

Serial to Ethernet bridging application manual

REV A

Publication Date: 2014/1/16
XMOS © 2014, All Rights Reserved.



Table of Contents

1	Application Overview	3
1.1	Introduction	3
1.2	Feature list	3
1.2.1	Supported	3
1.2.2	Not supported	4
2	Hardware platforms	5
2.1	Hardware requirements	5
2.2	Hardware setup	5
3	System description	7
3.1	Software architecture	7
3.1.1	Cores	7
3.1.2	Buffering	8
3.1.3	Communication model	8
3.2	Software components used	9
3.3	Resource usage	9
4	Programming guide	11
4.1	Getting started	11
4.1.1	Installation	11
4.1.2	Flash the web pages and device configuration	12
4.2	Source code structure	13
4.2.1	Directory structure	13
4.2.2	Key files	14
4.3	Notes on application interfaces	14
4.3.1	UART configuration	14
4.3.2	Webserver	14
4.3.3	Flash interface	15
5	API	16
5.1	Configuration defines	16
5.2	Constants	18
5.3	Data structures	18
5.4	Configuration functions	19
5.5	Interface functions	21
5.6	Module functions	27
6	Using the application	29
6.1	S2E device discovery	29
6.1.1	Running the UDP test server	29
6.1.2	Discover the S2E devices on the network	29
6.1.3	Modify IP address of a particular S2E device	30
6.1.4	Modify IP address of all S2E devices to use DHCP server	30
6.2	Data communication using S2E device	31
6.2.1	UART serial port setup	31
6.2.2	Telnet client setup	31
6.3	Device configuration using web interface	32
6.4	Device configuration using telnet interface	32

6.4.1	UART configuration	32
6.4.2	Sample usage	33
7	References	34

1 Application Overview

IN THIS CHAPTER

- Introduction
 - Feature list
-

1.1 Introduction

The Serial to Ethernet application (referred to as S2E) firmware serves as a reference design to add Ethernet connectivity to any serial device. The solution offers flexibility in configuring multiple UART devices (upto 8 UARTs) to bridge them to Ethernet networks. Design utilizes some of the existing XMOS xSOFTip for Ethernet, XTCP and MultiUART components.

The Firmware mainly functions as bridge between serial and Ethernet data communication end points. This application includes telnet server to facilitate data communication from host applications via separate telnet sockets for each of the configured UARTs. It also provides an embedded web server and a dedicated telnet socket for UART configuration management. UDP mode device discovery feature is provided in order to discover and configure the S2E devices available in the network.

1.2 Feature list

1.2.1 Supported

- Design fits in a single tile (U8) device
- 10/100 Mbit Ethernet port
- Supports up to 8 serial ports at the following baud rates: 115200, 57600, 38400, 28800, 19200, 14400, 9600, 7200, 4800, 2400, 1200, 600, 300, 150
- Device discovery and device IP configuration management using UDP
- Web page for UART configuration
- Telnet server functionality; supports data transfer via telnet socket for each UART
- Flash support for IP and UART configuration, web page storage and retrieval
- Telnet mode UART configuration support
- CMOS/TTL level and RS232 level communication for UARTs

1.2.2 Not supported

- ▶ UART to UART communication (serial extender or pair configuration)
- ▶ UDP mode data transfer
- ▶ VirtualCOM port for the UARTs for Configuration

2 Hardware platforms

IN THIS CHAPTER

- ▶ Hardware requirements
 - ▶ Hardware setup
-

2.1 Hardware requirements

- ▶ XP-SKC-L2 sliceKIT 1V2 L2 core board
- ▶ XA-SK-E100 Ethernet 1V1 sliceCARD
- ▶ XA-SK-UART-8 OctoUART 1V0 sliceCARD
- ▶ xTAG-2 and XA-SK-XTAG2 adapter
- ▶ Ethernet cable
- ▶ Power supply 5V

2.2 Hardware setup

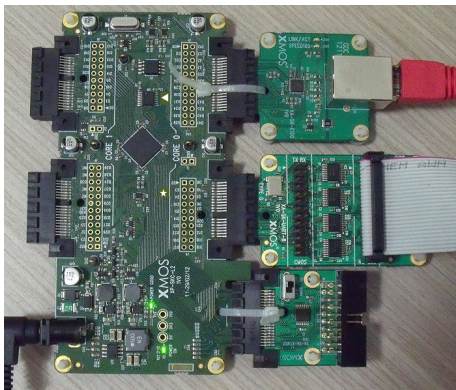
MultiUART component requires 8-bit ports for both transmit and receive ports. The current version of the *Serial to Ethernet* application runs on U8 (single tile). The sliceCARDs should be connected to the sliceKIT core board in the following manner:

sliceCARD	sliceKIT Connector	sliceKIT - Jumper
Ethernet	Triangle	J5
MultiUART	Star	J4

- ▶ CMOS TTL
- ▶ RS-232

By default, XA-SK-UART-8 sliceCARD uses the RS-232 levels. In order to use the CMOS TTL levels, short J3 pins (25-26) of the XA-SK-UART-8 sliceCARD. At a time, only one voltage level type can be used for all 8 UART channels (RS-232 or CMOS TTL). When using the RS-232 levels, UART device pins must be connected to J4 of the XA-SK-UART-8 sliceCARD. When using TTL levels, UART device pins must be connected to J3 of the XA-SK-UART-8 sliceCARD (along with J3 25-26 pins shorted). UART mapping information is as below:

Figure 1:
Hardware
setup



UART Channel	J3/J4 Pin no.(TX)	J3/J4 Pin no.(RX)
0	1	2
1	5	6
2	7	8
3	11	12
4	13	14
5	17	18
6	19	20
7	23	24

3 System description

IN THIS CHAPTER

- Software architecture
 - Software components used
 - Resource usage
-

This section briefly describes the software components used, thread diagram and resource usage details.

3.1 Software architecture

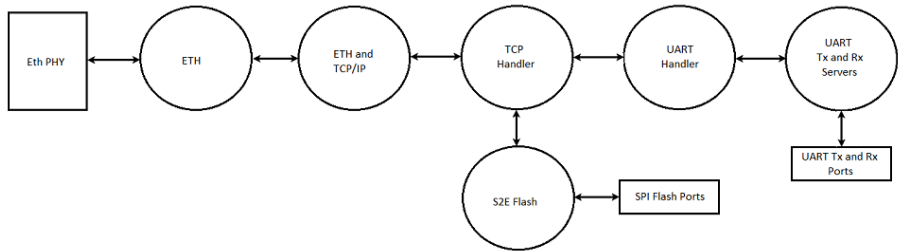


Figure 2:
Core usage

In order to achieve the desired data bridging, this application essentially maps each of the configured UART to a telnet socket and maintains application buffers (FIFOs) for each of such mapping. Whenever there is any UART data available, the application fills the appropriate UART buffer and notifies TCP client thread to consume this data. Similarly whenever there is any data from Ethernet packets, XTCP server notifies the application's TCP client about data availability and the client stores this data into respective application buffers. This data is then consumed by UART (client) handler.

3.1.1 Cores

The design uses seven logical cores as described below.

The MultiUART component comprises two logical cores, one acting as a transmit (TX) server for up to 8 uarts, and the other acting as a receive (RX) server for up to 8 uarts.

UART_handler is an application core that interfaces with the UART RX and TX servers. It handles UART configuration requests, facilitates to store the UART data received from UART RX server into respective application buffers and transfers the data received from TCP clients to UART TX server.

TCP_Handler is another application core that initializes and manages the application. It interfaces with the flash module for UART configuration storage and recovery, handles all the xtcp application events (application data and UI configuration requests) received from the xtcp server module. UDP discovery management, web server handling, telnet data extraction are all implemented in this logical core.

The XTCP server runs on a single logical core and connects to the Ethernet MAC component which uses a single logical core. It uses XC channel to communicate to clients (TCP_Handler in this case) using XTCP events.

The Flash core handles flash data storage and retrieval requests from TCP_handler core based on the application dynamics such as start-up or UI driven requests.

3.1.2 Buffering

Buffering for the TX server is managed by the UART_handler Core. Data is transferred to the UART TX logical core via a shared memory interface.

There is no buffering provided by the UART RX server. The UART_handler core is able to respond to the received data in real time and store them in buffer(s) available in the TCP_handler core via token notifications.

3.1.3 Communication model

The `sc_multi_uart` module utilises a combination of shared memory and channel communication. Channel communication is used on both the RX and TX servers to pause the logical core and subsequently release the logical core when required for reconfiguration. The primary means of data transfer for both the RX and TX logical cores is via shared memory. The RX logical core utilises a channel to notify any client (UART_handler in this case) of available data - this means that events can be utilised within an application to avoid the requirement for polling for received data.

XTCP module and flash core connects to TCP_handler client using their respective XC channels.

3.2 Software components used

Component	Description
sc_ethernet	Two thread version of the ethernet component implementing 10/100 MII Ethernet MAC and filters
sc_xtcp	Micro TCP/IP stack for use with sc_ethernet component
sc_multi_uart	Component for implementing multiple serial device communication
sc_util	General utility modules for developing for XMOS devices
sc_website	Component framework for Embedded web site development
sc_sliceKit_support	sliceKIT library to use L2 core board's flash for application
sc_otp	Library for reading MAC from sliceKIT core board's OTP memory

3.3 Resource usage

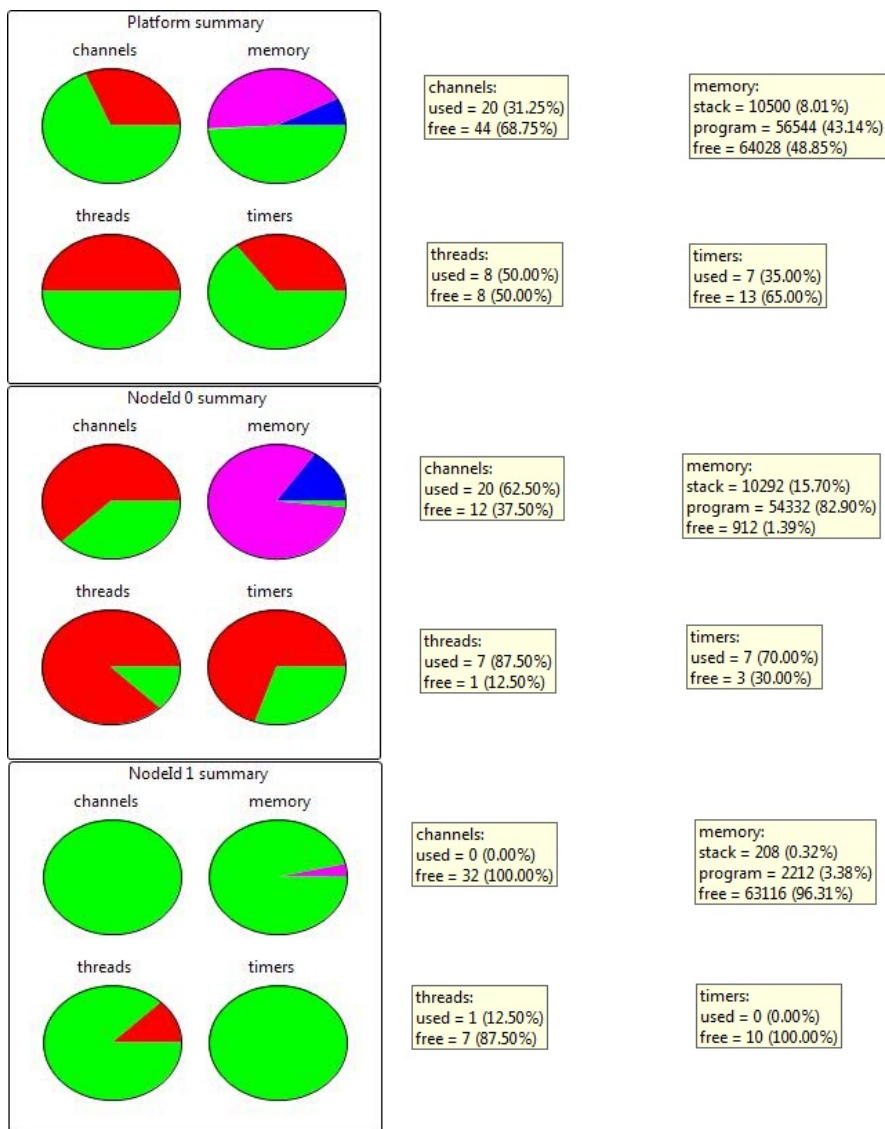


Figure 3:
Resource
usage

4 Programming guide

IN THIS CHAPTER

- ▶ Getting started
 - ▶ Source code structure
 - ▶ Notes on application interfaces
-

4.1 Getting started

4.1.1 Installation

- ▶ `sw_serial_to_ethernet`: `git://github.com/xcore/sw_serial_to_ethernet`
- ▶ `sc_multi_uart`: `git://github.com/xcore/sc_multi_uart.git`
- ▶ `sc_xtcp`: `git://github.com/xcore/sc_xtcp.git`
- ▶ `sc_ethernet`: `git://github.com/xcore/sc_ethernet.git`
- ▶ `sc_util`: `git://github.com/xcore/sc_util.git`
- ▶ `sc_website`: `git://github.com/xcore/sc_website.git`
- ▶ `sc_sliceit_support`: `git://github.com/xcore/sc_sliceit_support`
- ▶ `sc_otp`: `git://github.com/xcore/sc_otp`

All these components are packaged in the software distribution package (XM-004695-SM serial_to_ethernet_2.0.0). Once this zip file is selected, you can follow the below instructions to build and use the software.

4.1.1.1 Import and build procedure using xTIMEcomposer studio

To install the software, open the xTIMEcomposer (v13.0.0 or later) and follow these steps:

1. Open the xTIMEcomposer studio.
2. Open the *Edit* perspective (Window -> Open Perspective -> XMOS Edit).
3. Click *Import* option in the *Project Explorer* window (Import -> General -> Existing Projects into Workspace and click Next).
4. Choose *Select archive file* option and click *Browse* button.
5. Select s2e release zip file and click *Finish* button

6. The application is called as *app_serial_to_ethernet* in the *Project Explorer* window.

Build the serial to ethernet application:

1. Click on the *app_serial_to_ethernet* item in the *Project Explorer* window.
2. Click on the *Build* (indicated by a 'Hammer' picture) icon.
3. Check the *Console* window to verify that the application has built successfully.

4.1.2 Flash the web pages and device configuration

To flash the web pages and device configuration using xTIMEcomposer studio:

1. In the *Project Explorer* window, locate the *app_serial_to_ethernet.xe* and *web_data.bin* in the (*app_serial_to_ethernet* -> *bin*).
2. Right click on *app_serial_to_ethernet.xe* and click on (Flash As -> Flash Configurations...).
3. In the *Flash Configurations* window, double click the *xCORE Application* to create a new flash configuration.
4. Navigate to *XFlash Options* tab and apply the following settings:
 - ▶ Check *Boot partition size (bytes)*: and its value as 0x10000
 - ▶ *Other XFlash Options*: as -data bin/web_data.bin
5. Click on *Apply* and then *Flash* to the XMOS device.
6. Check the *Console* window to verify flashing progress.

4.1.2.1 Building from command line tool

To build from the command line, navigate to *app_serial_to_ethernet* directory and execute the command:

```
xmake all
```

In order to build the firmware with a static IP (say 169.254.196.178), execute the following command:

```
xmake all STATIC_IP=169.254.196.178
```

To flash the application, configuration and web pages, execute the command:

```
xmake flash
```

4.2 Source code structure

4.2.1 Directory structure

The source code is split into application source code and web pages. The application builds into a single executable using the source code from the modules. The modules used by an application are specified using the `USED_MODULES` variable in the application Makefile.

The source package contains:

Directory	Description
src	Source files that implement application functionality
web	Web (html) pages used by the web server module
images	Images that are displayed on the web (html) pages

Supported modules originate from other repositories:

Directory	Description	Repository
module_mii_singlethread	Lite version of low level ethernet interface over MII	sc_ethernet
module_ethernet_smi	A library code to communicate with ethernet phy using the SMI protocol	sc_ethernet
module_xtcp	XTCP TCP/IP stack	sc_xtcp
module_multi_uart	Multiple UART TX server and RX server	sc_multi_uart
module_xc_ptr	A library to allow XC code to access pointers via inline assembly calls.	sc_util
module_website	Embedded website component	sc_website
module_mutual_thread_comm	A protocol that allows two cores to communicate	sc_util

4.2.2 Key files

File	Description
<code>xtcp_client_conf.h</code>	Header file for clients of the TCP/IP stack.
<code>udp_discovery.h</code>	Header file for declaring UDP port, firmware version and function declarations required for s2e device discovery
<code>uart_config.h</code>	Header file containing declarations for UART data structures and interfacing with multi-uart server component
<code>web_server_conf.h</code>	Header file declaring all the functions called from the web pages
<code>telnet_config.h</code>	Header file for configuring UARTs via telnet socket
<code>multi_uart_rx_conf.h</code>	Header file for multiUART RX server configuration
<code>multi_uart_tx_conf.h</code>	Header file for multiUART TX server configuration
<code>s2e_conf.h</code>	Header file to enable debug options for s2e application

4.3 Notes on application interfaces

This section provides a brief description on main application interfaces.

4.3.1 UART configuration

The initialisation and configuration process for both the RX and TX operations is the same. The files `multi_uart_tx_conf.h` and `multi_uart_rx_conf.h` are used to configure multiUART TX and RX servers for the default values. For application configuration, the function `uart_config_init()` is used to apply configuration stored from flash or to use default application defined static configuration. The function `uart_set_config()` is utilised whenever there is a dynamic configuration change request (ie., a particular UART reconfiguration request). The flow is visualised in Figure 4.

4.3.2 Webserver

The webserver handles all HTTP requests from the web clients. A web client may request to change UART settings, save current settings, or apply the previously saved UART settings etc. Webserver identifies these requests, validates them and services those requests. It calls appropriate UART handler api's to retrieve and set channel settings. For example, for a 'Set' request from the web page, the webserver handler utilizes website component in order to parse the form data from web page, the required UART parameters are extracted and the UART_Handler's `uart_set_config` api is called to set the new UART configuration.

The Webserver interface utilises `sc_website` component to implement web server functionality. Refer to the Programming Guide section within the SliceKit GPIO Example Applications documentation linked

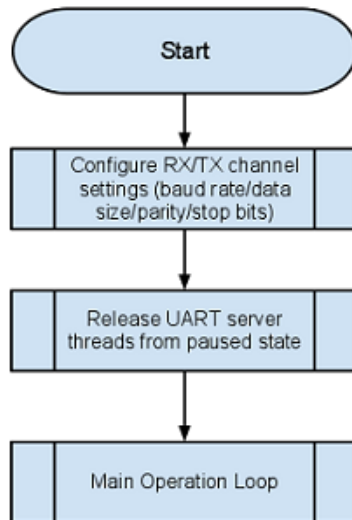


Figure 4:
UART
configuration
flow

from the front page documentation of `sw_gpio_examples` repository ([git://github.com/xcore/sw_gpio_examples.git](https://github.com/xcore/sw_gpio_examples.git)). This documentation contains more information on how to utilize the Embedded Webserver Function Library component in customizing web server applications.

4.3.3 Flash interface

- ▶ Request from web page (HTTP request)
- ▶ From Telnet configuration server
- ▶ Upon startup (to restore last saved settings)

IP configuration is saved via UDP server request and is requested from flash upon start-up.

5 API

IN THIS CHAPTER

- ▶ Configuration defines
 - ▶ Constants
 - ▶ Data structures
 - ▶ Configuration functions
 - ▶ Interface functions
 - ▶ Module functions
-

5.1 Configuration defines

The files `multi_uart_tx_conf.h` and `multi_uart_rx_conf.h` must be copied from `app_multi_uart_demo\src` folder to `app_serial_to_ethernet_demo\src` folder

UART_RX_CHAN_COUNT

Default value = 8

Define the number of channels that are to be supported, must fit in the port. Also, must be a power of 2 (i.e. 1,2,4,8) - not all channels have to be utilised

UART_TX_CHAN_COUNT

Default value = 8

Define the number of channels that are to be supported, must fit in the port. Also, must be a power of 2 (i.e. 1,2,4,8) - not all channels have to be utilised

UART_RX_MAX_BAUD

Default value = 115200

Define maximum baud rate to be supported for any of the UART channel

UART_TX_BUF_SIZE

Default value = 8

Define the buffer size in UART word entries - needs to be a power of 2 (i.e. 1,2,4,8,16,32 etc)

The file `xtcp_client_conf.h` defines configuration flexibility required for XTCP clients in order to use XTCP server functions

UIP_CONF_RECEIVE_WINDOW

Default value = 128

Define window length of TCP packets that the application will receive and process for each TCP packets received from XTCP clients. Note this value will be set as default length of application buffers that will be used to hold UART data collected from XTCP clients.

XTCP_CLIENT_BUF_SIZE

Default value = 650

Define MTU size used for XTCP device packet transmissions

The file `udp_discovery.h` defines ports used for UDP discovery. This file can set the following defines:

UDP_RECV_BUF_SIZE

Default value = 80

Define length of UDP message buffer which holds the incoming UDP test server request or its corresponding S2E response

INCOMING_UDP_PORT

Default value = 15534

Define incoming UDP port to listen to device discovery requests from UDP test server

OUTGOING_UDP_PORT

Default value = 15533

Define outgoing UDP port in order to send device response to UDP test server

S2E_FIRMWARE_VER

Current value = 1.1.2

Define to specify S2E firmware version. This shall be updated for every release

UART_RX_FLUSH_DELAY

Default value = 20000000

If UART data received is lesser than minimum configured packet size, this defines a minimum wait time to send this data to telnet handler

The file `telnet_config.h` defines ports used for Telnet socket used for UART configuration.

S2E_TELNET_CONFIG_PORT

Default value = 23

Define to specify telnet port to use for UART configuration

5.2 Constants

The file `srcs2e_flash.h` has some constants defined which may be used by other files while accessing flash.

S2E_FLASH_ERROR

Default value = -1

Any errors (like flash not present, wrong flash, unable to write, etc...) while accessing the flash fitted on the board, are represented using this constant.

S2E_FLASH_OK

Default value = 0

All successfull flash operations are represented using this constant.

UART_CONFIG

Default value = 0

To be used as a data type parameter while calling flash routines. This also represents the sector index (relative to webpage image) in the flash where UART configuration must be stored.

IPVER

Default value = 1

To be used as a data type parameter while calling flash routines. This also represents the sector index (relative to webpage image) in the flash where IP configuration must be stored.

FLASH_CMD_SAVE

Default value = 1

Flash command to 'save' configuration to flash.

FLASH_CMD_RESTORE

Default value = 2

Flash command to 'restore' configuration from flash.

FLASH_DATA_PRESENT

Default value = \$

While 'saving' settings to flash, this value is written as the first byte. So, on a 'restore' command, by reading for this sybol, we would know that some data is present in that sector of flash.

5.3 Data structures

uart_config_data_t

Structure to hold UART configuration details.

This structure has the following members:

int channel_id
UART Id.

e_uart_config_parity parity
One of valid parity: Odd, Even, Mark, Space, None.

e_uart_config_stop_bits stop_bits
Stop bits configuration: Single or Two.

e_uart_config_polarity polarity
Polarity setting: Start bit as 1 or 0.

int baud
Baud rate of UART channel.

int char_len
Number of bits each UART character contain.

5.4 Configuration functions

```
void uart_config_init(chanend c_uart_config,  
                     chanend ?c_flash_data,  
                     chanend ?c_xtcp,  
                     int &telnet_port_address)
```

This function retrieves configuration of each UARTs from flash, and applies it to the base configuration structure of type `uart_config_data_t`.

Telnet port applicable for corresponding UART is also fetched from flash and applied to repective UART configuration structure. If there is no valid flash configuration, default in-program configuration with following values EvenParity-1 StopBit-115200Baud-Start0Polarity-8UARTCharLen is used

This function has the following parameters:

c_uart_config
Channel-end to communicate UART configuration details between TCP handler and UART handler thread

c_flash_data
Channel-end to communicate UART configuration data stored in flash to TCP handler thread

c_xtcp
Channel-end between XTCP and TCP handler thread

telnet_port_address

Reference to structure holding Telnet port addresses mapped for UARTs

This function returns:

None

```
void s2e_webserver_init(chanend c_xtcp,  
                       chanend ?c_flash,  
                       chanend c_uart_config,  
                       chanend ?c_flash_data)
```

s2e_webserver_init The S2E webserver initialization routine.

Registers all channels used by it.

This function has the following parameters:

c_xtcp channel connecting to the xtcp module

c_flash channel for web page data

c_uart_config
 channel for UART configuration

c_flash_data
 channel for s2e flash data

This function returns:

none

```
void telnet_to_uart_init(chanend c_xtcp,  
                        chanend c_uart_data,  
                        int telnet_port_address[])
```

This function initializes UART state structure, assigns configured/default telnet ports and listens on these ports, shares UART buffer references to UART handler thread.

This function has the following parameters:

c_xtcp Channel-end between XTCP and TCP handler thread

c_uart_data Channel-end to communicate UART data between TCP handler and UART handler thread

telnet_port_address

Array containing telnet ports mapped for configured UARTs

This function returns:

None

```
void udp_discovery_init(chanend c_xtcp,  
                        chanend c_flash_data,  
                        xtcp_ipconfig_t &ipconfig)
```

This function initializes UDP discovery state, fetches S2E IP address stored in flash and references to ETH thread in order to configure device IP.

If valid IP is not present in flash, default configured IP defined from ipconfig is used

This function has the following parameters:

c_xtcp Channel between XTCP and TCP handler thread

c_flash_data Channel between Flash and TCP handler thread

ipconfig Reference to structure holding IP configuration info

This function returns:

None

5.5 Interface functions

```
void uart_handler(chanend c_uart_data,  
                  chanend c_uart_config,  
                  streaming chanend c_uart_rx,  
                  streaming chanend c_uart_tx)
```

This function configures UARTs, initializes application buffers states.

As a part of event handling, this function does the following: (i) handles incoming data from UARTs and stores into appropriate buffers (ii) in case of data transfer transaction, either notifies or acknowledges TCP handler about UART data; otherwise, listens from TCP handler to (a) collect telnet received data to UART (b) share appropriate UART buffer holding UART received data (iii) in case of UART configuration requests, collects appropriate UART configuration and reconfigures UART for received configuration (iv) sends a data byte from telnet UART buffers in a round robin basis, and notifies current transmit and receive transactions

This function has the following parameters:

c_uart_data	Channel-end to communicate UART data between TCP handler and UART handler thread
c_uart_config	Channel-end to communicate UART configuration details between TCP handler and UART handler thread
c_uart_rx	Channel primarily to send UART data from application to MultiUART RX server thread
c_uart_tx	Channel primarily to collect UART data from MultiUART TX server thread into UART handler thread

This function returns:

None

```
void tcp_handler(chanend c_xtcp,  
                chanend c_uart_data,  
                chanend c_uart_config,  
                chanend ?c_flash_web,  
                chanend ?c_flash_data)
```

This function handles TCP handler thread.

During initialization, IP configuration details stored in flash are retrieved from UDP discovery function and passed on to ETH thread. This thread mainly handles (a) all XTCP events and invokes respective sub-function handlers (b) notification messages from UART handler thread for UART data transactions.

This function has the following parameters:

c_xtcp	Channel-end between XTCP and TCP handler thread
c_uart_data	Channel-end to communicate UART data between TCP handler and UART handler thread
c_uart_config	Channel-end to communicate UART configuration data TCP handler and UART handler thread
c_flash_web	Channel-end to communicate Web page data stored in flash to TCP handler thread
c_flash_data	Channel-end to communicate UART configuration data stored in flash to TCP handler thread

This function returns:

None

```
void telnet_config_event_handler(chanend c_xtcp,  
                                chanend c_uart_config,  
                                chanend c_flash_data,  
                                xtcp_connection_t &conn)
```

This function handles UART configuration requests from telnet configuration client.

It initializes cofnig parse state machine, receives configuration request events, sends response back to the client

This function has the following parameters:

`c_xtcp` Channel-end between XTCP and TCP handler thread

`c_uart_config` Channel-end
to communicate UART configuration data TCP handler and UART
handler thread

`c_flash_data` Channel-end to communicate UART configuration data stored in
flash to TCP handler thread

`conn` Reference to structure holding IP configuration info

This function returns:

None

```
void s2e_webserver_event_handler(chanend c_xtcp,  
                                chanend ?c_flash,  
                                chanend c_uart_config,  
                                xtcp_connection_t &conn)
```

`s2e_webserver_event_handler` Handles webserver event.

This function has the following parameters:

`c_xtcp` channel connecting to the xtcp module

`c_flash` channel for web page data

`c_uart_config` channel for UART configuration

`c_flash_data` channel for s2e flash data

This function returns:

none

```
void telnet_to_uart_event_handler(chanend c_xtcp,  
                                chanend c_uart_data,  
                                xtcp_connection_t &conn)
```

This function handles XTCP events related to Telnet data communication.

For any event, a corresponding UART is mapped from XTCP connection. As a part of event handling, this function does the following: (i) for new connections, TCP connection details are stored and telnet parse state machine and TCP ack mode is initialized (ii) for receive events, data from TCP stack is collected into appropriate UART buffers. Received data is sent to telnet parser server in order to separate application data from telnet protocol. Initiates UART data transaction on c_uart_data channel in order to send data to UART (iii) for send requests initiated by telnet handler, this handler performs either of the following functionality (a) welcome messages are sent to respective telnet clients at the start of each session (b) collect outstanding data from appropriate UART RX active buffer and sends on XTCP connection

This function has the following parameters:

c_xtcp	Channel-end between XTCP and TCP handler thread
c_uart_data	Channel-end to communicate UART data between TCP handler and UART handler thread
conn	Reference to structure holding IP configuration info

This function returns:

None

```
void udp_discovery_event_handler(chanend c_xtcp,  
                                chanend c_flash_data,  
                                xtcp_connection_t &conn)
```

Handles events related to UDP discovery functionality.

Receives S2E identification and IP configuration requests from UDP test server, frames and sends S2E response messages

This function has the following parameters:

c_xtcp	Channel between XTCP and TCP handler thread
--------	---

`c_flash_data` Channel between Flash and TCP handler thread

`conn` Reference to UDP connection state

This function returns:

None

```
void s2e_flash(chanend c_flash_web,  
              chanend c_flash_data,  
              fl_SPIPorts &flash_ports)
```

`s2e_flash` The S2E flash thread will keep looking for data (or commands) on the `c_flash_data` channel.

This function has the following parameters:

`c_flash_web` channel for webpage data

`c_flash_data` channel for s2e data

`flash_ports` reference to flash ports used by the device

This function returns:

none

```
void send_data_to_flash_thread(chanend c_flash_data,  
                              uart_config_data_t &data)
```

`send_data_to_flash_thread` Send UART configuration data to flash.

Send one configuration at a time. In order to send configuration for all the channels, this routine must be called in a loop; each time sending the current channels config.

This function has the following parameters:

`c_flash_data` channel for s2e data

`data` reference to the current channel's config

This function returns:

none

```
void get_data_from_flash_thread(chanend c_flash_data,  
                               uart_config_data_t &data,  
                               int &telnet_port)
```

get_data_from_flash_thread Get UART configuration data from flash.

Get one configuration at a time. In order to get configuration for all the channels, this routine must be called in a loop; each time updating the current channels config. Telnet ports for each channel are also updated.

This function has the following parameters:

c_flash_data channel for s2e data

data reference to the current channel's config to update

telnet_port reference to current channel's telnet port to update

This function returns:

none

```
void send_ipconfig_to_flash_thread(chanend c_flash_data,  
                                   xtcp_ipconfig_t &ip)
```

send_ipconfig_to_flash_thread Send IP configuration data to flash.

This function has the following parameters:

c_flash_data channel for s2e data

ip reference to the current IP config

This function returns:

none

```
void get_ipconfig_from_flash_thread(chanend c_flash_data,  
                                    xtcp_ipconfig_t &ip)
```

get_ipconfig_from_flash_thread Get IP configuration data from flash.

This function has the following parameters:

c_flash_data channel for s2e data

ip reference to the current IP config

This function returns:

none

```
void send_cmd_to_flash_thread(chanend c_flash_data,
                             int data_type,
                             int command)
```

send_cmd_to_flash_thread Send command to flash thread.

This function has the following parameters:

c_flash_data channel for s2e data

data_type UART_CONFIG (or) IPVER

command FLASH_CMD_SAVE (or) FLASH_CMD_RESTORE

This function returns:

none

```
int get_flash_access_result(chanend c_flash_data)
```

get_flash_access_result Get the flash access result after performing certain command.

This function has the following parameters:

c_flash_data channel for s2e data

This function returns:

int S2E_FLASH_ERROR (or) S2E_FLASH_OK

5.6 Module functions

```
int parse_telnet_buffer(char data[],
                       int len,
                       int &parse_state,
                       int &close_request)
```

This function implements Telnet server functionality.

Mainly extracts application data from XTCP data packets in telnet protocol format.

This function has the following parameters:

`data` Buffer to store application data filtered from telnet server

`len` Number of bytes of received data

`parse_state` Current state of telnet parser state machine

`close_request` Flag to set when telnet client is suspended

This function returns:

Length of application buffer

6 Using the application

IN THIS CHAPTER

- ▶ S2E device discovery
 - ▶ Data communication using S2E device
 - ▶ Device configuration using web interface
 - ▶ Device configuration using telnet interface
-

This section details on how to use the application and its user interfaces.

6.1 S2E device discovery

S2E devices discovery on the network is performed by a UDP test server script (available at `sw_serial_to_ethernet -> tests -> udp_test_server`). This script needs to be executed on a host machine connected to a network router.

- ▶ Make sure the device is flashed with the firmware and web pages
- ▶ For Windows users, download *Serial_to_Ethernet_UDP_test_server* package (XM-004697-SM) and extract its contents to a directory. Navigate to (`udp_test_server -> windows -> udp_server.exe`), right-click on `udp-server.exe` and run as Administrator.
- ▶ For MAC or Linux users, it is recommended to install socket package for python, and run the script using the command `python udp_server.py`

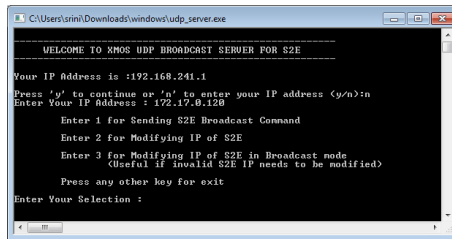
6.1.1 Running the UDP test server

1. Select an appropriate host specific option as described above
2. Script displays the selected network adapter on the console. If there are multiple network adapters on your host, ensure the ip address used by the script corresponds to the one used by your network adapter connected to the router
3. Script displays different options to choose as explained in the following sections:

6.1.2 Discover the S2E devices on the network

1. Key in option *1* from the choices.
2. Once the script is executed, it sends a broadcast request for all S2E devices in the network to respond. The message format is “XMOS S2E REPLY” broadcasted to 255.255.255.255

Figure 5:
S2E device
discovery
using
udp_test_server



3. XMOS S2E devices monitor this broadcast message and responds to the test server using the following format: "XMOS S2E VER:a.b.c;MAC:xx:xx:xx:xx:xx:xx;IP:abc.def.uvw.xyz"
4. The test server parses this response received from S2E devices available in the network and displays the following information on the console: VER -> Firmware Version MAC -> MAC Address IP -> IP address of the S2E device
5. The above information is displayed for all the S2E devices available in the network

6.1.3 Modify IP address of a particular S2E device

1. Key in option 2 from the choices.
2. The device discovery (option 1) should be used prior to using this option
3. Upon selecting the above option, ensure all available S2Es on the network are displayed
4. You can now select an appropriate S2E from the list and provide a new IP address for the selected S2E device. The server sends a unicast message using the format: "XMOS S2E IPCHANGE aaa.bbb.ccc.ddd"
5. Appropriate S2E device will receive this message, flash the new ip address, resets and starts with the new ip address
6. At the test server, you can now see S2E IP is changed to the new IP address by selecting the device discovery option again

6.1.4 Modify IP address of all S2E devices to use DHCP server

1. Key in option 3 from the choices.
2. This is a request and enables the s2e devices to DHCP mode. A DHCP server can be used to assign IP address to all S2E devices. The test server sends a broadcast message using the format: "XMOS S2E IPCHANGE 0.0.0.0"

3. It is important that only the intended S2Es for which the IP address is invalid should be made available in the network. All other S2Es should be removed from the network.
4. Once the S2E device's IP is changed to the DHCP assigned IP addresses, select the discovery option after some time in order to know the new IP addresses for the device(s).

6.2 Data communication using S2E device

Apart from the standard UART and Telnet clients available on the host, following tools may be installed on the host system in order to use the S2E application.

- ▶ For Win 7 users, Hercules Utility by HW-Group available at
▶ http://www.hw-group.com/products/hercules/index_en.html
- ▶ For MAC users, SecureCRT7.0 utility available at
▶ <http://www.vandyke.com/download/securecrt/>

The following example uses Hercules 3.2.5

6.2.1 UART serial port setup

1. Open the client application and change to Serial tab
2. Select appropriate options in the Serial pane. Apply the default settings (Data size = 8, Parity = Even, Handshake = Off, Mode = Free). Cross check these settings with the UART settings in the webpage.
3. Click Open

6.2.2 Telnet client setup

1. Open the client application
2. Switch to TCP Client tab
3. Key in the IP address (for e.g. 169.254.196.178) of the s2e device
4. Key in the port number configured for a particular UART (default configured values for each UART channel starts with 46)
5. Click Connect

Telnet client connection to the s2e server is now opened; now key in the data to be sent to a particular UART. Files can also be uploaded using this client by right-clicking (and selecting appropriate option) in the data pane of either sessions.

Software is tested for the following telnet clients

1. Putty

2. Hercules

6.3 Device configuration using web interface

1. Open the browser window

2. Key in the ip address (for e.g. <http://169.254.196.178/>) of the S2E device and press Enter.

Home page of the application appears

1. Click on a UART Channel to configure.

A new page for the selected channel appears with its settings. In order to change the UART parameters

1. Select UART parameters to change (Parity, Stop bits, Baud rate, Char Len or Telnet port)

2. Click Set.

3. If configuration is set successfully, the Response text will say 'Ok'

4. Click on Back to main config page to select a different UART channel or save the current settings to flash.

5. When clicked on Save in the main config page, current set configuration will be saved to flash. On successful save, the Response text will say 'Ok'

Software is tested for the following web browsers

1. Google Chrome

2. Mozilla Firefox

6.4 Device configuration using telnet interface

Telnet client can also be used for UART configuration or passing client data to UART channels (and vice versa). These are described as follows:

6.4.1 UART configuration

A separate telnet socket (default configured to port 23) is used for configuring UART channels via telnet client.

1. Open the telnet client (following example uses Hercules 3.2.5)

2. Switch to TCP Client tab

3. Key in the ip address (for e.g. 169.254.196.178)

4. Key in the port number (for UART config, it is 23)

5. Click **Connect**

UART configuration server's welcome message appears in the data pane of Telnet client

Use the following format for configuring an UART channel
~C~~P1~~P2~~P3~~P4~~P5~~P6~@

▶ ~ is the parameter separator

▶ @ is command termination marker

C [Command code]

1 : Get channel configuration for a particular channel 2 : Set channel configuration 3 : Save current configuration of all channels to flash 4 : Restore and set channel configuration from flash

▶ ▶ P1 : UART Channel Identifier (typical values range for 0 to 7)

P2 [Parity Configuration (typical values range for 0 to 4)]

0 : No Parity 1 : Odd Parity 2 : Even Parity 3 : Mark (always 1) parity bit 4 : Space (always 0) parity bit

P3 [Stop bits configuration (typical values are 0 or 1)]

0 : Single stop bit 1 : Two stop bits

P4 [Baud rate configuration. Typical values (bits per second) include]

115200 57600 38400 28800 19200 14400 9600 7200 4800 2400 1200 600 300 150

P5 [UART character length. Typical values include]

5 6 7 8 9

▶ ▶ ▶ ▶ ▶ P6 : Telnet port (typical values are 10 to 65536)

1. Click **Enter** to apply the configuration for the channel

6.4.2 Sample usage

Get: ~1~~0~@

Gets channel '0' configuration.

Set: ~2~~0~~2~~0~~115200~~8~~100~@

Sets channel '0' with: Even parity, single stop bits, 115200 baud, 8 character length and telnet port to communicate with this channel as 100.

Save: ~3~@

Save current set configuration of all channels to flash

Restore: ~4~@

Restores and sets channels configuration from flash

7 References

[XC09] Douglas Watt. Programming in XC on XMOS Devices. Xmos Ltd, 2009.

► http://www.xmos.com/published/xc_en

[ToolsUserGuide] Douglas Watt and Huw Geddes. The XMOS Tools User Guide. Xmos Ltd, 2010.

► http://www.xmos.com/published/xtools_en

[XM-001600-PC] Multi-UART Module Manual. Xmos Ltd, 2012.

► http://xcore.github.io/sc_multi_uart/



Copyright © 2014, All Rights Reserved.

Xmos Ltd. is the owner or licensee of this design, code, or Information (collectively, the "Information") and is providing it to you "AS IS" with no warranty of any kind, express or implied and shall have no liability in relation to its use. Xmos Ltd. makes no representation that the Information, or any particular implementation thereof, is or will be free from any claims of infringement and again, shall have no liability in relation to any such claims.

XMOS and the XMOS logo are registered trademarks of Xmos Ltd. in the United Kingdom and other countries, and may not be used without written permission. All other trademarks are property of their respective owners. Where those designations appear in this book, and XMOS was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.