RDK-S2E Firmware Development Package

USER'S GUIDE

SW-RDK-S2E-UG-8555

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 8555 of this document, last updated on January 28, 2012.

Table of Contents

Copyright **2**
Revision Information **2**
1 Introduction **5**
2 Example Applications **7**
2.1 Boot Loader (boot_eth) 7
2.2 Serial To Ethernet Module (ser2enet) 7
3 Development System Utilities **9**
4 Main Module **13**
4.1 Introduction 13
4.2 API Functions 13
5 Configuration Module **17**
5.1 Introduction 17
5.2 API Functions 17
6 File System Module **33**
6.1 Introduction 33
6.2 API Functions 33
7 Serial Port Module **37**
7.1 Introduction 37
7.2 API Functions 37
8 Telnet Port Module **47**
8.1 Introduction 47
8.2 API Functions 47
9 Universal Plug and Play Module **57**
9.1 Introduction 57
9.2 API Functions 57
10 Command Line Processing Module **59**
10.1 Introduction 59
10.2 API Functions 59
10.3 Programming Example 61
11 Flash Parameter Block Module **63**
11.1 Introduction 63
11.2 API Functions 63
11.3 Programming Example 65
12 lwIP Wrapper Module **67**
12.1 Introduction 67
12.2 API Functions 67
12.3 Programming Example 70
13 Ring Buffer Module **71**
13.1 Introduction 71
13.2 API Functions 71
13.3 Programming Example 77
14 Simple Task Scheduler Module **79**
14.1 Introduction 79
14.2 API Functions 79

14.3 Programming Example 84
15 Ethernet Software Update Module 87
15.1 Introduction 87
15.2 API Functions 87
15.3 Programming Example 89
16 TFTP Server Module 91
16.1 Introduction 91
16.2 Usage 91
16.3 API Functions 94
17 Micro Standard Library Module 99
17.1 Introduction 99
17.2 API Functions 99
17.3 Programming Example 108
18 UART Standard IO Module 111
18.1 Introduction 111
18.2 API Functions 112
18.3 Programming Example 118
IMPORTANT NOTICE 120

1 Introduction

The Texas Instruments® Stellaris® Serial to Ethernet Module is a ready to use module for converting an existing serial link into a Ethernet link. The module contains an ARM® Cortex™-M3-based microcontroller, a pair of serial ports (one at RS232 levels and one at CMOS/TTL levels), and an Ethernet port.

The software for this module provides the following:

- IP configuration via static IP assignment or DHCP.
- Web server for module configuration.
- UPnP responder for device discovery.
- Telnet server with RFC2217 (remote serial port configuration) support.
- Telnet client to create an Ethernet-based serial port extender using two modules (one as the client and one as the server).

This document describes the reference software provided for this module.

2 Example Applications

There is an IAR workspace file (`rdk-s2e.eww`) that contains the peripheral driver library project, along with the Serial to Ethernet software project, in a single, easy to use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`rdk-s2e.mpw`) that contains the peripheral driver library project, along with the Serial to Ethernet software project, in a single, easy to use workspace for use with uVision.

All of these examples reside in the `boards/rdk-s2e` subdirectory of the firmware development package source distribution.

2.1 Boot Loader (boot_eth)

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

2.2 Serial To Ethernet Module (ser2enet)

The Serial to Ethernet Converter provides a means of accessing the UART on a Stellaris device via a network connection. The UART can be connected to the UART on a non-networked device, providing the ability to access the device via a network. This can be useful to overcome the cable length limitations of a UART connection (in fact, the cable can become thousands of miles long) and to provide networking capability to existing devices without modifying the device's operation.

The converter can be configured to use a static IP configuration or to use DHCP to obtain its IP configuration. Since the converter is providing a telnet server, the effective use of DHCP requires a reservation in the DHCP server so that the converter gets the same IP address each time it is connected to the network.

3 Development System Utilities

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

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

Ethernet Flash Downloader

Usage:

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

Description:

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

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

Arguments:

--help displays usage information.

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

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

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

--mac=MAC specifies the MAC address

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

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

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

--verbose specifies that verbose output should be output.

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

INPUT FILE specifies the name of the firmware image file.

Example:

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

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

Finder

Usage:

```
finder
```

Description:

This program locates Stellaris boards on the local network that are running an lwIP-based application that includes the locator service. It will display the IP address, MAC address, client

address, and application description for each board that it finds. This is useful for easily finding the IP address that has been assigned to a board via DHCP or AutoIP without needing to display it from the application (which is difficult on boards that do not have a builtin display).

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

Example:

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

```
finder
```

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
```

4 Main Module

Introduction 13
API Functions 13

4.1 Introduction

The main application entry point is contained in `boards/rdk-s2e/ser2enet/ser2enet.c`. The various modules are initialized, including the serial port driver, the telnet driver, the Universal Plug and Play driver, along with the configuration web server.

To provide periodic processing that is required by lwIP, the system tick timer is programmed and the interrupt service routine for this timer is contained here.

The lwIP Abstraction Library also provides for a host callback routine that can be configured as a periodic callback run in the lwIP context in order to deal with the fact that lwIP is not re-entrant. The host callback routine defined here provides support for the telnet and upnp modules.

4.2 API Functions

Defines

- SYSTICKHZ
- SYSTICKMS

Functions

- void lwIPHostTimerHandler (void)
- int main (void)
- void SysTickIntHandler (void)

Variables

- volatile tBoolean g_bFirmwareUpdate
- volatile tBoolean g_bLinkStatusUp
- unsigned long g_ulSystemTimeMS

4.2.1 Define Documentation

4.2.1.1 SYSTICKHZ

Definition:

```
#define SYSTICKHZ
```

Description:
The number of times per second that the SysTick timer generates a processor interrupt.

4.2.1.2 SYSTICKMS

Definition:
```
#define SYSTICKMS
```

Description:
The number of milliseconds between SysTick generated processor interrupts.

4.2.2 Function Documentation

4.2.2.1 lwIPHostTimerHandler

Handles the Ethernet interrupt hooks for the client software.

Prototype:
```
void
lwIPHostTimerHandler(void)
```

Description:
This function will run any handlers that are required to run in the Ethernet interrupt context. All the actual TCP/IP processing occurs within this function (since lwIP is not re-entrant).

Returns:
None.

4.2.2.2 main

Handles the main application loop.

Prototype:
```
int
main(void)
```

Description:
This function initializes and configures the device and the software, then performs the main loop.

Returns:
Never returns.

4.2.2.3 SysTickIntHandler

Handles the SysTick interrupt.

Prototype:

```
void
SysTickIntHandler(void)
```

Description:

This function is called when the SysTick timer expires. It increments the lwIP timers and sets a flag indicating that the timeout functions need to be called if necessary.

Returns:

None.

4.2.3 Variable Documentation

4.2.3.1 g_bFirmwareUpdate

Definition:

```
volatile tBoolean g_bFirmwareUpdate
```

Description:

A flag indicating that a firmware update request has been received.

4.2.3.2 g_bLinkStatusUp

Definition:

```
volatile tBoolean g_bLinkStatusUp
```

Description:

A flag indicating the current link status.

4.2.3.3 g_ulSystemTimeMS

Definition:

```
unsigned long g_ulSystemTimeMS
```

Description:

Keeps track of elapsed time in milliseconds.

5 Configuration Module

Introduction 17
API Functions 17

5.1 Introduction

The configuration module defines and manages the global configuration parameter block, as well providing an abstraction layer for the non-volatile storage of this parameter block.

These functions are contained in `boards/rdk-s2e/ser2enet/config.c`, with `boards/rdk-s2e/ser2enet/config.h` containing the API definitions for use the remainder of the application.

5.2 API Functions

Data Structures

- tConfigParameters
- tPortParameters
- tStringMap

Defines

- CONFIG_FLAG_STATICIP
- CONFIG_RFC2217_ENABLED
- DEFAULT_CGI_RESPONSE
- FIRMWARE_UPDATE_RESPONSE
- FLASH_PB_END
- FLASH_PB_SIZE
- FLASH_PB_START
- IP_UPDATE_RESPONSE
- MAX_S2E_PORTS
- MAX_VARIABLE_NAME_LEN
- MISC_PAGE_URI
- NUM_CONFIG_CGI_URIS
- NUM_CONFIG_SSI_TAGS
- PARAM_ERROR_RESPONSE
- PIN_U0CTS_PIN
- PIN_U0CTS_PORT
- PIN_U0RTS_INT
- PIN_U0RTS_PIN
- PIN_U0RTS_PORT

- PIN_U0RX_PIN
- PIN_U0RX_PORT
- PIN_U0TX_PIN
- PIN_U0TX_PORT
- PIN_U1CTS_PIN
- PIN_U1CTS_PORT
- PIN_U1RTS_INT
- PIN_U1RTS_PIN
- PIN_U1RTS_PORT
- PIN_U1RX_PIN
- PIN_U1RX_PORT
- PIN_U1TX_PIN
- PIN_U1TX_PORT
- PIN_XVR_INV_N_PIN
- PIN_XVR_INV_N_PORT
- PIN_XVR_OFF_N_PIN
- PIN_XVR_OFF_N_PORT
- PIN_XVR_ON_PIN
- PIN_XVR_ON_PORT
- PIN_XVR_RDY_PIN
- PIN_XVR_RDY_PORT
- PORT_FLAG_PROTOCOL
- PORT_FLAG_TELNET_MODE
- RX_RING_BUF_SIZE
- TX_RING_BUF_SIZE

Functions

- void ConfigInit (void)
- void ConfigLoad (void)
- void ConfigLoadFactory (void)
- void ConfigSave (void)
- void ConfigWebInit (void)

Variables

- tBoolean g_bStartBootloader
- unsigned char g_cUpdateRequired
- const tConfigParameters * g_psDefaultParameters
- const tConfigParameters *const g_psFactoryParameters
- tConfigParameters g_sParameters
- const unsigned short g_usFirmwareVersion

5.2.1 Data Structure Documentation

5.2.1.1 tConfigParameters

Definition:

```
typedef struct
{
    unsigned char ucSequenceNum;
    unsigned char ucCRC;
    unsigned char ucVersion;
    unsigned char ucFlags;
    unsigned short usLocationURLPort;
    unsigned char ucReserved1[2];
    tPortParameters sPort[MAX_S2E_PORTS];
    unsigned char ucModName[MOD_NAME_LEN];
    unsigned long ulStaticIP;
    unsigned long ulGatewayIP;
    unsigned long ulSubnetMask;
    unsigned char ucReserved2[108];
}
tConfigParameters
```

Members:

ucSequenceNum The sequence number of this parameter block. When in RAM, this value is not used. When in flash, this value is used to determine the parameter block with the most recent information.

ucCRC The CRC of the parameter block. When in RAM, this value is not used. When in flash, this value is used to validate the contents of the parameter block (to avoid using a partially written parameter block).

ucVersion The version of this parameter block. This can be used to distinguish saved parameters that correspond to an old version of the parameter block.

ucFlags Character field used to store various bit flags.

usLocationURLPort The TCP port number to be used for access to the UPnP Location URL that is part of the discovery response message.

ucReserved1 Padding to ensure consistent parameter block alignment.

sPort The configuration parameters for each port available on the S2E module.

ucModName An ASCII string used to identify the module to users via UPnP and web configuration.

ulStaticIP The static IP address to use if DHCP is not in use.

ulGatewayIP The default gateway IP address to use if DHCP is not in use.

ulSubnetMask The subnet mask to use if DHCP is not in use.

ucReserved2 Padding to ensure the whole structure is 256 bytes long.

Description:

This structure contains the S2E module parameters that are saved to flash. A copy exists in RAM for use during the execution of the application, which is loaded from flash at startup. The modified parameter block can also be written back to flash for use on the next power cycle.

5.2.1.2 tPortParameters

Definition:

```
typedef struct
{
    unsigned long ulBaudRate;
    unsigned char ucDataSize;
    unsigned char ucParity;
    unsigned char ucStopBits;
    unsigned char ucFlowControl;
    unsigned long ulTelnetTimeout;
    unsigned short usTelnetLocalPort;
    unsigned short usTelnetRemotePort;
    unsigned long ulTelnetIPAddr;
    unsigned char ucFlags;
    unsigned char ucReserved0[19];
}
tPortParameters
```

Members:

ulBaudRate The baud rate to be used for the UART, specified in bits-per-second (bps).

ucDataSize The data size to be use for the serial port, specified in bits. Valid values are 5, 6, 7, and 8.

ucParity The parity to be use for the serial port, specified as an enumeration. Valid values are 1 for no parity, 2 for odd parity, 3 for even parity, 4 for mark parity, and 5 for space parity.

ucStopBits The number of stop bits to be use for the serial port, specified as a number. Valid values are 1 and 2.

ucFlowControl The flow control to be use for the serial port, specified as an enumeration. Valid values are 1 for no flow control and 3 for HW (RTS/CTS) flow control.

ulTelnetTimeout The timeout for the TCP connection used for the telnet session, specified in seconds. A value of 0 indicates no timeout is to be used.

usTelnetLocalPort The local TCP port number to be listened on when in server mode or from which the outgoing connection will be initiated in client mode.

usTelnetRemotePort The TCP port number to which a connection will be established when in telnet client mode.

ulTelnetIPAddr The IP address which will be connected to when in telnet client mode.

ucFlags Miscellaneous flags associated with this connection.

ucReserved0 Padding to ensure consistent parameter block alignment, and to allow for future expansion of port parameters.

Description:

This structure defines the port parameters used to configure the UART and telnet session associated with a single instance of a port on the S2E module.

5.2.1.3 tStringMap

Definition:

```
typedef struct
{
    const char *pcString;
    unsigned char ucId;
```

```
}
tStringMap
```

Members:

pcString A human readable string related to the identifier found in the ucId field.

ucId An identifier value associated with the string held in the pcString field.

Description:

Structure used in mapping numeric IDs to human-readable strings.

5.2.2 Define Documentation

5.2.2.1 CONFIG_FLAG_STATICIP

Definition:

```
#define CONFIG_FLAG_STATICIP
```

Description:

If this flag is set in the ucFlags field of tConfigParameters, the module uses a static IP. If not, DHCP and AutoIP are used to obtain an IP address.

5.2.2.2 CONFIG_RFC2217_ENABLED

Definition:

```
#define CONFIG_RFC2217_ENABLED
```

Description:

Enable RFC-2217 (COM-PORT-OPTION) in the telnet server code.

5.2.2.3 DEFAULT_CGI_RESPONSE

Definition:

```
#define DEFAULT_CGI_RESPONSE
```

Description:

The file sent back to the browser by default following completion of any of our CGI handlers.

5.2.2.4 FIRMWARE_UPDATE_RESPONSE

Definition:

```
#define FIRMWARE_UPDATE_RESPONSE
```

Description:

The file sent back to the browser to signal that the bootloader is being entered to perform a software update.

5.2.2.5 FLASH_PB_END

Definition:

```
#define FLASH_PB_END
```

Description:

The address of the last block of flash to be used for storing parameters. Since the end of flash is used for parameters, this is actually the first address past the end of flash.

5.2.2.6 FLASH_PB_SIZE

Definition:

```
#define FLASH_PB_SIZE
```

Description:

The size of the parameter block to save. This must be a power of 2, and should be large enough to contain the tConfigParameters structure.

5.2.2.7 FLASH_PB_START

Definition:

```
#define FLASH_PB_START
```

Description:

The address of the first block of flash to be used for storing parameters.

5.2.2.8 IP_UPDATE_RESPONSE

Definition:

```
#define IP_UPDATE_RESPONSE
```

Description:

The file sent back to the browser to signal that the IP address of the device is about to change and that the web server is no longer operating.

5.2.2.9 MAX_S2E_PORTS

Definition:

```
#define MAX_S2E_PORTS
```

Description:

The number of serial to Ethernet ports supported by this module.

5.2.2.10 MAX_VARIABLE_NAME_LEN

Definition:

```
#define MAX_VARIABLE_NAME_LEN
```

Description:

The maximum length of any HTML form variable name used in this application.

5.2.2.11 MISC_PAGE_URI

Definition:

```
#define MISC_PAGE_URI
```

Description:

The URI of the “Miscellaneous Settings” page offered by the web server.

5.2.2.12 NUM_CONFIG_CGI_URIS

Definition:

```
#define NUM_CONFIG_CGI_URIS
```

Description:

The number of individual CGI URIs that are used by our configuration web pages.

5.2.2.13 NUM_CONFIG_SSI_TAGS

Definition:

```
#define NUM_CONFIG_SSI_TAGS
```

Description:

The number of individual SSI tags that the HTTPD server can expect to find in our configuration pages.

5.2.2.14 PARAM_ERROR_RESPONSE

Definition:

```
#define PARAM_ERROR_RESPONSE
```

Description:

The file sent back to the browser in cases where a parameter error is detected by one of the CGI handlers. This should only happen if someone tries to access the CGI directly via the browser command line and doesn’t enter all the required parameters alongside the URI.

5.2.2.15 PIN_U0CTS_PIN

Definition:

```
#define PIN_U0CTS_PIN
```

Description:

The GPIO pin on which the U0CTS pin resides.

5.2.2.16 PIN_U0CTS_PORT

Definition:

```
#define PIN_U0CTS_PORT
```

Description:

The GPIO port on which the U0CTS pin resides.

5.2.2.17 PIN_U0RTS_INT

Definition:

```
#define PIN_U0RTS_INT
```

Description:

The interrupt on which the U0RTS pin resides.

5.2.2.18 PIN_U0RTS_PIN

Definition:

```
#define PIN_U0RTS_PIN
```

Description:

The GPIO pin on which the U0RTS pin resides.

5.2.2.19 PIN_U0RTS_PORT

Definition:

```
#define PIN_U0RTS_PORT
```

Description:

The GPIO port on which the U0RTS pin resides.

5.2.2.20 PIN_U0RX_PIN

Definition:

```
#define PIN_U0RX_PIN
```

Description:

The GPIO pin on which the U0RX pin resides.

5.2.2.21 PIN_U0RX_PORT

Definition:
```
#define PIN_U0RX_PORT
```

Description:
The GPIO port on which the U0RX pin resides.

5.2.2.22 PIN_U0TX_PIN

Definition:
```
#define PIN_U0TX_PIN
```

Description:
The GPIO pin on which the U0TX pin resides.

5.2.2.23 PIN_U0TX_PORT

Definition:
```
#define PIN_U0TX_PORT
```

Description:
The GPIO port on which the U0TX pin resides.

5.2.2.24 PIN_U1CTS_PIN

Definition:
```
#define PIN_U1CTS_PIN
```

Description:
The GPIO pin on which the U1CTS pin resides.

5.2.2.25 PIN_U1CTS_PORT

Definition:
```
#define PIN_U1CTS_PORT
```

Description:
The GPIO port on which the U1CTS pin resides.

5.2.2.26 PIN_U1RTS_INT

Definition:
```
#define PIN_U1RTS_INT
```

Description:
The interrupt on which the U1RTS pin resides.

5.2.2.27 PIN_U1RTS_PIN

Definition:

```
#define PIN_U1RTS_PIN
```

Description:

The GPIO pin on which the U1RTS pin resides.

5.2.2.28 PIN_U1RTS_PORT

Definition:

```
#define PIN_U1RTS_PORT
```

Description:

The GPIO port on which the U1RTS pin resides.

5.2.2.29 PIN_U1RX_PIN

Definition:

```
#define PIN_U1RX_PIN
```

Description:

The GPIO pin on which the U1RX pin resides.

5.2.2.30 PIN_U1RX_PORT

Definition:

```
#define PIN_U1RX_PORT
```

Description:

The GPIO port on which the U1RX pin resides.

5.2.2.31 PIN_U1TX_PIN

Definition:

```
#define PIN_U1TX_PIN
```

Description:

The GPIO pin on which the U1TX pin resides.

5.2.2.32 PIN_U1TX_PORT

Definition:

```
#define PIN_U1TX_PORT
```

Description:

The GPIO port on which the U1TX pin resides.

5.2.2.33 PIN_XVR_INV_N_PIN

Definition:

```
#define PIN_XVR_INV_N_PIN
```

Description:

The GPIO pin on which the XVR_INV_N pin resides.

5.2.2.34 PIN_XVR_INV_N_PORT

Definition:

```
#define PIN_XVR_INV_N_PORT
```

Description:

The GPIO port on which the XVR_INV_N pin resides.

5.2.2.35 PIN_XVR_OFF_N_PIN

Definition:

```
#define PIN_XVR_OFF_N_PIN
```

Description:

The GPIO pin on which the XVR_OFF_N pin resides.

5.2.2.36 PIN_XVR_OFF_N_PORT

Definition:

```
#define PIN_XVR_OFF_N_PORT
```

Description:

The GPIO port on which the XVR_OFF_N pin resides.

5.2.2.37 PIN_XVR_ON_PIN

Definition:

```
#define PIN_XVR_ON_PIN
```

Description:

The GPIO pin on which the XVR_ON pin resides.

5.2.2.38 PIN_XVR_ON_PORT

Definition:

```
#define PIN_XVR_ON_PORT
```

Description:

The GPIO port on which the XVR_ON pin resides.

5.2.2.39 PIN_XVR_RDY_PIN

Definition:

```
#define PIN_XVR_RDY_PIN
```

Description:

The GPIO pin on which the XVR_RDY pin resides.

5.2.2.40 PIN_XVR_RDY_PORT

Definition:

```
#define PIN_XVR_RDY_PORT
```

Description:

The GPIO port on which the XVR_RDY pin resides.

5.2.2.41 PORT_FLAG_PROTOCOL

Definition:

```
#define PORT_FLAG_PROTOCOL
```

Description:

Bit 1 of field ucFlags in tPortParameters is used to indicate whether to bypass the telnet protocol (raw data mode).

5.2.2.42 PORT_FLAG_TELNET_MODE

Definition:

```
#define PORT_FLAG_TELNET_MODE
```

Description:

Bit 0 of field ucFlags in tPortParameters is used to indicate whether to operate as a telnet server or a telnet client.

5.2.2.43 RX_RING_BUF_SIZE

Definition:

```
#define RX_RING_BUF_SIZE
```

Description:

The size of the ring buffers used for interface between the UART and telnet session (RX).

5.2.2.44 TX_RING_BUF_SIZE

Definition:

```
#define TX_RING_BUF_SIZE
```

Description:

The size of the ring buffers used for interface between the UART and telnet session (TX).

5.2.3 Function Documentation

5.2.3.1 ConfigInit

Initializes the configuration parameter block.

Prototype:

```
void
ConfigInit(void)
```

Description:

This function initializes the configuration parameter block. If the version number of the parameter block stored in flash is older than the current revision, new parameters will be set to default values as needed.

Returns:

None.

5.2.3.2 ConfigLoad

Loads the S2E parameter block from flash.

Prototype:

```
void
ConfigLoad(void)
```

Description:

This function is called to load the most recently saved parameter block from flash.

Returns:

None.

5.2.3.3 ConfigLoadFactory

Loads the S2E parameter block from factory-default table.

Prototype:

```
void
ConfigLoadFactory(void)
```

Description:

This function is called to load the factory default parameter block.

Returns:
None.

5.2.3.4 ConfigSave

Saves the S2E parameter block to flash.

Prototype:
```
void
ConfigSave(void)
```

Description:
This function is called to save the current S2E configuration parameter block to flash memory.

Returns:
None.

5.2.3.5 ConfigWebInit

Configures HTTPD server SSI and CGI capabilities for our configuration forms.

Prototype:
```
void
ConfigWebInit(void)
```

Description:
This function informs the HTTPD server of the server-side-include tags that we will be processing and the special URLs that are used for CGI processing for the web-based configuration forms.

Returns:
None.

5.2.4 Variable Documentation

5.2.4.1 g_bStartBootloader

Definition:
```
tBoolean g_bStartBootloader
```

Description:
A flag to the main loop indicating that it should enter the bootloader and perform a firmware update.

5.2.4.2 g_cUpdateRequired

Definition:

```
unsigned char g_cUpdateRequired
```

Description:

Flags sent to the main loop indicating that it should update the IP address after a short delay (to allow us to send a suitable page back to the web browser telling it the address has changed).

5.2.4.3 g_psDefaultParameters

Definition:

```
const tConfigParameters *g_psDefaultParameters
```

Description:

This structure instance points to the most recently saved parameter block in flash. It can be considered the default set of parameters.

5.2.4.4 g_psFactoryParameters

Definition:

```
const tConfigParameters *const g_psFactoryParameters
```

Description:

This structure instance points to the factory default set of parameters in flash memory.

5.2.4.5 g_sParameters

Definition:

```
tConfigParameters g_sParameters
```

Description:

This structure instance contains the run-time set of configuration parameters for S2E module. This is the active parameter set and may contain changes that are not to be committed to flash.

5.2.4.6 g_usFirmwareVersion

Definition:

```
const unsigned short g_usFirmwareVersion
```

Description:

The version of the firmware. Changing this value will make it much more difficult for Texas Instruments support personnel to determine the firmware in use when trying to provide assistance; it should only be changed after careful consideration.

6 File System Module

Introduction 33
API Functions 33

6.1 Introduction

This set of functions provides an interface to the built-in flash file system used by the lwIP-based web server. In addition to providing handles to the normal flash-based files, “special” files can be handled in such a way to provide dynamic content to the web client.

The original data for the web server file system can be found in the `fs` directory. These files are converted into a single C file, `fsdata-s2e.c` which is then included in `fs_s2e.c`. The file system data file `fsdata-s2e.c` can be generated in one of two ways depending upon the available tools. If cygwin and Perl are available, the script `makefsdata.pl`, found in the `third_party/lwip-1.3.2/apps/httpserver_raw` directory can be run. Alternatively, if Perl is not installed, the Windows command line utility, `makefsfile` found in the `boards/rdk-s2e` directory can be run.

Starting in the `boards/rdk-s2e/ser2enet` directory, the syntax of these two commands is as follow:

```
perl ../../../third_party/lwip-1.3.2/apps/httpserver_raw/makefsdata fs fsdata-s2e.c
```

or

```
../makefsfile -i fs -o fsdata-s2e.c
```

These functions are contained in `boards/rdk-s2e/ser2enet/fs_s2e.c`.

6.2 API Functions

Functions

- void fs_close (struct fs_file *file)
- fs_file * fs_open (char *name)
- int fs_read (struct fs_file *file, char *buffer, int count)

6.2.1 Function Documentation

6.2.1.1 fs_close

Close an opened file designated by the handle.

Prototype:

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

Parameters:

file is the pointer to the file handle to be closed.

Description:

This function will free the memory associated with the file handle, and perform any additional actions that are required for closing this handle.

Returns:

None.

6.2.1.2 fs_open

Open a file and return a handle to the file.

Prototype:

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

Parameters:

name is the pointer to the string that contains the file name.

Description:

This function will check the file name against a list of files that require "special" handling. If the file name matches this list, then the file extensions will be enabled for dynamic file content generation. Otherwise, the file name will be compared against the list of names in in the built-in flash file system. If the file is not found, a NULL handle will be returned.

Returns:

Returns the pointer to the file handle if found, otherwise NULL.

6.2.1.3 fs_read

Read data from the opened file.

Prototype:

```
int
fs_read(struct fs_file *file,
        char *buffer,
        int count)
```

Parameters:

file is the pointer to the file handle to be read from.

buffer is the pointer to data buffer to be filled.

count is the maximum number of data bytes to be read.

Description:

This function will fill in the buffer with up to "count" bytes of data. If there is "special" processing required for dynamic content, this function will also handle that processing as needed.

Returns:

the number of data bytes read, or -1 if the end of the file has been reached.

7 Serial Port Module

Introduction 37
API Functions 37

7.1 Introduction

The serial driver provides a ring buffer structure as the interface between the UART hardware and the UART client (for example, telnet session). A simple API requiring only the UART port number (for example, 0, 1) is all that is provide at this time.

These functions are contained in `boards/rdk-s2e/ser2enet/serial.c`, with `boards/rdk-s2e/ser2enet/serial.h` containing the API definitions for use the remainder of the application.

7.2 API Functions

Functions

- unsigned long SerialGetBaudRate (unsigned long ulPort)
- unsigned char SerialGetDataSize (unsigned long ulPort)
- unsigned char SerialGetFlowControl (unsigned long ulPort)
- unsigned char SerialGetFlowOut (unsigned long ulPort)
- unsigned char SerialGetParity (unsigned long ulPort)
- unsigned char SerialGetStopBits (unsigned long ulPort)
- void SerialGPIOAIntHandler (void)
- void SerialGPIOBIntHandler (void)
- void SerialInit (void)
- void SerialPurgeData (unsigned long ulPort, unsigned char ucPurgeCommand)
- long SerialReceive (unsigned long ulPort)
- unsigned long SerialReceiveAvailable (unsigned long ulPort)
- void SerialSend (unsigned long ulPort, unsigned char ucChar)
- tBoolean SerialSendFull (unsigned long ulPort)
- void SerialSetBaudRate (unsigned long ulPort, unsigned long ulBaudRate)
- void SerialSetCurrent (unsigned long ulPort)
- void SerialSetDataSize (unsigned long ulPort, unsigned char ucDataSize)
- void SerialSetDefault (unsigned long ulPort)
- void SerialSetFactory (unsigned long ulPort)
- void SerialSetFlowControl (unsigned long ulPort, unsigned char ucFlowControl)
- void SerialSetFlowOut (unsigned long ulPort, unsigned char ucFlowValue)
- void SerialSetParity (unsigned long ulPort, unsigned char ucParity)
- void SerialSetStopBits (unsigned long ulPort, unsigned char ucStopBits)
- void SerialUART0IntHandler (void)
- void SerialUART1IntHandler (void)

7.2.1 Function Documentation

7.2.1.1 SerialGetBaudRate

Queries the serial port baud rate.

Prototype:

```
unsigned long
SerialGetBaudRate(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the UART configuration and return the currently configured baud rate for the selected port.

Returns:

The current baud rate of the serial port.

7.2.1.2 SerialGetDataSize

Queries the serial port data size.

Prototype:

```
unsigned char
SerialGetDataSize(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will read the UART configuration and return the currently configured data size for the selected port.

Returns:

None.

7.2.1.3 SerialGetFlowControl

Queries the serial port flow control.

Prototype:

```
unsigned char
SerialGetFlowControl(unsigned long ulPort)
```

Parameters:

ulPort is the serial port number to be accessed.

Description:

This function will return the currently configured flow control for the selected port.

Returns:
None.

7.2.1.4 SerialGetFlowOut

Gets the serial port flow control output signal.

Prototype:
```
unsigned char
SerialGetFlowOut(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function will set the flow control output pin to a specified value.

Returns:
Returns **SERIAL_FLOW_OUT_SET** or **SERIAL_FLOW_OUT_CLEAR**.

7.2.1.5 SerialGetParity

Queries the serial port parity.

Prototype:
```
unsigned char
SerialGetParity(unsigned long ulPort)
```

Parameters:
ulPort is the serial port number to be accessed.

Description:
This function will read the UART configuration and return the currently configured parity for the selected port.

Returns:
Returns the current parity setting for the port. This will be one of **SERIAL_PARITY_NONE**, **SERIAL_PARITY_ODD**, **SERIAL_PARITY_EVEN**, **SERIAL_PARITY_MARK**, or **SERIAL_PARITY_SPACE**.

7.2.1.6 SerialGetStopBits

Queries the serial port stop bits.

Prototype:
```
unsigned char
SerialGetStopBits(unsigned long ulPort)
```

Parameters:
ulPort is the serial port number to be accessed.

Description:
This function will read the UART configuration and return the currently configured stop bits for the selected port.

Returns:
None.

7.2.1.7 SerialGPIOAIntHandler

Handles the GPIO A interrupt for flow control (port 1).

Prototype:
```
void
SerialGPIOAIntHandler(void)
```

Description:
This function is called when the GPIO port A generates an interrupt. An interrupt will be generated when the inbound flow control signal changes levels (rising/falling edge). A notification function will be called to inform the corresponding telnet session that the flow control signal has changed.

Returns:
None.

7.2.1.8 SerialGPIOBIntHandler

Handles the GPIO B interrupt for flow control (port 0).

Prototype:
```
void
SerialGPIOBIntHandler(void)
```

Description:
This function is called when the GPIO port B generates an interrupt. An interrupt will be generated when the inbound flow control signal changes levels (rising/falling edge). A notification function will be called to inform the corresponding telnet session that the flow control signal has changed.

Returns:
None.

7.2.1.9 SerialInit

Initializes the serial port driver.

Prototype:
```
void
SerialInit(void)
```

Description:
This function initializes and configures the serial port driver.

Returns:
None.

7.2.1.10 SerialPurgeData

Purges the serial port data queue(s).

Prototype:
```
void
SerialPurgeData(unsigned long ulPort,
                unsigned char ucPurgeCommand)
```

Parameters:
ulPort is the serial port number to be accessed.
ucPurgeCommand is the command indicating which queue's to purge.

Description:
This function will purge data from the tx, rx, or both serial port queues.

Returns:
None.

7.2.1.11 SerialReceive

Receives a character from the UART.

Prototype:
```
long
SerialReceive(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function receives a character from the relevant port's UART buffer.

Returns:
Returns -1 if no data is available or the oldest character held in the receive ring buffer.

7.2.1.12 SerialReceiveAvailable

Returns number of characters available in the serial ring buffer.

Prototype:
```
unsigned long
SerialReceiveAvailable(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed.

Description:

This function will return the number of characters available in the serial ring buffer.

Returns:

The number of characters available in the ring buffer..

7.2.1.13 SerialSend

Sends a character to the UART.

Prototype:

```
void
SerialSend(unsigned long ulPort,
           unsigned char ucChar)
```

Parameters:

ulPort is the UART port number to be accessed.

ucChar is the character to be sent.

Description:

This function sends a character to the UART. The character will either be directly written into the UART FIFO or into the UART transmit buffer, as appropriate.

Returns:

None.

7.2.1.14 SerialSendFull

Checks the availability of the serial port output buffer.

Prototype:

```
tBoolean
SerialSendFull(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed.

Description:

This function checks to see if there is room on the UART transmit buffer for additional data.

Returns:

Returns the number of bytes available in the serial transmit ring buffer.

7.2.1.15 SerialSetBaudRate

Configures the serial port baud rate.

Prototype:

```
void
SerialSetBaudRate(unsigned long ulPort,
                  unsigned long ulBaudRate)
```

Parameters:

ulPort is the serial port number to be accessed.

ulBaudRate is the new baud rate for the serial port.

Description:

This function configures the serial port's baud rate. The current configuration for the serial port will be read. The baud rate will be modified, and the port will be reconfigured.

Returns:

None.

7.2.1.16 SerialSetCurrent

Configures the serial port according to the current working parameter values.

Prototype:

```
void
SerialSetCurrent(unsigned long ulPort)
```

Parameters:

ulPort is the UART port number to be accessed. Valid values are 0 and 1.

Description:

This function configures the serial port according to the current working parameters in g_sParameters.sPort for the specified port. The actual parameter set is then read back and g_sParameters.sPort updated to ensure that the structure is correctly synchronized with the hardware.

Returns:

None.

7.2.1.17 SerialSetDataSize

Configures the serial port data size.

Prototype:

```
void
SerialSetDataSize(unsigned long ulPort,
                  unsigned char ucDataSize)
```

Parameters:

ulPort is the serial port number to be accessed.

ucDataSize is the new data size for the serial port.

Description:
This function configures the serial port's data size. The current configuration for the serial port will be read. The data size will be modified, and the port will be reconfigured.

Returns:
None.

7.2.1.18 SerialSetDefault

Configures the serial port to a default setup.

Prototype:
```
void
SerialSetDefault(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function resets the serial port to a default configuration.

Returns:
None.

7.2.1.19 SerialSetFactory

Configures the serial port to the factory default setup.

Prototype:
```
void
SerialSetFactory(unsigned long ulPort)
```

Parameters:
ulPort is the UART port number to be accessed.

Description:
This function resets the serial port to a default configuration.

Returns:
None.

7.2.1.20 SerialSetFlowControl

Configures the serial port flow control option.

Prototype:
```
void
SerialSetFlowControl(unsigned long ulPort,
                     unsigned char ucFlowControl)
```

Parameters:

ulPort is the UART port number to be accessed.

ucFlowControl is the new flow control setting for the serial port.

Description:

This function configures the serial port's flow control. This function will enable/disable the flow control interrupt and the UART transmitter based on the value of the flow control setting and/or the flow control input signal.

Returns:

None.

7.2.1.21 SerialSetFlowOut

Sets the serial port flow control output signal.

Prototype:

```
void
SerialSetFlowOut(unsigned long ulPort,
                 unsigned char ucFlowValue)
```

Parameters:

ulPort is the UART port number to be accessed.

ucFlowValue is the value to program to the flow control pin. Valid values are **SERIAL_FLOW_OUT_SET** and **SERIAL_FLOW_OUT_CLEAR**.

Description:

This function will set the flow control output pin to a specified value.

Returns:

None.

7.2.1.22 SerialSetParity

Configures the serial port parity.

Prototype:

```
void
SerialSetParity(unsigned long ulPort,
                unsigned char ucParity)
```

Parameters:

ulPort is the serial port number to be accessed.

ucParity is the new parity for the serial port.

Description:

This function configures the serial port's parity. The current configuration for the serial port will be read. The parity will be modified, and the port will be reconfigured.

Returns:

None.

7.2.1.23 SerialSetStopBits

Configures the serial port stop bits.

Prototype:

```
void
SerialSetStopBits(unsigned long ulPort,
                  unsigned char ucStopBits)
```

Parameters:

ulPort is the serial port number to be accessed.

ucStopBits is the new stop bits for the serial port.

Description:

This function configures the serial port's stop bits. The current configuration for the serial port will be read. The stop bits will be modified, and the port will be reconfigured.

Returns:

None.

7.2.1.24 SerialUART0IntHandler

Handles the UART0 interrupt.

Prototype:

```
void
SerialUART0IntHandler(void)
```

Description:

This function is called when the UART generates an interrupt. An interrupt will be generated when data is received and when the transmit FIFO becomes half empty. The transmit and receive FIFOs are processed as appropriate.

Returns:

None.

7.2.1.25 SerialUART1IntHandler

Handles the UART1 interrupt.

Prototype:

```
void
SerialUART1IntHandler(void)
```

Description:

This function is called when the UART generates an interrupt. An interrupt will be generated when data is received and when the transmit FIFO becomes half empty. The transmit and receive FIFOs are processed as appropriate.

Returns:

None.

8 Telnet Port Module

Introduction 47
API Functions 47

8.1 Introduction

The telnet protocol (as defined by RFC854) is used to make the connection across the network. In its simplest form, a telnet client is simply a TCP connect to the appropriate port. Telnet interprets 0xff as a command indicator (known as the Interpret As Command, or IAC, byte). Consecutive IAC bytes are used to transfer an actual 0xff byte; thus, the only special processing required is to translate 0xff to 0xff 0xff when sending, and to translate 0xff 0xff to 0xff when receiving.

The WILL, WONT, DO, DONT option negotiation protocol is also implemented. This is a simple means of determining if capabilities are present, and for enabling or disabling features that do not require configuration. Through the use of this negotiation protocol, telnet clients and servers are able to easily match capabilities and avoid trying to configure features that are not shared by both ends of the connection (which would therefore result in the negotiation sequence being sent as actual data instead of being absorbed by the client or server).

In this implementation, only the SUPPRESS_GA and RFC 2217 options are supported; all other options are negatively responded to in order to prevent the client from trying to use them.

These functions are contained in `boards/rdk-s2e/ser2enet/telnet.c`, with `boards/rdk-s2e/ser2enet/telnet.h` containing the API definitions for use the remainder of the application.

8.2 API Functions

Data Structures

- tTelnetSessionData

Defines

- OPT_FLAG_DO_SUPPRESS_GA
- OPT_FLAG_SERVER
- OPT_FLAG_WILL_SUPPRESS_GA

Enumerations

- tRFC2217State
- tTCPState
- tTelnetState

Functions

- void TelnetClose (unsigned long ulSerialPort)
- unsigned short TelnetGetLocalPort (unsigned long ulSerialPort)
- unsigned short TelnetGetRemotePort (unsigned long ulSerialPort)
- void TelnetHandler (void)
- void TelnetInit (void)
- void TelnetListen (unsigned short usTelnetPort, unsigned long ulSerialPort)
- void TelnetNotifyLinkStatus (tBoolean bLinkStatusUp)
- void TelnetNotifyModemState (unsigned long ulPort, unsigned char ucModemState)
- void TelnetOpen (unsigned long ulIPAddr, unsigned short usTelnetRemotePort, unsigned short usTelnetLocalPort, unsigned long ulSerialPort)
- void TelnetWriteDiagInfo (char *pcBuffer, int iLen, unsigned char ucPort)

Variables

- unsigned long g_ulSystemTimeMS

8.2.1 Data Structure Documentation

8.2.1.1 tTelnetSessionData

Definition:

```
typedef struct
{
    tcp_pcb *pConnectPCB;
    tcp_pcb *pListenPCB;
    tTCPState eTCPState;
    tTelnetState eTelnetState;
    unsigned short usTelnetLocalPort;
    unsigned short usTelnetRemotePort;
    unsigned long ulTelnetRemoteIP;
    unsigned char ucFlags;
    unsigned long ulConnectionTimeout;
    unsigned long ulMaxTimeout;
    unsigned long ulSerialPort;
    pbuf *pBufQ[PBUF_POOL_SIZE];
    int iBufQRead;
    int iBufQWrite;
    pbuf *pBufHead;
    pbuf *pBufCurrent;
    unsigned long ulBufIndex;
    unsigned long ulLastTCPSendTime;
    tBoolean bLinkLost;
    unsigned char ucErrorCount;
    unsigned char ucReconnectCount;
    unsigned char ucConnectCount;
    err_t eLastErr;
```

```
}
tTelnetSessionData
```

Members:

pConnectPCB This value holds the pointer to the TCP PCB associated with this connected telnet session.

pListenPCB This value holds the pointer to the TCP PCB associated with this listening telnet session.

eTCPState The current state of the TCP session.

eTelnetState The current state of the telnet option parser.

usTelnetLocalPort The listen port for the telnet server or the local port for the telnet client.

usTelnetRemotePort The remote port that the telnet client connects to.

ulTelnetRemoteIP The remote address that the telnet client connects to.

ucFlags Flags for various options associated with the telnet session.

ulConnectionTimeout A counter for the TCP connection timeout.

ulMaxTimeout The max time for TCP connection timeout counter.

ulSerialPort This value holds the UART Port Number for this telnet session.

pBufQ This value holds an array of pbufs.

iBufQRead This value holds the read index for the pbuf queue.

iBufQWrite This value holds the write index for the pbuf queue.

pBufHead This value holds the head of the pbuf that is currently being processed (that has been popped from the queue).

pBufCurrent This value holds the actual pbuf that is being processed within the pbuf chain pointed to by the pbuf head.

ulBufIndex This value holds the offset into the payload section of the current pbuf.

ulLastTCPSendTime The amount of time passed since rx byte count has changed.

bLinkLost The indication that link layer has been lost.

ucErrorCount Debug and diagnostic counters.

ucReconnectCount

ucConnectCount

eLastErr The last error reported by lwIP while attempting to make a connection.

Description:

This structure is used holding the state of a given telnet session.

8.2.2 Define Documentation

8.2.2.1 OPT_FLAG_DO_SUPPRESS_GA

Definition:

```
#define OPT_FLAG_DO_SUPPRESS_GA
```

Description:

The bit in the flag that is set when the remote client has sent a DO request for SUPPRESS_GA and the server has accepted it.

8.2.2.2 OPT_FLAG_SERVER

Definition:

```
#define OPT_FLAG_SERVER
```

Description:

The bit in the flag that is set when a connection is operating as a telnet server. If clear, this implies that this connection is a telnet client.

8.2.2.3 OPT_FLAG_WILL_SUPPRESS_GA

Definition:

```
#define OPT_FLAG_WILL_SUPPRESS_GA
```

Description:

The bit in the flag that is set when the remote client has sent a WILL request for SUPPRESS_GA and the server has accepted it.

8.2.3 Enumeration Documentation

8.2.3.1 tRFC2217State

Description:

The possible states of the telnet COM-PORT option parser.

Enumerators:

STATE_2217_GET_COMMAND The telnet COM-PORT option parser is ready to process the first byte of data, which is the sub-option to be processed.

STATE_2217_GET_DATA The telnet COM-PORT option parser is processing data bytes for the specified command/sub-option.

STATE_2217_GET_DATA_IAC The telnet COM-PORT option parser has received an IAC in the data stream.

8.2.3.2 tTCPState

Description:

The possible states of the TCP session.

Enumerators:

STATE_TCP_IDLE The TCP session is idle. No connection has been attempted, nor has it been configured to listen on any port.

STATE_TCP_LISTEN The TCP session is listening (server mode).

STATE_TCP_CONNECTING The TCP session is connecting (client mode).

STATE_TCP_CONNECTED The TCP session is connected.

8.2.3.3 tTelnetState

Description:
The possible states of the telnet option parser.

Enumerators:
STATE_NORMAL The telnet option parser is in its normal mode. Characters are passed as is until an IAC byte is received.
STATE_IAC The previous character received by the telnet option parser was an IAC byte.
STATE_WILL The previous character sequence received by the telnet option parser was IAC WILL.
STATE_WONT The previous character sequence received by the telnet option parser was IAC WONT.
STATE_DO The previous character sequence received by the telnet option parser was IAC DO.
STATE_DONT The previous character sequence received by the telnet option parser was IAC DONT.
STATE_SB The previous character sequence received by the telnet option parser was IAC SB.
STATE_SB_IGNORE The previous character sequence received by the telnet option parser was IAC SB n, where n is an unsupported option.
STATE_SB_IGNORE_IAC The previous character sequence received by the telnet option parser was IAC SB n, where n is an unsupported option.
STATE_SB_RFC2217 The previous character sequence received by the telnet option parser was IAC SB COM-PORT-OPTION (in other words, RFC 2217).

8.2.4 Function Documentation

8.2.4.1 TelnetClose

Closes an existing Ethernet connection.

Prototype:
```
void
TelnetClose(unsigned long ulSerialPort)
```

Parameters:
ulSerialPort is the serial port associated with this telnet session.

Description:
This function is called when the the Telnet/TCP session associated with the specified serial port is to be closed.

Returns:
None.

8.2.4.2 TelnetGetLocalPort

Gets the current local port for a connection's telnet session.

Prototype:

```
unsigned short
TelnetGetLocalPort(unsigned long ulSerialPort)
```

Parameters:

ulSerialPort is the serial port associated with this telnet session.

Description:

This function returns the local port in use by the telnet session associated with the given serial port. If operating as a telnet server, this port is the port that is listening for an incoming connection. If operating as a telnet client, this is the local port used to connect to the remote server.

Returns:

None.

8.2.4.3 TelnetGetRemotePort

Gets the current remote port for a connection's telnet session.

Prototype:

```
unsigned short
TelnetGetRemotePort(unsigned long ulSerialPort)
```

Parameters:

ulSerialPort is the serial port associated with this telnet session.

Description:

This function returns the remote port in use by the telnet session associated with the given serial port. If operating as a telnet server, this function will return 0. If operating as a telnet client, this is the server port that the connection is using.

Returns:

None.

8.2.4.4 TelnetHandler

Handles periodic task for telnet sessions.

Prototype:

```
void
TelnetHandler(void)
```

Description:

This function is called periodically from the lwIP timer thread context. This function will handle transferring data between the UART and the the telnet sockets. The time period for this should be tuned to the UART ring buffer sizes to maintain optimal throughput.

Returns:

None.

8.2.4.5 TelnetInit

Initializes the telnet session(s) for the Serial to Ethernet Module.

Prototype:

```
void
TelnetInit(void)
```

Description:
This function initializes the telnet session data parameter block.

Returns:
None.

8.2.4.6 TelnetListen

Opens a telnet server session (listen).

Prototype:

```
void
TelnetListen(unsigned short usTelnetPort,
             unsigned long ulSerialPort)
```

Parameters:
usTelnetPort is the telnet port number to listen on.
ulSerialPort is the serial port associated with this telnet session.

Description:
This function establishes a TCP session in listen mode as a telnet server.

Returns:
None.

8.2.4.7 TelnetNotifyLinkStatus

Handles link status notification.

Prototype:

```
void
TelnetNotifyLinkStatus(tBoolean bLinkStatusUp)
```

Parameters:
bLinkStatusUp is the boolean indicate of link layer status.

Description:
This function should be called when the link layer status has changed.

Returns:
None.

8.2.4.8 TelnetNotifyModemState

Handles RFC2217 modem state notification.

Prototype:

```
void
TelnetNotifyModemState(unsigned long ulPort,
                       unsigned char ucModemState)
```

Parameters:

ulPort is the serial port for which the modem state changed.

ucModemState is the new modem state.

Description:

This function should be called by the serial port code when the modem state changes. If RFC2217 is enabled, a notification message will be sent.

Returns:

None.

8.2.4.9 TelnetOpen

Opens a telnet server session (client).

Prototype:

```
void
TelnetOpen(unsigned long ulIPAddr,
           unsigned short usTelnetRemotePort,
           unsigned short usTelnetLocalPort,
           unsigned long ulSerialPort)
```

Parameters:

ulIPAddr is the IP address of the telnet server.

usTelnetRemotePort is port number of the telnet server.

usTelnetLocalPort is local port number to connect from.

ulSerialPort is the serial port associated with this telnet session.

Description:

This function establishes a TCP session by attempting a connection to a telnet server.

Returns:

None.

8.2.4.10 TelnetWriteDiagInfo

Format a block of HTML containing connection diagnostic information.

Prototype:

```
void
TelnetWriteDiagInfo(char *pcBuffer,
                    int iLen,
                    unsigned char ucPort)
```

Parameters:

pcBuffer is a pointer to a buffer into which the diagnostic text will be written.

iLen is the length of the buffer pointed to by *pcBuffer*.

ucPort is the port number whose diagnostics are to be written. Valid values are 0 or 1.

Description:

This function formats a block of HTML text containing diagnostic information on a given port's telnet connection status into the supplied buffer.

Returns:

None.

8.2.5 Variable Documentation

8.2.5.1 g_ulSystemTimeMS

Definition:

```
unsigned long g_ulSystemTimeMS
```

Description:

Keeps track of elapsed time in milliseconds.

9 Universal Plug and Play Module

Introduction 57
API Functions 57

9.1 Introduction

The Universal Plug and Play (UPnP) module provides the functions necessary to support UPnP on the network interface.

These functions are contained in `boards/rdk-s2e/ser2enet/upnp.c`, with `boards/rdk-s2e/ser2enet/upnp.h` containing the API definitions for use the remainder of the application.

9.2 API Functions

Functions

- void UPnPHandler (unsigned long ulTimeMS)
- void UPnPInit (void)
- void UPnPStart (void)
- void UPnPStop (void)

9.2.1 Function Documentation

9.2.1.1 UPnPHandler

Handles Ethernet interrupt for UPnP sessions.

Prototype:

```
void
UPnPHandler(unsigned long ulTimeMS)
```

Parameters:

ulTimeMS is the absolute time (as maintained by the lwip handler) in ms.

Description:

This function should be called on a regular periodic basis to handle the various timers and process any buffers for the UPnP sessions.

Returns:

None.

9.2.1.2 UPnPInit

Initializes the UPnP session for the Serial to Ethernet Module.

Prototype:

```
void
UPnPInit(void)
```

Description:

This function initializes and configures the UPnP session for the module.

Returns:

None.

9.2.1.3 UPnPStart

Starts listening for UPnP requests.

Prototype:

```
void
UPnPStart(void)
```

Description:

This function sets up the two ports which listen for UPnP location and discovery requests.

Returns:

None.

9.2.1.4 UPnPStop

Broadcasts a byebye message and stop UPnP discovery.

Prototype:

```
void
UPnPStop(void)
```

Description:

This function broadcasts an SSDP byebye message indicating that the UPnP device is no longer available then frees resources associated with UPnP discovery and location.

Returns:

None.

10 Command Line Processing Module

Introduction 59
API Functions 59
Programming Example 61

10.1 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.

10.2 API Functions

Data Structures

- tCmdLineEntry

Defines

- CMDLINE_BAD_CMD
- CMDLINE_TOO_MANY_ARGS

Functions

- int CmdLineProcess (char *pcCmdLine)

Variables

- tCmdLineEntry g_sCmdTable[ ]

10.2.1 Data Structure Documentation

10.2.1.1 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.

10.2.2 Define Documentation

10.2.2.1 CMDLINE_BAD_CMD

Definition:

```
#define CMDLINE_BAD_CMD
```

Description:

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

10.2.2.2 CMDLINE_TOO_MANY_ARGS

Definition:

```
#define CMDLINE_TOO_MANY_ARGS
```

Description:

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

10.2.3 Function Documentation

10.2.3.1 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.

10.2.4 Variable Documentation

10.2.4.1 g_sCmdTable

Definition:

```
tCmdLineEntry g_sCmdTable[]
```

Description:

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

10.3 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));
}
```

11 Flash Parameter Block Module

Introduction 63
API Functions 63
Programming Example 65

11.1 Introduction

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

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

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

11.2 API Functions

Functions

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

11.2.1 Function Documentation

11.2.1.1 FlashPBGet

Gets the address of the most recent parameter block.

Prototype:

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

Description:

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

Returns:

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

11.2.1.2 FlashPBInit

Initializes the flash parameter block.

Prototype:

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

Parameters:

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

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

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

Description:

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

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

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

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

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

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

Returns:

None.

11.2.1.3 FlashPBSave

Writes a new parameter block to flash.

Prototype:

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

Parameters:

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

Description:

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

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

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

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

Returns:

None.

11.3 Programming Example

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

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

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

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

12 lwIP Wrapper Module

Introduction 67
API Functions 67
Programming Example 70

12.1 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.

12.2 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)

12.2.1 Function Documentation

12.2.1.1 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.

12.2.1.2 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.

12.2.1.3 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.

12.2.1.4 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.

12.2.1.5 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.

12.2.1.6 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.

12.2.1.7 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.

12.3 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);
}
```

13 Ring Buffer Module

Introduction 71
API Functions 71
Programming Example 77

13.1 Introduction

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

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

13.2 API Functions

Functions

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

13.2.1 Function Documentation

13.2.1.1 RingBufAdvanceRead

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

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

None.

13.2.1.2 RingBufAdvanceWrite

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

Prototype:

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

Parameters:

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

ulNumBytes is the number of bytes added to the buffer.

Description:

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

Returns:

None.

13.2.1.3 RingBufContigFree

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

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to check.

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

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

13.2.1.4 RingBufContigUsed

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

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

Returns:
Returns the number of contiguous bytes available.

13.2.1.5 RingBufEmpty

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

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

Parameters:
ptRingBuf is the ring buffer object to empty.

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

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

13.2.1.6 RingBufFlush

Empties the ring buffer.

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

Discards all data from the ring buffer.

Returns:

None.

13.2.1.7 RingBufFree

Returns number of bytes available in a ring buffer.

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to check.

Description:

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

Returns:

Returns the number of bytes available in the ring buffer.

13.2.1.8 RingBufFull

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

Prototype:

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

Parameters:

ptRingBuf is the ring buffer object to empty.

Description:

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

Returns:

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

13.2.1.9 RingBufInit

Initialize a ring buffer object.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be initialized.

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

ulSize is the size of the buffer in bytes.

Description:

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

Returns:

None.

13.2.1.10 RingBufRead

Reads data from a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be read from.

pucData points to where the data should be stored.

ulLength is the number of bytes to be read.

Description:

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

Returns:

None.

13.2.1.11 RingBufReadOne

Reads a single byte of data from a ring buffer.

Prototype:

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

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

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

Returns:
The byte read from the ring buffer.

13.2.1.12 RingBufSize

Return size in bytes of a ring buffer.

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

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

13.2.1.13 RingBufUsed

Returns number of bytes stored in ring buffer.

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

Parameters:
ptRingBuf is the ring buffer object to check.

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

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

13.2.1.14 RingBufWrite

Writes data to a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be written to.

pucData points to the data to be written.

ulLength is the number of bytes to be written.

Description:

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

Returns:

None.

13.2.1.15 RingBufWriteOne

Writes a single byte of data to a ring buffer.

Prototype:

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

Parameters:

ptRingBuf points to the ring buffer to be written to.

ucData is the byte to be written.

Description:

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

Returns:

None.

13.3 Programming Example

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

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

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

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

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

14 Simple Task Scheduler Module

Introduction 79
API Functions 79
Programming Example 84

14.1 Introduction

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

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

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

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

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

14.2 API Functions

Data Structures

- tSchedulerTask

Functions

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

- unsigned long SchedulerTickCountGet (void)

Variables

- tSchedulerTask g_psSchedulerTable[ ]
- unsigned long g_ulSchedulerNumTasks

14.2.1 Data Structure Documentation

14.2.1.1 tSchedulerTask

Definition:

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

Members:

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

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

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

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

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

Description:

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

14.2.2 Function Documentation

14.2.2.1 SchedulerElapsedTicksCalc

Returns the number of ticks elapsed between two times.

Prototype:

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

Parameters:

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

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

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

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

14.2.2.2 SchedulerElapsedTicksGet

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

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

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

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

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

14.2.2.3 SchedulerInit

Initializes the task scheduler.

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

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

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

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

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

Returns:
None.

14.2.2.4 SchedulerRun

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

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

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

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

Returns:
None.

14.2.2.5 SchedulerSysTickIntHandler

Handles the SysTick interrupt on behalf of the scheduler module.

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

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

Returns:
None.

14.2.2.6 SchedulerTaskDisable

Disables a task and prevents the scheduler from calling it.

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

Parameters:

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

Description:

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

Returns:

None.

14.2.2.7 SchedulerTaskEnable

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

Prototype:

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

Parameters:

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

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

Description:

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

Returns:

None.

14.2.2.8 SchedulerTickCountGet

Returns the current system time in ticks since power on.

Prototype:

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

Description:

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

Returns:

Tick count since last boot.

14.2.3 Variable Documentation

14.2.3.1 g_psSchedulerTable

Definition:

```
tSchedulerTask g_psSchedulerTable[]
```

Description:

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

14.2.3.2 g_ulSchedulerNumTasks

Definition:

```
unsigned long g_ulSchedulerNumTasks
```

Description:

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

14.3 Programming Example

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

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

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

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

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

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

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

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

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

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

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

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

```
    //
    SchedulerInit(TICKS_PER_SECOND);

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

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

15 Ethernet Software Update Module

Introduction 87
API Functions 87
Programming Example 89

15.1 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.

15.2 API Functions

Functions

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

15.2.1 Function Documentation

15.2.1.1 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.

15.2.1.2 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.

15.3 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();
    }
```

16 TFTP Server Module

Introduction 91
Usage 91
API Functions 94

16.1 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.

16.2 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.

16.2.0.3 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.

16.2.0.4 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.

16.2.0.5 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.

16.2.0.6 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.

16.3 API Functions

Data Structures

- _tTFTPConnection

Defines

- TFTP_BLOCK_SIZE

Enumerations

- tTFTPError

Functions

- void TFTPInit (tTFTPRequest pfnRequest)

16.3.1 Data Structure Documentation

16.3.1.1 _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.

16.3.2 Define Documentation

16.3.2.1 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.

16.3.3 Typedef Documentation

16.3.3.1 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.

16.3.4 Enumeration Documentation

16.3.4.1 tTFTPError

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

16.3.4.2 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).

16.3.5 Function Documentation

16.3.5.1 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.

17 Micro Standard Library Module

Introduction 99
API Functions 99
Programming Example 108

17.1 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.

17.2 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)

17.2.1 Data Structure Documentation

17.2.1.1 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.

17.2.2 Function Documentation

17.2.2.1 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.

17.2.2.2 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).

17.2.2.3 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.

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

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

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

The function will copy at most *ulSize* - 1 characters into the buffer *pcBuf*. One space is reserved in the buffer for the null termination character.

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

Returns:

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

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

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

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

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

Returns:

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

17.2.2.6 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

17.2.2.7 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*.

17.2.2.8 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*.

17.2.2.9 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*.

17.2.2.10 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*.

17.2.2.11 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*.

17.2.2.12 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*.

17.2.2.13 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.

17.2.2.14 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.

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

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

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

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

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

Returns:

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

17.3 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);
```

18 UART Standard IO Module

Introduction 111
API Functions 112
Programming Example 118

18.1 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.

18.1.1 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.

18.1.2 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.

18.2 API Functions

Functions

- void UARTEchoSet (tBoolean bEnable)
- void UARTFlushRx (void)
- void UARTFlushTx (tBoolean bDiscard)
- unsigned char UARTgetc (void)
- int UARTgets (char *pcBuf, unsigned long ulLen)
- int UARTPeek (unsigned char ucChar)
- void UARTprintf (const char *pcString,...)
- int UARTRxBytesAvail (void)
- void UARTStdioInit (unsigned long ulPortNum)
- void UARTStdioInitExpClk (unsigned long ulPortNum, unsigned long ulBaud)
- void UARTStdioIntHandler (void)
- int UARTTxBytesFree (void)
- int UARTwrite (const char *pcBuf, unsigned long ulLen)

18.2.1 Function Documentation

18.2.1.1 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.

18.2.1.2 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.

18.2.1.3 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.

18.2.1.4 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.

18.2.1.5 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.

18.2.1.6 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.

18.2.1.7 UARTprintf

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

Prototype:

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

Parameters:

pcString is the format string.

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

Description:

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

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

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

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

Returns:

None.

18.2.1.8 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.

18.2.1.9 UARTStdioInit

Initializes the UART console.

Prototype:
```
void
UARTStdioInit(unsigned long ulPortNum)
```

Parameters:
ulPortNum is the number of UART port to use for the serial console (0-2)

Description:
This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 115200, 8-N-1. An application wishing to use a different baud rate may call UARTStdioInitExpClk() instead of this function.

This function or UARTStdioInitExpClk() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:
None.

18.2.1.10 UARTStdioInitExpClk

Initializes the UART console and allows the baud rate to be selected.

Prototype:
```
void
UARTStdioInitExpClk(unsigned long ulPortNum,
                    unsigned long ulBaud)
```

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

Description:
This function will initialize the specified serial port to be used as a serial console. The serial parameters will be set to 8-N-1 and the bit rate set according to the value of the *ulBaud* parameter.

This function or UARTStdioInit() must be called prior to using any of the other UART console functions: UARTprintf() or UARTgets(). In order for this function to work correctly, SysCtlClockSet() must be called prior to calling this function. An application wishing to use 115,200 baud may call UARTStdioInit() instead of this function but should not call both functions.

It is assumed that the caller has previously configured the relevant UART pins for operation as a UART rather than as GPIOs.

Returns:
None.

18.2.1.11 UARTStdioIntHandler

Handles UART interrupts.

Prototype:
```
void
UARTStdioIntHandler(void)
```

Description:
This function handles interrupts from the UART. It will copy data from the transmit buffer to the UART transmit FIFO if space is available, and it will copy data from the UART receive FIFO to the receive buffer if data is available.

Returns:
None.

18.2.1.12 UARTTxBytesFree

Returns the number of bytes free in the transmit buffer.

Prototype:
```
int
UARTTxBytesFree(void)
```

Description:
This function, available only when the module is built to operate in buffered mode using **UART_BUFFERED**, may be used to determine the amount of space currently available in the transmit buffer.

Returns:
Returns the number of free bytes.

18.2.1.13 UARTwrite

Writes a string of characters to the UART output.

Prototype:
```
int
UARTwrite(const char *pcBuf,
          unsigned long ulLen)
```

Parameters:
pcBuf points to a buffer containing the string to transmit.
ulLen is the length of the string to transmit.

Description:

This function will transmit the string to the UART output. The number of characters transmitted is determined by the *ulLen* parameter. This function does no interpretation or translation of any characters. Since the output is sent to a UART, any LF (/n) characters encountered will be replaced with a CRLF pair.

Besides using the *ulLen* parameter to stop transmitting the string, if a null character (0) is encountered, then no more characters will be transmitted and the function will return.

In non-buffered mode, this function is blocking and will not return until all the characters have been written to the output FIFO. In buffered mode, the characters are written to the UART transmit buffer and the call returns immediately. If insufficient space remains in the transmit buffer, additional characters are discarded.

Returns:

Returns the count of characters written.

18.3 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