# lib_ethernet: Ethernet library

Publication Date: 2025/3/15
Document Number: XM-006386-UG v4.0.1

XMOS

IN THIS DOCUMENT

# 1 Introduction

`lib_ethernet` allows interfacing to MII, RMII or RGMII Ethernet PHYs and provides the Media Access Control (MAC) function for the Ethernet stack.

Various MAC blocks are available depending on the XMOS architecture selected, desired PHY interface and line speed, as described in Table 1.

Table 1: Ethernet MAC support by XMOS device family

| XCORE Architecture | MII 100 Mb | RMII 100 Mb | GMII 1 Gb | RGMII 1 Gb |
|---|---|---|---|---|
| XS1 | Deprecated from version 4.0.0 | N/A | N/A | N/A |
| XS2 (xCORE-200) | Supported | N/A | N/A | Supported |
| XS3 (xcore.ai) | Supported | Supported | Contact XMOS | N/A |

The MII MAC is available as two types; a low resource usage version which provides standard layer 2 data access to an array of clients, and a real-time version which offers additional hardware features including:

- Hardware time-stamping of point of ingress and egress of frames supporting standards such as IEEE 802.1AS.
- Support for high priority send and receive queues and receive filtering. This allows time sensitive traffic to be prioritised over other traffic.
- Traffic shaping on egress using an IEEE 802.1Qav compliant credit based shaper.
- Configurable VLAN tag stripping on received frames.

All RMII and RGMII implementations offer the 'real-time' features as standard. See the *Real-Time MAC Features* section for more details.

In addition, all MACs support client specific filtering for both source MAC address and Ethertype. See the *Standard MAC Features* section for more details.

# 2 Usage

`lib_ethernet` is intended to be used with XCommon CMake , the *XMOS* application build and dependency management system.

To use `lib_ethernet` in an application, add `lib_ethernet`, to the list of dependent modules in the application's *CMakeLists.txt* file.

```
set(APP_DEPENDENT_MODULES "lib_ethernet")
```

All *lib_ethernet* functions can be accessed via the `ethernet.h` header file:

```
#include <ethernet.h>
```

# 3 Typical resource usage

Instantiating Ethernet on the XCORE requires resources in terms of memory, threads (MIPS), ports and other resources. The amount required depends on the feature set of the MAC. Table 2 summarises the main requirements.

Table 2: Ethernet MAC XCORE resource usage

| Configuration | Pins | Port Requirement | Clocks | RAM | Threads |
|---|---|---|---|---|---|
| 10/100 Mb/s standard MII | 13 | 5 (1-bit), 2 (4-bit), 1 (any-bit) | 2 | ~16 k | 2 |
| 10/100 Mb/s Real-Time MII | 13 | 5 (1-bit), 2 (4-bit) | 2 | ~23 k | 4 |
| 10/100 Mb/s Real-Time RMII | 7 | 3 (1-bit), 2 (4-bit) or 4 (1-bit) | 2 | ~25 k | 4 |
| 10/100/1000Mb/s RGMII | 12 | 8 (1-bit), 2 (4-bit), 2 (8-bit) | 4 | ~102 k | 8 |
| Raw MII | 13 | 5 (1-bit), 2 (4-bit) | 2 | ~10 k | 1 |
| SMI (MDIO) | 2 | 2 (1-bit) or 1 (multi-bit) | 0 | ~1 k | 0 |

**Note:** The RAM usage shown is for a typical usage rx and tx buffer size that can store multiple 1500 byte packets. The total RAM usage by the MAC will increase or decrease depending on buffer size settings which is set by the user.

**Note:** Not all ports are brought out to pins since they are used internally to the device. Hence the total port bit-count may not always match the required device pin count.

**Note:** The SMI configuration API allows the SMI task to be `distributed` which means, when placed on the same tile as the client (e.g. PHY management task), read and write API calls will be turned into function calls by the compiler. Consequently it doesn't normally require a dedicated xCORE thread. The read and write API calls always block until the last bit of the SMI transaction is complete.

# 4 Standard MAC Features

## 4.1 Feature Overview

All MACs in this library support a number of useful features which can be configured by clients.

- Support for multiple clients (Rx and Tx) allowing many tasks to share the MAC.
- Configurable Ethertype and MAC address filters for unicast, multicast and broadcast addresses and is configurable per client. The number of entries is configurable using `ETHERNET_MACADDR_FILTER_TABLE_SIZE`.
- Configurable source MAC address. This may be used in conjunction with, for example, lib_otp to provide a unique MAC address per XMOS chip.
- Link state detection allowing action to be taken by higher layers in the case of link state change.
- Separately configurable Rx and Tx buffer sizes (queues).
- VLAN aware received packet length calculation. If the VLAN tag (0x8100) is seen the header length is automatically extended by 4 octets to support the Tag Protocol Identifier (TPID) and Tag Control Information (TCI).

Transmission of packets is via an API that blocks until the frame has been copied into the transmit queue. This means the buffer size should be appropriately sized for your application or the application should tolerate blocking.

Reception of a packet blocks until a packet is available. It may however be combined with an asynchronous notification allowing the client to `select` on the XC interface whereupon it can then receive the waiting packet. This provides an efficient, event-driven API option. Please see the *API* for details on how to use the MAC.

In addition, the RMII RT MAC supports an exit command. This tears down all of the tasks associated with the MAC and frees the memory and XCORE resources used, including ports. After exit, the MAC may be re-started again. This can be helpful in cases where ports may be shared (eg. Flash memory) allowing DFU support in package constrained systems. It may also be used to support multiple connect PHY devices where redundancy is required, without costing the chip resources to support multiple MACs.

## 4.2 Receive Filter

All ethernet MACs support filtering of received packets. The filtering consists of two stages; first destination MAC address filtering followed by an optional Ethertype filter. Note that Ethertype filters are not supported on high priority receive queues.

By default the receive MAC address filter has no entries so all clients must register at least one MAC address filter entry to be able to receive packets. The size of the filter table is statically defined by `ETHERNET_MACADDR_FILTER_TABLE_SIZE` which is nominally set to 30 which is the maximum number of total entries for all clients that can be supported whilst still maintaining full line rate reception without packet drops. The MII and RMII MACs use a linear search whereas the RGMII MAC uses a hash table which offers higher performance required by the line speed at the cost of greater memory usage.

Multiple MAC addresses may be registered per client including the broadcast MAC address `FF:FF:FF:FF:FF:FF` which is needed by some protocols such as ARP, DHCP or WoL. Any MAC address matching one of the filter entries will be forwarded to the client that registered it. It is possible for multiple clients to register and share the same desti-

nation MAC address and receive the same packet. However, in the case of the real-time MACs, where a high priority receive queue is enabled, it is not possible for a mixture of high and low priority queues to filter by the same mac address; the user must choose either high priority or low priority for a given MAC address filter entry.

Once a packet has been filtered for destination MAC address and forwarded to the server for client reception, an additional Ethertype filter is optionally applied for low priority queues only. If no Ethertype filter has been registered for a client then the Ethertype field is ignored and all packets are forwarded to the client. A maximum of `ETHERNET_MAX_ETHERTYPE_FILTERS` (set to two statically) are supported per client.

For real-time MACs, VLAN tagged packets are automatically detected and the extracted Ethertype field position within the packet is automatically accounted for.

For details of the API regarding filter configuration, please see the *configuration API documentation*.

# 5 Real-Time MAC Features

In addition to all of the features outlined in the *Standard MAC Features* section, real-time (RT) MACs offer enhanced features which are useful in a number of applications such as industrial control, real-time networking and Audio/Video streaming cases. These specific features are introduced below.

## 5.1 Hardware Time Stamping

The XCORE contains architectural features supporting precise timing measurements. Specifically, a 100 MHz timer is included and the RT MACs make use of this to timestamp packets at the point of ingress and egress. This 100 MHz, 32-bit timer value has a resolution of 10 nanoseconds and the provided timestamp can be converted to nanoseconds by multiplying by 10.

When receiving packets, a reference to a structure of type `ethernet_packet_info_t` contains the timestamp of the received packet at point of ingress.

When transmitting packets, an additional Tx API is provided for the RT MAC which blocks until the packet has been transmitted and returns the time of egress.

These features, along with APIs to tune the ingress and egress latency offsets, can be used by higher layers such as IEEE 802.1AS (Timing and Synchronization) or PTP (IEEE 1588) to implement precise timing synchronisation across the network.

## 5.2 High Priority Queues

The RT MACs extend the standard client interfaces with the support of a dedicated High Priority (HP) queue. This queue allows traffic to be received or transmitted before lower priority traffic, which is useful in real-time applications where the network is shared with normal, lower priority, traffic. The MAC logic always prioritises HP packets and queues over low priority.

The dedicated HP client API uses streaming channels instead of XC interfaces which provide higher performance data transfer. A dedicated channel is used for each of the receive and transmit interfaces. Streaming channels offer higher performance at the cost of occupying a dedicated switch path which may require careful consideration if the client is placed on a different tile from the MAC. This is important due to the architectural limitation of a maximum of four inter-tile switch paths between tiles. A maximum of one HP receive and transmit client are be supported per MAC.

A flag in the filter table can manually be set when making filter entries which is then used to determine the priority level when receiving packets. This determines which queue to use.

The transmit HP queue is optionally rate limited using the Credit Based Shaper which is described below. Together, these features provide the mechanisms required by IEEE 802.1Qav enabling reliable, low-latency delivery of time-sensitive streams over Ethernet networks.

## 5.3 Credit Based Shaper

The Credit Based Shaper (CBS), in conjunction with the HP queue, limits the bandwidth of non-time-sensitive traffic and ensures a reserved bandwidth for high-priority streams.

The CBS uses the following mechanisms to manage egress rate:

- Credits: The high priority queue is assigned a "credit" that increases or decreases over time based on the network's traffic conditions.
- Idle Slope: Determines how quickly credit increases when the queue is idle (i.e., waiting to transmit).
- Transmission of data decreases credit proportionally to the number of bits sent.

If the credit is positive, the high priority stream is eligible for transmission and will always be transmitted before any low priority traffic. If the credit is negative, the high priority stream is paused until the credit returns to a positive state. By spreading traffic out evenly over time using a CBS, the queue size in each bridge and endpoint can be shorter, which in turn reduces the latency experienced by traffic as it flows through the system.

The RT MACs are passed an enum argument when instantiated which enables or disables the CBS. In addition the MAC provides an API which can adjust the high-priority transmit queue's CBS idle slope dynamically, for example, if a different bandwidth reservation is required.

The idle slope passed is a fractional value representing the number of bits per reference timer tick in a Q16.16 format defined by `MII_CREDIT_FRACTIONAL_BITS` allowing very precise control over bandwidth reservation. Please see *API* for details and an example showing how to convert from bits-per-second to the slope argument.

## 5.4 VLAN Tag Stripping

In addition to standard MAC VLAN awareness of received packets when calculating payload length, the RT MAC also includes a feature to optionally strip VLAN tags. If the VLAN tag (0x8100) is seen the header length is automatically extended by 4 octets to support the Tag Protocol Identifier (TPID) and Tag Control Information (TCI). This is done inside the MAC so that the application can directly utilise the incoming packet payload. VLAN stripping is dynamically controllable on a per-client basis.

# 6 External signal description

## 6.1 MII: Media Independent Interface

MII is an interface standardized by IEEE 802.3 that connects different types of PHYs to the same Ethernet Media Access Control (MAC). The MAC can interact with any PHY using the same hardware interface, independent of the media the PHYs are connected to.

The MII transfers data using 4 bit words (nibbles) in each direction, clocked at 25 MHz to achieve 100 Mb/s data rate.

An enable signal (TXEN) is set active to indicate start of frame and remains active until it is completed. A clock signal (TXCLK) clocks nibbles (TXD[3:0]) at 2.5 MHz for 10 Mb/s mode and 25 MHz for 100 Mb/s mode. The RXDV signal goes active when a valid frame starts and remains active throughout a valid frame duration. A clock signal (RXCLK) clocks the received nibbles (RXD[3:0]). Table 3 describes the MII signals:

Table 3: MII signals

| Port Requirement | Signal Name | Description |
|---|---|---|
| 4-bit port [Bit 3] | TXD3 | Transmit data bit 3 |
| 4-bit port [Bit 2] | TXD2 | Transmit data bit 2 |
| 4-bit port [Bit 1] | TXD1 | Transmit data bit 1 |
| 4-bit port [Bit 0] | TXD0 | Transmit data bit 0 |
| 1-bit port | TXCLK | Transmit clock (2.5/25 MHz) |
| 1-bit port | TXEN | Transmit data valid |
| 1-bit port | RXCLK | Receive clock (2.5/25 MHz) |
| 1-bit port | RXDV | Receive data valid |
| 1-bit port | RXERR | Receive data error |
| 4-bit port [Bit 3] | RX3 | Receive data bit 3 |
| 4-bit port [Bit 2] | RX2 | Receive data bit 2 |
| 4-bit port [Bit 1] | RX1 | Receive data bit 1 |
| 4-bit port [Bit 0] | RX0 | Receive data bit 0 |

Any unused 1-bit and 4-bit xCORE ports can be used for MII provided that they are on the same tile and there is enough resource to instantiate the relevant Ethernet MAC component on that tile.

## 6.2 RMII: Reduced Media Independent Interface

RMII is an interface standardized by IEEE 802.3 that connects different types of PHYs to the same Ethernet Media Access Control (MAC). The MAC can interact with any PHY using the same hardware interface, independent of the media the PHYs are connected to. It offers similar functionality to MII however offers a reduced pin-count.

The RMII transfers data using 2 bit words (half-nibbles) in each direction, clocked at 50 MHz to achieve 100 Mb/s data rate.

An enable signal (TXEN) is set active to indicate start of frame and remains active until it is completed. A common, externally provided, clock signal clocks 2 bits (TXD[1:0]) at 50 MHz for 100 Mb/s mode. The RXDV signal goes active when a valid frame starts and

remains active throughout a valid frame duration. The common clock signal clocks the received half-nibbles (RXD[1:0]).

Note that either half of a 4-bit port (upper or lower pins) may be used for data or alternatively two 1-bit ports may be used. This provides additional pinout flexibility which may be important in applications which use low pin-count packages. Both Rx and Tx have their port type set independently and can be mixed. Unused pins on a 4-bit port are ignored for Rx and driven low for Tx.

**Note:** By default most RMII PHYs supply a CRS_DV signal (carrier sense) instead of an RX_DV data valid strobe. This library requires the PHY to be configured so that the receive data strobe is set to RX_DV mode. Please check your chosen PHY supports this.

The RMII MAC requires a minimum thread speed of 75 MHz which allows all 8 hardware threads to be used on a 600 MHz xcore.ai device.

Table 4 describes the RMII signals:

Table 4: RMII signals

| Port Requirement | Signal Name | Description |
|---|---|---|
| 4-bit port [Bit 1 or 3] or 1-bit port | TXD1 | Transmit data bit 1 |
| 4-bit port [Bit 0 or 2] or 1-bit port | TXD0 | Transmit data bit 0 |
| 1-bit port | PHY_CLK | PHY clock (50 MHz) |
| 1-bit port | TXEN | Transmit data valid |
| 1-bit port | RXDV | Receive data valid |
| 4-bit port [Bit 1 or 3] or 1-bit port | RX1 | Receive data bit 1 |
| 4-bit port [Bit 0 or 2] or 1-bit port | RX0 | Receive data bit 0 |

Any unused 1-bit and 4-bit xCORE ports can be used for RMII provided that they are on the same tile and there are sufficient chip resources to instantiate the relevant Ethernet MAC component on that tile.

Port timing on xCORE devices typically becomes important above 20 MHz. Since RMII operates at 50 MHz, it is likely that port timings will need to be adjusted to center the data valid windows for both capture (RX) and presentation (TX) for maximum reliability. These timings are provided by a structure `port_timing` which has various members to control the on-chip delays.

The detail for how to set the values is outside the scope of this document, however the user is encouraged to consult the IO timings for xcore.ai document for further understanding. Aspects of the hardware design including PCB layout, clock skew, duty cycle and pin drive strength will affect these adjustments. The RMII MAC example in this repo shows an example port timing struct for a specific board.

In summary, the fields (and their uses) in the `port_timing` structure are as follows:

- clk_delay_tx_rising - The number of core clock cycles to delay the capture clock. Since no signal capture occurs in the TX section this value is not critical, however it should be set to the same as clk_delay_tx_falling.

- clk_delay_tx_falling - The number of core clock cycles to delay the drive clock falling edge. Increasing this value delays the presentation of the TX data and TXEN signal relative to the external ethernet clock.
- clk_delay_rx_rising - The number of core clock cycles to delay the capture clock. Increasing this value delays the point at which the RX data and RXDV are sampled relative to the external ethernet clock.
- clk_delay_rx_falling - The number of core clock cycles to delay the drive clock. Since no signal drive occurs in the RX section this value is not critical, however it should be set to the same as clk_delay_rx_rising.
- pad_delay_rx - The number of core clock cycles to delay the sampling of RX data and strobe. Because this setting delays the data and not the clock, it has the effect of adding negative clock delay, which can be useful in some cases.

## 6.3 RGMII: Reduced Gigabit Media Independent Interface

RGMII requires half the number of data pins used in GMII by clocking data on both the rising and the falling edges of the clock, and by eliminating non-essential signals (carrier sense and collision indication).

xCORE-200 XE/XEF devices have a set of pins that are dedicated to communication with a Gigabit Ethernet PHY or switch via RGMII, designed to comply with the timings in the RGMII v1.3 specification.

RGMII supports Ethernet speeds of 10 Mb/s, 100 Mb/s and 1000 Mb/s.

The Ethernet MAC implements ID mode as specified by RGMII. TX clock from xCORE to PHY is delayed. Default 10/100 and 1000 Mb/s delays are set in rgmii_consts.h to an integer number of system clock ticks (e.g. 1 x 2ns if system clock is 500MHz):

```
#define RGMII_DELAY 1
#define RGMII_DIVIDE_1G 3
#define RGMII_DELAY_100M 3
```

Note that some Ethernet PHY operate in “hybrid mode” and apply skew compensation on incoming TX clock. You may need to adjust this compensation, disable it, or set the above delay to 0 in the Ethernet MAC.

The Ethernet MAC will expect RX clock from PHY to xCORE be delayed by 1.2-2ns as specified by RGMII.

The pins and functions are listed in Table 2. When the 10/100/1000 Mb/s Ethernet MAC is instantiated these pins can no longer be used as GPIO pins, and will instead be driven directly from a Double Data Rate RGMII block, which in turn is interfaced to a set of ports on Tile 1. Table 5 describes the RGMII pins and signals:

Table 5: RGMII pins and signals

| Mandatory Pin | Signal Name | Description |
|---|---|---|
| X1D40 | TX3 | Transmit data bit 3 |
| X1D41 | TX2 | Transmit data bit 2 |
| X1D42 | TX1 | Transmit data bit 1 |
| X1D43 | TX0 | Transmit data bit 0 |
| X1D26 | TX_CLK | Transmit clock (2.5/25/125 MHz) |
| X1D27 | TX_CTL | Transmit data valid/error |
| X1D28 | RX_CLK | Receive clock (2.5/25/125 MHz) |
| X1D29 | RX_CTL | Receive data valid/error |
| X1D30 | RX3 | Receive data bit 3 |
| X1D31 | RX2 | Receive data bit 2 |
| X1D32 | RX1 | Receive data bit 1 |
| X1D33 | RX0 | Receive data bit 0 |

The RGMII block is connected to the ports on Tile 1 as shown in *RGMII port structure*. When the 10/100/1000 Mb/s Ethernet MAC is instantiated, the ports and IO pins shown can only be used by the MAC component. Other IO pins and ports are unaffected.

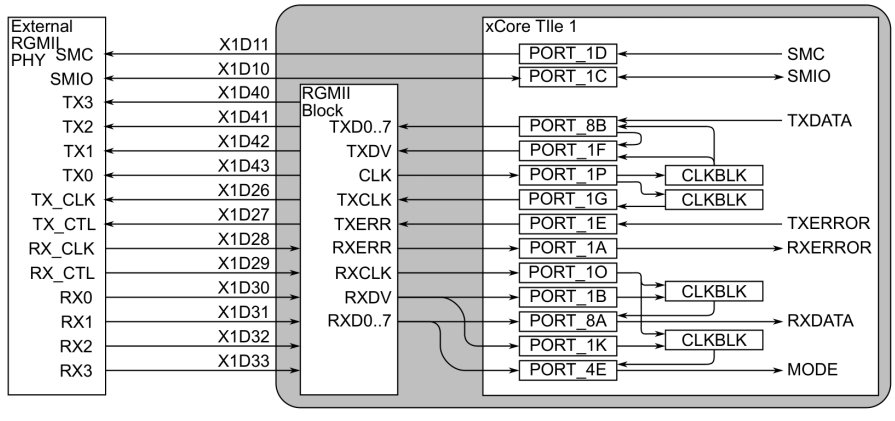


Fig. 1: RGMII port structure

# 7 Usage

## 7.1 10/100 Mb/s Ethernet MAC operation

There are two types of 10/100 Mb/s Ethernet MAC that are optimized for different feature sets. Both connect to a standard 10/100 Mb/s Ethernet PHY using the same MII interface described in *MII: Media Independent Interface*, or optionally an RMII interface for the real-time MAC running on xcore.ai.

The resource-optimized MAC described here is provided for applications that do not require real-time features, such as those required by the Audio Video Bridging standards. A simple webserver or low-bandwidth TCP traffic is a typical use for this MAC.

The same API is shared across all configurations of the Ethernet MACs. Additional API calls are available in the configuration interface of the real-time MACs that will cause a run-time assertion if called by the non-real-time configuration.

Ethernet MAC components are instantiated as parallel tasks that run in a `par` statement. The application can connect via a transmit, receive and configuration interface connection using the *ethernet_tx_if*, *ethernet_rx_if* and *The Ethernet MAC configuration interface* interface types, as shown in Fig. 2

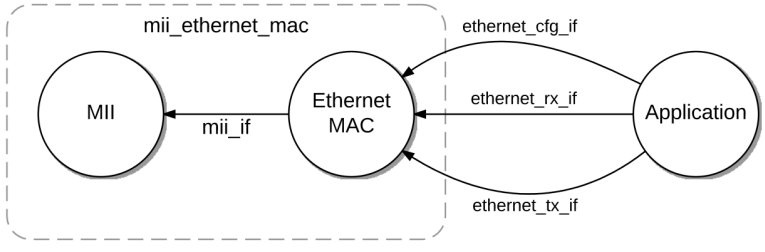


Fig. 2: 10/100 Mb/s Ethernet MAC task diagram

For example, the following code instantiates a standard Ethernet MAC component using MII and connects to it:

```
port p_eth_rxclk  = XS1_PORT_1J;
port p_eth_rxd    = XS1_PORT_4E;
port p_eth_txd    = XS1_PORT_4F;
port p_eth_rxdv   = XS1_PORT_1K;
port p_eth_txen   = XS1_PORT_1L;
port p_eth_txclk  = XS1_PORT_1I;
port p_eth_rxerr  = XS1_PORT_1P;
port p_eth_timing = XS1_PORT_8C;
clock eth_rxclk   = XS1_CLKBLK_1;
clock eth_txclk   = XS1_CLKBLK_2;

int main()
{
  ethernet_cfg_if i_cfg[1];
  ethernet_rx_if i_rx[1];
  ethernet_tx_if i_tx[1];
  par {
    mii_ethernet_mac(i_cfg, 1, i_rx, 1, i_tx, 1,
                     p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
                     p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
                     eth_rxclk, eth_txclk, 1600);
    application(i_cfg[0], i_rx[0], i_tx[0]);
  }
  return 0;
}
```

Note that the connections are arrays of interfaces, so several tasks can connect to the same component instance.

The application can use the client end of the interface connections to perform Ethernet MAC operations e.g.:

```
void application(client ethernet_cfg_if i_cfg,
                 client ethernet_rx_if i_rx,
                 client ethernet_tx_if i_tx)
{
  ethernet_macaddr_filter_t macaddr_filter;
  size_t index = i_rx.get_index();
  for (int i = 0; i < MACADDR_NUM_BYTES; i++)
    macaddr_filter.addr[i] = i;
  i_cfg.add_macaddr_filter(index, 0, macaddr_filter);

  while (1) {
    select {
    case i_rx.packet_ready():
      uint8_t rxbuf[ETHERNET_MAX_PACKET_SIZE];
      ethernet_packet_info_t packet_info;
      i_rx.get_packet(packet_info, rxbuf, ETHERNET_MAX_PACKET_SIZE);
      i_tx.send_packet(rxbuf, packet_info.len, ETHERNET_ALL_INTERFACES);
      break;
    }
  }
}
```

## 7.2 10/100 Mb/s real-time Ethernet MAC

The real-time 10/100 Mb/s Ethernet MAC supports additional features required to implement, for example, an AVB Talker and/or Listener endpoint, but has additional xCORE resource requirements compared to the non-real-time MAC.

The real-time MAC may support the RMII interface described in *RMII: Reduced Media Independent Interface* when targeting xcore.ai devices.

It is instantiated similarly to the non-real-time Ethernet MAC, with additional streaming channels for sending and receiving high-priority Ethernet traffic, as shown in Fig. 3:

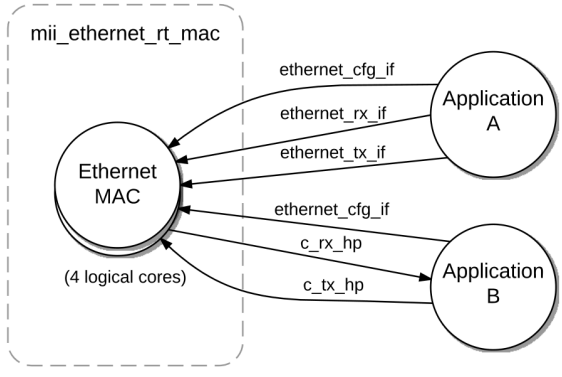


Fig. 3: 10/100 Mb/s real-time Ethernet MAC task diagram

For example, the following code instantiates a real-time Ethernet MAC component with connected via MII high and low-priority interfaces and connects to it:

```
port p_eth_rxclk  = XS1_PORT_1J;
port p_eth_rxd    = XS1_PORT_4E;
port p_eth_txd    = XS1_PORT_4F;
port p_eth_rxdv   = XS1_PORT_1K;
port p_eth_txen   = XS1_PORT_1L;
port p_eth_txclk  = XS1_PORT_1I;
port p_eth_rxerr  = XS1_PORT_1P;
clock eth_rxclk   = XS1_CLKBLK_1;
clock eth_txclk   = XS1_CLKBLK_2;

int main()
{
  ethernet_cfg_if i_cfg[1];
  ethernet_rx_if i_rx_lp[1];
  ethernet_tx_if i_tx_lp[1];
  streaming chan c_rx_hp;
  streaming chan c_tx_hp;
  par {
    mii_ethernet_rt_mac(i_cfg, 1, i_rx_lp, 1, i_tx_lp, 1,
                        c_rx_hp, c_tx_hp, p_eth_rxclk, p_eth_rxerr,
                        p_eth_rxd, p_eth_rxdv, p_eth_txclk,
                        p_eth_txen, p_eth_txd, eth_rxclk, eth_txclk,
                        4000, 4000, ETHERNET_ENABLE_SHAPER);
   application(i_cfg[0], i_rx_lp[0], i_tx_lp[0], c_rx_hp, c_tx_hp);
  }
}
```

Similarly the RMII real-time MAC may be instantiated (four bit port version shown):

```
port p_eth_clk = XS1_PORT_1J;
port p_eth_txd = XS1_PORT_4B;
port p_eth_rxd = XS1_PORT_4A;
port p_eth_rxdv = XS1_PORT_1K;
port p_eth_txen = XS1_PORT_1L;
clock eth_rxclk = XS1_CLKBLK_1;
clock eth_txclk = XS1_CLKBLK_2;

int main()
{
```

(continues on next page)

(continued from previous page)

```
  ethernet_cfg_if i_cfg[1];
  ethernet_rx_if i_rx_lp[1];
  ethernet_tx_if i_tx_lp[1];
  streaming chan c_rx_hp;
  streaming chan c_tx_hp;
  par {
    rmii_ethernet_rt_mac(i_cfg, 1, i_rx_lp, 1, i_tx_lp, 1,
                         c_rx_hp, c_tx_hp,
                         p_eth_clk,
                         p_eth_rxd, NULL, USE_UPPER_2B,
                         p_eth_rxdv,
                         p_eth_txen,
                         p_eth_txd, NULL, USE_UPPER_2B,
                         eth_rxclk, eth_txclk,
                         port_timing,
                         4000, 4000, ETHERNET_ENABLE_SHAPER);
   application(i_cfg[0], i_rx_lp[0], i_tx_lp[0], c_rx_hp, c_tx_hp);
  }
}
```

The application can use the other end of the streaming channels to send and receive high-priority traffic e.g.:

```
void application(client ethernet_cfg_if i_cfg,
                 client ethernet_rx_if i_rx,
                 client ethernet_tx_if i_tx,
                 streaming chanend c_rx_hp,
                 streaming chanend c_tx_hp)
{
  ethernet_macaddr_filter_t macaddr_filter;
  size_t index = i_rx.get_index();
  for (int i = 0; i < MACADDR_NUM_BYTES; i++)
    macaddr_filter.addr[i] = i;
  i_cfg.add_macaddr_filter(index, 1, macaddr_filter);

  while (1) {
    uint8_t rxbuf[ETHERNET_MAX_PACKET_SIZE];
    ethernet_packet_info_t packet_info;
    select {
    case ethernet_receive_hp_packet(c_rx_hp, rxbuf, packet_info):
      ethernet_send_hp_packet(c_tx_hp, rxbuf, packet_info.len,
                              ETHERNET_ALL_INTERFACES);
      break;
    }
  }
}
```

## 7.3 10/100/1000 Mb/s real-time Ethernet MAC

The 10/100/1000 Mb/s Ethernet MAC supports the same feature set and API as the 10/100 Mb/s real-time MAC but with higher throughput and lower end-to-end latency. The component connects to a Gigabit Ethernet PHY via an RGMII interface as described in *RGMII: Reduced Gigabit Media Independent Interface*.

It is instantiated similarly to the real-time Ethernet MAC, with an additional combinable task that allows the configuration interface to be shared with another slow interface such as SMI/MDIO. It must be instantiated on Tile 1 and the user application run on Tile 0, as shown in Fig. 4:

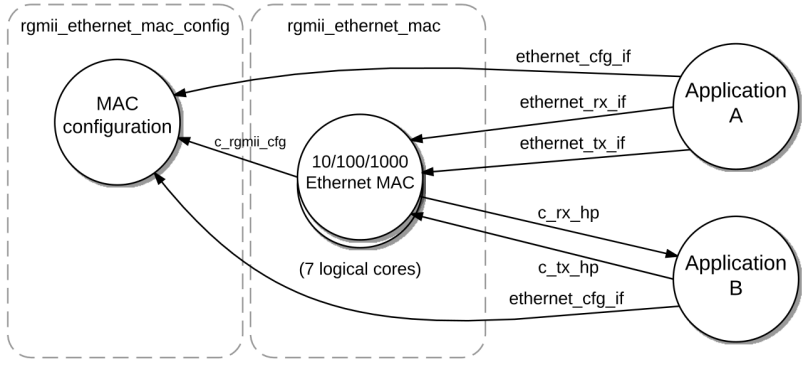


Fig. 4: 10/100/1000 Mb/s Ethernet MAC task diagram

For example, the following code instantiates a 10/100/1000 Mb/s Ethernet MAC component with high and low-priority interfaces and connects to it:

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

int main()
{
  ethernet_cfg_if i_cfg[1];
  ethernet_rx_if i_rx_lp[1];
  ethernet_tx_if i_tx_lp[1];
  streaming chan c_rx_hp;
  streaming chan c_tx_hp;
  streaming chan c_rgmii_cfg;
  par {
    on tile[1]: rgmii_ethernet_mac(i_rx, 1, i_tx, 1,
                                   c_rx_hp, c_tx_hp,
                                   c_rgmii_cfg, rgmii_ports,
                                   ETHERNET_ENABLE_SHAPER);
    on tile[1]: rgmii_ethernet_mac_config(i_cfg, 1, c_rgmii_cfg);
    on tile[0]: application(i_cfg[0], i_rx_lp[0], i_tx_lp[0], c_rx_hp, c_tx_hp);
  }
}
```

## 7.4 Raw MII interface

The raw MII interface implements a MII layer component with a basic buffering scheme that is shared with the application. It provides a direct access to the MII pins as described in *MII: Media Independent Interface*. It does not implement the buffering and filtering required by a compliant Ethernet MAC layer, and defers this to the application.

The buffering of this task is shared with the application it is connected to. It sets up an interrupt handler on the logical core the application is running on (via the **`init`** function on the *mii_if* interface connection) and also consumes some of the MIPs on that core in addition to the core *Raw MII interface* is running on (Fig. 5).

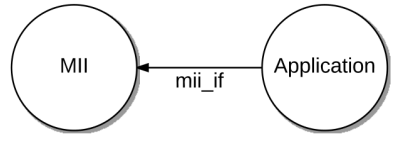


Fig. 5: MII task diagram

For example, the following code instantiates a MII component and connects to it:

```
port p_eth_rxclk  = XS1_PORT_1J;
port p_eth_rxd    = XS1_PORT_4E;
port p_eth_txd    = XS1_PORT_4F;
port p_eth_rxdv   = XS1_PORT_1K;
port p_eth_txen   = XS1_PORT_1L;
port p_eth_txclk  = XS1_PORT_1I;
port p_eth_rxerr  = XS1_PORT_1P;
port p_eth_timing = XS1_PORT_8C;
clock eth_rxclk   = XS1_CLKBLK_1;
clock eth_txclk   = XS1_CLKBLK_2;

int main()
{
  mii_if i_mii;
  par {
    mii(i_mii, p_eth_rxclk, p_eth_rxerr, p_eth_rxd, p_eth_rxdv,
        p_eth_txclk, p_eth_txen, p_eth_txd, p_eth_timing,
        eth_rxclk, eth_txclk, 4096);
    application(i_mii);
  }
  return 0;
}
```

More information on interfaces and tasks can be be found in the XMOS Programming Guide.

## 7.5 SMI/MDIO interface

The SMI (Serial Management Interface) is used in Ethernet systems for the management and configuration of PHY (physical layer) devices. It is part of the MDIO (Management Data Input/Output) system defined by the IEEE 802.3 standard and provides a mechanism for communication between a MAC (Media Access Control) layer and PHY devices in an Ethernet system.

The MDIO interface consists of clock (MDC) and data (MDIO) signals. MDC is always driven by the host whereas MDIO can be turned around so that it can read data from the PHY.

The SMI task is marked as `[[distributable]]` which means, if it is called from the same tile, does not occupy a hardware thread. Instead the call to the `read_reg()` or `write_reg()` methods are treated as function calls which return when the last bit of the SMI transaction is complete.

The interface uses two pins to communicate and there are two variants of the API provided, depending on whether you wish to use two one bit ports or two bits of a wider port. If you use two bits of a wider port, the remaining pins are not available for general use and should either be left disconnected or weakly pulled down.

---

**Note:** The standard SMI/MDIO specification requires use of a pull-up resistor on MDIO (typically 4.7 kOhm for a single PHY in a 3.3 V system). If using the single-port version then it is necessary to also connect a pull-up to the MDC line (typically 4.7 kOhm for 3.3 V systems). The reason for this is that xcore ports have only a single direction bit. So in order to sample the MDIO line with a known MDC state, an external resistor is required.

---

The speed of the interface is set conservatively at 1.66 MHz which supports slower PHY SMI interfaces (eg. LAN8710A) that have a relatively slow time to data valid. This speed is also chosen to support the single port version which has to sample read data at the falling edge, effectively reducing the maximum bit clock by a factor of two. If faster access is required and supported by the PHY, or the two port version is used, then it is possible to adjust the following define in `smi.xc` up to a maximum of around 4 MHz for xCORE-200 and 5 MHz for xcore.ai:

```
#define SMI_BIT_CLOCK_HZ 1660000
```

Increasing the bit clock may require use of smaller pull-up resistor(s) depending on board layout to ensure that the signal rise time is sufficient. If in doubt, either test operation using lower the bit rate by setting a smaller `SMI_BIT_CLOCK_HZ` or check with an oscilloscope to ensure that the MDC and MDIO lines are fully reaching the logic high state.

# 8 API

All Ethernet functions can be accessed via the **ethernet.h** header:

```
#include <ethernet.h>
```

You will also have to add **lib_ethernet** to the **USED_MODULES** field of your application Makefile.

## 8.1 Creating a 10/100 Mb/s Ethernet MAC instance

void **mii_ethernet_mac**(SERVER_INTERFACE(ethernet_cfg_if, i_cfg[*n_cfg*]), static_const_unsigned_t n_cfg, SERVER_INTERFACE(ethernet_rx_if, i_rx[*n_rx*]), static_const_unsigned_t n_rx, SERVER_INTERFACE(ethernet_tx_if, i_tx[*n_tx*]), static_const_unsigned_t n_tx, in_port_t p_rxclk, in_port_t p_rxer, in_port_t p_rxd, in_port_t p_rxdv, in_port_t p_txclk, out_port_t p_txen, out_port_t p_txd, port p_timing, clock rxclk, clock txclk, static_const_unsigned_t rx_bufsize_words)

10/100 Mb/s Ethernet MAC component that connects to an MII interface.

This function implements a 10/100 Mb/s Ethernet MAC component connected to an MII interface. Interaction to the component is via the connected configuration and data interfaces.

### Parameters

- **i_cfg** – Array of client configuration interfaces
- **n_cfg** – The number of configuration clients connected
- **i_rx** – Array of receive clients
- **n_rx** – The number of receive clients connected
- **i_tx** – Array of transmit clients
- **n_tx** – The number of transmit clients connected
- **p_rxclk** – MII RX clock port
- **p_rxer** – MII RX error port
- **p_rxd** – MII RX data port
- **p_rxdv** – MII RX data valid port
- **p_txclk** – MII TX clock port
- **p_txen** – MII TX enable port
- **p_txd** – MII TX data port
- **p_timing** – Internal timing port - this can be any xCORE port that is not connected to any external device.
- **rxclk** – Clock used for MII receive timing
- **txclk** – Clock used for MII transmit timing
- **rx_bufsize_words** – The number of words to used for a receive buffer. This should be at least 1500 words.

## 8.2 Creating a 10/100 Mb/s real-time Ethernet MAC instance

void **mii_ethernet_rt_mac**(SERVER_INTERFACE(ethernet_cfg_if, i_cfg[*n_cfg*]), static_const_unsigned_t n_cfg, SERVER_INTERFACE(ethernet_rx_if, i_rx_lp[*n_rx_lp*]), static_const_unsigned_t n_rx_lp, SERVER_INTERFACE(ethernet_tx_if, i_tx_lp[*n_tx_lp*]), static_const_unsigned_t n_tx_lp, nullable_streaming_chanend_t c_rx_hp, nullable_streaming_chanend_t c_tx_hp, in_port_t p_rxclk, in_port_t p_rxer, in_port_t p_rxd, in_port_t p_rxdv, in_port_t p_txclk, out_port_t p_txen, out_port_t p_txd, clock rxclk, clock txclk, static_const_unsigned_t rx_bufsize_words, static_const_unsigned_t tx_bufsize_words, enum *ethernet_enable_shaper_t* shaper_enabled)

10/100 Mb/s real-time Ethernet MAC component to connect to an MII interface.

This function implements a 10/100 Mb/s Ethernet MAC component, connected to an MII interface, with real-time features (priority queuing and traffic shaping). Interaction to the component is via the connected configuration and data interfaces.

### Parameters

- **i_cfg** – Array of client configuration interfaces
- **n_cfg** – The number of configuration clients connected
- **i_rx_lp** – Array of low priority receive clients
- **n_rx_lp** – The number of low priority receive clients connected
- **i_tx_lp** – Array of low priority transmit clients
- **n_tx_lp** – The number of low priority transmit clients connected
- **c_rx_hp** – Streaming channel end for high priority receive data
- **c_tx_hp** – Streaming channel end for high priority transmit data
- **p_rxclk** – MII RX clock port
- **p_rxer** – MII RX error port
- **p_rxd** – MII RX data port
- **p_rxdv** – MII RX data valid port
- **p_txclk** – MII TX clock port
- **p_txen** – MII TX enable port
- **p_txd** – MII TX data port
- **rxclk** – Clock used for MII receive timing
- **txclk** – Clock used for MII transmit timing
- **rx_bufsize_words** – The number of words to used for a receive buffer. This should be at least 500 words.
- **tx_bufsize_words** – The number of words to used for a transmit buffer. This should be at least 500 words.
- **shaper_enabled** – This should be set to `ETHERNET_ENABLE_SHAPER` or `ETHERNET_DISABLE_SHAPER` to either enable or disable the 802.1Qav traffic shaper within the MAC.

void **rmii_ethernet_rt_mac**(SERVER_INTERFACE(ethernet_cfg_if, i_cfg[*n_cfg*]), static_const_unsigned_t n_cfg, SERVER_INTERFACE(ethernet_rx_if, i_rx_lp[*n_rx_lp*]), static_const_unsigned_t n_rx_lp, SERVER_INTERFACE(ethernet_tx_if, i_tx_lp[*n_tx_lp*]), static_const_unsigned_t n_tx_lp, nullable_streaming_chanend_t c_rx_hp, nullable_streaming_chanend_t c_tx_hp, in_port_t p_clk, port p_rxd_0, NULLABLE_RESOURCE(port, p_rxd_1), *rmii_data_4b_pin_assignment_t* rx_pin_map, in_port_t p_rxdv, out_port_t p_txen, port p_txd_0, NULLABLE_RESOURCE(port, p_txd_1), *rmii_data_4b_pin_assignment_t* tx_pin_map, clock rxclk, clock txclk, *rmii_port_timing_t* port_timing, static_const_unsigned_t rx_bufsize_words, static_const_unsigned_t tx_bufsize_words, enum *ethernet_enable_shaper_t* shaper_enabled)

10/100 Mb/s real-time Ethernet MAC component to connect to an RMII interface.

This function implements a 10/100 Mb/s Ethernet MAC component, connected to an RMII interface, with real-time features (priority queuing and traffic shaping). Interaction to the component is via the connected configuration and data interfaces. Each of the 2 bit data ports may be defined either as half of a 4 bit port (upper or lower 2 bits) or a pair of 1 bit ports.

**Parameters**

- **i_cfg** – Array of client configuration interfaces
- **n_cfg** – The number of configuration clients connected
- **i_rx_lp** – Array of low priority receive clients
- **n_rx_lp** – The number of low priority receive clients connected
- **i_tx_lp** – Array of low priority transmit clients
- **n_tx_lp** – The number of low priority transmit clients connected
- **c_rx_hp** – Streaming channel end for high priority receive data
- **c_tx_hp** – Streaming channel end for high priority transmit data
- **p_clk** – RMII clock input port
- **p_rxd_0** – Port for data bit 0 (1 bit option) or entire port (4 bit option)
- **p_rxd_1** – Port for data bit 1 (1 bit option). Pass null if unused.
- **rx_pin_map** – Which pins to use in 4 bit case. USE_LOWER_2B or USE_HIGHER_2B. Ignored if 1 bit ports used.
- **p_rxdv** – RMII RX data valid port
- **p_txen** – RMII TX enable port
- **p_txd_0** – Port for data bit 0 (1 bit option) or entire port (4 bit option)
- **p_txd_1** – Port for data bit 1 (1 bit option). Pass null if unused.
- **tx_pin_map** – Which pins to use in 4 bit case. USE_LOWER_2B or USE_HIGHER_2B. Ignored if 1 bit ports used.
- **rxclk** – Clock used for RMII receive timing
- **txclk** – Clock used for RMII transmit timing
- **port_timing** – Struct used for initialising the clock blocks to ensure setup and hold times are met
- **rx_bufsize_words** – The number of words to used for a receive buffer. This should be at least 500 long words.
- **tx_bufsize_words** – The number of words to used for a transmit buffer. This should be at least 500 long words.
- **shaper_enabled** – This should be set to ETHERNET_ENABLE_SHAPER or

`ETHERNET_DISABLE_SHAPER` to either enable or disable the 802.1Qav traffic shaper within the MAC.

### Real-time Ethernet MAC supporting typedefs

enum **`ethernet_enable_shaper_t`**

Enum representing a flag to enable or disable the 802.1Qav credit based traffic shaper on the egress MAC port.

*Values:*

enumerator **`ETHERNET_DISABLE_SHAPER`**

Disable the credit based shaper

enumerator **`ETHERNET_ENABLE_SHAPER`**

Enable the credit based shaper

struct **`rmii_port_timing_t`**

Struct containing the clock delay settings for the Rx and Tx pins. This is needed to adjust port timings to ensure that the data is captured with sufficient setup and hold margin. This is required due to the relatively fast 50 MHz clock. Please consult the documentation for further details and suggested settings.

enum **`rmii_data_4b_pin_assignment_t`**

ENUM to determine which two bits of a four bit port are to be used as data lines in the case that a four bit port is specified for RMII. The other two pins of the four bit port cannot be used. For Rx the unused input bits are ignored. For Tx, the unused pins are always driven low.

*Values:*

enumerator **`USE_LOWER_2B`**

Use bit 0 and bit 1 of the four bit port for data bits 0 and 1

enumerator **`USE_UPPER_2B`**

Use bit 2 and bit 3 of the four bit port for data bits 0 and 1

## 8.3 Creating a 10/100/1000 Mb/s Ethernet MAC instance

struct **rgmii_ports_t**

Structure representing the port and clock resources required by RGMII

A macro to initialize this structure is provided:

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

void **rgmii_ethernet_mac**(SERVER_INTERFACE(ethernet_rx_if, i_rx_lp[*n_rx_lp*]), static_const_unsigned_t n_rx_lp, SERVER_INTERFACE(ethernet_tx_if, i_tx_lp[*n_tx_lp*]), static_const_unsigned_t n_tx_lp, nullable_streaming_chanend_t c_rx_hp, nullable_streaming_chanend_t c_tx_hp, streaming_chanend_t c_rgmii_cfg, REFERENCE_PARAM(*rgmii_ports_t*, rgmii_ports), enum *ethernet_enable_shaper_t* shaper_enabled)

10/100/1000 Mb/s Ethernet MAC component to connect to an RGMII interface.

This function implements a 10/100/1000 Mb/s Ethernet MAC component, connected to an RGMII interface, with real-time features. Interaction to the component is via the connected configuration and data interfaces.

**Parameters**

- **i_rx_lp** – Array of low priority receive clients
- **n_rx_lp** – The number of low priority receive clients connected
- **i_tx_lp** – Array of low priority transmit clients
- **n_tx_lp** – The number of low priority transmit clients connected
- **c_rx_hp** – Streaming channel end for high priority receive data
- **c_tx_hp** – Streaming channel end for high priority transmit data
- **c_rgmii_cfg** – A streaming channel end connected to rgmii_ethernet_mac_config()
- **rgmii_ports** – A *rgmii_ports_t* structure initialized with the RGMII_PORTS_INITIALIZER macro
- **shaper_enabled** – This should be set to `ETHERNET_ENABLE_SHAPER` or `ETHERNET_DISABLE_SHAPER` to either enable or disable the 802.1Qav traffic shaper within the MAC.

void **rgmii_ethernet_mac_config**(SERVER_INTERFACE(ethernet_cfg_if, i_cfg[*n*]), unsigned n, streaming_chanend_t c_rgmii_cfg)

RGMII Ethernet MAC configuration task

This function implements the server side of the **ethernet_cfg_if** interface and communicates internally with the RGMII Ethernet MAC via a streaming channel end.

The function can be combined with SMI from within the top level par.

**Parameters**

- **i_cfg** – Array of client configuration interfaces
- **n** – The number of configuration clients connected
- **c_rgmii_cfg** – A streaming channel end connected to rgmii_ethernet_mac()

## 8.4 The Ethernet MAC configuration interface

*group* **ethernet_config_if**

Ethernet MAC configuration interface.

This interface allows clients to configure the Ethernet MAC.

### Functions

void **set_macaddr**(size_t ifnum, const uint8_t mac_address[MACADDR_NUM_BYTES])

Set the source MAC address of the Ethernet MAC

**Parameters**

- **ifnum** – The index of the MAC interface to set
- **mac_address** – The six-octet MAC address to set Warning: The mac address set through this function is not used anywhere in the MAC other than for later retrieval by *get_macaddr()*. The client is expected to create the full packet including src and dest mac address for transmission. For setting a mac address filter when receiving packets, use add_macaddr_filter

void **get_macaddr**(size_t ifnum, uint8_t mac_address[MACADDR_NUM_BYTES])

Gets the source MAC address of the Ethernet MAC

**Parameters**

- **ifnum** – The index of the MAC interface to get
- **mac_address** – The six-octet MAC address of this interface

void **set_link_state**(int ifnum, *ethernet_link_state_t* new_state, *ethernet_speed_t* speed)

Set the current link state.

This function sets the current link state and speed of the PHY to the MAC.

**Parameters**

- **ifnum** – The index of the MAC interface to set
- **new_state** – The new link state for the port.
- **speed** – The active link speed and duplex of the PHY.

void **get_link_state**(int ifnum, REFERENCE_PARAM(unsigned, link_state), REFERENCE_PARAM(unsigned, link_speed))

Get the current link state.

This function Gets the current link state and speed of the PHY to the MAC.

**Parameters**

- **ifnum** – The index of the MAC interface to ge the link state for

**Returns**

Ethernet link state and speed

*ethernet_macaddr_filter_result_t* **add_macaddr_filter**(size_t client_num, int is_hp, *ethernet_macaddr_filter_t* entry)

Add MAC addresses to the filter. Only packets with the specified MAC address will be forwarded to the client.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.
- **is_hp** – Indicates whether the RX client is high priority. There is only one high priority client, so client_num must be 0 when is_hp is set. High priority queuing is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.
- **entry** – The filter entry to add.

**Returns**

ETHERNET_MACADDR_FILTER_SUCCESS when the entry is added or ETHERNET_MACADDR_FILTER_TABLE_FULL on failure.

void **del_macaddr_filter**(size_t client_num, int is_hp, *ethernet_macaddr_filter_t* entry)

Delete MAC addresses from the filter.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.
- **is_hp** – Indicates whether the RX client is high priority. There is only one high priority client, so client_num must be 0 when is_hp is set. High priority queueing is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.
- **entry** – The filter entry to delete.

void **del_all_macaddr_filters**(size_t client_num, int is_hp)

Delete all MAC addresses from the filter registered for this client.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.
- **is_hp** – Indicates whether the RX client is high priority. There is only one high priority client, so client_num must be 0 when is_hp is set. High priority queuing is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

void **add_ethertype_filter**(size_t client_num, uint16_t ethertype)

Add an Ethertype to the filter. This filter is applied after the MAC address filter and only if it is successful. Only packets with the specified Ethertypes will be forwarded to the client. A maximum of 2 Ethertype filters can be applied per client.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.
- **ethertype** – A two-octet Ethertype value to filter.

void **del_ethertype_filter**(size_t client_num, uint16_t ethertype)

Delete an Ethertype from the filter

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.
- **ethertype** – A two-octet Ethertype value to delete from filter.

void **get_tile_id_and_timer_value**(REFERENCE_PARAM(unsigned, tile_id), REFERENCE_PARAM(unsigned, time_on_tile))

Get the tile ID that the Ethernet MAC is running on and the current timer value on that tile. This function is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

**Parameters**

- **tile_id** – The tile ID returned from the Ethernet MAC
- **time_on_tile** – The current timer value from the Ethernet MAC

void **set_egress_qav_idle_slope**(size_t ifnum, unsigned slope)

Set the high-priority TX queue's credit based shaper idle slope value. See also *set_egress_qav_idle_slope_bps()* where the argument is bits per second. This function is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

**Parameters**

- **ifnum** – The index of the MAC interface to set the slope (always 0)
- **slope** – The slope value in bits per 100 MHz ref timer tick in MII_CREDIT_FRACTIONAL_BITS Q format.

void **set_egress_qav_idle_slope_bps**(size_t ifnum, unsigned bits_per_second)

Set the high-priority TX queue's credit based shaper idle slope in bits per second. This function is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

**Parameters**

- **ifnum** – The index of the MAC interface to set the slope (always 0)
- **slope** – The maximum number of bits per second to be set

void **set_egress_qav_credit_limit**(size_t ifnum, int payload_limit_bytes)

Sets the the high-priority TX queue's Qav credit limit in units of frame size bytes

**Parameters**

- **ifnum** – The index of the MAC interface to set the slope (always 0)
- **limit_bytes** – The credit limit in units of payload size in bytes to set as a credit limit, not including preamble, CRC and IFG. Set to 0 for no limit (default)

void **set_ingress_timestamp_latency**(size_t ifnum, *ethernet_speed_t* speed, unsigned value)

Set the ingress latency to correct for the offset between the timestamp measurement plane relative to the reference plane. See 802.1AS 8.4.3.

This latency can change at different PHY speeds, thus requires a latency value to be set for each speed in the **ethernet_speed_t** enum.

All ingress timestamps received by the client will be corrected with the set value. The latency is initialized to 0 for all speeds.

This function is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

**Parameters**

- **ifnum** – The index of the MAC interface to set the latency
- **speed** – The speed to set the latency for
- **value** – The latency value in nanoseconds

void **set_egress_timestamp_latency**(size_t ifnum, *ethernet_speed_t* speed, unsigned value)

Set the egress latency to correct for the offset between the timestamp measurement plane relative to the reference plane. See 802.1AS 8.4.3.

This latency can change at different PHY speeds, thus requires a latency value to be set for each speed in the **ethernet_speed_t** enum.

All egress timestamps received by the client will be corrected with the set value. The latency is initialized to 0 for all speeds.

This function is only available in the 10/100 Mb/s real-time and 10/100/1000 Mb/s MACs.

**Parameters**

- **ifnum** – The index of the MAC interface to set the latency
- **speed** – The speed to set the latency for
- **value** – The latency value in nanoseconds

void **enable_strip_vlan_tag**(size_t client_num)

Enable stripping of any VLAN tags on packets delivered to this client. This feature is available on the real-time 100 Mbps Ethernet MAC only.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.

void **disable_strip_vlan_tag**(size_t client_num)

Disable stripping of any VLAN tags on packets delivered to this client. This feature is available on the real-time 100 Mbps Ethernet MAC only.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.

void **enable_link_status_notification**(size_t client_num)

Enable notifications of link status changes. These will be sent over the RX interface using ETH_IF_STATUS packets.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.

void **disable_link_status_notification**(size_t client_num)

Disable notifications of link status changes.

**Parameters**

- **client_num** – The index into the set of RX clients. Can be acquired by calling the *get_index()* method.

void **exit**(void)

Exit ethernet MAC. Quits all of the associated sub tasks and frees memory. Allows the resources previously used by the MAC to be re-used by other tasks. Only supported on RMII real-time MACs. This command is ignored for other ethernet MACs.

enum **ethernet_link_state_t**

Type representing link events.

*Values:*

enumerator **ETHERNET_LINK_DOWN**

Ethernet link down event.

enumerator **ETHERNET_LINK_UP**

Ethernet link up event.

enum **ethernet_speed_t**

Type representing the PHY link speed and duplex

*Values:*

enumerator **LINK_10_MBPS_FULL_DUPLEX**

10 Mbps full duplex

enumerator **LINK_100_MBPS_FULL_DUPLEX**

100 Mbps full duplex

enumerator **LINK_1000_MBPS_FULL_DUPLEX**

1000 Mbps full duplex

enumerator **NUM_ETHERNET_SPEEDS**

Count of speeds in this enum

struct **ethernet_macaddr_filter_t**

Structure representing MAC address filter data that is registered with the Ethernet MAC

enum **ethernet_macaddr_filter_result_t**

Type representing the result of adding a filter entry to the Ethernet MAC

*Values:*

enumerator **ETHERNET_MACADDR_FILTER_SUCCESS**

The filter entry was added successfully

enumerator **ETHERNET_MACADDR_FILTER_TABLE_FULL**

The filter entry was not added because the filter table is full

## 8.5 The Ethernet MAC data handling interface

*group* **ethernet_tx_if**

Ethernet MAC data transmit interface

This interface allows clients to send packets to the Ethernet MAC for transmission

### Functions

void **_init_send_packet**(size_t n, size_t ifnum)

Internal API call. Do not use.

void **_complete_send_packet**(char packet[*n*], unsigned n, int request_timestamp, size_t ifnum)

Internal API call. Do not use.

unsigned **_get_outgoing_timestamp**()

Internal API call. Do not use.

inline void **send_packet**(CLIENT_INTERFACE(ethernet_tx_if, i), char packet[*n*], unsigned n, unsigned ifnum)

Function to send an Ethernet packet on the specified interface.

The call will block until a transmit buffer is available and the packet has been copied to the Ethernet MAC.

#### Parameters

- **packet** – A byte-array containing the Ethernet packet to send. Must include a valid Ethernet frame header.
- **n** – The number of bytes in the packet array to send
- **ifnum** – The index of the MAC interface to send the packet Use the `ETHERNET_ALL_INTERFACES` define to send to all interfaces.

inline unsigned **send_timed_packet**(CLIENT_INTERFACE(ethernet_tx_if, i), char packet[*n*], unsigned n, unsigned ifnum)

Function to send an Ethernet packet on the specified interface and return a timestamp when the packet was sent by the MAC.

The call will block until the packet has been sent and the egress timestamp retrieved.

#### Parameters

- **packet** – A byte-array containing the Ethernet packet to send. Must include a valid Ethernet frame header.
- **n** – The number of bytes in the packet array to send
- **ifnum** – The index of the MAC interface to send the packet Use the `ETHERNET_ALL_INTERFACES` define to send to all interfaces.

#### Returns

A 32-bit timestamp off a 100 MHz reference clock that represents the egress time. May be corrected for egress latency, see *set_egress_timestamp_latency()* on the **ethernet_cfg_if** interface.

*group* **ethernet_rx_if**

### Functions

size_t **get_index**()

Get the index of a given receiver client

void **packet_ready**()

Packet ready notification.

This notification will fire when a packet has been queued for this client and is ready to be received using *get_packet()*.

The event can be selected upon e.g.:

```
select {
  case i_eth_rx.packet_ready():
    ... // Get and handle the packet
  break;
}
```

void **get_packet**(REFERENCE_PARAM(*ethernet_packet_info_t*, desc), char packet[*n*], unsigned n)

Function to receive an Ethernet packet or status/control data from the MAC. Should be called after a *packet_ready()* notification.

#### Parameters

- **desc** – A descriptor containing metadata about the packet contents.
- **packet** – A byte-array containing the packet data.
- **n** – The number of bytes to receive. The **data** array must be large enough to receive the number of bytes specified.

enum **eth_packet_type_t**

Type representing the type of packet from the MAC

*Values:*

enumerator **ETH_DATA**

A packet containing data.

enumerator **ETH_IF_STATUS**

A control packet containing interface status information

enumerator **ETH_OUTGOING_TIMESTAMP_INFO**

A control packet containing an outgoing timestamp

enumerator **ETH_NO_DATA**

A packet containing no data.

struct **ethernet_packet_info_t**

Structure representing a received data or control packet from the Ethernet MAC

## 8.6 The Ethernet MAC high-priority data handling interface

inline void **ethernet_send_hp_packet**(streaming_chanend_t c_tx_hp, char packet[*n*], unsigned n, unsigned ifnum)

Function to send a priority-queued packet over a high priority channel from the 10/100 Mb/s real-time MAC.

**Parameters**

- **c_tx_hp** – A streaming channel end connected to the MAC.
- **packet** – A byte-array containing the Ethernet packet to send. Must include a valid Ethernet frame header.
- **n** – The number of bytes in the packet array to send
- **ifnum** – The index of the MAC interface to send the packet Use the `ETHERNET_ALL_INTERFACES` define to send to all interfaces.

inline void **ethernet_receive_hp_packet**(streaming_chanend_t c_rx_hp, char packet[], REFERENCE_PARAM(*ethernet_packet_info_t*, packet_info))

Function to receive a priority-queued packet over a high priority channel from the 10/100 Mb/s real-time MAC.

The packet can be split into two transactions due to internal buffering and therefore this function must be used to receive the packet.

**Parameters**

- **c_rx_hp** – A streaming channel end connected to the MAC.
- **packet** – A byte-array containing the packet data.
- **packet_info** – A descriptor containing metadata about the packet contents.

## 8.7 Creating a raw MII instance

All raw MII functions can be accessed via the **mii.h** header:

```
#include <mii.h>
```

void **mii**(SERVER_INTERFACE(mii_if, i_mii), in_port_t p_rxclk, in_port_t p_rxer, in_port_t p_rxd, in_port_t p_rxdv, in_port_t p_txclk, out_port_t p_txen, out_port_t p_txd, port p_timing, clock rxclk, clock txclk, static_const_unsigned_t rx_bufsize_words)

Raw MII component.

This function implements a MII layer component with a basic buffering scheme that is shared with the application. It provides a direct access to the MII pins. It does not implement the buffering and filtering required by a compliant Ethernet MAC layer, and defers this to the application.

The buffering of this task is shared with the application it is connected to. It sets up an interrupt handler on the logical core the application is running on via the **init** function on the **mii_if** interface connection) and also consumes some of the MIPs on that core in addition to the core **mii** is running on.

**Parameters**

- **i_mii** – The MII interface to connect to the application.
- **p_rxclk** – MII RX clock port
- **p_rxer** – MII RX error port
- **p_rxd** – MII RX data port
- **p_rxdv** – MII RX data valid port
- **p_txclk** – MII TX clock port
- **p_txen** – MII TX enable port
- **p_txd** – MII TX data port
- **p_timing** – Internal timing port - this can be any xCORE port that is not connected to any external device.
- **rxclk** – Clock used for MII receive timing
- **txclk** – Clock used for MII transmit timing
- **rx_bufsize_words** – The number of words to used for a receive buffer. This should be at least 1500 words.

## 8.8 The MII interface

*group* **mii_if**

Interface allowing access to the MII packet layer.

**Functions**

*mii_info_t* **init**()

Initialize the MII layer

This function initializes the MII layer. In doing so it will setup an interrupt handler on the current logical core that calls the function (so tasks on that core may be interrupted and can no longer rely on the deterministic runtime of the xCORE).

**Returns**

state structure to use in subsequent calls to send/receive packets.

void **release_packet**(int *data)

Get incoming packet from MII layer.

This function can be called after an event is triggered by the mii_incoming_packet() function. It gets the next incoming packet from the packet buffer of the MII layer.

This function will release a packet back to the MII layer to be used for buffering.

**Returns**

a tuple containing a pointer to the data (which is owned by the application until the *release_packet()* function is called), the number of bytes in the packet and a timestamp. If no packet is available then the first element will be a NULL pointer. Release a packet back to the MII layer.

**Parameters**

- **data** – The pointer to packet to return. This should be the same pointer returned by get_incoming_packet()

void **send_packet**(int *buf, size_t n)

Send a packet to the MII layer.

This function will send a packet over MII. It does not block and will return immediately with the MII layer now owning the memory of the packet. The function mii_packet_sent() should be subsequently called to determine when the packet has been transmitted and the application can use the buffer again.

**Parameters**

- **buf** – The pointer to the packet to be transferred to the MII layer.
- **n** – The number of bytes in the packet to send.

void **mii_incoming_packet**(*mii_info_t* info)

Event on/wait for an incoming packet.

This function waits for an incoming packet from the MII layer. It can be used in a select to detect an incoming packet e.g

```
mii_info_t mii_info = i_mii.init();
select {
  case mii_incoming_packet(mii_info):
       ...
       break;
...
```

void **mii_packet_sent**(*mii_info_t* info)

Event on/wait for a packet send to complete.

This function will wait for a packet transmitted with the `send_packet` function on the mii_interface to complete. It can be used in a select to event when the transmission is complete e.g

```
mii_info_t mii_info = i_mii.init();
select {
  case mii_packet_sent(mii_info):
       ...
       break;
...
```

typedef struct mii_lite_data_t ***mii_info_t**

## 8.9 Creating an SMI/MDIO instance

All SMI functions can be accessed via the `smi.h` header:

```
#include <smi.h>
```

void **smi**(SERVER_INTERFACE(smi_if, i_smi), port p_mdio, port p_mdc)

SMI component that connects to an Ethernet PHY or switch via MDIO on separate ports.

This function implements a SMI component that connects to an Ethernet PHY/ switch via MDIO/MDC connected on separate ports. Interaction to the component is via the connected SMI interface.

**Parameters**

- **i_smi** – Client register read/write interface
- **p_mdio** – SMI MDIO port
- **p_mdc** – SMI MDC port

void **smi_singleport**(SERVER_INTERFACE(smi_if, i_smi), port p_smi, unsigned mdio_bit, unsigned mdc_bit)

SMI component that connects to an Ethernet PHY or switch via MDIO on a shared multi-bit port.

Important!! This version requires a pull-up resistor on MDC to function.

This function implements a SMI component that connects to an Ethernet PHY/ switch via MDIO/MDC connected on the same multi-bit port. Interaction to the component is via the connected SMI interface. Unsed pins in the port are reserved and should be left unconnected or weakly pulled down.

**Parameters**

- **i_smi** – Client register read/write interface
- **p_smi** – The multi-bit port with MDIO/MDC pins
- **mdio_bit** – The MDIO bit position on the multi-bit port
- **mdc_bit** – The MDC bit position on the multi-bit port

## 8.10 The SMI/MDIO PHY interface

*group* **smi_if**

SMI register configuration interface.

This interface allows clients to read or write the PHY SMI registers

### Functions

uint16_t **read_reg**(uint8_t phy_address, uint8_t reg_address)

Read the specified SMI register in the PHY

#### Parameters

- **phy_address** – The 5-bit SMI address of the PHY
- **reg_address** – The 5-bit register address to read

#### Returns

The 16-bit data value read from the register

void **write_reg**(uint8_t phy_address, uint8_t reg_address, uint16_t val)

Write the specified SMI register in the PHY

#### Parameters

- **phy_address** – The 5-bit SMI address of the PHY
- **reg_address** – The 5-bit register address to write
- **val** – The 16-bit data value to write to the register

## 8.11 SMI PHY configuration helper functions

void **smi_configure**(CLIENT_INTERFACE(smi_if, *smi*), uint8_t phy_address, *ethernet_speed_t* speed_mbps, *smi_autoneg_t* auto_neg)

Function to configure the PHY speed/duplex with or without auto negotiation. The smi_phy_is_powered_down() function should be called to check that the PHY is not powered down before calling this function.

**Parameters**

- **smi** – An interface connection to the SMI component
- **phy_address** – The 5-bit SMI address of the PHY
- **speed_mbps** – If auto negotiation is disabled, the specified speed will be forced, otherwise the PHY will be configured to advertise as capable of all full-duplex speeds up to and including the specified speed.
- **auto_neg** – If set to `SMI_ENABLE_AUTONEG` auto negotiation is enabled, otherwise disabled if set to `SMI_DISABLE_AUTONEG`

enum **smi_autoneg_t**

Type representing PHY auto negotiation enable/disable flags

*Values:*

enumerator **SMI_DISABLE_AUTONEG**

Enable auto negotiation

enumerator **SMI_ENABLE_AUTONEG**

Disable auto negotiation

void **smi_set_loopback_mode**(CLIENT_INTERFACE(smi_if, *smi*), uint8_t phy_address, int enable)

Function to enable loopback mode with the Ethernet PHY.

**Parameters**

- **smi** – An interface connection to the SMI component
- **phy_address** – The 5-bit SMI address of the PHY
- **enable** – Loopback enable flag. If set to 1, loopback is enabled, otherwise 0 to disable

unsigned **smi_get_id**(CLIENT_INTERFACE(smi_if, *smi*), uint8_t phy_address)

Function to retrieve the PHY manufacturer ID number.

**Parameters**

- **smi** – An interface connection to the SMI component
- **phy_address** – The 5-bit SMI address of the PHY

**Returns**

The PHY manufacturer ID number

unsigned **smi_phy_is_powered_down**(CLIENT_INTERFACE(smi_if, *smi*), uint8_t phy_address)

Function to retrieve the power down status of the PHY.

**Parameters**

- **smi** – An interface connection to the SMI component
- **phy_address** – The 5-bit SMI address of the PHY

**Returns**

1 if the PHY is powered down, 0 otherwise

*ethernet_link_state_t* **smi_get_link_state**(CLIENT_INTERFACE(smi_if, *smi*), uint8_t phy_address)

Function to retrieve the link up/down status.

**Parameters**

- **smi** – An interface connection to the SMI component
- **phy_address** – The 5-bit SMI address of the PHY

**Returns**

ETHERNET_LINK_UP if the link is up, ETHERNET_LINK_DOWN if the link is down