XMOS TCP/IP STACK

Document Number: X5303A

Publication Date: 2012/3/14

SYNOPSIS

This documentation covers the xtcp component for XMOS Devices found at:

http://www.github.com/xcore/sc_xtcp

CONTENTS

1 Overview

IN THIS CHAPTER

The XMOS TCP/IP component provides a IP/UDP/TCP stack that connects to the XMOS ethernet component. It enables several clients to connect to it and send and receive on multiple TCP or UDP connections. The stack has been designed for a low memory embedded programming environment and despite its low memory footprint provides a complete stack including ARP, IP, UDP, TCP, DHCP, IPv4LL, ICMP and IGMP protocols.

The stack is based on the open-source stack uIP with modifications to work efficiently on XMOS architecture and communicate between threads using XC channels.

The TCP stack interfaces to the 5-thread XMOS ethernet stack via a pair of channels. Alternatively, an integrated 2-thread Ethernet plus TCP/IP is available for use in resource limited applications.

1.1 5 thread Ethernet plus separate TCP/IP stack properties

- Layer 2 packets can be sent and received independently of layer 3
- Integrated support for high priority Qtagged packets
- Integrated support for 802.1 Qav rate control
- Packet filtering in an independent threads
- Works on a 400 MHz part

1.2 Two thread ethernet plus integrated TCP/IP stack properties

- Uses only 2 threads
- High throughput
- Uses lower memory footprint
- Only TCP/IP sourced packets can be transmitted
- 500 MHz parts only (MII thread requires 62.5 MIPS)

1.3 Component Summary

Functionality	
Provides a lightweight IP/UDP/TCP stack	
Supported Standards	
IP, UDP, TCP, DHCP, IPv4LL, ICMP, IGMP	
Supported Devices	
Requirements	
XMOS Desktop Tools	v10.4 or later
XMOS Ethernet Component	2v0
Licensing and Support	
Component code provided without charge from XMOS. Component code is maintained by XMOS.	

2 Resource requirements

The resource requirements for the XTCP stack alone are:

Resource	Usage
Channels	1 per client
Memory	Between 20 and 45 KBytes
Timers	2
Clocks	0

When used with the single thread ethernet MII module, the combined usage is:

Resource	Usage
Channels	3 plus 1 per client
Memory	Between 26 and 50 KBytes
Timers	4
Clocks	1

The memory usage depends on the selection of different options at compile time, and on the amount of buffering chosen.

3 H/W Development Platforms

For initial development of ethernet applications the following XMOS development platforms can be used:

- XC-2 Ethernet Kit (http://www.xmos.com/products/development-kits/xc-2-ethernet-kit)
- XDK XS1-G Development Kit (http://www.xmos.com/products/development-kits/xs1-g-devel
- XP-DSC-BLDC Motor Control Platform (http://www.xmos.com/development-kits/motor-contro
- XK-AVB-LC-SYS AVB Audio Endpoint (http://www.xmos.com/products/reference-designs/avb
- XP-MC-CTRL-L2 Control Board Platform (http://www.xmos.com/development-kits/motor-cont

For developing an application specific board with ethernet please refer to the hardware guides for the above boards with example schematics, BOMs, design guidelines etc.

Note that the 2 thread version of the stack relies on the single thread MII ethernet component, which requires 62.5MIPS to run correctly. Consequently, the 2-thread version of the stack will not run on a 400MHz device.

4 TCP/IP Stack System Description

IN THIS CHAPTER

4.1 Software Architecture

The following Figure shows the architecture of the TCP/IP stack when attaching to an independent Ethernet stack through XC channel:

The server runs in a single thread and connects to the XMOS Ethernet MAC component (see [XEth10]). It can then connect to several client threads over XC channels.

Alternatively, the TCP/IP server and Ethernet server can be run as an integrated system of two threads. To enable this, the header file `uip_single_server.h` should be included, and the function uipSingleServer() used in place of the uip_server() and `ethernet_server` function from the ethernet repository. In addition, the `module_mii_singlethread` should be used instead of `module_ethernet` and `module_locks` in the Makefile. Finally, define the constant `UIP_USE_SINGLE_THREADED_ETHERNET` in your application's `xtcp_client_config.h` file.

4.2 IP Configuration

The server will determine its IP configuration based on the arguments passed into the uip_server() or uipSingleServer() function. If an address is supplied then that address will be used (a static IP address configuration).

If no address is supplied then the server will first try to find a DHCP server on the network to obtain an address automatically. If it cannot obtain an address from DHCP, it will determine a link local address (in the range 169.254/16) automatically using the Zeroconf IPV4LL protocol.

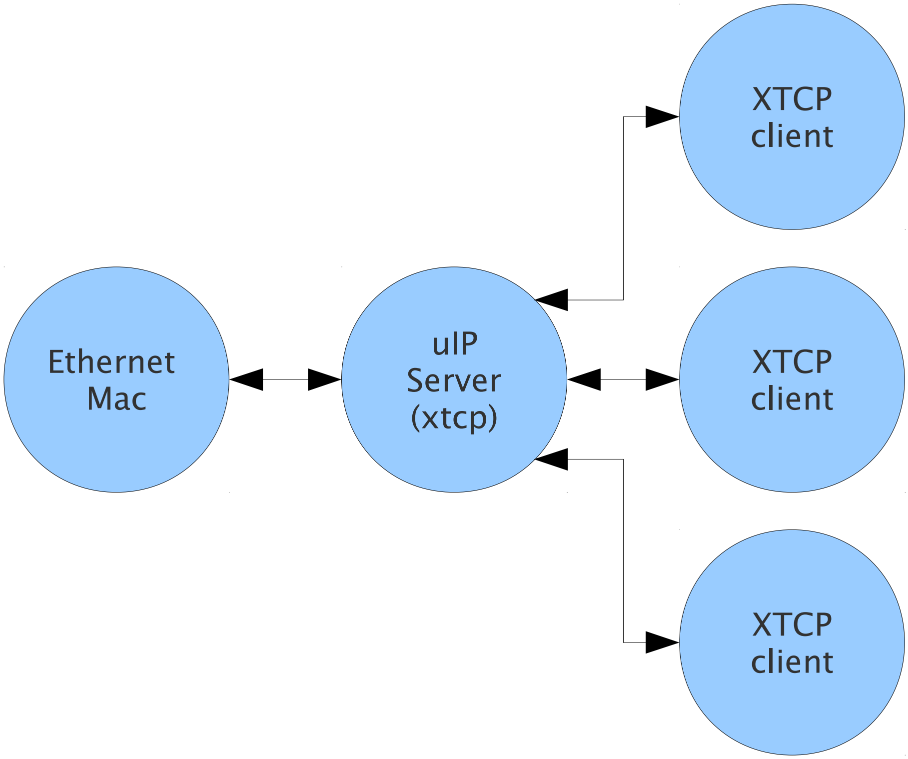

Figure 1: XTCP software architecture

To use dynamic address, the uip_server() can be passed a *null* to the ip configuration parameter. The uipSingleServer() must be passed a structure with 0 for all fields of the addresses.

4.3 Events and Connections

The TCP/IP stack client interface is a low-level event based interface. This is to allow applications to manage buffering and connection management in the most efficient way possible for the application.

Each client will receive *events* from the server. These events usually have an associated *connection*. In addition to receiving these events the client can send *commands* to the server to initiate new connections and so on.

Figure 2 shows an example event/command sequence of a client making a connection, sending some data, receiving some data and then closing the connection. Note that sending and receiving may be split into several events/commands since the server itself performs no buffering.

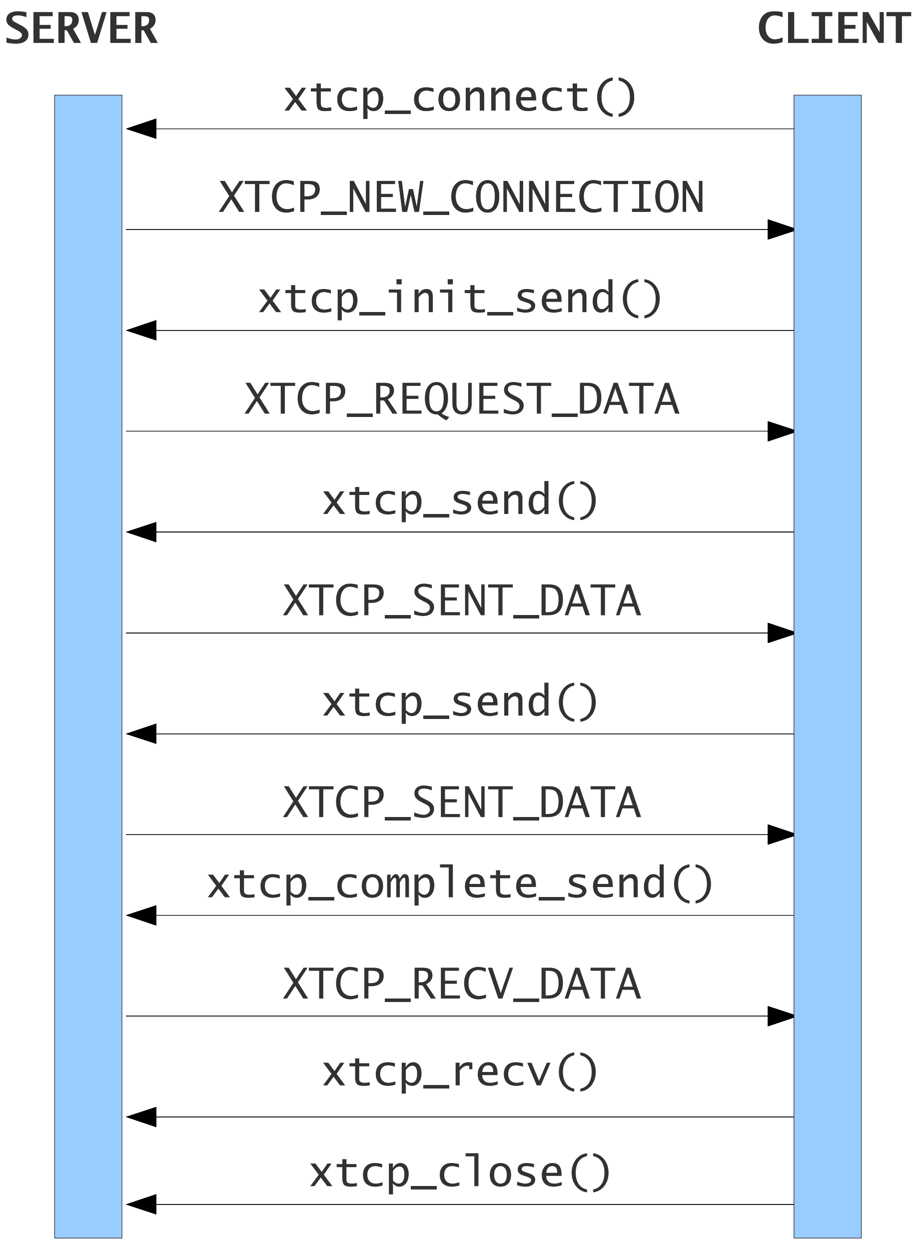

Figure 2: Example event sequence

If the client is handling multiple connections then the server may interleave events for each connection so the client has to hold a persistent state for each connection.

The connection and event model is the same from both TCP connections and UDP connections. Full details of both the possible events and possible commands can be found in Section §6.

4.4 TCP and UDP

The XTCP API treats UDP and TCP connections in the same way. The only difference is when the protocol is specified on initializing connections with xtcp_connect() or xtcp_listen().

4.5 New Connections

New connections are made in two different ways. Either the xtcp_connect() function is used to initiate a connection with a remote host as a client or the xtcp_listen() function is used to listen on a port for other hosts to connect to the application . In either case once a connection is established then the `XTCP_NEW_CONNECTION` event is triggered.

In the Berkley sockets API, a listening UDP connection merely reports data received on the socket, indepedent of the source IP address. In XTCP, a `XTCP_NEW_CONNECTION` event is sent each time data arrives from a new source. The API function xtcp_close() should be called after the connection is no longer needed.

4.6 Receiving Data

When data is received by a connection, the `XTCP_RECV_DATA` event is triggered and communicated to the client. At this point the client **must** call the xtcp_recv() function to receive the data.

Data is sent from host to client as the UDP or TCP packets come in. There is no buffering in the server so it will wait for the client to handle the event before processing new incoming packets.

As an alternative to the low level interface, a higher level buffered interface is available. See section §4.10.

4.7 Sending Data

When sending data, the client is responsible for dividing the data into chunks for the server and re-transmitting the previous chunk if a transmission error occurs.

Note that re-transmission may be needed on both TCP and UDP connections. On UDP connections, the transmission may fail if the server has not yet established a connection between the destination IP address and layer 2 MAC address.

The client can initiate a send transaction with the xtcp_init_send() function. At this point no sending has been done but the server is notified of a wish to send. The client must then wait for a `XTCP_REQUEST_DATA` event at which point it must respond with a call to xtcp_send().

After this data is sent to the server, two things can happen: Either the server will respond with an XTCP_SENT_DATA event, in which case the next chunk of data can be sent or with an XTCP_RESEND_DATA event in which case the client must re-transmit the previous chunk of data.

The command/event exchange continues until the client calls the xtcp_complete_send() function to finish the send transaction. After this the server will not trigger any more XTCP_SENT_DATA events.

4.8 Link Status Events

As well as events related to connections. The server may also send link status events to the client. The events XTCP_IFUP and XTCP_IFDOWN indicate to a client when the link goes up or down.

4.9 Configuration

The server is configured via arguments passed to the uip_server() function and the defines described in Section §6.1.

Client connections are configured via the client API described in Section §6.1.

4.10 Buffered API

As an alternative to the low level interface, a buffered interface is available as a utility layer.

To set up the buffered interface, the application must receive or make a new connection. As part of the new connection processing a buffer must be associated with it, by calling xtcp_buffered_set_rx_buffer() and xtcp_buffered_set_tx_buffer().

When sending using the buffered interface, a call to xtcp_buffered_send() is all that is required. When processing the XTCP_SENT_DATA, XTCP_REQUEST_DATA and XTCP_RESEND_DATA, the function xtcp_buffered_send_handler() should be called.

When processing a XTCP_RECV_DATA event, either the function xtcp_buffered_recv() or xtcp_buffered_recv_upto() can be called. These either return the data requested, or zero. If some data is returned, indicated by a non-zero return value, then the application should process the data, and call the receive function again. Only when the function returns zero can the application stop trying to receive and process the data.

Two example applications are provided. *app_buffered_protocol_demo* shows the use of the buffered API used with fixed length packets, and *app_buffered_protocol_demo_2* shows the use of the delimited token mechanism.

5 Programming Guide

IN THIS CHAPTER

5.1 Getting started

5.1.1 Installation

The xtcp component can be found at:

`http://www.github.com/xcore/sc_xtcp`

To install follow the instructions found on the community wiki at

`http://xcore.github.com`

5.1.2 Compilation and Demo Applications

The standalone XMOS TCP/IP package is supplied with several demo applications. These all start with the prefix `app_` and are described in the package README. This can be built in the XDE or from the command line by executing the command `xmake all`.

An example of the two-thread TCP plus Ethernet implementation can be found in the test_2_thread_example.

5.2 Source code structure

All the files for the stack are contains in the `module_xtcp` directory. The important header files that are used by applications are:

File	Description
`uip_server.h`	Header file containing prototype for uip_server().
`uip_single_server.h`	Header file containing prototype for uipSingleServer().
`xtcp_client.h`	Header file containing the client API described in §6.4.

5.3 An XTCP application (tutorial)

This tutorial walks through the a simple webserver application that uses the XMOS TCP/IP component. This can be found in the app_simple_webserver directory.

5.3.1 The toplevel main

The toplevel main of the application sets up the different components running on different threads on the device. It can be found in the file main.xc.

First the ethernet MAC is run on core 2. Details of the ethernet component can be found in [XEth10].

```
on stdcore[2]:
{
        int mac_address[2];

        ethernet_getmac_otp(otp_data, otp_addr, otp_ctrl,
                            (mac_address, char[]));

        phy_init(clk_smi,
#ifdef PORT_ETH_RST_N
                            p_mii_resetn,
#else
                            null,
#endif
                            smi, mii);

        ethernet_server(mii, mac_address,
                            mac_rx, 1, mac_tx, 1, smi,
                            connect_status);
}
```

The TCP/IP server is run using the uip_server() function.

```
on stdcore[3]: uip_server(mac_rx[0], mac_tx[0],
                 xtcp, 1, ipconfig,
                 connect_status);
```

Finally, the client to the TCP/IP server is run on a separate thread and connected to the TCP/IP server via the first element xtcp channel array. The function xhttpd implements the web server.

```
on stdcore[0]: xhttpd(xtcp[0]);
```

5.3.2 The webserver mainloop

The webserver is implemented in the xhttpd function in xhttpd.xc. This function implements a simple loop that just responds to events from the TCP/IP server. When an event occurs it is passed onto the httpd_handle_event handler.

```
void xhttpd(chanend tcp_svr)
{
  xtcp_connection_t conn;

  // Initiate the HTTP state
  httpd_init(tcp_svr);

  // Loop forever processing TCP events
  while(1)
    {
      select
        {
        case xtcp_event(tcp_svr, conn):
          httpd_handle_event(tcp_svr, conn);
          break;
        }
    }
}
```

5.3.3 The webserver event handler

The event handler is implemented in `httpd.c` and contains the main logic of the web server. The server can handle several connections at once. However, events for each connection may be interleaved so the handler needs to store separate state for each one. The `httpd_state_t` structures holds this state:

```
typedef struct httpd_state_t {
  int active;      //< Whether this state structure is being used
                   //  for a connection
  int conn_id;     //< The connection id
  char *dptr;      //< Pointer to the remaining data to send
  int dlen;        //< The length of remaining data to send
  char *prev_dptr; //< Pointer to the previously sent item of data
} httpd_state_t;

httpd_state_t connection_states[NUM_HTTPD_CONNECTIONS];
```

The `http_init` function is called at the start of the application. It initializes the connection state array and makes a request to accept incoming new TCP connections on port 80 (using the xtcp_listen() function):

```
void httpd_init(chanend tcp_svr)
{
  int i;
  // Listen on the http port
  xtcp_listen(tcp_svr, 80, XTCP_PROTOCOL_TCP);

  for ( i = 0; i < NUM_HTTPD_CONNECTIONS; i++ )
    {
      connection_states[i].active = 0;
      connection_states[i].dptr = NULL;
    }
```

```
}
```

When an event occurs the `httpd_handle_event` function is called. The behaviour of this function depends on the event type. Firstly, link status events are ignored:

```
void httpd_handle_event(chanend tcp_svr, xtcp_connection_t *conn)
{
  // We have received an event from the TCP stack, so respond
  // appropriately

  // Ignore events that are not directly relevant to http
  switch (conn->event)
    {
    case XTCP_IFUP:
    case XTCP_IFDOWN:
    case XTCP_ALREADY_HANDLED:
      return;
    default:
      break;
    }
```

For other events, we first check that the connection is definitely a http connection (is directed at port 80) and then call one of several event handlers for each type of event. The is a separate function for new connections, receiving data, sending data and closing connections:

```
if (conn->local_port == 80) {
  switch (conn->event)
    {
    case XTCP_NEW_CONNECTION:
      httpd_init_state(tcp_svr, conn);
      break;
    case XTCP_RECV_DATA:
      httpd_recv(tcp_svr, conn);
      break;
    case XTCP_SENT_DATA:
    case XTCP_REQUEST_DATA:
    case XTCP_RESEND_DATA:
        httpd_send(tcp_svr, conn);
        break;
    case XTCP_TIMED_OUT:
    case XTCP_ABORTED:
    case XTCP_CLOSED:
        httpd_free_state(conn);
        break;
    default:
      // Ignore anything else
      break;
    }
  conn->event = XTCP_ALREADY_HANDLED;
}
```

The following sections describe the four handler functions.

5.3.3.1 Handling Connections

When a XTCP_NEW_CONNECTION event occurs we need to associate some state with the connection. So the connection_states array is searched for a free state structure.

```
void httpd_init_state(chanend tcp_svr, xtcp_connection_t *conn)
{
  int i;

  // Try and find an empty connection slot
  for (i=0;i<NUM_HTTPD_CONNECTIONS;i++)
    {
      if (!connection_states[i].active)
        break;
    }
```

If we don't find a free state we cannot handle the connection so xtcp_abort()' is called to abort the connection.

```
if ( i == NUM_HTTPD_CONNECTIONS )
  {
    xtcp_abort(tcp_svr, conn);
  }
```

If we can allocate the state structure then the elements of the structure are initialized. The function xtcp_set_connection_appstate() is then called to associate the state with the connection. This means when a subsequent event is signalled on this connection the state can be recovered.

```
else
  {
    connection_states[i].active = 1;
    connection_states[i].conn_id = conn->id;
    connection_states[i].dptr = NULL;
    xtcp_set_connection_appstate(
         tcp_svr,
         conn,
         (xtcp_appstate_t) &connection_states[i]);
```

When a XTCP_TIMED_OUT, XTCP_ABORTED or XTCP_CLOSED event is received then the state associated with the connection can be freed up. This is done in the httpd_free_state function:

```
void httpd_free_state(xtcp_connection_t *conn)
{
  int i;
```

```
  for ( i = 0; i < NUM_HTTPD_CONNECTIONS; i++ )
    {
      if (connection_states[i].conn_id == conn->id)
        {
          connection_states[i].active = 0;
        }
    }
}
```

5.3.3.2 Receiving Data

When a XTCP_RECV_DATA event occurs the httpd_recv function is called. The first thing this function does is call the xtcp_recv() function to place the received data in the data array. (Note that all TCP/IP clients *must* call xtcp_recv() directly after receiving this kind of event).

```
void httpd_recv(chanend tcp_svr, xtcp_connection_t *conn)
{
  struct httpd_state_t *hs = (struct httpd_state_t *) conn->appstate;
  char data[XTCP_CLIENT_BUF_SIZE];
  int len;

  // Receive the data from the TCP stack
  len = xtcp_recv(tcp_svr, data);
```

The hs variable points to the connection state. This was recovered from the appstate member of the connection structure which was previously associated with application state when the connection was set up. As a safety check we only proceed if this state has been set up and the hs variable is non-null.

```
if ( hs == NULL || hs->dptr != NULL)
  {
    return;
  }
```

Now the connection state is known and the incoming data buffer filled. To keep things simple, this server makes the assumption that a single tcp packet gives us enough information to parse the http request. However, many applications will need to concatenate each tcp packet to a different buffer and handle data after several tcp packets have come in. The next step in the code is to call the parse_http_request function:

```
parse_http_request(hs, &data[0], len);
```

This function examines the incoming packet and checks if it is a GET request. If so, then it always serves the same page. We signal that a page is ready to the callee by setting the data pointer (dptr) and data length (dlen) members of the connection state.

```
void parse_http_request(httpd_state_t *hs, char *data, int len)
{
  // Return if we have data already
  if (hs->dptr != NULL)
    {
      return;
    }

  // Test if we received a HTTP GET request
  if (strncmp(data, "GET ", 4) == 0)
    {
      // Assign the default page character array as the data to send
      hs->dptr = &page[0];
      hs->dlen = strlen(&page[0]);
    }
  else
    {
      // We did not receive a get request, so do nothing
    }
}
```

The final part of the receive handler checks if the `parse_http_request` function set the `dptr` data pointer. If so, then it signals to the tcp/ip server that we wish to send some data on this connection. The actual sending of data is handled when an `XTCP_REQUEST_DATA` event is signalled by the tcp/ip server.

```
if (hs->dptr != NULL)
  {
    // Initate a send request with the TCP stack.
    // It will then reply with event XTCP_REQUEST_DATA
    // when it's ready to send
    xtcp_init_send(tcp_svr, conn);
  }
```

5.3.3.3 Sending Data

To send data the connection state keeps track of three variables:

Name	Description
`dptr`	A pointer to the next piece of data to send
`dlen`	The amount of data left to send
`prev_dptr`	The previous value of `dptr` before the last send

We keep the previous value of `dptr` in case the tcp/ip server asks for a resend.

On receiving a `XTCP_REQUEST_DATA`, `XTCP_SENT_DATA` or `XTCP_RESEND_DATA` event the function `httpd_send` is called.

The first thing the function does is check whether we have been asked to resend data. In this case it sends the previous amount of data using the `prev_dptr` pointer.

```
if (conn->event == XTCP_RESEND_DATA) {
  xtcp_send(tcp_svr, hs->prev_dptr, (hs->dptr - hs->prev_dptr));
  return;
}
```

If the request is for the next piece of data, then the function first checks that we have data left to send. If not, the function xtcp_complete_send() is called to finish the send transaction and then the connection is closed down with xtcp_close() (since HTTP only does one transfer per connection).

```
if (hs->dlen == 0 || hs->dptr == NULL)
  {
    // Terminates the send process
    xtcp_complete_send(tcp_svr);
    // Close the connection
    xtcp_close(tcp_svr, conn);
  }
```

If we have data to send, then first the amount of data to send is calculated. This is based on the amount of data we have left (`hs->dlen`) and the maximum we can send (`conn->mss`). Having calculated this length, the data is sent using the xtcp_send() function.

Once the data is sent, all that is left to do is update the `dptr`, `dlen` and `prev_dptr` variables in the connection state.

```
else {
  int len = hs->dlen;

  if (len > conn->mss)
    len = conn->mss;

  xtcp_send(tcp_svr, hs->dptr, len);

  hs->prev_dptr = hs->dptr;
  hs->dptr += len;
  hs->dlen -= len;
}
```

5.3.3.4 Converting to use the two thread version of the stack

In order to convert the application to use the two threaded version, the following changes would be made.

First, add the *UIP_USE_SINGLE_THREADED_ETHERNET* constant to the *xtcp_client_config.h* file found in the application's source code.

:: #define UIP_USE_SINGLE_THREADED_ETHERNET

Next, replace the 5-thread ethernet server, and the TCP/IP server, with a single call to the 2 thread stack.

:: on stdcore[0]: {

char mac_address[6];

ethernet_getmac_otp(otp_ports, mac_address);

// Bring PHY out of reset p_reset <: 0x2;

// Start server uipSingleServer(clk_smi, null, smi, mii, xtcp, 1, ipconfig, mac_address); }

All other parts of the system will remain the same, as the client-server interface between the XTCP server and the application remains the same.

6 API

IN THIS CHAPTER

6.1 Configuration Defines

The following defines can be set by adding the file `xtcp_client_conf.h` into your application and setting the defines within that file.

Define	Description
XTCP_CLIENT_BUF_SIZE	The buffer size used for incoming packets. This has a maximum valu
UIP_CONF_MAX_CONNECTIONS	The maximum number of UDP or TCP connections the server can han
UIP_CONF_MAX_LISTENPORTS	The maximum number of UDP or TCP ports the server can listen to s
UIP_USE_SINGLE_THREADED_ETHERNET	Defining this, and using the uipSingleServer function in the multithre
XTCP_EXCLUDE_LISTEN	Exclude support for the listen command from the server, reducing m
XTCP_EXCLUDE_UNLISTEN	Exclude support for the unlisten command from the server, reducing
XTCP_EXCLUDE_CONNECT	Exclude support for the connect command from the server, reducing
XTCP_EXCLUDE_BIND_REMOTE	Exclude support for the bind_remote command from the server, redu
XTCP_EXCLUDE_BIND_LOCAL	Exclude support for the bind_local command from the server, reduci
XTCP_EXCLUDE_INIT_SEND	Exclude support for the init_send command from the server, reducin
XTCP_EXCLUDE_SET_APPSTATE	Exclude support for the set_appstate command from the server, redu
XTCP_EXCLUDE_ABORT	Exclude support for the abort command from the server, reducing m
XTCP_EXCLUDE_CLOSE	Exclude support for the close command from the server, reducing m
XTCP_EXCLUDE_SET_POLL_INTERVAL	Exclude support for the set_poll_interval command from the server,
XTCP_EXCLUDE_JOIN_GROUP	Exclude support for the join_group command from the server, reduc
XTCP_EXCLUDE_LEAVE_GROUP	Exclude support for the leave_group command from the server, redu
XTCP_EXCLUDE_GET_MAC_ADDRESS	Exclude support for the get_mac_address command from the server,
XTCP_EXCLUDE_GET_IPCONFIG	Exclude support for the get_ipconfig command from the server, redu

Define	Description
XTCP_EXCLUDE_ACK_RECV	Exclude support for the ack_recv command from the server, reducing
XTCP_EXCLUDE_ACK_RECV_MODE	Exclude support for the ack_recv_mode command from the server, re

6.2 Data Structures/Types

`xtcp_ipaddr_t`

XTCP IP address.

This data type represents a single ipv4 address in the XTCP stack.

`xtcp_ipconfig_t`

IP configuration information structure.

This structure describes IP configuration for an ip node.

This structure has the following members:

`xtcp_ipaddr_t ipaddr`
The IP Address of the node.

`xtcp_ipaddr_t netmask`
The netmask of the node.

The mask used to determine which address are routed locally.

`xtcp_ipaddr_t gateway`
The gateway of the node.

`xtcp_protocol_t`

XTCP protocol type.

This determines what type a connection is: either UDP or TCP.

This type has the following values:

`XTCP_PROTOCOL_TCP`
Transmission Control Protocol.

`XTCP_PROTOCOL_UDP`
User Datagram Protocol.

`xtcp_event_type_t`

XTCP event type.

The event type represents what event is occuring on a particualr connection. It is instantiated when an event is received by the client using the xtcp_event() function.

This type has the following values:

`XTCP_NEW_CONNECTION`
This event represents a new connection has been made.

In the case of a TCP server connections it occurs when a remote host firsts makes contact with the local host. For TCP client connections it occurs when a stream is setup with the remote host. For UDP connections it occurs as soon as the connection is created.

`XTCP_RECV_DATA`
This event occurs when the connection has received some data.

The client *must* follow receipt of this event with a call to xtcp_recv() before any other interaction with the server.

`XTCP_REQUEST_DATA`
This event occurs when the server is ready to send data and is requesting that the client send data.

This event happens after a call to xtcp_init_send() from the client. The client *must* follow receipt of this event with a call to xtcp_send() before any other interaction with the server.

`XTCP_SENT_DATA`
This event occurs when the server has successfully sent the previous piece of data that was given to it via a call to xtcp_send().

The server is now requesting more data so the client must* follow receipt of this event with a call to xtcp_send() before any other interaction with the server.

`XTCP_RESEND_DATA`
This event occurs when the server has failed to send the previous piece of data that was given to it via a call to xtcp_send().

The server is now requesting for the same data to be sent again. The client must* follow receipt of this event with a call to xtcp_send() before any other interaction with the server.

`XTCP_TIMED_OUT`
This event occurs when the connection has timed out with the remote host (TCP only).

This event represents the closing of a connection and is the last event that will occur on an active connection.

`XTCP_ABORTED`
This event occurs when the connection has been aborted by the local or remote host (TCP only).

This event represents the closing of a connection and is the last event that will occur on an active connection.

`XTCP_CLOSED`
This event occurs when the connection has been closed by the local or remote host.

This event represents the closing of a connection and is the last event that will occur on an active connection.

`XTCP_POLL` This event occurs at regular intervals per connection.

Polling can be initiated and the interval can be set with xtcp_set_poll_interval()

`XTCP_IFUP` This event occurs when the link goes up (with valid new ip address).

This event has no associated connection.

`XTCP_IFDOWN` This event occurs when the link goes down.

This event has no associated connection.

`XTCP_ALREADY_HANDLED`
This event type does not get set by the server but can be set by the client to show an event has been handled.

`xtcp_connection_type_t`

Type representing a connection type.

This type has the following values:

`XTCP_CLIENT_CONNECTION`
A client connection.

`XTCP_SERVER_CONNECTION`
A server connection.

`xtcp_connection_t`

This type represents a TCP or UDP connection.

This is the main type containing connection information for the client to handle. Elements of this type are instantiated by the xtcp_event() function which informs the client about an event and the connection the event is on.

This structure has the following members:

`int id` A unique identifier for the connection.

`xtcp_protocol_t protocol`
The protocol of the connection (TCP/UDP).

`xtcp_connection_type_t connection_type`
The type of connection (client/sever).

`xtcp_event_type_t event`
The last reported event on this connection.

`xtcp_appstate_t appstate`
The application state associated with the connection.

This is set using the xtcp_set_connection_appstate() function.

`xtcp_ipaddr_t remote_addr`
The remote ip address of the connection.

`unsigned int remote_port`
The remote port of the connection.

`unsigned int local_port`
The local port of the connection.

`unsigned int mss`
The maximum size in bytes that can be send using xtcp_send() after a send event.

6.3 Server API

```
void uip_server(chanend mac_rx,
                chanend mac_tx,
                chanend xtcp[],
                int num_xtcp_clients,
                xtcp_ipconfig_t & ?ipconfig,
                chanend connect_status)
```

uIP based xtcp server.

This function implements an xtcp tcp/ip server in a thread. It uses a port of the uIP stack which is then interfaces over the xtcp channel array.

The IP setup is based on the ipconfig parameter. If this parameter is NULL then an automatic IP address is found (using dhcp or ipv4 link local addressing if no dhcp server is present). Otherwise it uses the ipconfig structure to allocate a static ip address.

The clients can communicate with the server using the API found in xtcp_client.h

This function has the following parameters:

`mac_rx` Rx channel connected to ethernet server

`mac_tx` Tx channel connected to ethernet server

`xtcp` Client channel array

`num_xtcp_clients`
The number of clients connected to the server

`ipconfig` An data structure representing the IP config (ip address, netmask and gateway) of the device. Leave NULL for automatic address allocation.

`connect_status`
This chanend needs to be connected to the connect status output of the ethernet mac.

```
void uipSingleServer(clock clk_smi,
                     out port ?p_mii_resetn,
                     smi_interface_t &smi,
                     mii_interface_t &mii,
                     chanend xtcp[],
                     int num_xtcp,
                     xtcp_ipconfig_t &ipconfig,
                     char mac_address[6])
```

The top level thread function for a single threaded XTCP server, interfacing to a single thread ethernet MII.

The xtcp_client_conf.h file should define the compile time flag UIP_USE_SINGLE_THREADED_ETHERNET in order to use this function.

This function should be used in a top level 'par' in the main function. It will internally create the single MII thread.

This function has the following parameters:

`clk_smi`	Clock block for controlling the SMI interface
`p_mii_resetn`	Optional port which resets the PHY
`smi`	Structure describing the SMI ports
`mii`	Structure describing the MII ports
`xtcp`	Array of client comms channels
`num_xtcp`	The number of TCP client channels
`ipconfig`	The configuration structure for the IP address
`mac_address`	The unicast MAC address for this device

6.4 Client API

6.4.1 Event Receipt

```
transaction xtcp_event(chanend c_xtcp, xtcp_connection_t &conn)
```

Receive the next connect event.

Upon receiving the event, the xtcp_connection_t structure conn is instatiated with information of the event and the connection it is on.

This can be used in a select statement.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	the connection relating to the current event

6.4.2 Setting Up Connections

```
void xtcp_listen(chanend c_xtcp, int port_number, xtcp_protocol_t proto)
```

Listen to a particular incoming port.

After this call, when a connection is established an XTCP_NEW_CONNECTION event is signalled.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`port_number`	the local port number to listen to
`proto`	the protocol to listen to (TCP or UDP)

```
void xtcp_unlisten(chanend c_xtcp, int port_number)
```

Stop listening to a particular incoming port.

Applies to TCP connections only.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`port_number`	local port number to stop listening on

```
void xtcp_connect(chanend c_xtcp,
                  int port_number,
                  xtcp_ipaddr_t ipaddr,
                  xtcp_protocol_t proto)
```

Try to connect to a remote port.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`port_number`	the remote port to try to connect to
`ipaddr`	the ip addr of the remote host
`proto`	the protocol to connect with (TCP or UDP)

```
void xtcp_bind_local(chanend c_xtcp,
                     xtcp_connection_t &conn,
                     int port_number)
```

Bind the local end of a connection to a particular port (UDP).

This function has the following parameters:

c_xtcp	chanend connected to the xtcp server
conn	the connection
port_number	the local port to set the connection to

```
void xtcp_bind_remote(chanend c_xtcp,
                      xtcp_connection_t &conn,
                      xtcp_ipaddr_t addr,
                      int port_number)
```

Bind the remote end of a connection to a particular port and ip address.

This is only valid for XTCP_PROTOCOL_UDP connections. After this call, packets sent to this connection will go to the specified address and port

This function has the following parameters:

c_xtcp	chanend connected to the xtcp server
conn	the connection
addr	the intended remote address of the connection
port_number	the intended remote port of the connection

```
void xtcp_set_connection_appstate(chanend c_xtcp,
                                  xtcp_connection_t &conn,
                                  xtcp_appstate_t appstate)
```

Set the connections application state data item.

After this call, subsequent events on this connection will have the appstate field of the connection set

This function has the following parameters:

c_xtcp	chanend connected to the xtcp server
conn	the connection
appstate	An unsigned integer representing the state. In C this is usually a pointer to some connection dependent information.

6.4.3 Receiving Data

`int xtcp_recv(chanend c_xtcp, char data[])`

Receive data from the server.

This can be called after an XTCP_RECV_DATA event.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`data`	A array to place the received data into

This function returns:

The length of the received data in bytes

`int xtcp_recvi(chanend c_xtcp, char data[], int i)`

Receive data from the xtcp server.

This can be called after an XTCP_RECV_DATA event.

The data is put into the array data starting at index i i.e. the first byte of data is written to data[i].

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`data`	A array to place the received data into
`i`	The index where to start filling the data array

This function returns:

The length of the received data in bytes

`int xtcp_recv_count(chanend c_xtcp, char data[], int count)`

Receive a number of bytes of data from the xtcp server.

This can be called after an XTCP_RECV_DATA event.

Data is pulled from the xtcp server and put into the array, until either there is no more data to pull, or until count bytes have been received. If there are more bytes to be received from the server then the remainder are discarded. The return value reflects the number of bytes pulled from the server, not the number stored in the buffer. From this the user can determine if they have lost some data.

see the buffer client protocol for a mechanism for receiving bytes without discarding the extra ones.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`data`	A array to place the received data into
`count`	The number of bytes to receive

This function returns:

The length of the received data in bytes, whether this was more or less than the requested amount.

6.4.4 Sending Data

`void xtcp_init_send(chanend c_xtcp, xtcp_connection_t &conn)`

Initiate sending data on a connection.

After making this call, the server will respond with a XTCP_REQUEST_DATA event when it is ready to accept data.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	the connection

`void xtcp_send(chanend c_xtcp, char NULLABLE data[], int len)`

Send data to the xtcp server.

Send data to the server. This should be called after a XTCP_REQUEST_DATA, XTCP_SENT_DATA or XTCP_RESEND_DATA event (alternatively xtcp_write_buf can be called). To finish sending this must be called with a length of zero or call the xtcp_complete_send() function.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`data`	An array of data to send
`len`	The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.

`void xtcp_sendi(chanend c_xtcp, char NULLABLE data[], int i, int len)`

Send data to the xtcp server.

Send data to the server. This should be called after a XTCP_REQUEST_DATA, XTCP_SENT_DATA or XTCP_RESEND_DATA event (alternatively xtcp_write_buf can be called). The data is sent starting from index i i.e. data[i] is the first byte to be sent. To finish sending this must be called with a length of zero.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp serve
`data`	An array of data to send
`i`	The index at which to start reading from the data array
`len`	The length of data to send. If this is 0, no data will be sent and a XTCP_SENT_DATA event will not occur.

```
void xtcp_complete_send(chanend c_xtcp)
```

Complete a send transaction with the server.

This function can be called after a XTCP_REQUEST_DATA, XTCP_SENT_DATA or XTCP_RESEND_DATA event to finish any sending on the connection that the event related to.

This function has the following parameters:

`c_xtcp`	chanend connected to the tcp server

6.4.5 Other Connection Management

```
void xtcp_set_poll_interval(chanend c_xtcp,
                            xtcp_connection_t &conn,
                            int poll_interval)
```

Set UDP poll interval.

When this is called then the udp connection will cause a poll event every poll_interval milliseconds.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	the connection
`poll_interval`	the required poll interval in milliseconds

```
void xtcp_close(chanend c_xtcp, xtcp_connection_t &conn)
```

Close a connection.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	the connection

```
void xtcp_abort(chanend c_xtcp, xtcp_connection_t &conn)
```

Abort a connection.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	the connection

`void xtcp_pause(chanend c_xtcp, xtcp_connection_t &conn)`

pause a connection.

No further reads and writes will occur on the network.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	tcp connection structure

`void xtcp_unpause(chanend c_xtcp, xtcp_connection_t &conn)`

unpause a connection

Activity is resumed on a connection.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`conn`	tcp connection structure

6.4.6 Other General Client Functions

`void xtcp_join_multicast_group(chanend c_xtcp, xtcp_ipaddr_t addr)`

Subscribe to a particular ip multicast group address.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`addr`	The address of the multicast group to join. It is assumed that this is a multicast IP address.

`void xtcp_leave_multicast_group(chanend c_xtcp, xtcp_ipaddr_t addr)`

Unsubscribe to a particular ip multicast group address.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`addr`	The address of the multicast group to leave. It is assumed that this is a multicast IP address which has previously been joined.

`void xtcp_get_mac_address(chanend c_xtcp, unsigned char mac_addr[])`

Get the current host MAC address of the server.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`mac_addr`	the array to be filled with the mac address

```
void xtcp_get_ipconfig(chanend c_xtcp, xtcp_ipconfig_t &ipconfig)
```

Get the IP config information into a local structure.

Get the current host IP configuration of the server.

This function has the following parameters:

`c_xtcp`	chanend connected to the xtcp server
`ipconfig`	the structure to be filled with the IP configuration information

6.4.7 High-level blocking client API

```
void xtcp_wait_for_ifup(chanend tcp_svr)
```

block until the xtcp interface has come up

This means, amongst other things, that it has acquired an IP address using whatever scheme was necessary

```
xtcp_connection_t xtcp_wait_for_connection(chanend tcp_svr)
```

Block until a connection attempt to is made.

```
int xtcp_write(chanend tcp_svr,
               xtcp_connection_t &conn,
               unsigned char buf[],
               int len)
```

Write a buffer of data to a TCP connection.

This is a blocking write of data to the given xtcp connection

This function has the following parameters:

`tcp_svr`	The xtcp control channel
`conn`	The xtcp server connection structure
`buf`	The buffer to write
`len`	The length of data to send

This function returns:

1 for success, 0 for failure

```
int xtcp_read(chanend tcp_svr,
```

```
xtcp_connection_t &conn,
unsigned char buf[],
int minlen)
```

Receive data from xtcp connection.

This is a blocking read from the xtcp stack

This function has the following parameters:

tcp_svr	The xtcp control channel
conn	The xtcp server connection structure
buf	The buffer to read into
minlen	The minimim length of data to receive

This function returns:

The number of bytes received

6.4.8 High-level buffered client API

```
void xtcp_buffered_set_rx_buffer(chanend tcp_svr,
                                 xtcp_connection_t *conn,
                                 xtcp_bufinfo_t *bufinfo,
                                 char *buf,
                                 int buflen)
```

set the location and size of the receiver buffer

This function has the following parameters:

tcp_svr	the xtcp server control channel
conn	a pointer to the xtcp connection info structure
bufinfo	a pointer to the buffered API control structure
buf	a pointer to the buffer to use for received data
buflen	the length of the receive buffer in bytes

```
void xtcp_buffered_set_tx_buffer(chanend tcp_svr,
                                 xtcp_connection_t *conn,
                                 xtcp_bufinfo_t *bufinfo,
                                 char *buf,
                                 int buflen,
                                 int lowmark)
```

set the location and size of the transmission buffer

the size of the buffer should probably be no smaller than XTCP_CLIENT_BUF_SIZE plus the maximum buffered message length. if it is, then buffer overflow can be detected and data will be lost.

This function has the following parameters:

`tcp_svr`	the xtcp server control channel
`conn`	a pointer to the xtcp connection info structure
`bufinfo`	a pointer to the buffered API control structure
`buf`	a pointer to the buffer to use for received data
`buflen`	the length of the receive buffer in bytes
`lowmark`	if the number of spare bytes in the buffer falls below this, TCP pauses the stream

```
int xtcp_buffered_recv(chanend tcp_svr,
                       xtcp_connection_t *conn,
                       xtcp_bufinfo_t *bufinfo,
                       char **buf,
                       int len,
                       int *overflow)
```

Pull a buffer of data out of the received data buffer.

This pulls a specified length of data from the data buffer. It is most useful for protocols where the packet format is known, or at least where variable sized data blocks are preceeded by a length field. A good example is DHCP.

When calling this in response to a XTCP_RECV_DATA event, and you must keep calling it until it returns zero.

The return value is either:

when the user wants to pull N bytes from the buffer, but less than N have been received into it, then the function returns zero. In this case, a calling function would typically not process further until another receive event was detected, indicating that there is some more data available in to read, and therefore that the number of bytes requested can now be fullfilled.

consider the data pointed to by the buf parameter to be read only. It points into the allocated buffer

This function has the following parameters:

`tcp_svr`	the xtcp server control channel

`conn`	a pointer to the xtcp connection info structure
`bufinfo`	a pointer to the buffered API control structure
`buf`	on return this points to the received data.
`len`	length of the buffer to receive into
`overflow`	pointer to an int which is set to non-zero if the buffer overflowed

This function returns:

the number of characters received in the buffer, or zero if we have used up all of the data, or the space available when receiving more data from xtcp would overflow the buffer

```
int xtcp_buffered_recv_upto(chanend tcp_svr,
                            xtcp_connection_t *conn,
                            xtcp_bufinfo_t *bufinfo,
                            char **buf,
                            char delim,
                            int *overflow)
```

Receive data from the receive buffer, up to a given delimiter character.

Many protocols, eg SMTP, FTP, HTTP, have variable length records with delimiters at the end of the record. This function can be used to fetch data from that type of data stream.

When calling this in response to a XTCP_RECV_DATA event, and you must keep calling it until it returns zero.

The returned length contains the delimiter

This function has the following parameters:

`tcp_svr`	the xtcp server control channel
`conn`	a pointer to the xtcp connection info structure
`bufinfo`	a pointer to the buffered API control structure
`buf`	on return this points to the received data.
`delim`	a character to receive data until
`overflow`	pointer to an int which is set to non-zero if the buffer overflowed

This function returns:

the number of characters in the returned data (including delimiter), or zero when there is nothing to receive, or the space available when receiving more data from xtcp would overflow the buffer

```
int xtcp_buffered_send(chanend tcp_svr,
                       xtcp_connection_t *conn,
                       xtcp_bufinfo_t *bufinfo,
                       char *buf,
                       int len)
```

Add more data to the send buffer.

This function has the following parameters:

`tcp_svr`	the xtcp server control channel
`conn`	a pointer to the xtcp connection info structure
`bufinfo`	a pointer to the buffered API control structure
`buf`	a buffer of data to queue for sending
`len`	the length of the data in the buffer

This function returns:

1 if the data was able to be buffered for send, 0 otherwise

```
void xtcp_buffered_send_handler(chanend tcp_svr,
                                xtcp_connection_t *conn,
                                xtcp_bufinfo_t *bufinfo)
```

The handler function for transmission requests from the xtcp stack.

When one of the following event types is received from the xtcp server channel then this method should be called.

XTCP_SENT_DATA XTCP_REQUEST_DATA XTCP_RESEND_DATA

This function has the following parameters:

`tcp_svr`	the xtcp server control channel
`conn`	a pointer to the xtcp connection info structure
`bufinfo`	a pointer to the buffered API control structure

```
int xtcp_buffered_send_buffer_remaining(xtcp_bufinfo_t *bufinfo)
```

Get the remaining amount of space in the send buffer.

A client can use this to determine whether the outgoing buffer has enough space to accept more data before the call to send that data is made.

This function has the following parameters:

`bufinfo`	a pointer to the buffered API control structure

This function returns:

the number of bytes remaining in the send buffer

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

[XEth10] XMOS Ethernet Component Design Guide. Xmos Ltd. 2010.

Table of Contents

X5303A