a software UART peripheral. Multiple software UART modules can be in use simultaneously, limited only by the availability of resources (RAM for the per-instance data structure, pins for the interface, timers, and processor cycles to execute the code). The software UART module supports five through eight data bits, a varity of parity modes (odd, even, one, zero, and none), and one or two stop bits. A callback mechanism is used to simulate the interrupts that would be provided by a hardware UART module.

The API for the software UART module has been constructed to be as close as possible to the API provided in the Stellaris Peripheral Driver Library for the hardware UART module. The two notable difference are the function prefix being “SoftUART” instead of “UART”, and the first argument of each API is a pointer to the tSoftUART data structure instead of the base address of the hardware module.

The software UART transmitter and receiver are handled independently (because of the asynchronous nature of the two). As a result, there are separate timers for each, and if only one is required then the other does not need to be utilized.

Timing for the software UART transmitter is provided by the application. The SoftUARTTxTimerTick() function must be called on a periodic basis to provide the timing for the software UART transmitter. The timer tick function must be called at the desired UART baud rate; for example, to operate the software UART transmitter at 38,400 baud, the tick function must be called at a 38,400 Hz rate. Because the application provides the timing, the timer resource can be flexible and multiple software UART transmitters can be driven from a single timer resource.

Timing for the software UART receiver is also provided by the application. Initially, the Rx pin is configured by the software UART module for a GPIO edge interrupt. The GPIO edge interrupt handler must be provided by the application (so that it can be shared with other possible GPIO interrupts on that port). When the interrupt occurs, a timer must be started at the desired baud rate (i.e. for 38,400 baud, it must run at 38,400 Hz) and the SoftUARTRxTick() function must be called. Then, whenever the timer interrupt occurs, the SoftUARTRxTick() function must be called. The timer is disabled whenver SoftUARTRxTick() indicates that it is no longer needed. Because the application provides the timing, the timer resource can beflexible. However, each software UART receiver must have its own timer resource.

The software UART module requires one or two GPIO pins. The following table shows the possible pin usages for the software UART module:

Tx	Rx	Pins	Description
yes		1	transmit only
	yes	1	receive only
yes	yes	2	transmit and receive

The per-instance data structure is approximately 52 bytes in length (the actual length depends on how the structure is packed by the compiler in use).

The following table shows some approximate measurements of the processor usage of the software UART module at various baud rates with the processor running at 50 MHz. Actual processor usage may vary, depending on how the application uses the software UART module, processor clock speed, interrupt priority, and compiler in use.

UART Baud Rate	% Of Processor	Million Cycles Per Second
9600	5.32	2.66
14400	7.99	3.99
19200	10.65	5.32
28800	15.96	7.98
38400	21.28	10.64
57600	32.00	16.00
115200	64.04	32.02

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

32.2 API Functions

Data Structures

- tSoftUART

Functions

- void SoftUARTBreakCtl (tSoftUART *pUART, tBoolean bBreakState)
- tBoolean SoftUARTBusy (tSoftUART *pUART)
- void SoftUARTCallbackSet (tSoftUART *pUART, void (*pfnCallback)(void))
- long SoftUARTCharGet (tSoftUART *pUART)
- long SoftUARTCharGetNonBlocking (tSoftUART *pUART)
- void SoftUARTCharPut (tSoftUART *pUART, unsigned char ucData)
- tBoolean SoftUARTCharPutNonBlocking (tSoftUART *pUART, unsigned char ucData)
- tBoolean SoftUARTCharsAvail (tSoftUART *pUART)
- void SoftUARTConfigGet (tSoftUART *pUART, unsigned long *pulConfig)
- void SoftUARTConfigSet (tSoftUART *pUART, unsigned long ulConfig)
- void SoftUARTDisable (tSoftUART *pUART)
- void SoftUARTEnable (tSoftUART *pUART)
- void SoftUARTFIFOLevelGet (tSoftUART *pUART, unsigned long *pulTxLevel, unsigned long *pulRxLevel)
- void SoftUARTFIFOLevelSet (tSoftUART *pUART, unsigned long ulTxLevel, unsigned long ulRxLevel)
- void SoftUARTInit (tSoftUART *pUART)
- void SoftUARTIntClear (tSoftUART *pUART, unsigned long ulIntFlags)
- void SoftUARTIntDisable (tSoftUART *pUART, unsigned long ulIntFlags)
- void SoftUARTIntEnable (tSoftUART *pUART, unsigned long ulIntFlags)

- unsigned long SoftUARTIntStatus (tSoftUART *pUART, tBoolean bMasked)
- unsigned long SoftUARTParityModeGet (tSoftUART *pUART)
- void SoftUARTParityModeSet (tSoftUART *pUART, unsigned long ulParity)
- void SoftUARTRxBufferSet (tSoftUART *pUART, unsigned short *pusRxBuffer, unsigned short usLen)
- void SoftUARTRxErrorClear (tSoftUART *pUART)
- unsigned long SoftUARTRxErrorGet (tSoftUART *pUART)
- void SoftUARTRxGPIOSet (tSoftUART *pUART, unsigned long ulBase, unsigned char ucPin)
- unsigned long SoftUARTRxTick (tSoftUART *pUART, tBoolean bEdgeInt)
- tBoolean SoftUARTSpaceAvail (tSoftUART *pUART)
- void SoftUARTTxBufferSet (tSoftUART *pUART, unsigned char *pucTxBuffer, unsigned short usLen)
- void SoftUARTTxGPIOSet (tSoftUART *pUART, unsigned long ulBase, unsigned char ucPin)
- void SoftUARTTxTimerTick (tSoftUART *pUART)

32.2.1 Data Structure Documentation

32.2.1.1 tSoftUART

Definition:

```
typedef struct
{
    void (*pfnIntCallback)(void);
    unsigned long ulTxGPIO;
    unsigned long ulRxGPIOPort;
    unsigned char *pucTxBuffer;
    unsigned short *pusRxBuffer;
    unsigned short usTxBufferLen;
    unsigned short usTxBufferRead;
    unsigned short usTxBufferWrite;
    unsigned short usTxBufferLevel;
    unsigned short usRxBufferLen;
    unsigned short usRxBufferRead;
    unsigned short usRxBufferWrite;
    unsigned short usRxBufferLevel;
    unsigned short usIntStatus;
    unsigned short usIntMask;
    unsigned short usConfig;
    unsigned char ucFlags;
    unsigned char ucTxState;
    unsigned char ucTxNext;
    unsigned char ucTxData;
    unsigned char ucRxPin;
    unsigned char ucRxState;
    unsigned char ucRxData;
    unsigned char ucRxFlags;
    unsigned char ucRxStatus;
}
tSoftUART
```

Members:

pfnIntCallback The address of the callback function that is called to simulate the interrupts that would be produced by a hardware UART implementation. This address can be set via a direct structure access or using the SoftUARTCallbackSet function.

ulTxGPIO The address of the GPIO pin to be used for the Tx signal. This member can be set via a direct structure access or using the SoftUARTTxGPIOSet function.

ulRxGPIOPort The address of the GPIO port to be used for the Rx signal. This member can be set via a direct structure access or using the SoftUARTRxGPIOSet function.

pucTxBuffer The address of the data buffer used for the transmit buffer. This member can be set via a direct structure access or using the SoftUARTTxBufferSet function.

pusRxBuffer The address of the data buffer used for the receive buffer. This member can be set via a direct structure access or using the SoftUARTRxBufferSet function.

usTxBufferLen The length of the transmit buffer. This member can be set via a direct structure access or using the SoftUARTTxBufferSet function.

usTxBufferRead The index into the transmit buffer of the next character to be transmitted. This member should not be accessed or modified by the application.

usTxBufferWrite The index into the transmit buffer of the next location to store a character into the buffer. This member should not be accessed or modified by the application.

usTxBufferLevel The transmit buffer level at which the transmit interrupt is asserted. This member should not be accessed or modified by the application.

usRxBufferLen The length of the receive buffer. This member can be set via a direct structure access or using the SoftUARTRxBufferSet function.

usRxBufferRead The index into the receive buffer of the next character to be read from the buffer. This member should not be accessed or modified by the application.

usRxBufferWrite The index into the receive buffer of the lcoation to store the next character received. This member should not be accessed or modified by the application.

usRxBufferLevel The receive buffer level at which the receive interrupt is asserted. This member should not be accessed or modified by the application.

usIntStatus The set of virtual interrupts that are currently asserted. This member should not be accessed or modified by the application.

usIntMask The set of virtual interrupts that should be sent to the callback function. This member should not be accessed or modified by the application.

usConfig The configuration of the SoftUART module. This member can be set via the SoftUARTConfigSet and SoftUARTFIFOLevelSet functions.

ucFlags The flags that control the operation of the SoftUART module. This member should not be be accessed or modified by the application.

ucTxState The current state of the SoftUART transmit state machine. This member should not be accessed or modified by the application.

ucTxNext The value that is written to the Tx pin at the start of the next transmit timer tick. This member should not be accessed or modified by the application.

ucTxData The character that is currently be sent via the Tx pin. This member should not be accessed or modified by the application.

ucRxPin The GPIO pin to be used for the Rx signal. This member can be set via a direct structure access or using the SoftUARTRxGPIOSet function.

ucRxState The current state of the SoftUART receive state machine. This member should not be accessed or modified by the application.

ucRxData The character that is currently being received via the Rx pin. This member should not be accessed or modified by the application.

ucRxFlags The flags that indicate any errors that have occurred during the reception of the current character via the Rx pin. This member should not be accessed or modified by the application.

ucRxStatus The receive error status. This member should only be accessed via the SoftUARTRxErrorGet and SoftURATRxErrorClear functions.

Description:
This structure contains the state of a single instance of a SoftUART module.

32.2.2 Function Documentation

32.2.2.1 SoftUARTBreakCtl

Causes a BREAK to be sent.

Prototype:
```
void
SoftUARTBreakCtl(tSoftUART *pUART,
                 tBoolean bBreakState)
```

Parameters:
pUART specifies the SoftUART data structure.
bBreakState controls the output level.

Description:
Calling this function with *bBreakState* set to **true** asserts a break condition on the SoftUART. Calling this function with *bBreakState* set to **false** removes the break condition. For proper transmission of a break command, the break must be asserted for at least two complete frames.

Returns:
None.

32.2.2.2 SoftUARTBusy

Determines whether the UART transmitter is busy or not.

Prototype:
```
tBoolean
SoftUARTBusy(tSoftUART *pUART)
```

Parameters:
pUART specifies the SoftUART data structure.

Description:
Allows the caller to determine whether all transmitted bytes have cleared the transmitter hardware. If **false** is returned, the transmit buffer is empty and all bits of the last transmitted character, including all stop bits, have left the hardware shift register.

Returns:
Returns **true** if the UART is transmitting or **false** if all transmissions are complete.

32.2.2.3 SoftUARTCallbackSet

Sets the callback used by the SoftUART module.

Prototype:

```
void
SoftUARTCallbackSet(tSoftUART *pUART,
                    void (*pfnCallback)(void))
```

Parameters:

pUART specifies the SoftUART data structure.

pfnCallback is a pointer to the callback function.

Description:

This function sets the address of the callback function that is called when there is an “interrupt” produced by the SoftUART module.

Returns:

None.

32.2.2.4 SoftUARTCharGet

Waits for a character from the specified port.

Prototype:

```
long
SoftUARTCharGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

Gets a character from the receive buffer for the specified port. If there are no characters available, this function waits until a character is received before returning.

Returns:

Returns the character read from the specified port, cast as a *long*.

32.2.2.5 SoftUARTCharGetNonBlocking

Receives a character from the specified port.

Prototype:

```
long
SoftUARTCharGetNonBlocking(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

Gets a character from the receive buffer for the specified port.

Returns:
Returns the character read from the specified port, cast as a *long*. A **-1** isreturned if there are no characters present in the receive buffer. The SoftUARTCharsAvail() function should be called before attempting to call this function.

32.2.2.6 SoftUARTCharPut

Waits to send a character from the specified port.

Prototype:
```
void
SoftUARTCharPut(tSoftUART *pUART,
                unsigned char ucData)
```

Parameters:
pUART specifies the SoftUART data structure.
ucData is the character to be transmitted.

Description:
Sends the character *ucData* to the transmit buffer for the specified port. If there is no space available in the transmit buffer, this function waits until there is space available before returning.

Returns:
None.

32.2.2.7 SoftUARTCharPutNonBlocking

Sends a character to the specified port.

Prototype:
```
tBoolean
SoftUARTCharPutNonBlocking(tSoftUART *pUART,
                           unsigned char ucData)
```

Parameters:
pUART specifies the SoftUART data structure.
ucData is the character to be transmitted.

Description:
Writes the character *ucData* to the transmit buffer for the specified port. This function does not block, so if there is no space available, then a **false** is returned, and the application must retry the function later.

Returns:
Returns **true** if the character was successfully placed in the transmit buffer or **false** if there was no space available in the transmit buffer.

32.2.2.8 SoftUARTCharsAvail

Determines if there are any characters in the receive buffer.

Prototype:

```
tBoolean
SoftUARTCharsAvail(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function returns a flag indicating whether or not there is data available in the receive buffer.

Returns:

Returns **true** if there is data in the receive buffer or **false** if there is no data in the receive buffer.

32.2.2.9 SoftUARTConfigGet

Gets the current configuration of a UART.

Prototype:

```
void
SoftUARTConfigGet(tSoftUART *pUART,
                  unsigned long *pulConfig)
```

Parameters:

pUART specifies the SoftUART data structure.

pulConfig is a pointer to storage for the data format.

Description:

Returns the data format of the SoftUART. The data format returned in *pulConfig* is enumerated the same as the *ulConfig* parameter of SoftUARTConfigSet().

Returns:

None.

32.2.2.10 SoftUARTConfigSet

Sets the configuration of a SoftUART module.

Prototype:

```
void
SoftUARTConfigSet(tSoftUART *pUART,
                  unsigned long ulConfig)
```

Parameters:

pUART specifies the SoftUART data structure.

ulConfig is the data format for the port (number of data bits, number of stop bits, and parity).

Description:

This function configures the SoftUART for operation in the specified data format, as specified in the *ulConfig* parameter.

The *ulConfig* parameter is the logical OR of three values: the number of data bits, the number of stop bits, and the parity. **SOFTUART_CONFIG_WLEN_8**, **SOFTUART_CONFIG_WLEN_7**, **SOFTUART_CONFIG_WLEN_6**, and **SOFTUART_CONFIG_WLEN_5** select from eight to five data bits per byte (respectively). **SOFTUART_CONFIG_STOP_ONE** and **SOFTUART_CONFIG_STOP_TWO** select one or two stop bits (respectively). **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, and **SOFTUART_CONFIG_PAR_ZERO** select the parity mode (no parity bit, even parity bit, odd parity bit, parity bit always one, and parity bit always zero, respectively).

Returns:

None.

32.2.2.11 SoftUARTDisable

Disables the SoftUART.

Prototype:

```
void
SoftUARTDisable(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function disables the SoftUART after waiting for it to become idle.

Returns:

None.

32.2.2.12 SoftUARTEnable

Enables the SoftUART.

Prototype:

```
void
SoftUARTEnable(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function enables the SoftUART, allowing data to be transmitted and received.

Returns:

None.

32.2.2.13 SoftUARTFIFOLevelGet

Gets the buffer level at which “interrupts” are generated.

Prototype:
```
void
SoftUARTFIFOLevelGet(tSoftUART *pUART,
                     unsigned long *pulTxLevel,
                     unsigned long *pulRxLevel)
```

Parameters:

pUART specifies the SoftUART data structure.

pulTxLevel is a pointer to storage for the transmit buffer level, returned as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

pulRxLevel is a pointer to storage for the receive buffer level, returned as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function gets the buffer level at which transmit and receive “interrupts” are generated.

Returns:

None.

32.2.2.14 SoftUARTFIFOLevelSet

Sets the buffer level at which “interrupts” are generated.

Prototype:
```
void
SoftUARTFIFOLevelSet(tSoftUART *pUART,
                     unsigned long ulTxLevel,
                     unsigned long ulRxLevel)
```

Parameters:

pUART specifies the SoftUART data structure.

ulTxLevel is the transmit buffer “interrupt” level, specified as one of **UART_FIFO_TX1_8**, **UART_FIFO_TX2_8**, **UART_FIFO_TX4_8**, **UART_FIFO_TX6_8**, or **UART_FIFO_TX7_8**.

ulRxLevel is the receive buffer “interrupt” level, specified as one of **UART_FIFO_RX1_8**, **UART_FIFO_RX2_8**, **UART_FIFO_RX4_8**, **UART_FIFO_RX6_8**, or **UART_FIFO_RX7_8**.

Description:

This function sets the buffer level at which transmit and receive “interrupts” are generated.

Returns:

None.

32.2.2.15 SoftUARTInit

Initializes the SoftUART module.

Prototype:
```
void
SoftUARTInit(tSoftUART *pUART)
```

Parameters:
pUART specifies the soft UART data structure.

Description:
This function initializes the data structure for the SoftUART module, putting it into the default configuration.

Returns:
None.

32.2.2.16 SoftUARTIntClear

Clears SoftUART “interrupt” sources.

Prototype:
```
void
SoftUARTIntClear(tSoftUART *pUART,
                 unsigned long ulIntFlags)
```

Parameters:
pUART specifies the SoftUART data structure.
ulIntFlags is a bit mask of the “interrupt” sources to be cleared.

Description:
The specified SoftUART “interrupt” sources are cleared, so that they no longer assert. This function must be called in the callback function to keep the “interrupt” from being recognized again immediately upon exit.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to SoftUARTIntEnable().

Returns:
None.

32.2.2.17 SoftUARTIntDisable

Disables individual SoftUART “interrupt” sources.

Prototype:
```
void
SoftUARTIntDisable(tSoftUART *pUART,
                   unsigned long ulIntFlags)
```

Parameters:
pUART specifies the SoftUART data structure.
ulIntFlags is the bit mask of the "interrupt" sources to be disabled.

Description:
Disables the indicated SoftUART "interrupt" sources. Only the sources that are enabled can be reflected to the SoftUART callback.

The *ulIntFlags* parameter has the same definition as the *ulIntFlags* parameter to SoftUARTIntEnable().

Returns:
None.

32.2.2.18 SoftUARTIntEnable

Enables individual SoftUART "interrupt" sources.

Prototype:
```
void
SoftUARTIntEnable(tSoftUART *pUART,
                  unsigned long ulIntFlags)
```

Parameters:
pUART specifies the SoftUART data structure.
ulIntFlags is the bit mask of the "interrupt" sources to be enabled.

Description:
Enables the indicated SoftUART "interrupt" sources. Only the sources that are enabled can be reflected to the SoftUART callback.

The *ulIntFlags* parameter is the logical OR of any of the following:

- **SOFTUART_INT_OE** - Overrun Error "interrupt"
- **SOFTUART_INT_BE** - Break Error "interrupt"
- **SOFTUART_INT_PE** - Parity Error "interrupt"
- **SOFTUART_INT_FE** - Framing Error "interrupt"
- **SOFTUART_INT_RT** - Receive Timeout "interrupt"
- **SOFTUART_INT_TX** - Transmit "interrupt"
- **SOFTUART_INT_RX** - Receive "interrupt"

Returns:
None.

32.2.2.19 SoftUARTIntStatus

Gets the current SoftUART "interrupt" status.

Prototype:
```
unsigned long
SoftUARTIntStatus(tSoftUART *pUART,
                  tBoolean bMasked)
```

Parameters:

pUART specifies the SoftUART data structure.

bMasked is **false** if the raw “interrupt” status is required and **true** if the masked “interrupt” status is required.

Description:

This returns the “interrupt” status for the SoftUART. Either the raw “interrupt” status or the status of “interrupts” that are allowed to reflect to the SoftUART callback can be returned.

Returns:

Returns the current “interrupt” status, enumerated as a bit field of values described in SoftUARTIntEnable().

32.2.2.20 SoftUARTParityModeGet

Gets the type of parity currently being used.

Prototype:

```
unsigned long
SoftUARTParityModeGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function gets the type of parity used for transmitting data and expected when receiving data.

Returns:

Returns the current parity settings, specified as one of **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, or **SOFTUART_CONFIG_PAR_ZERO**.

32.2.2.21 SoftUARTParityModeSet

Sets the type of parity.

Prototype:

```
void
SoftUARTParityModeSet(tSoftUART *pUART,
                      unsigned long ulParity)
```

Parameters:

pUART specifies the SoftUART data structure.

ulParity specifies the type of parity to use.

Description:

Sets the type of parity to use for transmitting and expect when receiving. The *ulParity* parameter must be one of **SOFTUART_CONFIG_PAR_NONE**, **SOFTUART_CONFIG_PAR_EVEN**, **SOFTUART_CONFIG_PAR_ODD**, **SOFTUART_CONFIG_PAR_ONE**, or **SOFTUART_CONFIG_PAR_ZERO**. The last two allow direct control of the parity bit; it is always either one or zero based on the mode.

Returns:
None.

32.2.2.22 SoftUARTRxBufferSet

Sets the receive buffer for a SoftUART module.

Prototype:
```
void
SoftUARTRxBufferSet(tSoftUART *pUART,
                    unsigned short *pusRxBuffer,
                    unsigned short usLen)
```

Parameters:
pUART specifies the SoftUART data structure.
pusRxBuffer is the address of the receive buffer.
usLen is the size, in 16-bit half-words, of the receive buffer.

Description:
This function sets the address and size of the receive buffer. It also resets the read and write pointers, marking the receive buffer as empty.

Returns:
None.

32.2.2.23 SoftUARTRxErrorClear

Clears all reported receiver errors.

Prototype:
```
void
SoftUARTRxErrorClear(tSoftUART *pUART)
```

Parameters:
pUART specifies the SoftUART data structure.

Description:
This function is used to clear all receiver error conditions reported via SoftUARTRxErrorGet(). If using the overrun, framing error, parity error or break interrupts, this function must be called after clearing the interrupt to ensure that later errors of the same type trigger another interrupt.

Returns:
None.

32.2.2.24 SoftUARTRxErrorGet

Gets current receiver errors.

Prototype:

```
unsigned long
SoftUARTRxErrorGet(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function returns the current state of each of the 4 receiver error sources. The returned errors are equivalent to the four error bits returned via the previous call to SoftUARTCharGet() or SoftUARTCharGetNonBlocking() with the exception that the overrun error is set immediately when the overrun occurs rather than when a character is next read.

Returns:

Returns a logical OR combination of the receiver error flags, **SOFTUART_RXERROR_FRAMING**, **SOFTUART_RXERROR_PARITY**, **SOFTUART_RXERROR_BREAK** and **SOFTUART_RXERROR_OVERRUN**.

32.2.2.25 SoftUARTRxGPIOSet

Sets the GPIO pin to be used as the SoftUART Rx signal.

Prototype:

```
void
SoftUARTRxGPIOSet(tSoftUART *pUART,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pUART specifies the SoftUART data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used when the SoftUART must sample the Rx signal. If there is not a GPIO pin allocated for Rx, the SoftUART module will not read data from the slave device.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

32.2.2.26 SoftUARTRxTick

Performs the periodic update of the SoftUART receiver.

Prototype:

```
unsigned long
SoftUARTRxTick(tSoftUART *pUART,
               tBoolean bEdgeInt)
```

Parameters:

pUART specifies the SoftUART data structure.

bEdgeInt should be **true** if this function is being called because of a GPIO edge interrupt and **false** if it is being called because of a timer interrupt.

Description:

This function performs the periodic, time-based updates to the SoftUART receiver. The reception of data to the SoftUART is performed by the state machine in this function.

This function must be called by the GPIO interrupt handler, and then periodically at the desired SoftUART baud rate. For example, to run the SoftUART at 115,200 baud, this function must be called at a 115,200 Hz rate.

Returns:

Returns **SOFTUART_RXTIMER_NOP** if the receive timer should continue to operate or **SOFTUART_RXTIMER_END** if it should be stopped.

32.2.2.27 SoftUARTSpaceAvail

Determines if there is any space in the transmit buffer.

Prototype:

```
tBoolean
SoftUARTSpaceAvail(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function returns a flag indicating whether or not there is space available in the transmit buffer.

Returns:

Returns **true** if there is space available in the transmit buffer or **false** if there is no space available in the transmit buffer.

32.2.2.28 SoftUARTTxBufferSet

Sets the transmit buffer for a SoftUART module.

Prototype:

```
void
SoftUARTTxBufferSet(tSoftUART *pUART,
                    unsigned char *pucTxBuffer,
                    unsigned short usLen)
```

Parameters:

pUART specifies the SoftUART data structure.

pucTxBuffer is the address of the transmit buffer.

usLen is the size, in 8-bit bytes, of the transmit buffer.

Description:

This function sets the address and size of the transmit buffer. It also resets the read and write pointers, marking the transmit buffer as empty.

Returns:

None.

32.2.2.29 SoftUARTTxGPIOSet

Sets the GPIO pin to be used as the SoftUART Tx signal.

Prototype:

```
void
SoftUARTTxGPIOSet(tSoftUART *pUART,
                  unsigned long ulBase,
                  unsigned char ucPin)
```

Parameters:

pUART specifies the SoftUART data structure.

ulBase is the base address of the GPIO module.

ucPin is the bit-packed representation of the pin to use.

Description:

This function sets the GPIO pin that is used when the SoftUART must assert the Tx signal.

The pin is specified using a bit-packed byte, where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Returns:

None.

32.2.2.30 SoftUARTTxTimerTick

Performs the periodic update of the SoftUART transmitter.

Prototype:

```
void
SoftUARTTxTimerTick(tSoftUART *pUART)
```

Parameters:

pUART specifies the SoftUART data structure.

Description:

This function performs the periodic, time-based updates to the SoftUART transmitter. The transmission of data from the SoftUART is performed by the state machine in this function.

This function must be called at the desired SoftUART baud rate. For example, to run the SoftUART at 115,200 baud, this function must be called at a 115,200 Hz rate.

Returns:

None.

32.3 Programming Example

The following example shows how to configure the software UART module and transmit some data to an external peripheral. This example uses Timer 0 as the timing source.

```
//
// The instance data for the software UART.
//
tSoftUART g_sUART;

//
// The buffer used to hold the transmit data.
//
unsigned char g_pucTxBuffer[16];

//
// The buffer used to hold the receive data.
//
unsigned short g_pusRxBuffer[16];

//
// The number of processor clocks in the time period of a single bit on the
// software UART interface.
//
unsigned long g_ulBitTime;

//
// The transmit timer tick function.
//
void
Timer0AIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART transmit timer tick function.
    //
    SoftUARTTxTimerTick(&g_sUART);
}

//
// The receive timer tick function.
//
void
Timer0BIntHandler(void)
{
    //
    // Clear the timer interrupt.
    //
    TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    //
    // Call the software UART receive timer tick function, and see if the
    // timer should be disabled.
    //
    if(SoftUARTRxTick(&g_sUART, false) == SOFTUART_RXTIMER_END)
    {
        //
        // Disable the timer interrupt since the software UART doesn't need
        // it any longer.
        //
```

```
        TimerDisable(TIMER0_BASE, TIMER_B);
    }
}

//
// The interrupt handler for the software UART GPIO edge interrupt.
//
void
GPIOIntHandler(void)
{
    //
    // Configure the software UART receive timer so that it samples at the
    // mid-bit time of this character.
    //
    TimerDisable(TIMER0_BASE, TIMER_B);
    TimerLoadSet(TIMER0_BASE, TIMER_B, g_ulBitTime);
    TimerIntClear(TIMER0_BASE, TIMER_TIMB_TIMEOUT);
    TimerEnable(TIMER0_BASE, TIMER_B);

    //
    // Call the software UART receive timer tick function.
    //
    SoftUARTRxTick(&g_sUART, true);
}

//
// The callback function for the software UART.  This function is
// equivalent to the interrupt handler for a hardware UART.
//
void
UARTCallback(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = SoftUARTIntStatus(&g_sUART, true);

    //
    // Clear the asserted interrupt sources.
    //
    SoftUARTIntClear(&g_sUART, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the software UART and send some data.
//
void
TestSoftUART(void)
{
    //
    // Initialize the software UART instance data.
    //
    SoftUARTInit(&g_sUART);

    //
    // Set the callback function used for this software UART.
    //
    SoftUARTCallbackSet(&g_sUART, UARTCallback);
```

```
//
// Configure the pins used for the software UART.  This example uses
// pins PD0 and PE1.
//
SoftUARTTxGPIOSet(&g_sUART, GPIO_PORTD_BASE, GPIO_PIN_0);
SoftUARTRxGPIOSet(&g_sUART, GPIO_PORTE_BASE, GPIO_PIN_1);

//
// Configure the data buffers used as the transmit and receive buffers.
//
SoftUARTTxBufferSet(&g_sUART, g_pucTxBuffer, 16);
SoftUARTRxBufferSet(&g_sUART, g_pusRxBuffer, 16);

//
// Enable the GPIO modules that contains the GPIO pins to be used by
// the software UART.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOD);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);

//
// Configure the software UART module: 8 data bits, no parity, and one
// stop bit.
//
SoftUARTConfigSet(&g_sUART,
                  (SOFTUART_CONFIG_WLEN_8 | SOFTUART_CONFIG_PAR_NONE |
                   SOFTUART_CONFIG_STOP_ONE));

//
// Compute the bit time for 38,400 baud.
//
g_ulBitTime = (SysCtlClockGet() / 38400) - 1;

//
// Configure the timers used to generate the timing for the software
// UART.  The interface in this example is run at 38,400 baud,
// requiring a timer tick at 38,400 Hz.
//
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
TimerConfigure(TIMER0_BASE,
               (TIMER_CFG_16_BIT_PAIR | TIMER_CFG_A_PERIODIC |
                TIMER_CFG_B_PERIODIC));
TimerLoadSet(TIMER0_BASE, TIMER_A, g_ulBitTime);
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT | TIMER_TIMB_TIMEOUT);
TimerEnable(TIMER0_BASE, TIMER_A);

//
// Set the priorities of the interrupts associated with the software
// UART.  The receiver is higher priority than the transmitter, and the
// receiver edge interrupt is higher priority than the receiver timer
// interrupt.
//
IntPrioritySet(INT_GPIOE, 0x00);
IntPrioritySet(INT_TIMER0B, 0x40);
IntPrioritySet(INT_TIMER0A, 0x80);

//
// Enable the interrupts associated with the software UART.
//
IntEnable(INT_GPIOE);
IntEnable(INT_TIMER0A);
IntEnable(INT_TIMER0B);

//
// Enable the transmit FIFO half full interrupt in the software UART.
//
```

```
    SoftUARTIntEnable(&g_sUART, SOFTUART_INT_TX);

    //
    // Write some data into the software UART transmit FIFO.
    //
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);
    SoftUARTCharPut(&g_sUART, 0x55);
    SoftUARTCharPut(&g_sUART, 0xaa);

    //
    // Wait until the software UART is idle.  The transmit FIFO half full
    // interrupt is sent to the callback function prior to exiting this
    // loop.
    //
    while(SoftUARTBusy(&g_sUART))
    {
    }
}
```

As a comparison, the following is the equivalent code using the hardware UART module and the Stellaris Peripheral Driver Library.

```
//
// The interrupt handler for the hardware UART.
//
void
UART0IntHandler(void)
{
    unsigned long ulInts;

    //
    // Read the asserted interrupt sources.
    //
    ulInts = UARTIntStatus(UART0_BASE, true);

    //
    // Clear the asserted interrupt sources.
    //
    UARTIntClear(UART0_BASE, ulInts);

    //
    // Handle the asserted interrupts.
    //
    ...
}

//
// Setup the hardware UART and send some data.
//
void
TestUART(void)
{
    //
    // Enable the GPIO module that contains the GPIO pins to be used by
    // the UART, as well as the UART module.
    //
```

```
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);

    //
    // Configure the GPIO pins for use by the UART module.
    //
    GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //
    // Initalize the hardware UART module: 8 data bits, no parity, one stop
    // bit, and 38,400 baud rate.
    //
    UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 38400,
                        (UART_CONFIG_WLEN_8 | UART_CONFIG_PAR_NONE |
                         UART_CONFIG_STOP_ONE));

    //
    // Enable the transmit FIFO half full interrupt in the hardware UART.
    //
    UARTIntEnable(UART0_BASE, UART_INT_TX);
    IntEnable(INT_UART0);

    //
    // Write some data into the hardware UART transmit FIFO.
    //
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);
    UARTCharPut(UART0_BASE, 0x55);
    UARTCharPut(UART0_BASE, 0xaa);

    //
    // Wait until the hardware UART is idle.  The transmit FIFO half full
    // interrupt is sent to the interrupt handler prior to exiting this
    // loop.
    //
    while(UARTBusy(UART0_BASE))
    {
    }
}
```

33 Ethernet Software Update Module

33.1 Introduction

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

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

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

33.2 API Functions

Functions

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

33.2.1 Function Documentation

33.2.1.1 SoftwareUpdateBegin

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

Prototype:

```
void
SoftwareUpdateBegin(void)
```

Description:

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

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

Note:

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

Returns:

Never returns.

33.2.1.2 SoftwareUpdateInit

Initializes the remote Ethernet software update notification feature.

Prototype:

```
void
SoftwareUpdateInit(tSoftwareUpdateRequested pfnCallback)
```

Parameters:

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

Description:

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

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

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

Note:

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

Returns:

None.

33.3 Programming Example

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

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

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

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

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

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

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

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

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

```
        }

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

34 TFTP Server Module

34.1 Introduction

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

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

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

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

34.2 Usage

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

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

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

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

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

34.2.0.3 pfnRequest

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

Prototype:

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

Parameters:

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

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

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

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

Description:

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

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

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

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

Returns:

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

34.2.0.4 pfnGetData

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

Prototype:

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

Parameters:

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

Description:

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

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

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

Returns:

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

34.2.0.5 pfnPutData

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

Prototype:

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

Parameters:

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

Description:

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

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

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

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

34.2.0.6 pfnClose

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

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

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

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

Returns:
None.

34.3 API Functions

Data Structures

- _tTFTPConnection

Defines

- TFTP_BLOCK_SIZE

Enumerations

- tTFTPError

Functions

- void TFTPInit (tTFTPRequest pfnRequest)

34.3.1 Data Structure Documentation

34.3.1.1 _tTFTPConnection

Definition:

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

Members:

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

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

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

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

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

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

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

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

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

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

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

34.3.2 Define Documentation

34.3.2.1 TFTP_BLOCK_SIZE

Definition:
```
#define TFTP_BLOCK_SIZE
```

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

34.3.3 Typedef Documentation

34.3.3.1 tTFTPConnection

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

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

34.3.4 Enumeration Documentation

34.3.4.1 tTFTPError

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

34.3.4.2 enum tTFTPMode

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

34.3.5 Function Documentation

34.3.5.1 void TFTPInit (tTFTPRequest *pfnRequest*)

Initializes the TFTP server module.

Parameters:

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

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

Returns:

None.

35 Micro Standard Library Module

35.1 Introduction

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

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

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

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

35.2 API Functions

Data Structures

- tTime

Functions

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

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

35.2.1 Data Structure Documentation

35.2.1.1 tTime

Definition:

```
typedef struct
{
    unsigned short usYear;
    unsigned char ucMon;
    unsigned char ucMday;
    unsigned char ucWday;
    unsigned char ucHour;
    unsigned char ucMin;
    unsigned char ucSec;
}
tTime
```

Members:

usYear The number of years since 0 AD.
ucMon The month, where January is 0 and December is 11.
ucMday The day of the month.
ucWday The day of the week, where Sunday is 0 and Saturday is 6.
ucHour The number of hours.
ucMin The number of minutes.
ucSec The number of seconds.

Description:

A structure that contains the broken down date and time.

35.2.2 Function Documentation

35.2.2.1 ulocaltime

Converts from seconds to calendar date and time.

Prototype:

```
void
ulocaltime(unsigned long ulTime,
           tTime *psTime)
```

Parameters:

ulTime is the number of seconds.
psTime is a pointer to the time structure that is filled in with the broken down date and time.

Description:

This function converts a number of seconds since midnight GMT on January 1, 1970 (traditional Unix epoch) into the equivalent month, day, year, hours, minutes, and seconds representation.

Returns:

None.

35.2.2.2 umktime

Converts calendar date and time to seconds.

Prototype:

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

Parameters:

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

Description:

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

Returns:

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

35.2.2.3 urand

Generate a new (pseudo) random number

Prototype:

```
int
urand(void)
```

Description:

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

Returns:

A pseudo-random number will be returned.

35.2.2.4 usnprintf

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

Prototype:

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

Parameters:

pcBuf is the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

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

Description:

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

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

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

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

The function will copy at most *ulSize* - 1 characters into the buffer *pcBuf*. One space is reserved in the buffer for the null termination character.

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

Returns:

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

35.2.2.5 usprintf

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

Prototype:

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

Parameters:

pcBuf is the buffer where the converted string is stored.

pcString is the format string.

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

Description:

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

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

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

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

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

Returns:

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

35.2.2.6 usrand

Set the random number generator seed.

Prototype:

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

Parameters:

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

Description:

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

Returns:

None

35.2.2.7 ustrcasecmp

Compares two strings without regard to case.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

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

Returns:

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

35.2.2.8 ustrcmp

Compares two strings.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

Description:

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

Returns:

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

35.2.2.9 ustrlen

Retruns the length of a null-terminated string.

Prototype:

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

Parameters:

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

Description:

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

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

Returns:

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

35.2.2.10 ustrncmp

Compares two strings.

Prototype:

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

Parameters:

pcStr1 points to the first string to be compared.

pcStr2 points to the second string to be compared.

iCount is the maximum number of characters to compare.

Description:

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

Returns:

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

35.2.2.11 ustrncpy

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

Prototype:

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

Parameters:

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

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

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

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

Returns:
Returns *pcDst*.

35.2.2.12 ustrnicmp

Compares two strings without regard to case.

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

Parameters:
pcStr1 points to the first string to be compared.
pcStr2 points to the second string to be compared.
iCount is the maximum number of characters to compare.

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

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

35.2.2.13 ustrstr

Finds a substring within a string.

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

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

Description:

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

Returns:

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

35.2.2.14 ustrtoul

Converts a string into its numeric equivalent.

Prototype:

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

Parameters:

pcStr is a pointer to the string containing the integer.

ppcStrRet is a pointer that will be set to the first character past the integer in the string.

iBase is the radix to use for the conversion; can be zero to auto-select the radix or between 2 and 16 to explicitly specify the radix.

Description:

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

Returns:

Returns the result of the conversion.

35.2.2.15 uvsnprintf

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

Prototype:

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

Parameters:

pcBuf points to the buffer where the converted string is stored.

ulSize is the size of the buffer.

pcString is the format string.

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

Description:

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

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

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

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

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

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

Returns:

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

35.3 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);
```

36 UART Standard IO Module

36.1 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.

36.1.1 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.

36.1.2 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.

36.2 API Functions

Functions

- void UARTEchoSet (tBoolean bEnable)
- void UARTFlushRx (void)
- void UARTFlushTx (tBoolean bDiscard)
- unsigned char UARTgetc (void)
- int UARTgets (char *pcBuf, unsigned long ulLen)
- int UARTPeek (unsigned char ucChar)
- void UARTprintf (const char *pcString,...)
- int UARTRxBytesAvail (void)
- void UARTStdioConfig (unsigned long ulPortNum, unsigned long ulBaud, unsigned long ulSrcClock)
- void UARTStdioInit (unsigned long ulPortNum)
- void UARTStdioInitExpClk (unsigned long ulPortNum, unsigned long ulBaud)
- void UARTStdioIntHandler (void)
- int UARTTxBytesFree (void)
- int UARTwrite (const char *pcBuf, unsigned long ulLen)

36.2.1 Function Documentation

36.2.1.1 UARTEchoSet

Enables or disables echoing of received characters to the transmitter.

Prototype:

```
void
UARTEchoSet(tBoolean bEnable)
```

Parameters:

bEnable must be set to **true** to enable echo or **false** to disable it.

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to control whether or not received characters are automatically echoed back to the transmitter. By default, echo is enabled and this is typically the desired behavior if the module is being used to support a serial command line. In applications where this module is being used to provide a convenient, buffered serial interface over which application-specific binary protocols are being run, however, echo may be undesirable and this function can be used to disable it.

Returns:

None.

36.2.1.2 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.

36.2.1.3 UARTFlushTx

Flushes the transmit buffer.

Prototype:
```
void
UARTFlushTx(tBoolean bDiscard)
```

Parameters:
bDiscard indicates whether any remaining data in the buffer should be discarded (**true**) or transmitted (**false**).

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to flush the transmit buffer, either discarding or transmitting any data received via calls to UARTprintf() that is waiting to be transmitted. On return, the transmit buffer will be empty.

Returns:
None.

36.2.1.4 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.

36.2.1.5 UARTgets

A simple UART based get string function, with some line processing.

Prototype:
```
int
UARTgets(char *pcBuf,
         unsigned long ulLen)
```

Parameters:
pcBuf points to a buffer for the incoming string from the UART.
ulLen is the length of the buffer for storage of the string, including the trailing 0.

Description:
This function will receive a string from the UART input and store the characters in the buffer pointed to by *pcBuf*. The characters will continue to be stored until a termination character is received. The termination characters are CR, LF, or ESC. A CRLF pair is treated as a single termination character. The termination characters are not stored in the string. The string will be terminated with a 0 and the function will return.

In both buffered and unbuffered modes, this function will block until a termination character is received. If non-blocking operation is required in buffered mode, a call to UARTPeek() may be made to determine whether a termination character already exists in the receive buffer prior to calling UARTgets().

Since the string will be null terminated, the user must ensure that the buffer is sized to allow for the additional null character.

Returns:
Returns the count of characters that were stored, not including the trailing 0.

36.2.1.6 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.

36.2.1.7 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.

36.2.1.8 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.

36.2.1.9 UARTStdioConfig

Configures the UART console.

Prototype:

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

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

ulBaud is the bit rate that the UART is to be configured to use.

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

Description:

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

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

Returns:

None.

36.2.1.10 UARTStdioInit

Initializes the UART console.

Prototype:

```
void
UARTStdioInit(unsigned long ulPortNum)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

Description:

This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 115200, 8-N-1. An application wishing to use a different baud rate may call UARTStdioInitExpClk() instead of this function.

This function or UARTStdioInitExpClk() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

36.2.1.11 UARTStdioInitExpClk

Initializes the UART console and allows the baud rate to be selected.

Prototype:

```
void
UARTStdioInitExpClk(unsigned long ulPortNum,
                    unsigned long ulBaud)
```

Parameters:

ulPortNum is the number of UART port to use for the serial console (0-2)

ulBaud is the bit rate that the UART is to be configured to use.

Description:

This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 8-N-1 and the bit rate set according to the value of the *ulBaud* parameter.

This function or UARTStdioInit() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function. An application wishing to use 115,200 baud may call UARTStdioInit() instead of this function but should not call both functions.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:

None.

36.2.1.12 UARTStdioIntHandler

Handles UART interrupts.

Prototype:

```
void
UARTStdioIntHandler(void)
```

Description:

This function handles interrupts from the UART. It will copy data from the transmit buffer to the UART transmit FIFO if space is available, and it will copy data from the UART receive FIFO to the receive buffer if data is available.

Returns:

None.

36.2.1.13 UARTTxBytesFree

Returns the number of bytes free in the transmit buffer.

Prototype:

```
int
UARTTxBytesFree(void)
```

Description:

This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to determine the amount of space currently available in the transmit buffer.

Returns:

Returns the number of free bytes.

36.2.1.14 UARTwrite

Writes a string of characters to the UART output.

Prototype:

```
int
UARTwrite(const char *pcBuf,
          unsigned long ulLen)
```

Parameters:

pcBuf points to a buffer containing the string to transmit.

ulLen is the length of the string to transmit.

Description:

This function will transmit the string to the UART output. The number of characters transmitted is determined by the *ulLen* parameter. This function does no interpretation or translation of any characters. Since the output is sent to a UART, any LF (/n) characters encountered will be replaced with a CRLF pair.

Besides using the *ulLen* parameter to stop transmitting the string, if a null character (0) is encountered, then no more characters will be transmitted and the function will return.

In non-buffered mode, this function is blocking and will not return until all the characters have been written to the output FIFO. In buffered mode, the characters are written to the UART transmit buffer and the call returns immediately. If insufficient space remains in the transmit buffer, additional characters are discarded.

Returns:

Returns the count of characters written.

36.3 Programming Example

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

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

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

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

IMPORTANT NOTICE

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

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

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

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

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

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

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

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

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

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

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

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