Application Note: AN00120

XMOS Ethernet MAC library Application Note

Ethernet connectivity is an essential part of the explosion of connected devices known collectively as the Internet of Things (IoT). XMOS technology is perfectly suited to these applications - offering future proof and reliable ethernet connectivity whilst offering the flexibility to interface to a huge variety of "Things".

This application note shows a simple example that demonstrates the use of the XMOS MAC library to create a layer 2 ethernet MAC interface on an XMOS multicore microcontroller.

The code associated with this application note provides an example of using the MAC Library to provide a framework for the creation of an ethernet Media Independent Interface (MII) and MAC interface for 100Mbps.

The MAC framework uses XMOS libraries to provide a simple IP stack capable of responding to an ICMP ping message. The code used in the application note provides both MII communication to the PHY and a MAC transport layer for ethernet packets and enables a client to connect to it and send/receive packets.

Required tools and libraries

- xTIMEcomposer Tools - Version 13.1.0
- XMOS Ethernet library - Version 2.3.2rc0
- XMOS OTP Function Library - Version 1.0.0rc0
- XMOS Slicekit Core Board Support Library - Version 1.0.3rc0
- XMOS General utility: modules for developing XMOS devices
  - module_locks - Version 1.0.4rc0

Required hardware

This application note is designed to run on an XMOS xCORE-L (General Purpose family) series device. The example code provided with the application has been implemented and tested on the xCORE-L2 sliceKIT 1V2 (XP-SKC-L2) core board using ethernet sliceCARD 1V1 (XA-SK-E100). There is no dependancy on this core board - it can be modified to run on any (XMOS) development board which has the option to connect to the ethernet sliceCARD.

Prerequisites

- This document assumes familarity with the XMOS xCORE architecture, the Ethernet standards IEEE 802.3u (MII), 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 a description of XMOS related terms found in this document please see the XMOS Glossary[1].
- For an overview of Ethernet MAC library please see the *XMOS Layer 2 Ethernet MAC Component*[2].

[1] http://www.xmos.com/published/glossary
[2] https://www.xmos.com/published/xmos-layer-2-ethernet-mac-component

1 Overview

1.1 Introduction

XMOS MAC library supports two independent implementations.

- *The FULL implementation runs on five logical cores, allows multiple clients with independent buffering per client and supports accurate packet timestamping, priority queuing and 802.1Qav traffic shaping.*
- *The LITE implementation runs on two logical cores, does not support timestamping or multiple queues/buffering.*

This application note provides detail about a feature-rich (FULL) implementation which runs on five logical cores. This allows multiple clients with independent buffering per client and supports accurate packet timestamping, priority queuing and 802.1Qav traffic shaping. The less demanding applications, a LITE implementation can be used that runs on two logical cores. This implementation is restricted to a single receive and transmit client and does not support certain advanced features.

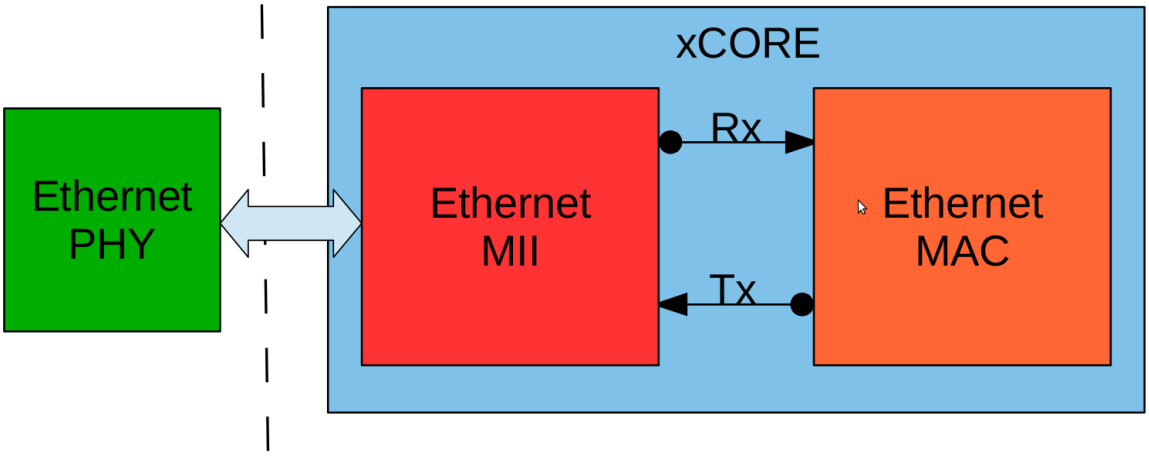

Figure 1: Ethernet MII and MAC layers

MII provides the management interface and data transfer signals between the Ethernet PHY (Physical Layer Device or transceiver) and the xCORE device. The MII layer receives packets of data which are then routed by an Ethernet MAC layer to multiple processes running on the xCORE.

1.2 Block Diagram

- **Hardware Timestamp:**
  On receiption/transmission of a ethernet frame over MII a timestamp is taken from the 100MHz reference timer on the core that the ethernet server is running on. The timestamp is taken at the end of the preamble immediately before the frame itself. This timestamp will be accurate to within 40ns.
  Functions *mac_rx_timed()* and *mac_tx_timed()* can be used from client to retrieve reception and transmission timestamp respectively.
  Both of these cores are mainly functional under the Ethernet MII layer, where it handles the incoming and outgoing of ethernet packets from ethernet PHY.

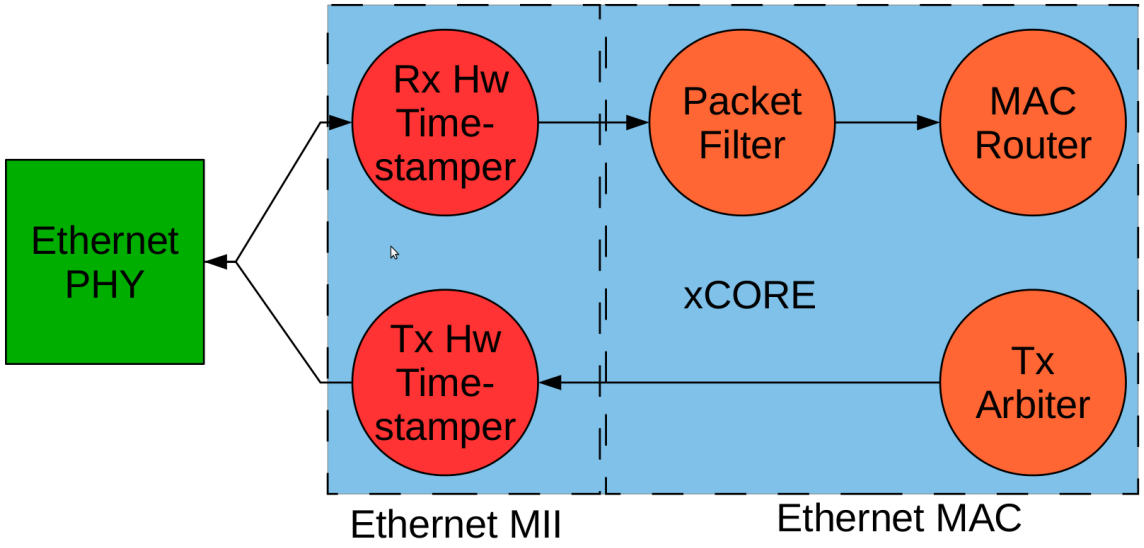

Figure 2: Block diagram of XMOS MAC library (FULL Implementation)

- **Packet Filter:**
  On the reception of every ethernet packet from MII layer, an initial filter is available where the packet is dropped unless
  1. The packet is destined for the host's MAC address or
  2. The packet is destined for a MAC address with the broadcast bit set

  After this initial filter, a user filter is supplied. To maintain the maximum amount of flexibility and efficiency the user must supply custom code to perform this filtering.
  - The function *mac_set_custom_filter* allows the user to enable/disable the filter for the incoming packets. A value *one* which is passed to this function, enables filtering and *zero* disables the filter, which intern drops the incoming packet.
  - If filter is enabled, then user must supply a definition of the function **mac_custom_filter()**. This function will inspect incoming packets like *ARP*, *ICMP* etc.

  This ethernet filter core verifies the CRC checksum of the received packets and identifies whether it is an *broadcast* or *unicast* message received. Incase of broadcast/unicast message, the custom filter updates the filter result with ethernet type. If not custom filter value is updated with *zero* value.
- **MAC Router:**
  This is basically an ethernet Rx server core which services the link commands, process the received frames w.r.t the custom filter set on *demo.xc* and the updated filter result from ethernet filter core. It also notifies the client about the rx packets that gets queued in rx buffer when the filters match, otherwise it ignores/drops the incoming ethernet packet(s).
  The *mac_rx* function requests for a rx frame to the server via client.
- **Tx Arbiter:**
  This is an ethernet Tx server core basically which handles the tx frame request from client. On request of tx frame, it checks for the availability of free buffer to queue the tx ethernet packet. When there is sufficient buffer available, it gets data from client and commits to MII that data is queued into tx buffer. On the event of insufficient tx buffer it ignores/drops the outgoing ethernet packet(s).
  The *max_tx* function requests for a tx frame to the server via client.

2 XMOS MAC library application note

The demo in this note uses the XMOS ethernet library and shows a simple program that communicates to client tasks over channels. The server can connect to several clients and each channel connection to the sever is for either RX (receiving packets from the MAC) or TX (transmitting packets to the MAC) operation.

To start using the MAC library, you need to add `module_ethernet`, `module_ethernet_board_support`, `module_otp_board_info`, `module_slicekit_support` to your makefile:

```
USED_MODULES = module_ethernet module_ethernet_board_support
               module_otp_board_info module_slicekit_support
```

You can then access the Ethernet MAC library functions in your source code via the ethernet_board_support.h, ethernet.h header files:

```
#include "ethernet_board_support.h"
#include "ethernet.h"
```

2.1 Configuring the stack

To configure the ethernet stack, **ethernet_conf.h** file is found in the *src/* directory of the application. Under CONFIG_FULL implementation, we can set the maximum packet size we can receive and the maximum number of ethernet clients (channel ends we can connect to the ethernet server).

Note: Even though it is set to 4, we have only one client in this demo.

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#ifdef CONFIG_FULL
#define ETHERNET_DEFAULT_IMPLEMENTATION full
#define MAX_ETHERNET_PACKET_SIZE (1518)
#define MAX_ETHERNET_CLIENTS   (4)
#else
#define ETHERNET_DEFAULT_IMPLEMENTATION lite
#endif
```

2.2 Setting The IP Address

Set an IP address that is routable in the network that the demo is to be tested on.

```
//::ip_address_define
// NOTE: YOU MAY NEED TO REDEFINE THIS TO AN IP ADDRESS THAT WORKS
// FOR YOUR NETWORK
#define OWN_IP_ADDRESS {169, 254, 8, 100}
//::
```

2.3 Ethernet Server and PHY configuration

The ethernet server **ethernet_server()** provides both MII layer and MAC layer functions. It runs in five cores and communicates to client(s) over the channel array(s). The function **otp_board_info_get_mac()** reads the device mac address from ROM and functions **eth_phy_reset()**, **smi_config()** and **eth_phy_config()** initialize the PHY

```
//::ethernet
on ETHERNET_DEFAULT_TILE:
{
  char mac_address[6];
  otp_board_info_get_mac(otp_ports, 0, mac_address);
  eth_phy_reset(eth_rst);
  smi_init(smi);
  eth_phy_config(1, smi);
  ethernet_server(mii, null, mac_address, rx, 1, tx, 1);
}
//::
```

2.4 Ethernet Client and Filter

A simple ethernet client **demo()** function is a task which takes ethernet packets and responds to ping requests.

```
//::demo
on ETHERNET_DEFAULT_TILE : demo(tx[0], rx[0]);
//::
```

This demo() function does the actual ethernet packet processing. First the application gets the device mac address from the ethernet server using **mac_get_macaddr()**.:

```
mac_get_macaddr(tx, own_mac_addr);
```

Then the packet filter is enabled. Filter is used to filter the MAC layer packets.

```
//::setup-filter
#ifdef CONFIG_FULL
mac_set_custom_filter(rx, 0x1);
#endif
//::
```

The implementation on **mac_custom_filter()** filters the incoming packet for *ARP* and *ICMP* messages.

```
//::custom-filter
int mac_custom_filter(unsigned int data[]){
        if (is_ethertype((data,char[]), ethertype_arp)){
                return SUCCESS;
        }else if (is_ethertype((data,char[]), ethertype_ip)){
                return SUCCESS;
        }

        return FAILURE;
}
//::
```

2.5 Packet Processing

When the packet is received it may be of any ethernet type. The applications which are capable of processing several ethernet types should build their own processing techniques as per their requirements.

For example in this demo we have taken ethernet types of Address Resolution Protocol (ARP) and Internet Control Message Protocol (ICMP) packet as ethernet types, where ARP is a part of Link Layer and ICMP is a part Internet Layer under Internet Protocol Suite. ICMP for Internet Protocol version 4 (IPv4) is also known

as ICMPv4.

Ethernet packets can be received into the Rx buffer using the **mac_rx()** function. Type of ethernet is checked using the function(s) **is_valid_arp_response()** and/or **is_valid_icmp_packet()**.

If an ARP packet is received, a response is sent to the Tx buffer using the **mac_tx()**. Similiarly if it is not ARP packet, then the code checks for an ICMP packet responds appropriately.

```
mac_get_macaddr(tx, own_mac_addr);
```

```
 //::arp_packet_check
  if (is_valid_arp_packet((rxbuf,char[]), nbytes))
    {
      build_arp_response((rxbuf,char[]), txbuf, own_mac_addr);
      mac_tx(tx, txbuf, nbytes, ETH_BROADCAST);
      printstr("ARP response sent\n");
    }
//::icmp_packet_check
```

2.6 The application main() function

The *main()* is splitted into two section *(1) Ethernet* and *(2) demo*.

On *Ethernet*: All the lowlevel initializations are done which generally includes,

1. Getting board MAC address from OTP
2. Resetting ethernet PHY device
3. Initializing SMI interface
4. Configuring the PHY and
5. Enables the ethernet server

On *demo:* [Refer: §2.7]

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

```
int main()
{
  chan rx[1], tx[1];

  par
    {
      //::ethernet
      on ETHERNET_DEFAULT_TILE:
      {
        char mac_address[6];
        otp_board_info_get_mac(otp_ports, 0, mac_address);
        eth_phy_reset(eth_rst);
        smi_init(smi);
        eth_phy_config(1, smi);
        ethernet_server(mii, null, mac_address, rx, 1, tx, 1);
      }
      //::

      //::demo
      on ETHERNET_DEFAULT_TILE : demo(tx[0], rx[0]);
      //::
    }

        return 0;
}
```

The MDIO Serial Management Interface (SMI) is used to transfer management information between MAC and PHY. At powerup the PHY usually adapts to whatever it's connected to (Autonegotiation) unless user alter settings via the MDIO interface.

2.7 The demo() function

The second section of this application is the *demo()* which covers,

1. Checks the MAC address on the incoming packet.
2. Configures the custom filter
3. Receives ethernet packet from ethernet server.
4. Verifies which ethernet type packet is received.(Here, it is ARP or ICMP)
5. Builds response according to ethernet type received and
6. Sends the response to the ethernet server to broadcast on network

This *demo()* acts as a ethernet client which continues to run between step 3 to 6 later.

Below is the source code for the demo (a simple ethernet client) function of this application, which is taken from the source file **demo.xc**

```
//::client_demo
void demo(chanend tx, chanend rx)
{
  unsigned int rxbuf[1600/4];
  unsigned int txbuf[1600/4];

  //::get-macaddr
  mac_get_macaddr(tx, own_mac_addr);
  //::

  //::setup-filter
#ifdef CONFIG_FULL
  mac_set_custom_filter(rx, 0x1);
#endif
  //::
  printstr("Test started\n");

  //::mainloop
  while (1)
  {
    unsigned int src_port;
    unsigned int nbytes;
    mac_rx(rx, (rxbuf,char[]), nbytes, src_port);
#ifdef CONFIG_LITE
    if (!is_broadcast((rxbuf,char[])) && !is_mac_addr((rxbuf,char[]), own_mac_addr))
      continue;
    if (mac_custom_filter(rxbuf) != SUCCESS)
      continue;
#endif


   //::arp_packet_check
    if (is_valid_arp_packet((rxbuf,char[]), nbytes))
      {
        build_arp_response((rxbuf,char[]), txbuf, own_mac_addr);
        mac_tx(tx, txbuf, nbytes, ETH_BROADCAST);
        printstr("ARP response sent\n");
      }
  //::icmp_packet_check
    else if (is_valid_icmp_packet((rxbuf,char[]), nbytes))
      {
        build_icmp_response((rxbuf,char[]), (txbuf, unsigned char[]), own_mac_addr);
        mac_tx(tx, txbuf, nbytes, ETH_BROADCAST);
        printstr("ICMP response sent\n");
      }
  //::
  }
}
//::end
```

APPENDIX A - Demo Hardware Setup

- To run the demo, connect the XTAG-2 USB debug adapter to the sliceKIT via the supplied adaptor board
- Connect the XTAG-2 to the host PC (using USB extension cable if desired)
- Connect the ethernet sliceCARD to the **CIRCLE** slot of the sliceKIT. Then, connect the slice to the host PC or to the network switch using an ethernet cable.
- On the xCORE-L series sliceKIT ensure that the xCONNECT LINK (xSCOPE) switch is set to ON, as per the image, to allow xSCOPE to function.

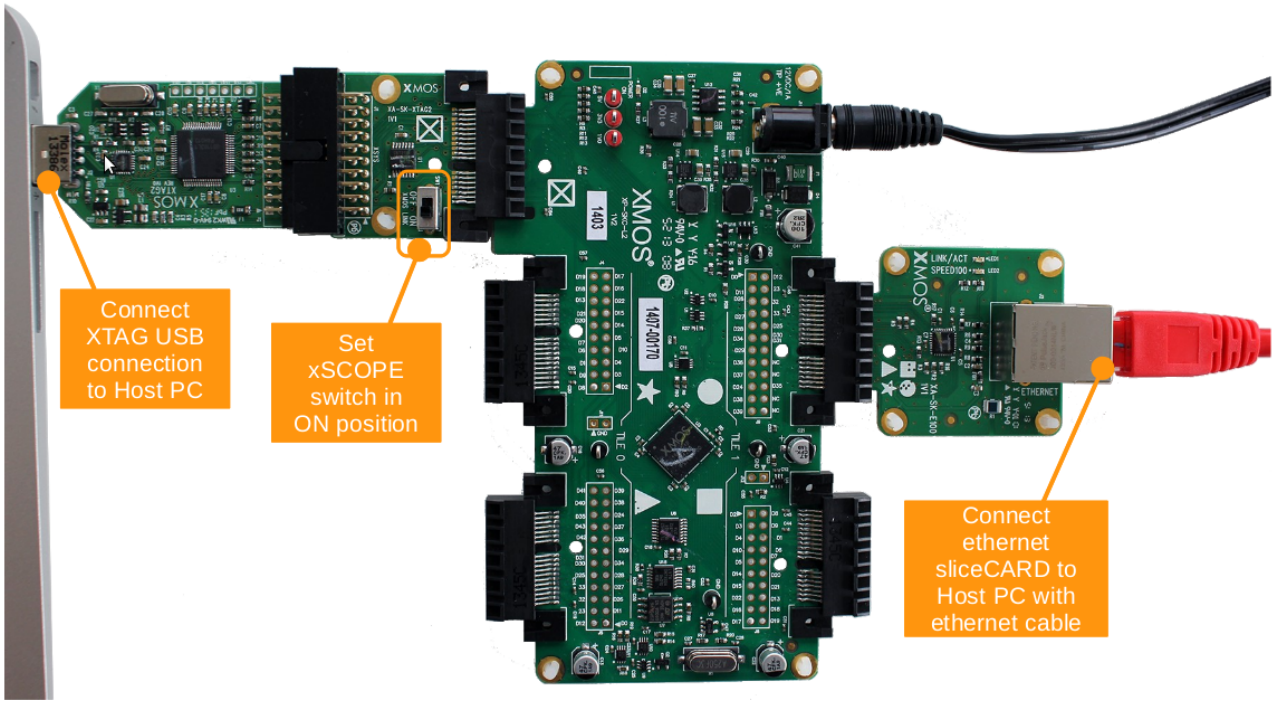

Figure 3: Hardware Setup for XMOS MAC library demo

APPENDIX B - Launching the demo device

Once the application source code is imported into the tools you can 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 code onto the xCORE-L sliceKIT. Here you use the tools to load the application over JTAG onto the xCORE device.

- Select **Run Configuration**.
- In **Main** menu, enable **JTAG** in Target I/O options.
- In **XScope** menu, enable **Offline [XScope] Mode**.
- Click **Apply** and then **Run**.

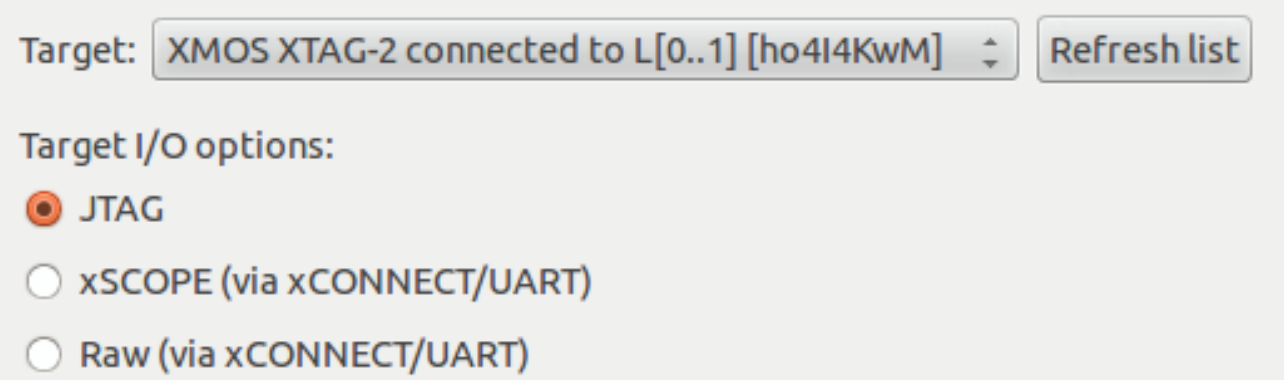

Figure 4: xTIMEcomposer Target I/O configuration

When the processor has finished booting you will see the following text in the xTIMEcomposer console window when **ping**-ed from the host.

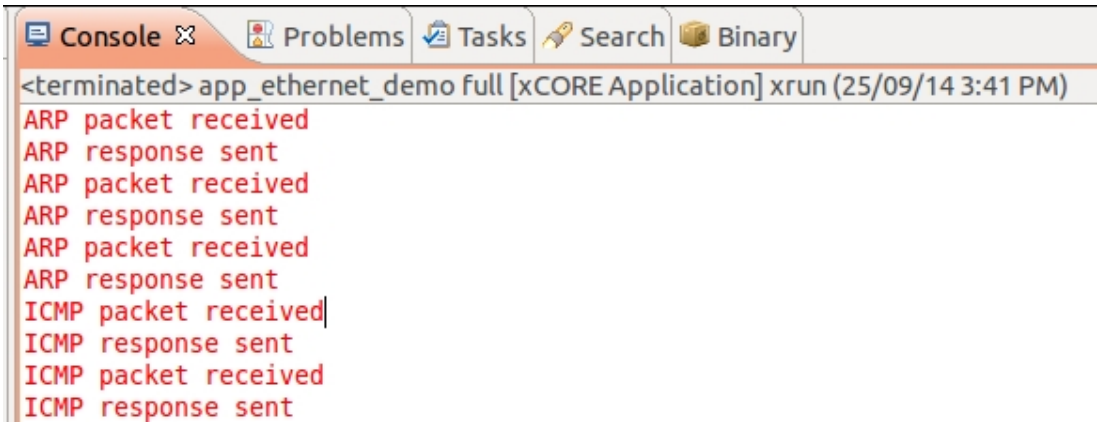

Figure 5: Response on xTIMEcomposer console for **Ping**

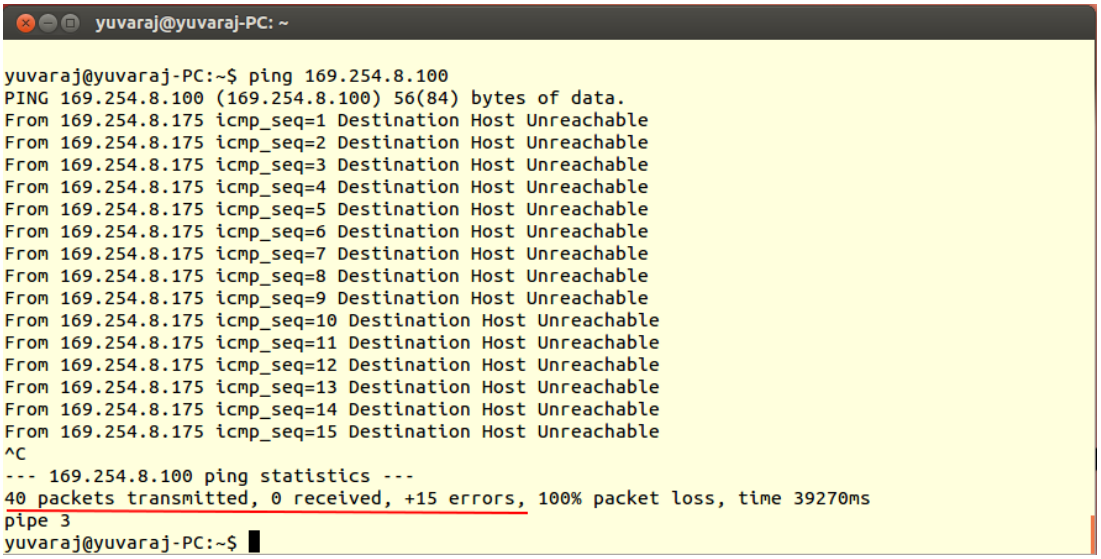

Figure 6: Response on Host PC for **Ping**

On the above *ping* response on host, out of 40 packets transmitted to device first 15 packet couldn't able to reach the device. During this period, device will respond as **ARP packet received**. After that device (remaining 35 packets) responds as **ICMP packet received**. (Figure 5)

APPENDIX C - FAQs

1. What do I do if I need to change the ethernet sliceCARD slot?

   By default the code assumes that the ethernet sliceCARD is connected to the **CIRCLE** slot. If the user needs to change to slots other than CIRCLE, then he/she has to **#define** it on **ethernet_conf.h** file available in the *src/* directory.:

```
ETHERNET_USE_STAR_SLOT
ETHERNET_USE_TRIANGLE_SLOT
ETHERNET_USE_CIRCLE_SLOT
ETHERNET_USE_SQUAR_SLOT
```

   Based on the type of slot and sliceKIT core board choosen, tile allocation then happens automatically.

2. How to implement LITE configuration?

   There is a build option CONFIG_LITE provided in the **Makefile** with this demo. ALternatively the user can undefine CONFIG_FULL on the **ethernet_conf.h** header file available in the *src/* directory. The LITE implementation does not support timestamping or multiple queues/buffering. The MAC will filter packets based on MAC address (including the broadcast bit). Any further filtering must be done by the single receive client of the ethernet server.

3. How do I define a MAC Custom Filter for the LITE implementation?

   The mac_custom_filter function allows the user to decide which packets get passed through the MAC. To do this, we have to provide the **mac_custom_filter.h** headerfile and a definition of the mac_custom_filter function itself. (By default filtering in enabled in this implementation)

   The header file in this example simply prototypes the *mac_custom_filter* function itself.

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

extern int mac_custom_filter(unsigned int data[]);
```

   The module requires the application to provide the header to cater for the case where the function is described as an inline function for performance. In this case it is just prototyped and the definition of *mac_custom_filter* is in our main application code file **demo.xc**

```
#ifdef CONFIG_LITE
if (!is_broadcast((rxbuf,char[])) && !is_mac_addr((rxbuf,char[]), own_mac_addr))
  continue;
if (mac_custom_filter(rxbuf) != SUCCESS)
  continue;
#endif
```

   This function returns *zero* if we do not want to handle the packet and *non-zero* otherwise. The non-zero value is used later to distribute to different clients. In this case we detect ARP packets and ICMP packets which match our own mac address as a destination (wherein the function returns *one*). The defintions of *is_broadcast*, *is_ethertype* and *is_mac_addr* are in **demo.xc**

4. What are Buffers and Queues? Why are they needed?

   The MAC maintains two sets of buffers: one for the incoming packets and the other for outgoing packets.

For the incoming packets:

- Empty buffers are in the incoming queue awaiting a packet coming from the MII interfaces.
- Buffers received from the MII interface are filtered and moved into a forwarding queue if appropriate.
- Buffers in the forwarding queue are moved into a client queue depending on which client registered for that type of packet.
- Once the data from a buffer has been sent to a client, the buffer is moved back into the incoming queue.

For the outgoing packets:

- Empty buffers are stored in an empty queue awaiting a packet from the client.
- Once the data is received the buffer is moved into a transmit queue awaiting output on the MII interface.
- Once the data is transmitted, the buffer is released back to the empty queue.

5. How to get timestamp values?

- **mac_rx_timed()** - On reception of a ethernet frame, the timestamp is stored with the buffer can be retrieved by the client.
- **mac_tx_timed()** - On transmission of a ethernet frame, the timestamp is stored and placed in a queue to sent back to the client.

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 Layer 2 Ethernet MAC Component

https://www.xmos.com/published/xmos-layer-2-ethernet-mac-component

IEEE 802.3 Ethernet Standards

http://standards.ieee.org/about/get/802/802.3.html

Ethernet Basics

http://homepage.smc.edu/morgan_david/linux/n-protocol-09-ethernet.pdf

Ethernet Frame

http://en.wikipedia.org/wiki/Ethernet_frame

Ethernet Timestamps

http://m.eetindia.co.in/STATIC/PDF/200906/EEIOL_2009JUN03_NETD_TA_01.pdf?SOURCES=DOWNLOAD

MAC address

http://en.wikipedia.org/wiki/MAC_address

Ethernet Type

http://en.wikipedia.org/wiki/EtherType

Internet Control Message Protocol

http://en.wikipedia.org/wiki/Internet_Control_Message_Protocol

APPENDIX E - Full Source code listing

E.1 Source code for demo.xc

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

/*************************************************************************
 *
 * Ethernet MAC Layer Client Test Code
 * IEEE 802.3 MAC Client
 *
 *
 ************************************************************************
 *
 * ARP/ICMP demo
 * Note: Only supports unfragmented IP packets
 *
 ************************************************************************/

#include <xs1.h>
#include <xclib.h>
#include <print.h>
#include <platform.h>
#include <stdlib.h>
#include "otp_board_info.h"
#include "ethernet.h"
#include "ethernet_board_support.h"
#include "checksum.h"
#include "xscope.h"

#define SUCCESS      1
#define FAILURE      0

// If you have a board with the xscope xlink enabled (e.g. the XC-2) then
// change this define to 0, make sure you also remove the -lxscope from
// the build flags in the Makefile
#define USE_XSCOPE 0


#if USE_XSCOPE
void xscope_user_init(void) {
  xscope_register(0);
  xscope_config_io(XSCOPE_IO_BASIC);
}
#endif

// Port Definitions

// These ports are for accessing the OTP memory
on ETHERNET_DEFAULT_TILE: otp_ports_t otp_ports = OTP_PORTS_INITIALIZER;

// Here are the port definitions required by ethernet
// The intializers are taken from the ethernet_board_support.h header for
// XMOS dev boards. If you are using a different board you will need to
// supply explicit port structure intializers for these values
smi_interface_t smi = ETHERNET_DEFAULT_SMI_INIT;
mii_interface_t mii = ETHERNET_DEFAULT_MII_INIT;
ethernet_reset_interface_t eth_rst = ETHERNET_DEFAULT_RESET_INTERFACE_INIT;

//::ip_address_define
// NOTE: YOU MAY NEED TO REDEFINE THIS TO AN IP ADDRESS THAT WORKS
// FOR YOUR NETWORK
#define OWN_IP_ADDRESS {169, 254, 8, 100}
//::


unsigned char ethertype_ip[] = {0x08, 0x00};
unsigned char ethertype_arp[] = {0x08, 0x06};

unsigned char own_mac_addr[6];
```

```
#define ARP_RESPONSE 1
#define ICMP_RESPONSE 2
#define UDP_RESPONSE 3

void demo(chanend tx, chanend rx);

#pragma unsafe arrays
int is_ethertype(unsigned char data[], unsigned char type[]){
        int i = 12;
        return data[i] == type[0] && data[i + 1] == type[1];
}

#pragma unsafe arrays
int is_mac_addr(unsigned char data[], unsigned char addr[]){
        for (int i=0;i<6;i++){
          if (data[i] != addr[i]){
                        return FAILURE;
                }
        }

        return SUCCESS;
}

#pragma unsafe arrays
int is_broadcast(unsigned char data[]){
        for (int i=0;i<6;i++){
          if (data[i] != 0xFF){
                        return FAILURE;
                }
        }

        return SUCCESS;
}

//::custom-filter
int mac_custom_filter(unsigned int data[]){
        if (is_ethertype((data,char[]), ethertype_arp)){
                return SUCCESS;
        }else if (is_ethertype((data,char[]), ethertype_ip)){
                return SUCCESS;
        }

        return FAILURE;
}
//::


int build_arp_response(unsigned char rxbuf[], unsigned int txbuf[], const unsigned char own_mac_addr[6])
{
  unsigned word;
  unsigned char byte;
  const unsigned char own_ip_addr[4] = OWN_IP_ADDRESS;
  int pad;

  for (int i = 0; i < 6; i++)
    {
      byte = rxbuf[22+i];
      (txbuf, unsigned char[])[i] = byte;
      (txbuf, unsigned char[])[32 + i] = byte;
    }
  word = (rxbuf, const unsigned[])[7];
  for (int i = 0; i < 4; i++)
    {
      (txbuf, unsigned char[])[38 + i] = word & 0xFF;
      word >>= 8;
    }

  (txbuf, unsigned char[])[28] = own_ip_addr[0];
  (txbuf, unsigned char[])[29] = own_ip_addr[1];
  (txbuf, unsigned char[])[30] = own_ip_addr[2];
  (txbuf, unsigned char[])[31] = own_ip_addr[3];

  for (int i = 0; i < 6; i++)
  {
```

```
    (txbuf, unsigned char[])[22 + i] = own_mac_addr[i];
    (txbuf, unsigned char[])[6 + i] = own_mac_addr[i];
  }
  txbuf[3] = 0x01000608;
  txbuf[4] = 0x04060008;
  (txbuf, unsigned char[])[20] = 0x00;
  (txbuf, unsigned char[])[21] = 0x02;

  // Typically 48 bytes (94 for IPv6)
  for (pad = 42; pad < 64; pad++)
  {
    (txbuf, unsigned char[])[pad] = 0x00;
  }

  return pad;
}


int is_valid_arp_packet(const unsigned char rxbuf[], int nbytes)
{
  static const unsigned char own_ip_addr[4] = OWN_IP_ADDRESS;

  if (rxbuf[12] != 0x08 || rxbuf[13] != 0x06)
    return FAILURE;

  printstr("ARP packet received\n");

  if ((rxbuf, const unsigned[])[3] != 0x01000608)
  {
    printstr("Invalid et_htype\n");
    return FAILURE;
  }
  if ((rxbuf, const unsigned[])[4] != 0x04060008)
  {
    printstr("Invalid ptype_hlen\n");
    return FAILURE;
  }
  if (((rxbuf, const unsigned[])[5] & 0xFFFF) != 0x0100)
  {
    printstr("Not a request\n");
    return FAILURE;
  }
  for (int i = 0; i < 4; i++)
  {
    if (rxbuf[38 + i] != own_ip_addr[i])
    {
      printstr("Not for us\n");
      return FAILURE;
    }
  }

  return SUCCESS;
}


int build_icmp_response(unsigned char rxbuf[], unsigned char txbuf[], const unsigned char own_mac_addr[6])
{
  static const unsigned char own_ip_addr[4] = OWN_IP_ADDRESS;
  unsigned icmp_checksum;
  int datalen;
  int totallen;
  const int ttl = 0x40;
  int pad;

  // Precomputed empty IP header checksum (inverted, bytereversed and shifted right)
  unsigned ip_checksum = 0x0185;

  for (int i = 0; i < 6; i++)
    {
      txbuf[i] = rxbuf[6 + i];
    }
  for (int i = 0; i < 4; i++)
    {
      txbuf[30 + i] = rxbuf[26 + i];
    }
```

```
  icmp_checksum = byterev((rxbuf, const unsigned[])[9]) >> 16;
  for (int i = 0; i < 4; i++)
    {
      txbuf[38 + i] = rxbuf[38 + i];
    }
  totallen = byterev((rxbuf, const unsigned[])[4]) >> 16;
  datalen = totallen - 28;
  for (int i = 0; i < datalen; i++)
    {
      txbuf[42 + i] = rxbuf[42+i];
    }

  for (int i = 0; i < 6; i++)
  {
    txbuf[6 + i] = own_mac_addr[i];
  }
  (txbuf, unsigned[])[3] = 0x00450008;
  totallen = byterev(28 + datalen) >> 16;
  (txbuf, unsigned[])[4] = totallen;
  ip_checksum += totallen;
  (txbuf, unsigned[])[5] = 0x01000000 | (ttl << 16);
  (txbuf, unsigned[])[6] = 0;
  for (int i = 0; i < 4; i++)
  {
    txbuf[26 + i] = own_ip_addr[i];
  }
  ip_checksum += (own_ip_addr[0] | own_ip_addr[1] << 8);
  ip_checksum += (own_ip_addr[2] | own_ip_addr[3] << 8);
  ip_checksum += txbuf[30] | (txbuf[31] << 8);
  ip_checksum += txbuf[32] | (txbuf[33] << 8);

  txbuf[34] = 0x00;
  txbuf[35] = 0x00;

  icmp_checksum = (icmp_checksum + 0x0800);
  icmp_checksum += icmp_checksum >> 16;
  txbuf[36] = icmp_checksum >> 8;
  txbuf[37] = icmp_checksum & 0xFF;

  while (ip_checksum >> 16)
  {
    ip_checksum = (ip_checksum & 0xFFFF) + (ip_checksum >> 16);
  }
  ip_checksum = byterev(~ip_checksum) >> 16;
  txbuf[24] = ip_checksum >> 8;
  txbuf[25] = ip_checksum & 0xFF;

  for (pad = 42 + datalen; pad < 64; pad++)
  {
    txbuf[pad] = 0x00;
  }
  return pad;
}


int is_valid_icmp_packet(const unsigned char rxbuf[], int nbytes)
{
  static const unsigned char own_ip_addr[4] = OWN_IP_ADDRESS;
  unsigned totallen;


  if (rxbuf[23] != 0x01)
    return FAILURE;

  printstr("ICMP packet received\n");

  if ((rxbuf, const unsigned[])[3] != 0x00450008)
  {
    printstr("Invalid et_ver_hdrl_tos\n");
    return FAILURE;
  }
  if (((rxbuf, const unsigned[])[8] >> 16) != 0x0008)
  {
    printstr("Invalid type_code\n");
    return FAILURE;
```

```
  }
  for (int i = 0; i < 4; i++)
  {
    if (rxbuf[30 + i] != own_ip_addr[i])
    {
      printstr("Not for us\n");
      return FAILURE;
    }
  }

  totallen = byterev((rxbuf, const unsigned[])[4]) >> 16;
  if (nbytes > 60 && nbytes != totallen + 14)
  {
    printstr("Invalid size\n");
    printintln(nbytes);
    printintln(totallen+14);
    return FAILURE;
  }
  if (checksum_ip(rxbuf) != 0)
  {
    printstr("Bad checksum\n");
    return FAILURE;
  }

  return SUCCESS;
}

//::client_demo
void demo(chanend tx, chanend rx)
{
  unsigned int rxbuf[1600/4];
  unsigned int txbuf[1600/4];

  //::get-macaddr
  mac_get_macaddr(tx, own_mac_addr);
  //::

  //::setup-filter
#ifdef CONFIG_FULL
  mac_set_custom_filter(rx, 0x1);
#endif
  //::
  printstr("Test started\n");

  //::mainloop
  while (1)
  {
    unsigned int src_port;
    unsigned int nbytes;
    mac_rx(rx, (rxbuf,char[]), nbytes, src_port);
#ifdef CONFIG_LITE
    if (!is_broadcast((rxbuf,char[])) && !is_mac_addr((rxbuf,char[]), own_mac_addr))
      continue;
    if (mac_custom_filter(rxbuf) != SUCCESS)
      continue;
#endif


   //::arp_packet_check
    if (is_valid_arp_packet((rxbuf,char[]), nbytes))
      {
        build_arp_response((rxbuf,char[]), txbuf, own_mac_addr);
        mac_tx(tx, txbuf, nbytes, ETH_BROADCAST);
        printstr("ARP response sent\n");
      }
  //::icmp_packet_check
    else if (is_valid_icmp_packet((rxbuf,char[]), nbytes))
      {
        build_icmp_response((rxbuf,char[]), (txbuf, unsigned char[]), own_mac_addr);
        mac_tx(tx, txbuf, nbytes, ETH_BROADCAST);
        printstr("ICMP response sent\n");
      }
  //::
  }
}
```

```
//::end

int main()
{
  chan rx[1], tx[1];

  par
    {
      //::ethernet
      on ETHERNET_DEFAULT_TILE:
      {
        char mac_address[6];
        otp_board_info_get_mac(otp_ports, 0, mac_address);
        eth_phy_reset(eth_rst);
        smi_init(smi);
        eth_phy_config(1, smi);
        ethernet_server(mii, null, mac_address, rx, 1, tx, 1);
      }
      //::

      //::demo
      on ETHERNET_DEFAULT_TILE : demo(tx[0], rx[0]);
      //::
    }

        return 0;
}
```

E.2 Source code for checksum.c

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#include <xclib.h>
#include "checksum.h"

#define u16_t unsigned short

/**
   * This implementation exploits various properties of the internet checksum
   * described in RFC 1071.
   * It assumes a little endian machine with no support for misaligned loads.
   *
   * We do as much as possible using 16bit loads. Fetching a 16bit value
   * will swap the bytes since data is stored in network order. Therefore the
   * the resulting sum will be swapped (Byte Order Independence).
   * We also use Byte Order Independence to aligned terms of the sum.
   * The sum:
   * [A, B] +' [C, D] +' ... +' [Y, Z] [1]
   * is equal the following sum with the bytes reversed
   * [0, A] +' [B, C] +' ... +' [Z, 0] [2]
   * If the terms in sum [1] are not 16bit aligned then the terms in sum [2]
   * will be, with the exception of the first and last bytes which can be dealt
   * with seperately. By testing for 16bit alignment and choosing to group
   * terms appropriately we can use 16bit loads for the majority of the data.
   * Finally the sum uses 32bit accumulator which is folded back into 16 bits
   * This saves us having to check for carry on each iteration of the loop.
   *
   * It might be possible to improve performance using parallel summation
   * (i.e. using 32bit addition), although this would need investigation.
   */
unsigned short checksum(const unsigned char data[], int skip, unsigned short len)
{
  int swap = 1;
  unsigned accum = 0;
  data += skip;
  const unsigned char *endptr = data + len - 1;

  if (len == 0)
    return 0;
```

```
  // Test for misaligned data
  if ((int)data % sizeof(u16_t) != 0)
  {
    swap = 0;
    accum += data[0] << 8;
    data++;
  }

  // At least two more bytes
  while (data < endptr)
  {
    accum += *((u16_t*)data);
    data += 2;
  }

  // Deal with misaligned end
  if (data == endptr)
    accum += data[0];

  // Fold carry into 16bits
  while (accum >> 16)
  {
    accum = (accum & 0xFFFF) + (accum >> 16);
  }

  if (swap)
    accum = byterev(~accum) >> 16;
  else
    accum = ~accum & 0xFFFF;

  return accum;
}

unsigned short checksum_ip(const unsigned char frame[])
{
  int i;
  unsigned accum = 0;

  for (i = 14; i < 34; i += 2)
  {
    accum += *((u16_t*)(frame + i));
  }

  // Fold carry into 16bits
  while (accum >> 16)
  {
    accum = (accum & 0xFFFF) + (accum >> 16);
  }

  accum = byterev(~accum) >> 16;

  return accum;
}

unsigned short checksum_udp(const unsigned char frame[], int udplen)
{
  int i;
  const unsigned char *endptr = frame + 34 + udplen - 1;
  unsigned accum = 0x1100;

  accum += (byterev((unsigned)udplen) >> 16) << 1;
  accum += *((u16_t*)(frame + 26));
  accum += *((u16_t*)(frame + 28));
  accum += *((u16_t*)(frame + 30));
  accum += *((u16_t*)(frame + 32));
  accum += *((u16_t*)(frame + 34));
  accum += *((u16_t*)(frame + 36));
  for (i = 42; frame + i < endptr; i += 2)
  {
    accum += *((u16_t*)(frame + i));
  }

  // Deal with misaligned end
  if (frame + i == endptr)
    accum += *endptr;
```

```
  // Fold carry into 16bits
  while (accum >> 16)
  {
    accum = (accum & 0xFFFF) + (accum >> 16);
  }

  accum = byterev(~accum) >> 16;

  return accum;
}
```

E.3 Source code for ethernet_conf.h

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#ifdef CONFIG_FULL
#define ETHERNET_DEFAULT_IMPLEMENTATION full
#define MAX_ETHERNET_PACKET_SIZE (1518)
#define MAX_ETHERNET_CLIENTS   (4)
#else
#define ETHERNET_DEFAULT_IMPLEMENTATION lite
#endif
```

E.4 Source code for mac_custom_filter.h

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

extern int mac_custom_filter(unsigned int data[]);
```

E.5 Source code for checksum.h

```
// Copyright (c) 2011, XMOS Ltd, All rights reserved
// This software is freely distributable under a derivative of the
// University of Illinois/NCSA Open Source License posted in
// LICENSE.txt and at <http://github.xcore.com/>

#ifndef _checksum_h_
#define _checksum_h_

unsigned short checksum_ip(const unsigned char frame[]);
unsigned short checksum_udp(const unsigned char frame[], int udplen);

#endif
```