XMOS Layer 2 Ethernet MAC Component

REV A

Publication Date: 2013/7/8
XMOS © 2013, All Rights Reserved.

Table of Contents

1 Ethernet Layer 2 MAC Overview **3**
1.1 Component Summary . . . 3

2 Ethernet Mac Description **4**
2.1 FULL Implementation . . . 4
2.1.1 Buffers and Queues . . . 4
2.1.2 Filtering . . . 5
2.1.3 Timestamping . . . 5
2.2 LITE implementation . . . 6
2.3 MAC Address Storage . . . 6

3 Ethernet Programming Guide **7**
3.1 Getting started . . . 7
3.1.1 Installation . . . 7
3.2 Source code structure . . . 7
3.2.1 Key Files . . . 7
3.3 A Sample Ethernet Application (tutorial) . . . 7
3.3.1 Makefile . . . 8
3.3.2 ethernet_conf.h . . . 9
3.3.3 mac_custom_filter . . . 9
3.3.4 Top level program structure . . . 10
3.3.5 Ethernet packet processing . . . 11
3.3.6 Running the application . . . 12

4 Ethernet API **13**
4.1 Configuration Defines . . . 13
4.2 Configuration Defines for FULL implementation . . . 13
4.3 Configuration defines for LITE implementation . . . 15
4.4 Custom Filter Function . . . 15
4.5 Data Structures . . . 15
4.6 MAC Server API . . . 17
4.7 RX Client API . . . 17
4.7.1 Packet Receive Functions . . . 17
4.7.2 Configuration Functions . . . 20
4.8 TX Client API . . . 21
4.8.1 Packet Transmit Functions . . . 21
4.8.2 Configuration Functions . . . 22

5 SMI Component API **24**
5.1 Configuration Defines . . . 24
5.2 Data Structures . . . 24
5.3 Phy API . . . 25

6 XMOS Development Board Support Component **27**
6.1 sliceKIT Core Board . . . 27

1 Ethernet Layer 2 MAC Overview

IN THIS CHAPTER

- Component Summary

The layer 2 MAC component implements a layer 2 ethernet MAC. It provides both MII communication to the PHY and MAC transport layer for ethernet packets and enables several clients to connect to it and send and receive packets.

Two independent implementations are available. The FULL implementation runs on 5 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 but is restricted to a single receive and trasnmit client and does not support any advanced features.

1.1 Component Summary

Functionality	
Provides MII ethernet interface and MAC with customizable filtering and accurate packet timestamping.	
Supported Standards	
Ethernet	IEEE 802.3u (MII)
Supported Devices	
XMOS Devices	XS1-G4 XS1-L2 XS1-L1
Requirements	
XMOS Desktop Tools	v12.0 or later
Ethernet	MII compatible 100Mbit PHY
Licensing and Support	
Component code provided without charge from XMOS. Component code is maintained by XMOS.	

2 Ethernet Mac Description

IN THIS CHAPTER

- FULL Implementation
- LITE implementation
- MAC Address Storage

The ethernet MAC runs on two or five logical cores depending on the chosen implementation and communicates to client tasks over channels. The server can connect to several clients and each channel connection to the server is for either RX (receiving packets from the MAC) or TX (transmitting packets to the MAC) operation.

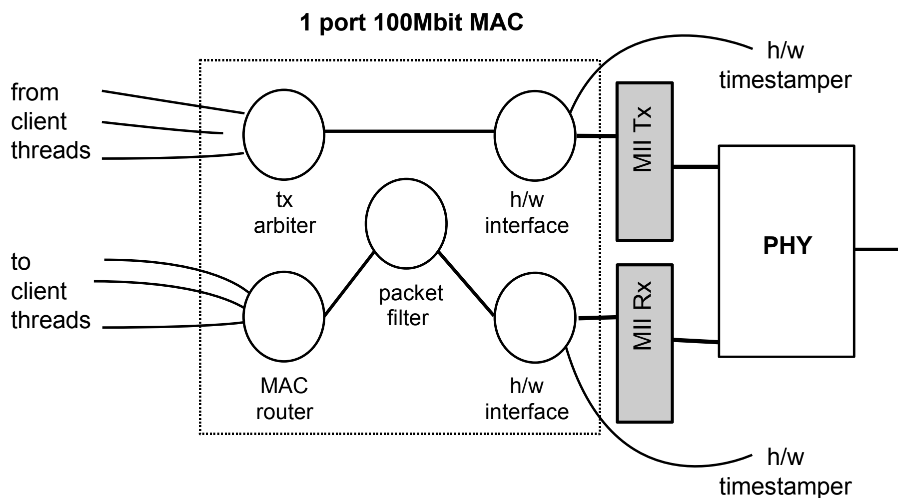

Figure 1: MAC component (FULL implementation)

2.1 FULL Implementation

2.1.1 Buffers and Queues

The MAC maintains a two sets of buffers: one for incoming packets and one for outgoing packets. These buffers are arranged into several queues.

Incoming buffers move around the different queues as follows:

- Empty buffers are in the incoming queue awaiting a packet coming in from the MII interfaces

- Buffers received from the MII interface are filtered (see below) and if they need to be kept then are moved into a forwarding queue.
- 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.

Outgoing buffers move around the different queues as follows:

- Empty buffers are an empty queue awaiting a packet coming in from a 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.

The number of buffers available can be set in the `ethernet_conf.h` configuration file (see §4.1).

2.1.2 Filtering

After incoming packets are received they are filtered. An initial filter is done 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 application must supply custom code to perform this filtering.

The user must supply a definition of the function mac_custom_filter(). This function can inspect incoming packets in any manner suitable for applications and then returns either 0 if the packet is to be dropped or a number which the clients can then use to determine which packets they wish to receive (using the client function mac_set_custom_filter().

2.1.3 Timestamping

On receipt of a ethernet frame over MII a timestamp is taken of 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. The timestamp is stored with the buffer data and can be retrieved by the client by using the mac_rx_timed() function.

On transmission of a ethernet frame over MII a timestamp is also taken. The timestamp is also taken at the end of the preamble immediately before the frame itself and is accurate to within 40ns. The client can retreive the timestamp using

the mac_tx_timed() function. In this case the timestamp is stored on transmission and placed in a queue to be sent back to the client thread.

2.2 LITE implementation

The LITE implementation does not support timestamping or multiple queues/buffering. The MAC will filter packets based on MAC address and the broadcast bit of the incoming MAC address. Any further filtering must be done by the single receive client of the ethernet server.

2.3 MAC Address Storage

The MAC address used for the server is set on instantiation of the server (as an argument to the ethernet_server() function). This address should be unique for each device. For all XMOS develop boards, a unique mac address is stored in the one time programmable rom (OTP). To retreive this address `otp_board_info_get_mac` function is provided in the module `module_otp_board_info`.

For information on programming MAC addresses into OTP please contact XMOS for detalis.

3 Ethernet Programming Guide

IN THIS CHAPTER

- Getting started
- Source code structure
- A Sample Ethernet Application (tutorial)

This section provides information on how to program applications using the ethernet MAC component.

3.1 Getting started

3.1.1 Installation

You can import the layer 2 MAC component and example applications from the xSOFTip browser in the xTIMEcomposer tool.

3.2 Source code structure

Source code can be found across several modules:

- `module_ethernet` contains the main MAC code
- `module_ethernet_smi` contains the code for controlling an ethernet phy via the SMI configuration protocol
- `module_ethernet_board_support` contains header files for common XMOS development boards allowing easy initialization of port structures.

Which modules are compiled into the application is controlled by the `USED_MODULES` define in your application Makefile.

3.2.1 Key Files

The following header files contain prototypes of all functions required to use the ethernet component. The API is described in §4.

3.3 A Sample Ethernet Application (tutorial)

This tutorial describes a demo included in the xmos ethernet package. The demo can be found in the directory app_ethernet_demo and provides a simple ethernet application that responds to ICMP ping requests. It assumes a basic knowledge of

File	Description
ethernet.h	Ethernet main header file (includes other headers)
ethernet_server.h	Ethernet Server API header file
ethernet_rx_client.h	Ethernet Client API header file (RX)
ethernet_tx_client.h	Ethernet Client API header file (TX)

Figure 2: Key Files

XC programming. For information on XMOS programming, you can find reference material at the XMOS website[1].

To write an ethernet enabled application for an XMOS device requires several things:

1. Write a Makefile for our application
2. Provide an ethernet_conf.h configuration file
3. Provide a custom filter function
4. Write the application code that uses the component

3.3.1 Makefile

The Makefile is found in the top level directory of the application. It uses the general XMOS makefile in module_xmos_common which compiles all the source files in the application and the modules that the application uses. We only have to add a couple of configuration options.

Firstly, this application is for a sliceKIT Core Board (the SLICEKIT-L2 target) so the TARGET variable needs to be set in the Makefile.

```
# The TARGET variable determines what target system the application is
# compiled for. It either refers to an XN file in the source directories
# or a valid argument for the --target option when compiling.

TARGET = SLICEKIT-L2
```

Secondly, the application will use the ethernet module (and the locks module which is required by the ethernet module). So we state that the application uses these.

```
# The USED_MODULES variable lists other module used by the application.

USED_MODULES = module_ethernet module_ethernet_board_support \
               module_otp_board_info module_slicekit_support
```

[1] http://www.xmos.com/support/documentation

Given this information, the common Makefiles will build all the files in the required modules when building the application. This works from the command line (using xmake) or from Eclipse.

3.3.2 ethernet_conf.h

The ethernet_conf.h file is found in the src/ directory of the application. This file contains a series of #defines that configure the ethernet stack. The possible #defines that can be set are described in §4.1.

Within this application we set the maximum packet size we can receive to be the maximum possible allowed in the ethernet standard and set the number of buffers to be 5 packets for incoming packets and 5 for outgoing.

The maximum number of ethernet clients (chanends we can connect to the ethernet server) is set to 4 (even though we only have one client in this example).

```
// 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
```

This application has two build configurations - one for the full implementation and one for the lite.

3.3.3 mac_custom_filter

The mac_custom_filter function allows use to decide which packets get passed through the MAC. To do this, we have to provide the mac_custom_filter.h header file and a definition of the mac_custom_filter function itself.

The header file in this example just 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 describe 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

```
int mac_custom_filter(unsigned int data[]){
        if (is_ethertype((data,char[]), ethertype_arp)){
                return 1;
        }else if (is_ethertype((data,char[]), ethertype_ip)){
                return 1;
        }

        return 0;
}
```

This function returns 0 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. In this case the function returns 1. The defintions os is_broadcast, is_ethertype and is_mac_addr are in demo.xc

3.3.4 Top level program structure

Now that we have the basic ethernet building blocks, we can build our application. This application is contained in demo.xc. Within this file is the main() function which declares some variables (primarily XC channels). It also contains a top level par construct which sets the various functional units running that make up the program.

We run the ethernet server (this is set to run on the tile `ETHERNET_DEFAULT_TILE` which is supplied by the board support module). First, the function `otp_board_info_get_mac()` reads the device mac address from ROM. The functions `eth_phy_reset()`, `smi_config()` and eth_phy_config() initialize the phy and then the main function ethernet_server() runs the ethernet component. The server communicates with other tasks via the rx and tx channel arrays.

```
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);
}
```

On tile 0 we run the demo() function as a task which takes ethernet packets and responds to ICMP ping requests. This function is described in the next section.

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

3.3.5 Ethernet packet processing

The demo() function does the actual ethernet packet processing. First the application gets the device mac address from the ethernet server.

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

Then the packet filter is set up. The mask value passed to mac_set_custom_filter() is used within the mac. After the custom_mac_filter function is run, if the result is non-zero then the result is and-ed against the mask. If this is non-zero then the packet is forwarded to the client.

So in this case, the mask is 1 so all packets that get a result of 1 from custom_mac_filter function will get passed to this client.

```
#ifdef CONFIG_FULL
mac_set_custom_filter(rx, 0x1);
#endif
```

Note that this is only for build configuration that uses the FULL configuration. If we are using the LITE configuration the filtering is done after the client recieves the packet later on.

After we are set up to receive the correct packets we can go into the main loop that responds to ARP and ICMP packets.

The first task in the loop is to receive a packet into the rxbuf buffer using the mac_rx() function.

```
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) != 0x1)
    continue;
#endif
```

Here we can see the filtering that needs to be done for the LITE configuration.

When the packet is received it may be an ARP or IP packet since both get past our filter. First we check if it is an ARP packet, if so then we build the response (in

the txbuf array) and send it out over ethernet using the mac_tx() function. The functions is_valid_arp_packet and build_arp_response are defined demo.xc.

```
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");
  }
```

If the packet is not an ARP packet we check if it is an ICMP packet and in the same way build a response and send it out.

```
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");
  }
```

3.3.6 Running the application

To test the application the following define in demo.xc needs to be set to an IP address that is routable in the network that the application is to be tested on.

```
// NOTE: YOU MAY NEED TO REDEFINE THIS TO AN IP ADDRESS THAT WORKS
// FOR YOUR NETWORK
#define OWN_IP_ADDRESS {192, 168, 1, 178}
```

Once this is done, the demo can be compiled and the XC-2 connected to a PC. Pinging the IP address defined should now get a response e.g.:

```
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=2.97 ms
64 bytes from 192.168.0.3: icmp_seq=2 ttl=64 time=2.93 ms
64 bytes from 192.168.0.3: icmp_seq=3 ttl=64 time=2.91 ms
64 bytes from 192.168.0.3: icmp_seq=4 ttl=64 time=2.96 ms
...
```

4 Ethernet API

IN THIS CHAPTER

- Configuration Defines
- Configuration Defines for FULL implementation
- Configuration defines for LITE implementation
- Custom Filter Function
- Data Structures
- MAC Server API
- RX Client API
- TX Client API

4.1 Configuration Defines

The file `ethernet_conf.h` may be provided in the application source code. This file can set the following defines:

ETHERNET_DEFAULT_IMPLEMENTATION

This define can be set to `full` or `lite` and determines which implementation is chosen by default when the application makes calls to `ethernet_server` etc.

4.2 Configuration Defines for FULL implementation

MAX_ETHERNET_PACKET_SIZE

This define sets the largest packet size in bytes that the ethernet mac will receive. The default is the largest possible ethernet packet size (1518 bytes). Setting this to a smaller value will save memory but restrict the type of packets you can receieve.

NUM_MII_RX_BUF

Number of incoming packets that will be buffered within the MAC.

NUM_MII_TX_BUF

Number of outgoing packets that will be buffered within the MAC.

MAX_ETHERNET_CLIENTS

The maximum number of clients that can be connected to the ethernet_server() function via the rx and tx channel arrays.

NUM_ETHERNET_PORTS

The number of ethernet ports to support. Maximum value is 2 in the current implementation.

ETHERNET_TX_HP_QUEUE

Define this constant to include the high priority transmit queueing mechanism. This enables frames which have an ethernet VLAN priority tag to be queued in a high priority queue, which in turn can be managed with the 802.1qav transmit traffic shaper.

ETHERNET_RX_HP_QUEUE

Define this constant to include high priority reception of ethernet VLAN priority tagged traffic. This traffic will be queued into a fast queue and delivered to the clients ahead of non-tagged traffic.

ETHERNET_TRAFFIC_SHAPER

If high priority transmit queueing is in use (see *ETHERNET_TX_HP_QUEUE*) then this enables the 802.1qav traffic shaping algorithm.

MII_RX_BUFSIZE_HIGH_PRIORITY

The number of quadlets (4 byte integers) of space in the high priority receive buffer. The buffer will actually be two full packets longer than this to avoid the need to be circular. This constant applies when the high priority receive queue is in use.

MII_RX_BUFSIZE_LOW_PRIORITY

The number of quadlets (4 byte integers) of space in the low priority receive buffer. The buffer will actually be two full packets longer than this to avoid the need to be circular. This constant applies when the high priority receive is in use.

MII_RX_BUFSIZE

The number of quadlets (4 byte integers) of space in the low priority receive buffer. The buffer will actually be two full packets longer than this to avoid the need to be circular. This constant applies when the high priority receive is not in use.

MII_TX_BUFSIZE

The number of quadlets (4 byte integers) of space in the low priority transmit buffer. The buffer will actually be two full packets longer than this to avoid the need to be circular.

MII_TX_BUFSIZE_HIGH_PRIORITY

The number of quadlets (4 byte integers) of space in the high priority transmit buffer. The buffer will actually be two full packets longer than this to avoid the need to be circular. This constant applies when the high priority receive is in use.

ENABLE_ETHERNET_SOURCE_ADDRESS_WRITE

By defining this preprocessor symbol, the source MAC address will be automatically filled in with the MAC address passed to the port during initialization.

4.3 Configuration defines for LITE implementation

4.4 Custom Filter Function

For the FULL implementation, every application is required to provide this function. It also needs to be prototyped (or defined as an inline definition) in the header file `mac_custom_filter.h`.

`int mac_custom_filter(unsigned int data[])`

This function examines an ethernet packet and returns a filter number to allow different clients to obtain different types of packet. The function *must* run within 6us to allow 100Mbit filtering of packets.

This function has the following parameters:

`data` This array contains the ethernet packet. It does not include the preamble but does include the layer 2 header or the packet.

This function returns:

0 if the packet is not wanted by the application or a number that can be registed by mac_set_custom_filter() by a client. Clients register a mask so the number is usually made up of a bit per unique client destination for the packet.

4.5 Data Structures

Depending on the implementation you must supply a different port structure. The type `mii_interface_t` will be set to one of this structures depending on the `ETHERNET_DEFAULT_IMPLEMENTATION` define.

`mii_interface_full_t`

Structure containing resources required for the MII ethernet interface.

This structure contains resources required to make up an MII interface. It consists of 7 ports and 2 clock blocks.

The clock blocks can be any available clock blocks and will be clocked of incoming rx/tx clock pins.

This structure has the following members:

`clock clk_mii_rx`
MII RX Clock Block.

`clock clk_mii_tx`
MII TX Clock Block.

`in port p_mii_rxclk`
MII RX clock wire.

`in port p_mii_rxer`
MII RX error wire.

`in buffered port:32 p_mii_rxd`
: MII RX data wire.

`in port p_mii_rxdv`
: MII RX data valid wire.

`in port p_mii_txclk`
: MII TX clock wire.

`out port p_mii_txen`
: MII TX enable wire.

`out buffered port:32 p_mii_txd`
: MII TX data wire.

`mii_interface_lite_t`

This structure has the following members:

`clock clk_mii_rx`
: MII RX Clock Block.

`clock clk_mii_tx`
: MII TX Clock Block.

`in port p_mii_rxclk`
: MII RX clock wire.

`in port p_mii_rxer`
: MII RX error wire.

`in buffered port:32 p_mii_rxd`
: MII RX data wire.

`in port p_mii_rxdv`
: MII RX data valid wire.

`in port p_mii_txclk`
: MII TX clock wire.

`out port p_mii_txen`
: MII TX enable wire.

`out buffered port:32 p_mii_txd`
: MII TX data wire.

`in port p_mii_timing`
: A port that is not used for anything, used by the LLD for timing purposes.

  Must be clocked of the reference clock

4.6 MAC Server API

```
void ethernet_server(mii_interface_t &mii,
                     smi_interface_t & ?smi,
                     char mac_address[],
                     chanend rx[],
                     int num_rx,
                     chanend tx[],
                     int num_tx)
```

Single MII port MAC/ethernet server.

This function provides both MII layer and MAC layer functionality. It runs in 5 threads and communicates to clients over the channel array parameters.

The clients connected via the rx/tx channels can communicate with the server using the APIs found in ethernet_rx_client.h and ethernet_tx_client.h

This function has the following parameters:

`mii`	The mii interface resources that the server will connect to
`mac_address`	The mac_address the server will use. This should be a two-word array that stores the 6-byte macaddr in a little endian manner (so reinterpreting the array as a char array is as one would expect)
`rx`	An array of chanends to connect to clients of the server who wish to receive packets.
`num_rx`	The number of clients connected to the rx array
`tx`	An array of chanends to connect to clients of the server who wish to transmit packets.
`num_tx`	The number of clients connected to the txx array
`smi`	An optional parameter of resources to connect to a PHY (via SMI) to check when the link is up.

4.7 RX Client API

4.7.1 Packet Receive Functions

```
void mac_rx(chanend c_mac,
            unsigned char buffer[],
            unsigned int &len,
            unsigned int &src_port)
```

This function receives a complete frame (i.e.

src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32 from the ethernet server.

This function is selectable i.e. it can be used as a case in a select e.g.

```
select {
   ...
   case mac_rx(...):
      break;
    ...
  }
```

This function has the following parameters:

`c_mac`	A chanend connected to the ethernet server
`buffer`	The buffer to fill with the incoming packet
`src_port`	A reference parameter to be filled with the ethernet port the packet came from.
`len`	A reference parameter to be filled with the length of the received packet in bytes.

```
void mac_rx_timed(chanend c_mac,
                  unsigned char buffer[],
                  unsigned int &len,
                  unsigned int &time,
                  unsigned int &src_port)
```

This function receives a complete frame (i.e.

src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32. It also timestamps the arrival of the frame.

This function is selectable.

This function has the following parameters:

`c_mac`	A chanend connected to the ethernet server
`buffer`	The buffer to fill with the incoming packet
`time`	A reference parameter to be filled with the timestamp of the packet
`len`	A reference parameter to be filled with the length of the received packet in bytes.
`src_port`	A reference parameter to be filled with the ethernet port the packet came from.

```
void safe_mac_rx(chanend c_mac,
                 unsigned char buffer[],
                 unsigned int &len,
                 unsigned int &src_port,
```

```
    int n)
```

This function receives a complete frame (i.e.

src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32. In addition it will only fill the given buffer up to a specified length.

This function is selectable.

This function has the following parameters:

c_mac	A chanend connected to the ethernet server
buffer	The buffer to fill with the incoming packet
len	A reference parameter to be filled with the length of the received packet in bytes.
src_port	A reference parameter to be filled with the ethernet port the packet came from.
n	The maximum number of bytes to fill the supplied buffer with.

```
void safe_mac_rx_timed(chanend c_mac,
                       unsigned char buffer[],
                       unsigned int &len,
                       unsigned int &time,
                       unsigned int &src_port,
                       int n)
```

This function receives a complete frame (i.e.

src/dest MAC address, type & payload), excluding pre-amble, SoF & CRC32 from the ethernet server. In addition it will only fill the given buffer up to a specified length.

This function is selectable i.e. it can be used as a case in a select.

This function has the following parameters:

c_mac	A chanend connected to the ethernet server
buffer	The buffer to fill with the incoming packet
src_port	A reference parameter to be filled with the ethernet port the packet came from.
len	A reference parameter to be filled with the length of the received packet in bytes.
n	The maximum number of bytes to fill the supplied buffer with.

```
void mac_rx_offset2(chanend c_mac,
                    unsigned char buffer[],
                    unsigned int &len,
                    unsigned int &src_port)
```

Receive a packet starting at the second byte of a buffer.

This is useful when the contents of the packet should be aligned on a different boundary.

This function has the following parameters:

`c_mac`	chanend of receive server.
`buffer`	The buffer to fill with the incoming packet
`len`	A reference parameter to be filled with the length of the received packet in bytes.
`src_port`	A reference parameter to be filled with the ethernet port the packet came from.

4.7.2 Configuration Functions

```
void mac_set_drop_packets(chanend c_mac_svr, int x)
```

Setup whether a link should drop packets or block if the link is not ready.

NOTE: setting no dropped packets does not mean no packets will be dropped. If packets are not dropped at the mac layer, it will block the mii fifo. The Mii fifo could possibly overflow and frames for other links could be dropped.

This function has the following parameters:

`c_mac_svr`	chanend of receive server.
`x`	boolean value as to whether packets should be dropped at mac layer.

```
void mac_set_queue_size(chanend c_mac_svr, int x)
```

Setup the size of the buffer queue within the mac attached to this link.

This function has the following parameters:

`c_mac_svr`	chanend connected to the mac
`x`	the required size of the queue

```
void mac_set_custom_filter(chanend c_mac_svr, int x)
```

Setup the custom filter up on a link.

For each packet, the filter value is &-ed against the result of the mac_custom_filter function. If the result is non-zero then the packet is transmitted down the link.

This function has the following parameters:

`c_mac_svr`	chanend of receive server.
`x`	filter value

4.8 TX Client API

4.8.1 Packet Transmit Functions

```
void mac_tx(chanend c_mac, unsigned int buffer[], int nbytes, int ifnum)
```

Sends an ethernet frame.

Frame includes dest/src MAC address(s), type and payload.

This function has the following parameters:

`c_mac`	channel end to tx server.
`buffer[]`	byte array containing the ethernet frame. *This must be word aligned*
`nbytes`	number of bytes in buffer
`ifnum`	the number of the eth interface to transmit to (use ETH_BROADCAST transmits to all ports)

```
void mac_tx_timed(chanend c_mac,
                  unsigned int buffer[],
                  int nbytes,
                  unsigned int &time,
                  int ifnum)
```

Sends an ethernet frame and gets the timestamp of the send.

Frame includes dest/src MAC address(s), type and payload.

This is a blocking call and returns the *actual time* the frame is sent to PHY according to the XCore 100Mhz 32-bit timer on the core the ethernet server is running.

NOTE: This function will block until the packet is sent to PHY.

This function has the following parameters:

`c_mac`	channel end connected to ethernet server.
`buffer[]`	byte array containing the ethernet frame. *This must be word aligned*
`nbytes`	number of bytes in buffer
`ifnum`	the number of the eth interface to transmit to (use ETH_BROADCAST transmits to all ports)

`time` A reference paramater that is set to the time the packet is sent to the phy

```
void mac_tx_offset2(chanend c_mac,
                    unsigned int buffer[],
                    int nbytes,
                    int ifnum)
```

Sends an ethernet frame.

Frame includes dest/src MAC address(s), type and payload.

The packet should start at offset 2 in the buffer. This allows the packet to be constructed with alignment on a different boundary, allowing for more efficient construction where many word values are not naturally aligned on word boundaries.

This function has the following parameters:

`c_mac` channel end to tx server.

`buffer[]` byte array containing the ethernet frame. *This must be word aligned*

`nbytes` number of bytes in buffer

`ifnum` the number of the eth interface to transmit to (use ETH_BROADCAST transmits to all ports)

Figure 3: Ethernet function synonyms

Synonym	Function
ethernet_send_frame	ethernet_send_frame
ethernet_send_frame_getTime	ethernet_send_frame_getTime
ethernet_send_frame_offset2	mac_tx_offset2
ethernet_get_my_mac_adrs	mac_get_macaddr

4.8.2 Configuration Functions

```
int mac_get_macaddr(chanend c_mac, unsigned char macaddr[])
```

Get the device MAC address.

This function gets the MAC address of the device (the address passed into the ethernet_server() function.

This function has the following parameters:

`c_mac` chanend end connected to ethernet server

`macaddr[]` an array of type char where the MAC address is placed (in network order).

This function returns:

zero on success and non-zero on failure.

5 SMI Component API

IN THIS CHAPTER

- Configuration Defines
- Data Structures
- Phy API

The module module_ethernet_smi is written to support SMI independently of the MII interface. Typically, Ethernet PHYs are configured on reset automatically, but the SMI interface may be useful for setting and testing register values dynamically.

There are two ways to interface SMI: using a pair of 1-bit ports, or using a single multi-bit port.

5.1 Configuration Defines

These defines can either be set in ethernet_conf.h or smi_conf.h from within your application directory.

SMI_COMBINE_MDC_MDIO

This define should be set to 1 if you want to combine MDC and MDIO onto a single bit port.

SMI_MDC_BIT

This defines the bit number on the shared port where the MDC line is. Only define this if you have a port that drives both MDC and MDIO.

SMI_MDIO_BIT

This defines the bit number on the shared port where the MDIO line is. Only define this if you have a port that drives both MDC and MDIO.

5.2 Data Structures

smi_interface_t

Structure containing resources required for the SMI ethernet phy interface.

This structure can be filled in two ways. One indicate that the SMI interface is connected using two 1-bit port, the other indicates that the interface is connected using a single multi-bit port.

If used with two 1-bit ports, set the phy_address, p_smi_mdio and p_smi_mdc as normal.

If SMI_COMBINE_MDC_MDIO is 1 then `p_smi_mdio` is ommited and `p_mdc` is assumbed to multibit port containing both mdio and mdc.

This structure has the following members:

`int phy_address`
: Address of PHY, typically 0 or 0x1F.

`port p_smi_mdio`
: MDIO port.

`port p_smi_mdc`
: MDC port.

5.3 Phy API

`void smi_init(smi_interface_t &smi)`

Function that configures the SMI ports.

No clock block is needed. Note that there is no deinit function.

This function has the following parameters:

`smi` structure containing the clock and data ports for SMI.

`void eth_phy_config(int eth100, smi_interface_t &smi)`

Function that configures the Ethernet PHY explicitly to set to autonegotiate.

This function has the following parameters:

`If` eth100 is non-zero, 100BaseT is advertised to the link peer Full duplex is always advertised

`smi` structure that defines the ports to use for SMI

`void eth_phy_config_noauto(int eth100, smi_interface_t &smi)`

Function that configures the Ethernet PHY to not autonegotiate.

This function has the following parameters:

`If` eth100 is non-zero, it is set to 100, else to 10 Mbits/s

`smi` structure that defines the ports to use for SMI

`void eth_phy_loopback(int enable, smi_interface_t &smi)`

Function that can enable or disable loopback in the phy.

This function has the following parameters:

`enable` boolean; set to 1 to enable loopback, or 0 to disable loopback.

`smi`	structure containing the ports

`int eth_phy_id(smi_interface_t &smi)`

Function that returns the PHY identification.

This function has the following parameters:

`smi`	structure containing the ports

This function returns:

the 32-bit identifier.

`int smi_check_link_state(smi_interface_t &smi)`

Function that polls whether the link is alive.

This function has the following parameters:

`smi`	structure containing the ports

This function returns:

non-zero if the link is alive; zero otherwise.

6 XMOS Development Board Support Component

IN THIS CHAPTER

- sliceKIT Core Board

The module `module_ethernet_board_support` provides defines to allow you to easily use an XMOS development board. To use the module include the following header:

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

The contents of this header varies depending on the `TARGET` defined in your `Makefile`.

With this header included you can intialize ethernet port structures using the following defines:

```
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;
```

You can also use the define `ETHERNET_DEFAULT_TILE` to refer to the tile that the ethernet ports are on.

6.1 sliceKIT Core Board

For the sliceKIT Core Board the ethernet slice could be in any of the four slots. To choose which slot the defines refer to you can set the define one of the following defines to be 1 in `ethernet_conf.h`:

- `ETHERNET_USE_CIRCLE_SLOT`
- `ETHERNET_USE_SQUARE_SLOT`
- `ETHERNET_USE_STAR_SLOT` (Not compatible with the 1v1 Core Board)
- `ETHERNET_USE_TRIANGLE_SLOT` (Not compatible with the 1v1 Core Board)

Copyright © 2013, All Rights Reserved.

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

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