Application Note: AN00199

XMOS Gigabit Ethernet 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 Ethernet library to create a *gigabit* layer 2 ethernet MAC interface on an XMOS multicore microcontroller.

The code associated with this application note provides an example of using the Ethernet Library to provide a framework for the creation of an ethernet Reduced Gigabit Media Independent Interface (RGMII) and MAC interface for gigabit data rates.

The applcation note 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

The code in this application note is known to work on version 14.2.4 of the xTIMEcomposer tools suite, it may work on other versions.

The application depends on the following libraries:

- lib_ethernet (>=3.3.0)
- lib_otpinfo (>=2.0.0)

Required hardware

This application note is designed to run on an XMOS xCORE-200 series device. The example code provided with the application has been implemented and tested on the xCORE-200 Explorer development kit. There is no dependancy on this board - it can be modified to run on any xCORE-200 series device with gigabit Ethernet capability.

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 the Ethernet library, please see the Ethernet library user guide.

[1] http://www.xmos.com/published/glossary

1 Overview

1.1 Introduction

The application note shows the use of the XMOS Ethernet library. The library allows multiple clients to access the Ethernet hardware. This application note uses the gigabit RGMII Ethernet MAC which uses 8 logical cores on a single tile (although one of those core can be shared with the application). The Ethernet library also provides 100Mbit MAC components which consume less resource on the xCORE device.

The gigabit MAC can handle line rate data packets through to the application layer and provides high performance streaming data, accurate packet timestamping, priority queuing and 802.1Qav traffic shaping.

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

1.2 Block Diagram

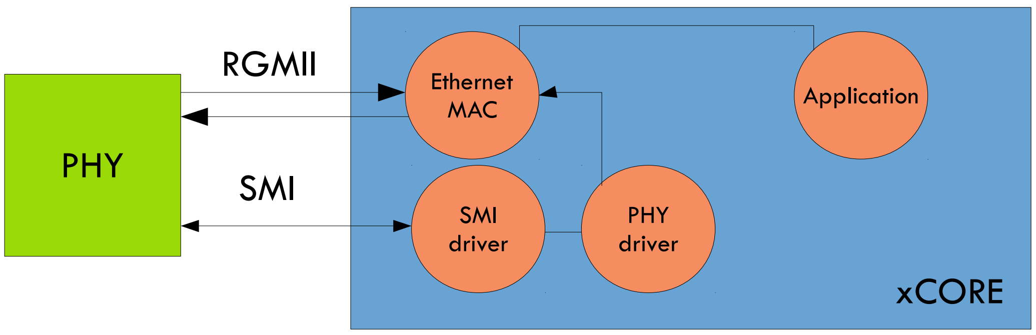


Figure 1: Application block diagram

The application communicates with the Ethernet MAC that drives the RGMII data interface to the PHY. A separate PHY driver configures the PHY via the SMI serial interface.

2 Gigabit Ethernet library demo

2.1 The Makefile

The demo in this note uses the XMOS Ethernet library and shows a simple program communicating with the Ethernet library.

To start using the Ethernet library, you need to add `lib_ethernet` to you Makefile:

```
USED_MODULES = .. lib_ethernet ...
```

This demo also gets the MAC address out of the xCORE OTP rom and is for the sliceKIT. So the Makefile also includes the OTP reading library (`lib_otpinfo`):

```
USED_MODULES = .. lib_otpinfo ...
```

2.2 Allocating hardware resources

The Ethernet library requires the user to declare the ports that the RGMII MAC uses to communicate with the Ethernet PHY. These ports are fixed on the xCORE-200 series so in the main program they are declared using the RGMII_PORTS_INITIALIZER macro provided by the library. This means the application just needs to declare a structure in `main.xc` to pass to the MAC component.

```
rgmii_ports_t rgmii_ports = on tile[1]: RGMII_PORTS_INITIALIZER;
```

The application needs to control the reset line of the PHY and configure the phy via the MDIO Serial Management Interface (SMI):

```
port p_smi_mdio   = on tile[1]: XS1_PORT_1C;
port p_smi_mdc    = on tile[1]: XS1_PORT_1D;
port p_eth_reset  = 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[0]: OTP_PORTS_INITIALIZER;
```

2.3 The application main() function

The main function in the program sets up the tasks in the application.

```
int main()
{
  ethernet_cfg_if i_cfg[NUM_CFG_CLIENTS];
  ethernet_rx_if i_rx[NUM_ETH_CLIENTS];
  ethernet_tx_if i_tx[NUM_ETH_CLIENTS];
  streaming chan c_rgmii_cfg;
  smi_if i_smi;

  par {
    on tile[1]: rgmii_ethernet_mac(i_rx, NUM_ETH_CLIENTS,
                                   i_tx, NUM_ETH_CLIENTS,
                                   null, null,
                                   c_rgmii_cfg,
                                   rgmii_ports,
                                   ETHERNET_DISABLE_SHAPER);
    on tile[1].core[0]: rgmii_ethernet_mac_config(i_cfg, NUM_CFG_CLIENTS, c_rgmii_cfg);
    on tile[1].core[0]: ar8035_phy_driver(i_smi, i_cfg[CFG_TO_PHY_DRIVER]);

    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    on tile[0]: icmp_server(i_cfg[CFG_TO_ICMP],
                            i_rx[ETH_TO_ICMP], i_tx[ETH_TO_ICMP],
                            ip_address, otp_ports);
  }
  return 0;
}
```

The `rgmii_ethernet_mac` and `rgmii_ethernet_mac_config` tasks communicate with the PHY and connect to the application via the three interfaces. The previously declared RGMII ports as arguments as well as an argument that determines whether the 802.1Qav traffic shaper is enabled.

The `smi` task is part of the Ethernet library and controls the SMI protocol to configure the PHY. It connects to the `ar8035_phy_driver` task which connects configuration of the PHY

The RGMII MAC is split into two tasks so that other tasks can be placed on the same logical core as the config task. In this example, the PHY driver task is placed on that core.

In this example, the `rgmii_ethernet_mac` task has two `null` arguments. These are two optional streaming channels parameters that are not used in this example but can be used for high priority, high speed traffic (see the Ethernet library user guide for details).

2.4 The PHY driver

The PHY drive task connects to both the Ethernet MAC (via the `ethernet_cfg_if` interface for configuration) and the SMI driver (via the `smi_if` interface):

```
[[combinable]]
void ar8035_phy_driver(client interface smi_if smi,
                client interface ethernet_cfg_if eth) {
  ethernet_link_state_t link_state = ETHERNET_LINK_DOWN;
  ethernet_speed_t link_speed = LINK_1000_MBPS_FULL_DUPLEX;
  const int phy_reset_delay_ms = 1;
  const int link_poll_period_ms = 1000;
  const int phy_address = 0x4;
  timer tmr;
  int t;
  tmr :> t;
  p_eth_reset <: 0;
  delay_milliseconds(phy_reset_delay_ms);
  p_eth_reset <: 1;

  while (smi_phy_is_powered_down(smi, phy_address));
  smi_configure(smi, phy_address, LINK_1000_MBPS_FULL_DUPLEX, SMI_ENABLE_AUTONEG);

  while (1) {
    select {
    case tmr when timerafter(t) :> t:
      ethernet_link_state_t new_state = smi_get_link_state(smi, phy_address);
      // Read AR8035 status register bits 15:14 to get the current link speed
      if (new_state == ETHERNET_LINK_UP) {
        link_speed = (ethernet_speed_t)(smi.read_reg(phy_address, 0x11) >> 14) & 3;
      }
      if (new_state != link_state) {
        link_state = new_state;
        eth.set_link_state(0, new_state, link_speed);
      }
      t += link_poll_period_ms * XS1_TIMER_KHZ;
      break;
    }
  }
}
```

The first action the drive does is wait for the PHY to power up and then configure the PHY. This is done via library functions provided by the Ethernet library.

The main body of the drive is an infinite loop that periodically reacts to a timer event in an xC `select` statement. A a set period it checks the state of the PHY over SMI and then informs the MAC of this state via the `eth.set_link_state` call. This way the MAC can know about link up/down events or change of link speed.

2.5 ICMP Packet Processing

The packet processing in the application is handled by the `icmp_server` task which is defined in the file `icmp.xc`. This function connects to the ethernet MAC via a transmit, receive and configuration interface:

```
[[combinable]]
void icmp_server(client ethernet_cfg_if cfg,
                 client ethernet_rx_if rx,
                 client ethernet_tx_if tx,
                 const unsigned char ip_address[4],
                 otp_ports_t &otp_ports)
{
```

The first thing the task performs is configuring its connection to the MAC. The MAC address is configured by reading a MAC address out of OTP (using the `otp_board_info_get_mac` function from the OTP reading library) and then calling the `set_macaddr` interface function:

```
unsigned char mac_address[MACADDR_NUM_BYTES];
ethernet_macaddr_filter_t macaddr_filter;

// Get the mac address from OTP and set it in the ethernet component
otp_board_info_get_mac(otp_ports, 0, mac_address);

size_t index = rx.get_index();
cfg.set_macaddr(0, mac_address);
```

After this, the task configures filters to determine which type of packets is will receive from the MAC:

```
memcpy(macaddr_filter.addr, mac_address, sizeof mac_address);
cfg.add_macaddr_filter(index, 0, macaddr_filter);

// Add broadcast filter
memset(macaddr_filter.addr, 0xff, sizeof mac_address);
cfg.add_macaddr_filter(index, 0, macaddr_filter);

// Only allow ARP and IP packets to the app
cfg.add_ethertype_filter(index, 0x0806);
cfg.add_ethertype_filter(index, 0x0800);
```

The task then proceeds into an infinite loop that waits for a packet from the MAC and then processes it:

```
  while (1)
  {
    select {
    case rx.packet_ready():
      unsigned char rxbuf[ETHERNET_MAX_PACKET_SIZE];
      unsigned char txbuf[ETHERNET_MAX_PACKET_SIZE];
      ethernet_packet_info_t packet_info;
      rx.get_packet(packet_info, rxbuf, ETHERNET_MAX_PACKET_SIZE);

      if (packet_info.type != ETH_DATA)
        continue;

      if (is_valid_arp_packet(rxbuf, packet_info.len, ip_address))
      {
        int len = build_arp_response(rxbuf, txbuf, mac_address, ip_address);
        tx.send_packet(txbuf, len, ETHERNET_ALL_INTERFACES);
        debug_printf("ARP response sent\n");
      }
      else if (is_valid_icmp_packet(rxbuf, packet_info.len, ip_address))
      {
        int len = build_icmp_response(rxbuf, txbuf, mac_address, ip_address);
        tx.send_packet(txbuf, len, ETHERNET_ALL_INTERFACES);
        debug_printf("ICMP response sent\n");
      }
      break;
    }
  }
}
```

The xC `select` statement will wait for the event `rx.packet_ready()` which is a receive notification from the MAC (see the Ethernet library user guide for details of the ethernet receive interface). When a packet arrives the `rx.get_packet` call will retreive the packet from the MAC.

After the packet is processed the `tx.send_packet` call will send the created reponse packet to the MAC.

Details of the packet processing functions `is_valid_arp_packet`, `build_arp_response`, `is_valid_icmp_packet` and `build_icmp_response` can be found in the `icmp.xc` file. The functions implement the ICMP protocol.

APPENDIX A - Demo Hardware Setup

- To run the demo, connect the PC to the XTAG USB debug adapter to xCORE-200 explorer XSYS connector
- Connect the XTAG to the host PC using a USB cable
- Connect the ethernet jack to the host PC or to the network switch using an ethernet cable.

APPENDIX B - Launching the demo device

Once the application source code is imported into the tools you can edit the demo to configure the IP address the ICMP code uses. This is declared as a data structure in `main.xc` (which is then passed to the `icmp_server` function):

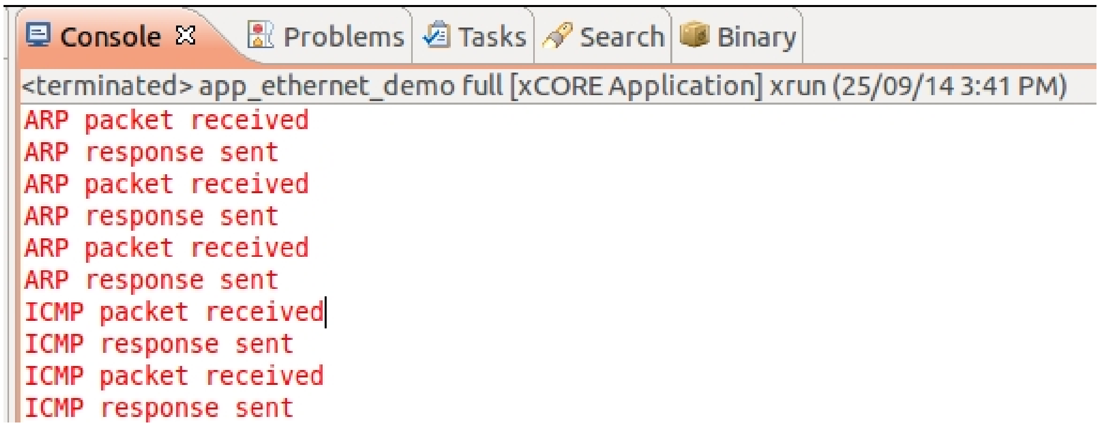
```
static unsigned char ip_address[4] = {192, 168, 1, 178};
```

Alter this value to an IP address that works on your network.

You can now 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-200 explorer. Here you use the tools to load the application over JTAG onto the xCORE device.

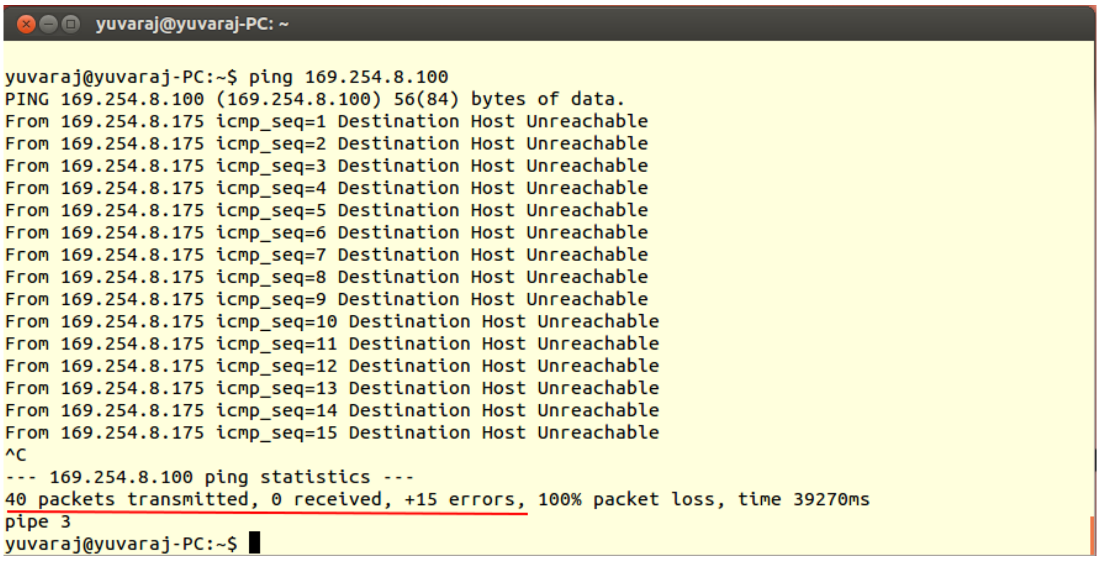
- Select **Run Configuration**.
- In **Main** menu, enable **xSCOPE** in Target I/O options.
- In **XScope** menu, enable **Offline [XScope] Mode**.
- Click **Apply** and then **Run**.

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



```
<terminated> app_ethernet_demo full [xCORE Application] xrun (25/09/14 3:41 PM)
ARP packet received
ARP response sent
ARP packet received
ARP response sent
ARP packet received
ARP response sent
ICMP packet received
ICMP response sent
ICMP packet received
ICMP response sent
```

Figure 2: Response on xTIMEcomposer console for **Ping**



```
yuvaraj@yuvaraj-PC:~$ ping 169.254.8.100
PING 169.254.8.100 (169.254.8.100) 56(84) bytes of data.
From 169.254.8.175 icmp_seq=1 Destination Host Unreachable
From 169.254.8.175 icmp_seq=2 Destination Host Unreachable
From 169.254.8.175 icmp_seq=3 Destination Host Unreachable
From 169.254.8.175 icmp_seq=4 Destination Host Unreachable
From 169.254.8.175 icmp_seq=5 Destination Host Unreachable
From 169.254.8.175 icmp_seq=6 Destination Host Unreachable
From 169.254.8.175 icmp_seq=7 Destination Host Unreachable
From 169.254.8.175 icmp_seq=8 Destination Host Unreachable
From 169.254.8.175 icmp_seq=9 Destination Host Unreachable
From 169.254.8.175 icmp_seq=10 Destination Host Unreachable
From 169.254.8.175 icmp_seq=11 Destination Host Unreachable
From 169.254.8.175 icmp_seq=12 Destination Host Unreachable
From 169.254.8.175 icmp_seq=13 Destination Host Unreachable
From 169.254.8.175 icmp_seq=14 Destination Host Unreachable
From 169.254.8.175 icmp_seq=15 Destination Host Unreachable
^C
--- 169.254.8.100 ping statistics ---
40 packets transmitted, 0 received, +15 errors, 100% packet loss, time 39270ms
pipe 3
yuvaraj@yuvaraj-PC:~$
```

Figure 3: 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 2)

APPENDIX C - 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 D - Full Source code listing

D.1 Source code for main.xc

```
// Copyright (c) 2015-2016, XMOS Ltd, All rights reserved
#include <xs1.h>
#include <platform.h>
#include "otp_board_info.h"
#include "ethernet.h"
#include "icmp.h"
#include "smi.h"
#include "debug_print.h"

// These ports are for accessing the OTP memory
otp_ports_t otp_ports = on tile[0]: OTP_PORTS_INITIALIZER;

rgmii_ports_t rgmii_ports = on tile[1]: RGMII_PORTS_INITIALIZER;

port p_smi_mdio   = on tile[1]: XS1_PORT_1C;
port p_smi_mdc    = on tile[1]: XS1_PORT_1D;
port p_eth_reset  = on tile[1]: XS1_PORT_1N;

static unsigned char ip_address[4] = {192, 168, 1, 178};


// An enum to manage the array of connections from the ethernet component
// to its clients.
enum eth_clients {
  ETH_TO_ICMP,
  NUM_ETH_CLIENTS
};

enum cfg_clients {
  CFG_TO_ICMP,
  CFG_TO_PHY_DRIVER,
  NUM_CFG_CLIENTS
};

[[combinable]]
void ar8035_phy_driver(client interface smi_if smi,
                client interface ethernet_cfg_if eth) {
  ethernet_link_state_t link_state = ETHERNET_LINK_DOWN;
  ethernet_speed_t link_speed = LINK_1000_MBPS_FULL_DUPLEX;
  const int phy_reset_delay_ms = 1;
  const int link_poll_period_ms = 1000;
  const int phy_address = 0x4;
  timer tmr;
  int t;
  tmr :> t;
  p_eth_reset <: 0;
  delay_milliseconds(phy_reset_delay_ms);
  p_eth_reset <: 1;

  while (smi_phy_is_powered_down(smi, phy_address));
  smi_configure(smi, phy_address, LINK_1000_MBPS_FULL_DUPLEX, SMI_ENABLE_AUTONEG);

  while (1) {
    select {
    case tmr when timerafter(t) :> t:
      ethernet_link_state_t new_state = smi_get_link_state(smi, phy_address);
      // Read AR8035 status register bits 15:14 to get the current link speed
      if (new_state == ETHERNET_LINK_UP) {
        link_speed = (ethernet_speed_t)(smi.read_reg(phy_address, 0x11) >> 14) & 3;
      }
      if (new_state != link_state) {
        link_state = new_state;
        eth.set_link_state(0, new_state, link_speed);
      }
      t += link_poll_period_ms * XS1_TIMER_KHZ;
      break;
    }
  }
}
```

```
int main()
{
  ethernet_cfg_if i_cfg[NUM_CFG_CLIENTS];
  ethernet_rx_if i_rx[NUM_ETH_CLIENTS];
  ethernet_tx_if i_tx[NUM_ETH_CLIENTS];
  streaming chan c_rgmii_cfg;
  smi_if i_smi;

  par {
    on tile[1]: rgmii_ethernet_mac(i_rx, NUM_ETH_CLIENTS,
                                   i_tx, NUM_ETH_CLIENTS,
                                   null, null,
                                   c_rgmii_cfg,
                                   rgmii_ports,
                                   ETHERNET_DISABLE_SHAPER);
    on tile[1].core[0]: rgmii_ethernet_mac_config(i_cfg, NUM_CFG_CLIENTS, c_rgmii_cfg);
    on tile[1].core[0]: ar8035_phy_driver(i_smi, i_cfg[CFG_TO_PHY_DRIVER]);

    on tile[1]: smi(i_smi, p_smi_mdio, p_smi_mdc);

    on tile[0]: icmp_server(i_cfg[CFG_TO_ICMP],
                            i_rx[ETH_TO_ICMP], i_tx[ETH_TO_ICMP],
                            ip_address, otp_ports);
  }
  return 0;
}
```

D.2 Source code for icmp.xc

```
// Copyright (c) 2013-2017, XMOS Ltd, All rights reserved
#include <debug_print.h>
#include <xclib.h>
#include <stdint.h>
#include <ethernet.h>
#include <otp_board_info.h>
#include <string.h>

static unsigned short checksum_ip(const unsigned char frame[34])
{
  int i;
  unsigned accum = 0;

  for (i = 14; i < 34; i += 2)
  {
    accum += *((const uint16_t*)(frame + i));
  }

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

  accum = byterev(~accum) >> 16;

  return accum;
}


static int build_arp_response(unsigned char rxbuf[64],
                              unsigned char txbuf[64],
                              const unsigned char own_mac_addr[MACADDR_NUM_BYTES],
                              const unsigned char own_ip_addr[4])
{
  unsigned word;
  unsigned char byte;

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

  txbuf[28] = own_ip_addr[0];
  txbuf[29] = own_ip_addr[1];
  txbuf[30] = own_ip_addr[2];
  txbuf[31] = own_ip_addr[3];

  for (int i = 0; i < MACADDR_NUM_BYTES; i++)
  {
    txbuf[22 + i] = own_mac_addr[i];
    txbuf[6 + i] = own_mac_addr[i];
  }
  txbuf[12] = 0x08;
  txbuf[13] = 0x06;
  txbuf[14] = 0x00;
  txbuf[15] = 0x01;
  txbuf[16] = 0x08;
  txbuf[17] = 0x00;
  txbuf[18] = 0x06;
  txbuf[19] = 0x04;
  txbuf[20] = 0x00;
  txbuf[21] = 0x02;

  // Typically 48 bytes (94 for IPv6)
  for (int i = 42; i < 64; i++)
  {
    txbuf[i] = 0x00;
  }

  return 64;
}


static int is_valid_arp_packet(const unsigned char rxbuf[nbytes],
                               unsigned nbytes,
                               const unsigned char own_ip_addr[4])
{
  if (rxbuf[12] != 0x08 || rxbuf[13] != 0x06)
    return 0;

  debug_printf("ARP packet received\n");

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

  return 1;
}
```

```
static int build_icmp_response(unsigned char rxbuf[], unsigned char txbuf[],
                               const unsigned char own_mac_addr[MACADDR_NUM_BYTES],
                               const unsigned char own_ip_addr[4])
{
  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 < MACADDR_NUM_BYTES; i++)
  {
    txbuf[i] = rxbuf[6 + i];
  }
  for (int i = 0; i < 4; i++)
  {
    txbuf[30 + i] = rxbuf[26 + i];
  }
  icmp_checksum = byterev(((const unsigned int *) rxbuf)[9]) >> 16;
  for (int i = 0; i < 4; i++)
  {
    txbuf[38 + i] = rxbuf[38 + i];
  }
  totallen = byterev(((const unsigned int *) rxbuf)[4]) >> 16;
  datalen = totallen - 28;
  for (int i = 0; i < datalen; i++)
  {
    txbuf[42 + i] = rxbuf[42+i];
  }

  for (int i = 0; i < MACADDR_NUM_BYTES; i++)
  {
    txbuf[6 + i] = own_mac_addr[i];
  }
  ((unsigned int *) txbuf)[3] = 0x00450008;
  totallen = byterev(28 + datalen) >> 16;
  ((unsigned int *) txbuf)[4] = totallen;
  ip_checksum += totallen;
  ((unsigned int *) txbuf)[5] = 0x01000000 | (ttl << 16);
  ((unsigned int *) txbuf)[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;
}
```

```
static int is_valid_icmp_packet(const unsigned char rxbuf[nbytes],
                                unsigned nbytes,
                                const unsigned char own_ip_addr[4])
{
  unsigned totallen;
  if (rxbuf[23] != 0x01)
    return 0;

  debug_printf("ICMP packet received\n");

  if (((const unsigned int *) rxbuf)[3] != 0x00450008)
  {
    debug_printf("Invalid et_ver_hdrl_tos\n");
    return 0;
  }
  if ((((const unsigned int *) rxbuf)[8] >> 16) != 0x0008)
  {
    debug_printf("Invalid type_code\n");
    return 0;
  }
  for (int i = 0; i < 4; i++)
  {
    if (rxbuf[30 + i] != own_ip_addr[i])
    {
      debug_printf("Not for us\n");
      return 0;
    }
  }

  totallen = byterev(((const unsigned int *) rxbuf)[4]) >> 16;
  if (nbytes > 60 && nbytes != totallen + 14)
  {
    debug_printf("Invalid size (nbytes:%d, totallen:%d)\n", nbytes, totallen+14);
    return 0;
  }
  if (checksum_ip(rxbuf) != 0)
  {
    debug_printf("Bad checksum\n");
    return 0;
  }

  return 1;
}

[[combinable]]
void icmp_server(client ethernet_cfg_if cfg,
                 client ethernet_rx_if rx,
                 client ethernet_tx_if tx,
                 const unsigned char ip_address[4],
                 otp_ports_t &otp_ports)
{
  unsigned char mac_address[MACADDR_NUM_BYTES];
  ethernet_macaddr_filter_t macaddr_filter;

  // Get the mac address from OTP and set it in the ethernet component
  otp_board_info_get_mac(otp_ports, 0, mac_address);

  size_t index = rx.get_index();
  cfg.set_macaddr(0, mac_address);

  memcpy(macaddr_filter.addr, mac_address, sizeof mac_address);
  cfg.add_macaddr_filter(index, 0, macaddr_filter);

  // Add broadcast filter
  memset(macaddr_filter.addr, 0xff, sizeof mac_address);
  cfg.add_macaddr_filter(index, 0, macaddr_filter);

  // Only allow ARP and IP packets to the app
  cfg.add_ethertype_filter(index, 0x0806);
  cfg.add_ethertype_filter(index, 0x0800);

  debug_printf("Test started\n");
  while (1)
  {
    select {
    case rx.packet_ready():
      unsigned char rxbuf[ETHERNET_MAX_PACKET_SIZE];
      unsigned char txbuf[ETHERNET_MAX_PACKET_SIZE];
      ethernet_packet_info_t packet_info;
      rx.get_packet(packet_info, rxbuf, ETHERNET_MAX_PACKET_SIZE);

      if (packet_info.type != ETH_DATA)
        continue;

      if (is_valid_arp_packet(rxbuf, packet_info.len, ip_address))
      {
        int len = build_arp_response(rxbuf, txbuf, mac_address, ip_address);
        tx.send_packet(txbuf, len, ETHERNET_ALL_INTERFACES);
        debug_printf("ARP response sent\n");
      }
      else if (is_valid_icmp_packet(rxbuf, packet_info.len, ip_address))
      {
        int len = build_icmp_response(rxbuf, txbuf, mac_address, ip_address);
        tx.send_packet(txbuf, len, ETHERNET_ALL_INTERFACES);
        debug_printf("ICMP response sent\n");
      }
      break;
    }
  }
}
```