Application Note: AN00121

Using XMOS TCP/IP Library for UDP-based Networking

This application note demonstrates the use of XMOS TCP/IP stack on an XMOS multicore micro controller to communicate on an ethernet-based network.

The code associated with this application note provides an example of using the XMOS TCP/IP (XTCP) Library and the ethernet board support component to provide a communication framework. It demonstrates how to broadcast and receive text messages from and to the XMOS device in the network using the UDP stack of XTCP library. The XTCP library features low memory footprint but provides a complete stack of various protocols.

On an XMOS xCORE, all the endpoint activities are implemented as concurrent real-time processes allowing the network data to be placed on the wire or received from the wire with negligible latency. Moreover, unlike conventional interrupt-driven processors, the deterministic nature of event-driven XMOS processors meets the precise timing requirements of the real-time data transmission over networks.

Note: This application note requires an application to be run on the host machine to test the communication with the XMOS device.

Required tools and libraries

- xTIMEcomposer Tools - Version 14.0.0
- XMOS TCP/IP library - Version 4.0.0

Required hardware

This application note is designed to run on an XMOS xCORE General-Purpose device.

The example code provided with the application has been implemented and tested on the xCORE General-Purpose sliceKIT (XP-SKC-L2) with an ethernet sliceCARD (XA-SK-E100) but there is no dependancy on this board and it can be modified to run on any development board which uses an xCORE device.

Prerequisites

- This document assumes familiarity with the XMOS xCORE architecture, the XMOS tool chain and the xC language. Documentation related to these aspects which are not specific to this application note are linked to in the references appendix.
- For descriptions of XMOS related terms found in this document please see the *XMOS glossary*[1].
- For an overview of XTCP TCP/IP stack please see the *XMOS TCP/IP stack design guide*[2] for reference.

[1] http://www.xmos.com/published/glossary
[2] https://www.xmos.com/published/xmos-tcpip-stack-design-guide

1 Overview

1.1 Introduction

Transmission Control Protocol/Internet Protocol (TCP/IP) is an internetworking protocol that allows cross-platform or heterogeneous networking. It manages the flow of data in packets with headers giving the source and destination information. It provides a reliable stream delivery using sequenced acknowledgment and error detection technique. TCP/IP offers the *User Datagram Protocol* (UDP), a minimal transport protocol, wherein the packet headers contain just enough information to forward the datagrams and their error checking. UDP does not support flow control and acknowledgment.

1.2 XMOS TCP/IP (XTCP)

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, IPv4, IPv6, UDP, TCP, DHCP, IPv4LL, ICMP and IGMP protocols. The stack is based on the open-source uIP stack with modifications to work efficiently on the XMOS architecture and communicate between tasks using xC channels.

1.3 Block diagram

The application firmware organization is shown in Figure 1.

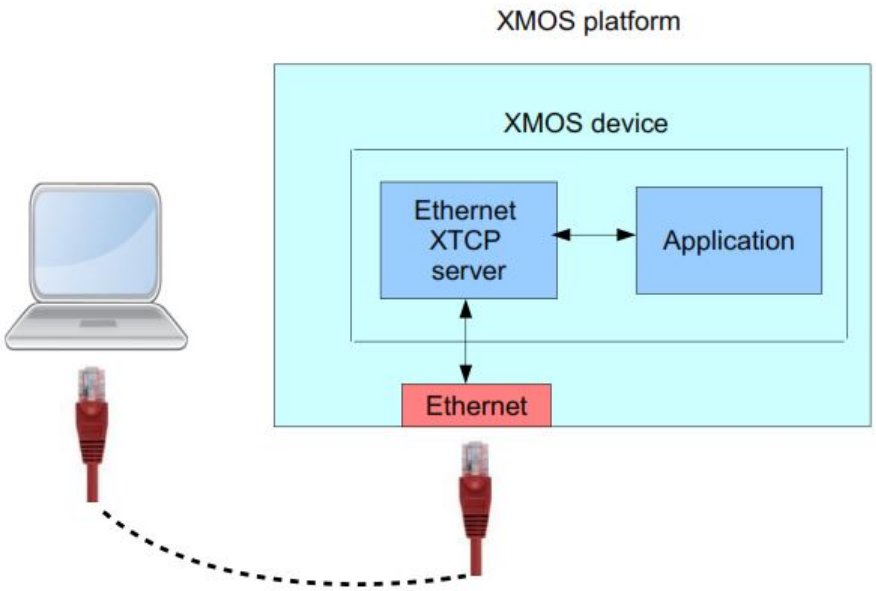

Figure 1: Networking with an XMOS device

2 The UDP example

The example in this note uses the XMOS TCP/IP library and shows a simple program that creates connections for data communication between the XMOS device and a computer in the network using UDP protocol.

To start using the XTCP library, you need to add `lib_xtcp` to your `Makefile`:

```
USED_MODULES = ... lib_xtcp ...
```

You can then access the XTCP functions in your source code via the xtcp.h header file:

```
#include <xtcp.h>
```

2.1 Allocating hardware resources

The TCP/IP stack connects to the MII task from the Ethernet library which requires several ports to communicate with the Ethernet PHY. These ports are declared in the main program file (`main.xc`). In this demo the ports are set up for the Ethernet slice connected to the CIRCLE slot of the sliceKIT:

```
// Here are the port definitions required by ethernet. This port assignment
// is for the L16 sliceKIT with the ethernet slice plugged into the
// CIRCLE slot.
port p_eth_rxclk  = on tile[1]: XS1_PORT_1J;
port p_eth_rxd    = on tile[1]: XS1_PORT_4E;
port p_eth_txd    = on tile[1]: XS1_PORT_4F;
port p_eth_rxdv   = on tile[1]: XS1_PORT_1K;
port p_eth_txen   = on tile[1]: XS1_PORT_1L;
port p_eth_txclk  = on tile[1]: XS1_PORT_1I;
port p_eth_int    = on tile[1]: XS1_PORT_1O;
port p_eth_rxerr  = on tile[1]: XS1_PORT_1P;
port p_eth_timing = on tile[1]: XS1_PORT_8C;

clock eth_rxclk   = on tile[1]: XS1_CLKBLK_1;
clock eth_txclk   = on tile[1]: XS1_CLKBLK_2;
```

Note that the port p_eth_dummy does not need to be connected to external hardware - it is just used internally by the Ethernet library.

The MDIO Serial Management Interface (SMI) is used to transfer management information between MAC and PHY. This interface consists of two signals which are connected to two ports:

```
port p_smi_mdio = on tile[1]: XS1_PORT_1M;
port p_smi_mdc  = on tile[1]: XS1_PORT_1N;
```

The final ports used in the application are the ones to access the internal OTP memory on the xCORE. These ports are fixed and can be intialized with the OTP_PORTS_INITIALIZER macro supplied by the `lib_otpinfo` OTP reading library.

```
// These ports are for accessing the OTP memory
otp_ports_t otp_ports = on tile[1]: OTP_PORTS_INITIALIZER;
```

2.2 IP configuration

The IP is configured via a structure passed to the xtcp task to suit the network where the XMOS device is used.

It is initialized to all zeros when DHCP/AutoIP is used.

```
// IP Config - change this to suit your network.  Leave with all
// 0 values to use DHCP/AutoIP
xtcp_ipconfig_t ipconfig = {
        { 0, 0, 0, 0 }, // ip address (eg 192,168,0,2)
        { 0, 0, 0, 0 }, // netmask (eg 255,255,255,0)
        { 0, 0, 0, 0 } // gateway (eg 192,168,0,1)
};
```

2.2.1 Application configuration

The final part of the application setup in main.xc is a set of defines that assigns port numbers to the broadcast and incoming ports and also assigns the broadcast address. The size of data buffer is also set. These defines are used by the application.

```
// Defines
#define RX_BUFFER_SIZE 300
#define INCOMING_PORT 15533
#define BROADCAST_INTERVAL 600000000
#define BROADCAST_PORT 15534
#define BROADCAST_ADDR {255,255,255,255}
#define BROADCAST_MSG "XMOS Broadcast\n"
```

2.3 The application main() function

For the UDP-based application example, the system comprises four tasks running on three separate logical cores of a xCORE General-Purpose multicore microcontroller.

The tasks perform the following operations.

- The MII task which handles MII/Ethernet traffic.
- The SMI task which drives the Ethernet PHY. This does not run on a logical core on its own.
- The ethernet TCP/IP server.
- The UDP reflector application running in a single core.

These tasks communicate via the use of xC channels and interface connections which allow data to be passed between application code running on separate logical cores.

Below is the source code for the main function of this application, which is taken from the source file `main.xc`

```
int main(void) {
  chan c_xtcp[1];
  mii_if i_mii;
  smi_if i_smi;
  par {
    // MII/ethernet driver
    on tile[1]: mii(i_mii, p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
                    p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
                    eth_rxclk, eth_txclk, XTCP_MII_BUFSIZE)

    // SMI/ethernet phy driver
    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    // TCP component
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);

    // The simple udp reflector thread
    on tile[0]: udp_reflect(c_xtcp[0]);

  }
  return 0;
}
```

Looking at this in more detail you can see the following:

- Four separate tasks are run in parallel within `par`
- There is a function call to execute the ethernet XTCP server: `xtcp()`
- There is a function to deal with XTCP events based on network protocol UDP for communication with a computer in the network: `udp_reflector()`
- The channels and interfaces to connect the tasks together are set up at the beginning of `main()`
- The `xtcp()` task is connected to the the `smi` and `mii` tasks. This is so it can drive the Ethernet PHY.
- The IP configuration is passed to the function `xtcp()`. It also takes the OTP ports so it can configure the MAC address based on reading the OTP memory of the device.

2.4 The UDP reflector

The application code for receiving UDP packets of data from a network machine and reflecting it back to the same machine is implemented in the file `main.xc`. Further, a fixed packet is sent periodically to a broadcast IP address. The code performing these tasks is contained within the function `udp_reflect()` which is shown in the following pages:

```
void udp_reflect(chanend c_xtcp)
{
  xtcp_connection_t conn;  // A temporary variable to hold
                           // connections associated with an event
  xtcp_connection_t responding_connection; // The connection to the remote end
                                           // we are responding to
  xtcp_connection_t broadcast_connection; // The connection out to the broadcast
                                          // address
  xtcp_ipaddr_t broadcast_addr = BROADCAST_ADDR;
  int send_flag = FALSE;  // This flag is set when the thread is in the
                      // middle of sending a response packet
  int broadcast_send_flag = FALSE; // This flag is set when the thread is in the
                               // middle of sending a broadcast packet
  timer tmr;
  unsigned int time;

  // The buffers for incoming data, outgoing responses and outgoing broadcast
  // messages
  char rx_buffer[RX_BUFFER_SIZE];
  char tx_buffer[RX_BUFFER_SIZE];
  char broadcast_buffer[RX_BUFFER_SIZE] = BROADCAST_MSG;

  int response_len;  // The length of the response the thread is sending
  int broadcast_len; // The length of the broadcast message the thread is
                     // sending


  // Maintain track of two connections. Initially they are not initialized
  // which can be represented by setting their ID to -1
  responding_connection.id = INIT_VAL;
  broadcast_connection.id = INIT_VAL;

  // Instruct server to listen and create new connections on the incoming port
  xtcp_listen(c_xtcp, INCOMING_PORT, XTCP_PROTOCOL_UDP);

  tmr :> time;
```

In this segment of the code you can see the following.

- The network connections, broadcast address and the buffers for holding the transmitted and received data are declared
- The XTCP server is instructed to listen and create new connections on the incoming port.

```
while (1) {
  select {
  // Respond to an event from the tcp server
  case xtcp_event(c_xtcp, conn):
    switch (conn.event)
      {
      case XTCP_IFUP:
        // When the interface goes up, set up the broadcast connection.
        // This connection will persist while the interface is up
        // and is only used for outgoing broadcast messages
        xtcp_connect(c_xtcp,
                     BROADCAST_PORT,
                     broadcast_addr,
                     XTCP_PROTOCOL_UDP);
        break;
              ..................
              ..................
              (the complete code is given in the Appendix)
              ..................
              ..................
       case XTCP_ALREADY_HANDLED:
        break;
    }
    break;
  // This is the periodic case, it occurs every BROADCAST_INTERVAL
  // timer ticks
  case tmr when timerafter(time + BROADCAST_INTERVAL) :> void:
    // A broadcast message can be sent if the connection is established
    // and one is not already being sent on that connection
    if (broadcast_connection.id != INIT_VAL && !broadcast_send_flag)  {
      printstrln("Sending broadcast message");
      broadcast_len = strlen(broadcast_buffer);
      xtcp_init_send(c_xtcp, broadcast_connection);
      broadcast_send_flag = TRUE;
    }
    tmr :> time;
    break;
  }
 }
}
```

You can see the following in the rest of the code.

- The `while` loop waits for an XTCP event and performs the appropriate function.
- The events pertaining to setting up and closing connections, reception of data and the request to send data.
- Periodical broadcast of a fixed data is done through timer events.

APPENDIX A - Demo Hardware Setup

1. Connect the xTAG-2 to XP-SKC-L2 sliceKIT through the xTAG-2 adaptor XA-SK-XTAG2.
2. Connect the xTAG-2 USB connector to the USB connector on your development PC using a USB cable.
3. Connect the ethernet slice XA-SK-E100 to the CIRCLE slot of the sliceKIT.
4. Connect the slice to the PC or to the network switch using an ethernet cable.
5. Connect the 12V power supply to the sliceKIT.
6. Switch on the power supply.

The demo code can be modified to run on any development board with an xCORE device and an ethernet connector.

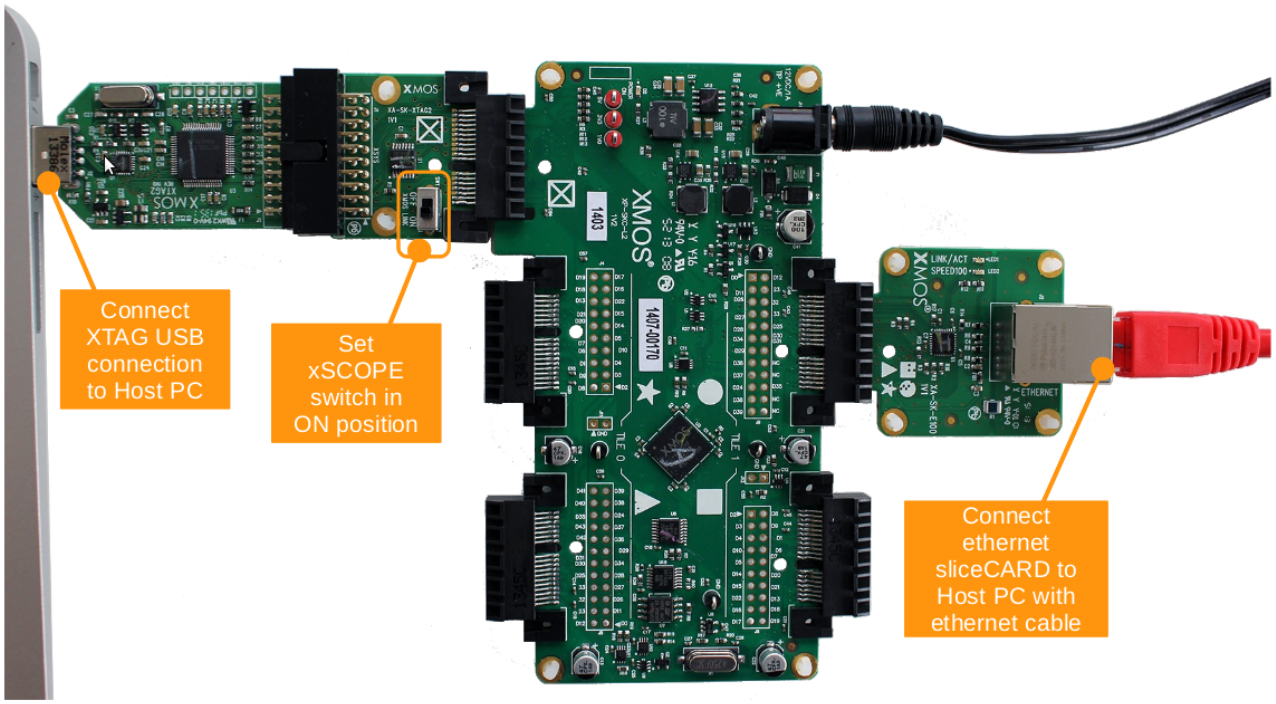

Figure 2: XMOS hardware setup for UDP demo

APPENDIX B - Launching the demo device

Once the application source code is imported into the tools xyou can then build the project which will generate the binary file required to run the demo application.

Once the application has been built you need to download the application binary code onto the xCORE General-Purpose sliceKIT. Here you use the tools to load the application over JTAG onto the xCORE multi-core microcontroller.

Then click **Run**. When the processor has finished booting you will see the following text in the xTIMEcomposer console window:

```
Address: 0.0.0.0
Gateway: 0.0.0.0
Netmask: 0.0.0.0
dhcp: 172.17.0.115
New broadcast connection established:1
Sending broadcast message
Sent Broadcast
Sending broadcast message
Sent Broadcast
```

The IP address assigned to the XMOS device through DHCP is displayed. The broadcast message sending is repeated periodically.

APPENDIX C - Host Application

C.1 Windows Host

On the host machine connected to the same network of the XMOS device,

- Download and run *Hercules terminal*[3]
- Click on the tab UDP
- Enter the IP address of the XMOS device under `Module IP`, incoming port number under `Port` and broadcast port number under `Local port`. Click on the `Listen` button.

The broadcast message `XMOS Broadcast` set in `main.xc` is displayed on the `Received data` space at regular intervals.

C.2 Testing data reception

On the Hercules terminal, type `Hello World` under Send and click on the Send button. You will see the following text in the xTIMEcomposer console window:

```
New connection to listening port:15533
Got data: 11 bytes
Responding
Sent Response
Closed connection:2
```

Since the demo code reflects the received data back to the sending machine, you can also see the `Hello World` displayed on the `Received data` space of the Hercules terminal.

The data reception can also be tested by running the python script `test_udp.py` instead of the Hercules terminal on the host machine. The python script is given below.

```
#!/usr/bin/python
import socket,sys

# This simple script sends a UDP packet to port 15533 at the
# IP address given as the first argument to the script
# This is to test the simple UDP example XC program

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

print ("Connecting..")
sock.connect((sys.argv[1], 15533))
print ("Connected")

msg = "hello world"
print ("Sending message: " + msg)
sock.send(bytes(msg,"utf-8"))

print ("Closing...")
sock.close()
print ("Closed")
```

[3] http://www.hw-group.com/products/hercules/index_en.html

APPENDIX D - References

XMOS Tools User Guide

http://www.xmos.com/published/xtimecomposer-user-guide

XMOS xCORE Programming Guide

http://www.xmos.com/published/xmos-programming-guide

XMOS TCP/IP stack design guide:

https://www.xmos.com/published/xmos-tcpip-stack-design-guide

APPENDIX E - Full source code listing

E.1 Source code for main.xc

```
// Copyright (c) 2016, XMOS Ltd, All rights reserved

#include <platform.h>
#include <string.h>
#include <print.h>
#include <xtcp.h>

// Here are the port definitions required by ethernet. This port assignment
// is for the L16 sliceKIT with the ethernet slice plugged into the
// CIRCLE slot.
port p_eth_rxclk  = on tile[1]: XS1_PORT_1J;
port p_eth_rxd    = on tile[1]: XS1_PORT_4E;
port p_eth_txd    = on tile[1]: XS1_PORT_4F;
port p_eth_rxdv   = on tile[1]: XS1_PORT_1K;
port p_eth_txen   = on tile[1]: XS1_PORT_1L;
port p_eth_txclk  = on tile[1]: XS1_PORT_1I;
port p_eth_int    = on tile[1]: XS1_PORT_1O;
port p_eth_rxerr  = on tile[1]: XS1_PORT_1P;
port p_eth_timing = on tile[1]: XS1_PORT_8C;

clock eth_rxclk   = on tile[1]: XS1_CLKBLK_1;
clock eth_txclk   = on tile[1]: XS1_CLKBLK_2;

port p_smi_mdio = on tile[1]: XS1_PORT_1M;
port p_smi_mdc  = on tile[1]: XS1_PORT_1N;

// These ports are for accessing the OTP memory
otp_ports_t otp_ports = on tile[1]: OTP_PORTS_INITIALIZER;

// IP Config - change this to suit your network.  Leave with all
// 0 values to use DHCP/AutoIP
xtcp_ipconfig_t ipconfig = {
        { 0, 0, 0, 0 }, // ip address (eg 192,168,0,2)
        { 0, 0, 0, 0 }, // netmask (eg 255,255,255,0)
        { 0, 0, 0, 0 } // gateway (eg 192,168,0,1)
};

// Defines
#define RX_BUFFER_SIZE 300
#define INCOMING_PORT 15533
#define BROADCAST_INTERVAL 600000000
#define BROADCAST_PORT 15534
#define BROADCAST_ADDR {255,255,255,255}
#define BROADCAST_MSG "XMOS Broadcast\n"
#define INIT_VAL -1

enum flag_status {TRUE=1, FALSE=0};

/** Simple UDP reflection thread.
 *
 * This thread does two things:
 *
 *   - Reponds to incoming packets on port INCOMING_PORT and
```

```
 *     with a packet with the same content back to the sender.
 *   - Periodically sends out a fixed packet to a broadcast IP address.
 *
 */
void udp_reflect(chanend c_xtcp)
{
  xtcp_connection_t conn;  // A temporary variable to hold
                           // connections associated with an event
  xtcp_connection_t responding_connection; // The connection to the remote end
                                           // we are responding to
  xtcp_connection_t broadcast_connection; // The connection out to the broadcast
                                          // address
  xtcp_ipaddr_t broadcast_addr = BROADCAST_ADDR;
  int send_flag = FALSE;  // This flag is set when the thread is in the
                      // middle of sending a response packet
  int broadcast_send_flag = FALSE; // This flag is set when the thread is in the
                               // middle of sending a broadcast packet
  timer tmr;
  unsigned int time;

  // The buffers for incoming data, outgoing responses and outgoing broadcast
  // messages
  char rx_buffer[RX_BUFFER_SIZE];
  char tx_buffer[RX_BUFFER_SIZE];
  char broadcast_buffer[RX_BUFFER_SIZE] = BROADCAST_MSG;

  int response_len;  // The length of the response the thread is sending
  int broadcast_len; // The length of the broadcast message the thread is
                     // sending


  // Maintain track of two connections. Initially they are not initialized
  // which can be represented by setting their ID to -1
  responding_connection.id = INIT_VAL;
  broadcast_connection.id = INIT_VAL;

  // Instruct server to listen and create new connections on the incoming port
  xtcp_listen(c_xtcp, INCOMING_PORT, XTCP_PROTOCOL_UDP);

  tmr :> time;
  while (1) {
    select {

    // Respond to an event from the tcp server
    case xtcp_event(c_xtcp, conn):
      switch (conn.event)
        {
        case XTCP_IFUP:
          // When the interface goes up, set up the broadcast connection.
          // This connection will persist while the interface is up
          // and is only used for outgoing broadcast messages
          xtcp_connect(c_xtcp,
                       BROADCAST_PORT,
                       broadcast_addr,
                       XTCP_PROTOCOL_UDP);
          break;

        case XTCP_IFDOWN:
```

```
          // Tidy up and close any connections we have open
          if (responding_connection.id != INIT_VAL) {
            xtcp_close(c_xtcp, responding_connection);
            responding_connection.id = INIT_VAL;
          }
          if (broadcast_connection.id != INIT_VAL) {
            xtcp_close(c_xtcp, broadcast_connection);
            broadcast_connection.id = INIT_VAL;
          }
          break;

        case XTCP_NEW_CONNECTION:

          // The tcp server is giving us a new connection.
          // It is either a remote host connecting on the listening port
          // or the broadcast connection the threads asked for with
          // the xtcp_connect() call
          if (XTCP_IPADDR_CMP(conn.remote_addr, broadcast_addr)) {
            // This is the broadcast connection
            printstr("New broadcast connection established:");
            printintln(conn.id);
            broadcast_connection = conn;
         }
          else {
            // This is a new connection to the listening port
            printstr("New connection to listening port:");
            printintln(conn.local_port);
            if (responding_connection.id == INIT_VAL) {
              responding_connection = conn;
            }
            else {
              printstr("Cannot handle new connection");
              xtcp_close(c_xtcp, conn);
            }
          }
          break;

        case XTCP_RECV_DATA:
          // When we get a packet in:
          //
          //  - fill the tx buffer
          //  - initiate a send on that connection
          //
          response_len = xtcp_recv_count(c_xtcp, rx_buffer, RX_BUFFER_SIZE);
          printstr("Got data: ");
          printint(response_len);
          printstrln(" bytes");

          for (int i=0;i<response_len;i++)
            tx_buffer[i] = rx_buffer[i];

          if (!send_flag) {
            xtcp_init_send(c_xtcp, conn);
            send_flag = TRUE;
            printstrln("Responding");
          }
          else {
            // Cannot respond here since the send buffer is being used
```

```
        }
        break;

    case XTCP_REQUEST_DATA:
    case XTCP_RESEND_DATA:
      // The tcp server wants data, this may be for the broadcast connection
      // or the reponding connection

      if (conn.id == broadcast_connection.id) {
        xtcp_send(c_xtcp, broadcast_buffer, broadcast_len);
      }
      else {
        xtcp_send(c_xtcp, tx_buffer, response_len);
      }
      break;

    case XTCP_SENT_DATA:
      xtcp_complete_send(c_xtcp);
      if (conn.id == broadcast_connection.id) {
        // When a broadcast message send is complete the connection is kept
        // open for the next one
        printstrln("Sent Broadcast");
        broadcast_send_flag = FALSE;
      }
      else {
        // When a reponse is sent, the connection is closed opening up
        // for another new connection on the listening port
        printstrln("Sent Response");
        xtcp_close(c_xtcp, conn);
        responding_connection.id = INIT_VAL;
        send_flag = FALSE;
      }
      break;

    case XTCP_TIMED_OUT:
    case XTCP_ABORTED:
    case XTCP_CLOSED:
      printstr("Closed connection:");
      printintln(conn.id);
      break;

    case XTCP_ALREADY_HANDLED:
        break;
    }
    break;

  // This is the periodic case, it occurs every BROADCAST_INTERVAL
  // timer ticks
  case tmr when timerafter(time + BROADCAST_INTERVAL) :> void:

    // A broadcast message can be sent if the connection is established
    // and one is not already being sent on that connection
    if (broadcast_connection.id != INIT_VAL && !broadcast_send_flag)  {
      printstrln("Sending broadcast message");
      broadcast_len = strlen(broadcast_buffer);
      xtcp_init_send(c_xtcp, broadcast_connection);
      broadcast_send_flag = TRUE;
    }
```

```
      tmr :> time;
      break;
    }
  }
}

#define XTCP_MII_BUFSIZE (4096)
#define ETHERNET_SMI_PHY_ADDRESS (0)

int main(void) {
  chan c_xtcp[1];
  mii_if i_mii;
  smi_if i_smi;
  par {
    // MII/ethernet driver
    on tile[1]: mii(i_mii, p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
                    p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
                    eth_rxclk, eth_txclk, XTCP_MII_BUFSIZE)

    // SMI/ethernet phy driver
    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    // TCP component
    on tile[1]: xtcp(c_xtcp, 1, i_mii,
                     null, null, null,
                     i_smi, ETHERNET_SMI_PHY_ADDRESS,
                     null, otp_ports, ipconfig);

    // The simple udp reflector thread
    on tile[0]: udp_reflect(c_xtcp[0]);

  }
  return 0;
}
```