Ethernet MAC library

The Ethernet MAC library provides a complete, software defined, Ethernet MAC that supports 10/100/1000 Mb/s data rates and is designed to IEEE Std 802.3-2002 specifications.

Features

- 10/100/1000 Mb/s full-duplex operation
- Media Independent Interface (MII) and Reduced Gigabit Media Independent Interface (RGMII) to the physical layer
- Configurable Ethertype and MAC address filters for unicast, multicast and broadcast addresses
- Frame alignment, CRC, and frame length error detection
- IEEE 802.1Q Audio Video Bridging priority queueing and credit based traffic shaper
- Support for VLAN-tagged frames
- Transmit and receive frame timestamp support for IEEE 1588 and 802.1AS
- Management Data Input/Output (MDIO) Interface for physical layer management

Components

- 10/100 Mb/s Ethernet MAC
- 10/100 Mb/s Ethernet MAC with real-time features
- 10/100/1000 Mb/s Ethernet MAC with real-time features (xCORE-200 XE/XEF)
- Raw MII interface

Software version and dependencies

This document pertains to version 3.0.2 of this library. It is known to work on version 14.0.2 of the xTIMEcomposer tools suite, it may work on other versions.

This library depends on the following other libraries:

- lib_logging (>=2.0.0)
- lib_locks (>=2.0.0)
- lib_xassert (>=2.0.0)
- lib_gpio (>=1.0.0)

Typical Resource Usage

This following table shows typical resource usage in some different configurations. Exact resource usage will depend on the particular use of the library by the application.

Configuration	Pins	Ports	Clocks	Ram	Logical cores
10/100 Mb/s	13	5 (1-bit), 2 (4-bit), 1 (any-bit)	2	~15.9K	2
10/100 Mb/s real-time	13	5 (1-bit), 2 (4-bit)	2	~22.6K	4
10/100/1000 Mb/s	12	8 (1-bit), 2 (4-bit), 2 (8-bit)	4	~101.4K	8
Raw MII	13	5 (1-bit), 2 (4-bit)	2	~10.0K	1
SMI (MDIO)	2	2 (1-bit) or 1 (multi-bit)	0	~0.7K	0

Related application notes

The following application notes use this library:

- AN00120 - How to use the Ethernet MAC library

1 External signal description

1.1 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 1 below describes the 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

Table 1: MII signals

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

1.2 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:

http://www.hp.com/rnd/pdfs/RGMIIv1_3.pdf

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

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.

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

Table 2: RGMII pins and signals

The RGMII block is connected to the ports on Tile 1 as shown in Figure 1. 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.

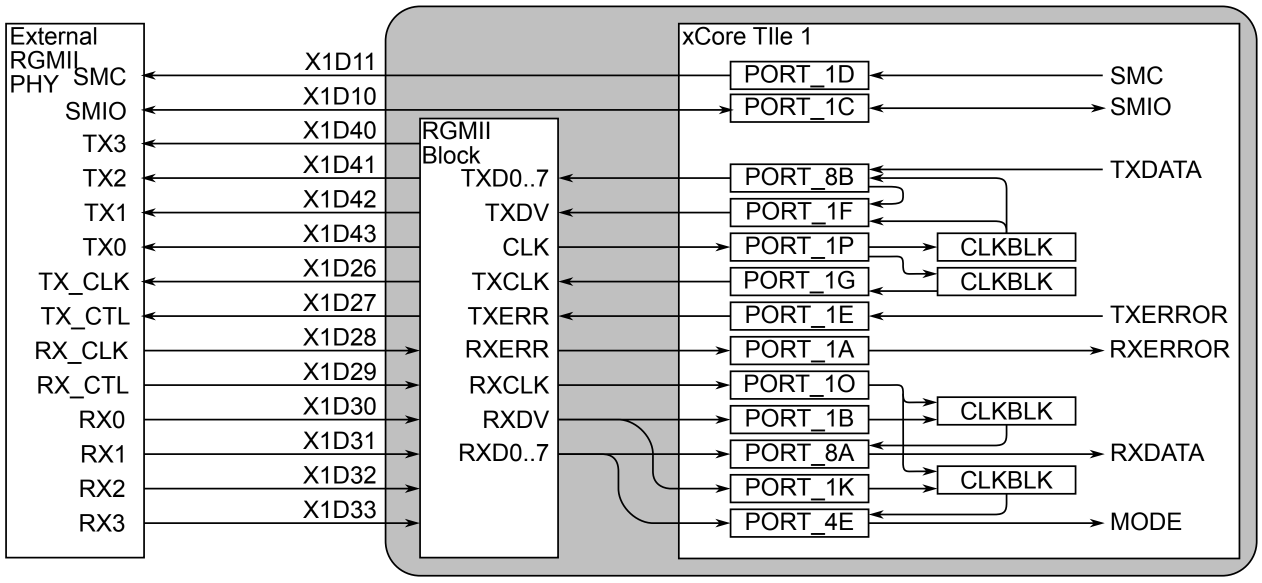

Figure 1: RGMII port structure

1.3 PHY Serial Management Interface (MDIO)

The MDIO interface consists of clock (MDC) and data (MDIO) signals. Both should be connected to two one-bit ports that are configured as open-drain IOs, using external pull-ups to either 3.3V or 2.5V (RGMII).

2 Usage

2.1 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 §1.1.

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.

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 ethernet_cfg_if interface types:

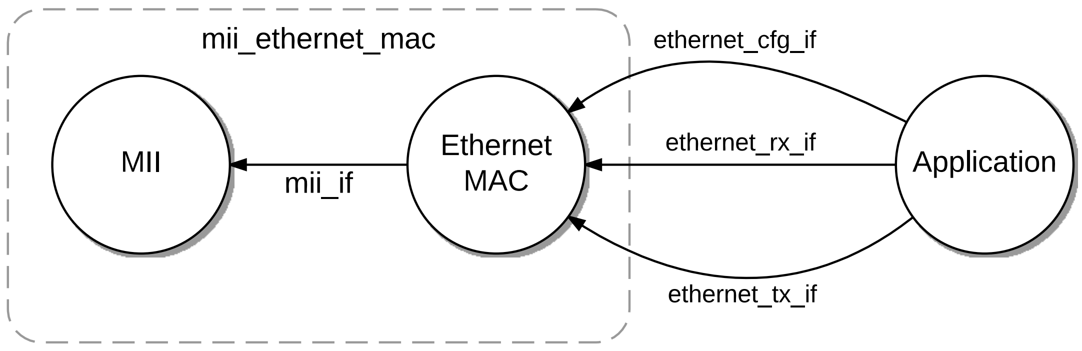

Figure 2: 10/100 Mb/s Ethernet MAC task diagram

For example, the following code instantiates a standard Ethernet MAC 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()
{
  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 < 6; 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;
    }
  }
}
```

2.2 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.

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

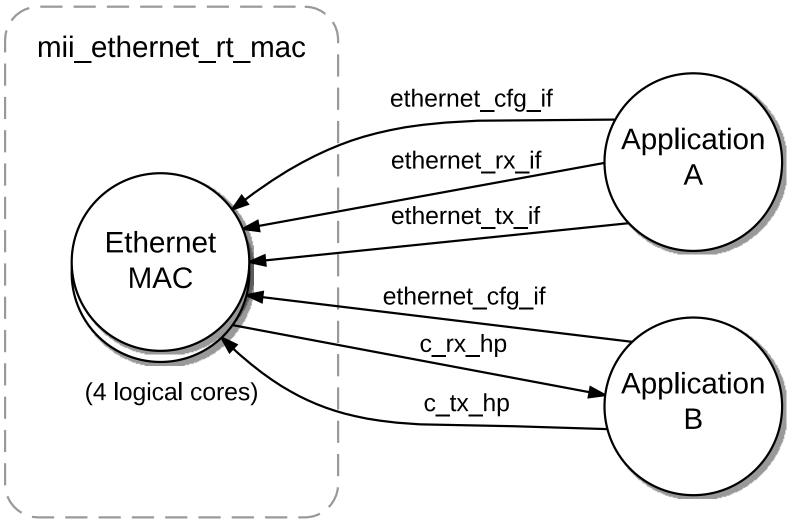

Figure 3: 10/100 Mb/s real-time Ethernet MAC task diagram

For example, the following code instantiates a real-time Ethernet MAC component with 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);
  }
}
```

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 < 6; 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;
    }
  }
}
```

2.3 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 §1.2.

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:

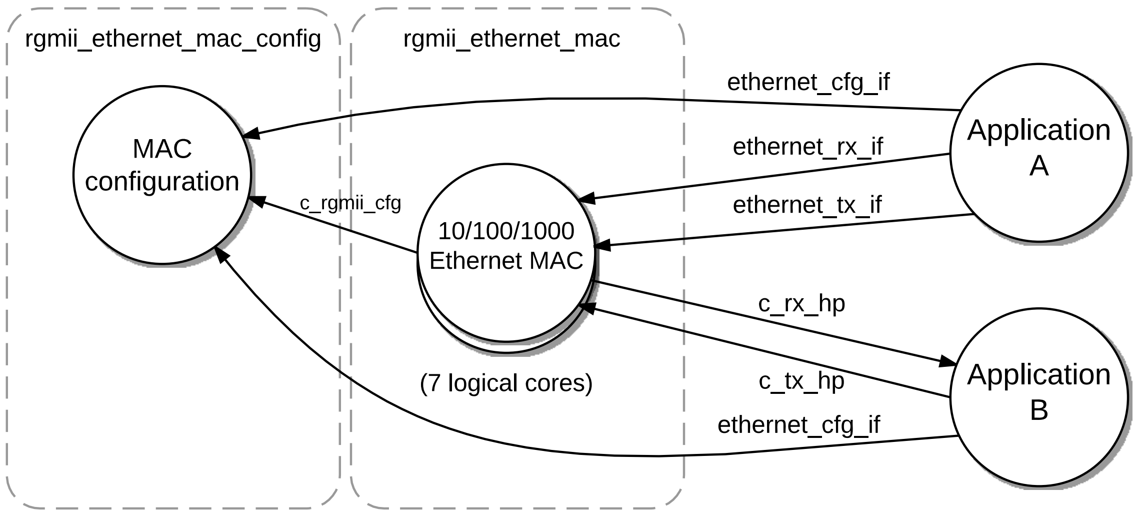

Figure 4: 10/100/1000 Mb/s Ethernet MAC task diagram

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);
  }
}
```

2.4 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 §1.1. 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.

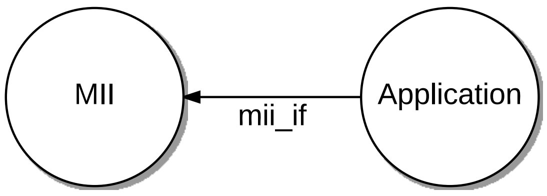

Figure 5: MII task diagram

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 (see XM-004440-PC).

3 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.

3.1 Creating a 10/100 Mb/s Ethernet MAC instance

Function	**mii_ethernet_mac**
Description	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.

Type	`void`
	`mii_ethernet_mac(server ethernet_cfg_if i_cfg[n_cfg],`
	`static const unsigned n_cfg,`
	`server ethernet_rx_if i_rx[n_rx],`
	`static const unsigned n_rx,`
	`server ethernet_tx_if i_tx[n_tx],`
	`static const unsigned n_tx,`
	`in port p_rxclk,`
	`in port p_rxer,`
	`in port p_rxd,`
	`in port p_rxdv,`
	`in port p_txclk,`
	`out port p_txen,`
	`out port p_txd,`
	`port p_timing,`
	`clock rxclk,`
	`clock txclk,`
	`static const unsigned rx_bufsize_words)`

Continued on next page

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.

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

Function	**mii_ethernet_rt_mac**
Description	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.

Type	`void`
	`mii_ethernet_rt_mac(server ethernet_cfg_if i_cfg[n_cfg],`
	`   static const unsigned n_cfg,`
	`   server ethernet_rx_if i_rx_lp[n_rx_lp],`
	`   static const unsigned n_rx_lp,`
	`   server ethernet_tx_if i_tx_lp[n_tx_lp],`
	`   static const unsigned n_tx_lp,`
	`   streaming chanend ?c_rx_hp,`
	`   streaming chanend ?c_tx_hp,`
	`   in port p_rxclk,`
	`   in port p_rxer,`
	`   in port p_rxd,`
	`   in port p_rxdv,`
	`   in port p_txclk,`
	`   out port p_txen,`
	`   out port p_txd,`
	`   clock rxclk,`
	`   clock txclk,`
	`   static const unsigned rx_bufsize_words,`
	`   static const unsigned tx_bufsize_words,`
	`   enum ethernet_enable_shaper_t shaper_enabled)`

Continued on next page

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.

3.3 Real-time supporting typedefs

Type	**ethernet_enable_shaper_t**
Description	Enum representing a flag to enable or disable the 802.1Qav credit based traffic shaper on the egress MAC port.
Values	ETHERNET_ENABLE_SHAPER Enable the credit based shaper. ETHERNET_DISABLE_SHAPER Disable the credit based shaper.

3.4 Creating a 10/100/1000 Mb/s Ethernet MAC instance

Type	**rgmii_ports_t**
Description	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;`

Fields	`in port p_rxclk`
	RX clock port.
	
	`in buffered port:1 p_rxer`
	RX error port.
	
	`in buffered port:32 p_rxd_1000`
	1Gb RX data port
	
	`in buffered port:32 p_rxd_10_100`
	10/100Mb RX data port
	
	`in buffered port:4 p_rxd_interframe`
	Interframe RX data port.
	
	`in port p_rxdv`
	RX data valid port.
	
	`in port p_rxdv_interframe`
	Interframe RX data valid port.
	
	`in port p_txclk_in`
	TX clock input port.
	
	`out port p_txclk_out`
	TX clock output port.
	
	`out port p_txer`
	TX error port.
	
	`out port p_txen`
	TX enable port.
	
	`out buffered port:32 p_txd`
	TX data port.
	
	`clock rxclk`
	Clock used for receive timing.

Continued on next page

	`clock rxclk_interframe` Clock used for interframe receive timing. `clock txclk` Clock used for transmit timing. `clock txclk_out` Second clock used for transmit timing.

Function	**rgmii_ethernet_mac**
Description	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.

Type	`void`
	`rgmii_ethernet_mac(server ethernet_rx_if i_rx_lp[n_rx_lp],`
	`   static const unsigned n_rx_lp,`
	`   server ethernet_tx_if i_tx_lp[n_tx_lp],`
	`   static const unsigned n_tx_lp,`
	`   streaming chanend ?c_rx_hp,`
	`   streaming chanend ?c_tx_hp,`
	`   streaming chanend c_rgmii_cfg,`
	`   rgmii_ports_t &rgmii_ports,`
	`   enum ethernet_enable_shaper_t shaper_enabled)`

Continued on next page

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.

Function	**rgmii_ethernet_mac_config**
Description	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.

Type	`[[combinable]]`
	`void`
	`rgmii_ethernet_mac_config(server ethernet_cfg_if i_cfg[n],`
	`   unsigned n,`
	`   streaming chanend c_rgmii_cfg)`

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()

3.5 The Ethernet MAC configuration interface

Type	**ethernet_cfg_if**
Description	Ethernet MAC configuration interface. This interface allows clients to configure the Ethernet MAC.
Functions	

Function	**set_macaddr**
Description	Set the source MAC address of the Ethernet MAC.

Type	`void set_macaddr(size_t ifnum,`
	`                  uint8_t mac_address[6])`

Parameters	`ifnum` The index of the MAC interface to set `mac_address` The six-octet MAC address to set

Function	**get_macaddr**
Description	Gets the source MAC address of the Ethernet MAC.

Type	`void get_macaddr(size_t ifnum,`
	`                  uint8_t mac_address[6])`

Parameters	`ifnum` The index of the MAC interface to get `mac_address` The six-octet MAC address of this interface

Function	**set_link_state**
Description	Set the current link state. This function sets the current link state and speed of the PHY to the MAC.

Type	`void set_link_state(int ifnum,`
	`                     ethernet_link_state_t new_state,`
	`                     ethernet_speed_t speed)`

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.

Continued on next page

Type	ethernet_cfg_if (continued)

Function	add_macaddr_filter
Description	Add MAC addresses to the filter.

Type	`ethernet_macaddr_filter_result_t`
	`add_macaddr_filter(size_t client_num,`
	`                   int is_hp,`
	`                   ethernet_macaddr_filter_t entry)`

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. `entry` The filter entry to add.
Returns	ETHERNET_MACADDR_FILTER_SUCCESS when the entry is added or ETHERNET_MACADDR_FILTER_TABLE_FULL on failure.

Function	del_macaddr_filter
Description	Delete MAC addresses from the filter.

Type	`void`
	`del_macaddr_filter(size_t client_num,`
	`                   int is_hp,`
	`                   ethernet_macaddr_filter_t entry)`

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. `entry` The filter entry to delete.

Continued on next page

Type	ethernet_cfg_if (continued)

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

Type	`void`
	`del_all_macaddr_filters(size_t client_num,`
	`int is_hp)`

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.

Function	add_ethertype_filter
Description	Add an Ethertype to the filter.

Type	`void`
	`add_ethertype_filter(size_t client_num,`
	`uint16_t ethertype)`

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.

Function	del_ethertype_filter
Description	Delete an Ethertype from the filter.

Type	`void`
	`del_ethertype_filter(size_t client_num,`
	`uint16_t ethertype)`

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.

Continued on next page

Type	ethernet_cfg_if (continued)

Function	get_tile_id_and_timer_value
Description	Get the tile ID that the Ethernet MAC is running on and the current timer value on that tile.
Type	`void get_tile_id_and_timer_value(unsigned &tile_id, unsigned &time_on_tile)`
Parameters	`tile_id` The tile ID returned from the Ethernet MAC `time_on_tile` The current timer value from the Ethernet MAC

Function	set_egress_qav_idle_slope
Description	Set the high-priority TX queue's credit based shaper idle slope.
Type	`void set_egress_qav_idle_slope(size_t ifnum, unsigned slope)`
Parameters	`ifnum` The index of the MAC interface to set the slope `slope` The slope value

Continued on next page

Type	ethernet_cfg_if (continued)

Function	**set_ingress_timestamp_latency**
Description	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.

Type	`void`
	`set_ingress_timestamp_latency(size_t ifnum,`
	`ethernet_speed_t speed,`
	`unsigned value)`

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

Function	**set_egress_timestamp_latency**
Description	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.

Type	`void`
	`set_egress_timestamp_latency(size_t ifnum,`
	`ethernet_speed_t speed,`
	`unsigned value)`

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

Type	**ethernet_link_state_t**
Description	Type representing link events.
Values	`ETHERNET_LINK_DOWN` Ethernet link down event. `ETHERNET_LINK_UP` Ethernet link up event.

Type	**ethernet_speed_t**
Description	Type representing the PHY link speed and duplex.
Values	`LINK_10_MBPS_FULL_DUPLEX` 10 Mbps full duplex `LINK_100_MBPS_FULL_DUPLEX` 100 Mbps full duplex `LINK_1000_MBPS_FULL_DUPLEX` 1000 Mbps full duplex `NUM_ETHERNET_SPEEDS` Count of speeds in this enum.

Type	**ethernet_macaddr_filter_t**
Description	Structure representing MAC address filter data that is registered with the Ethernet MAC.
Fields	`uint8_t addr` Six-octet destination MAC address to filter to the client that registers it. `unsigned appdata` An optional word of user data that is stored by the Ethernet MAC and returned to the client when a packet is received with the destination MAC address indicated by the addr field.

Type	**ethernet_macaddr_filter_result_t**
Description	Type representing the result of adding a filter entry to the Ethernet MAC.

Continued on next page

Values	ETHERNET_MACADDR_FILTER_SUCCESS The filter entry was added succesfully. ETHERNET_MACADDR_FILTER_TABLE_FULL The filter entry was not added because the filter table is full.

3.6 The Ethernet MAC data handling interface

Type	**ethernet_tx_if**
Description	Ethernet MAC data transmit interface. This interface allows clients to send packets to the Ethernet MAC for transmission
Functions	

Function	**_init_send_packet**
Description	Internal API call. Do not use.
Type	`void _init_send_packet(size_t n, size_t ifnum)`

Function	**_complete_send_packet**
Description	Internal API call. Do not use.
Type	`void _complete_send_packet(char packet[n], unsigned n, int request_timestamp, size_t ifnum)`

Function	**_get_outgoing_timestamp**
Description	Internal API call. Do not use.
Type	`unsigned _get_outgoing_timestamp()`

Continued on next page

Type	ethernet_tx_if (continued)

Function	send_packet
Description	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.

Type	`void send_packet(char packet[n],`
	`                 unsigned n,`
	`                 unsigned ifnum)`

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.

Function	send_timed_packet
Description	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.

Type	`unsigned send_timed_packet(char packet[n],`
	`                            unsigned n,`
	`                            unsigned ifnum)`

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.

Type	**ethernet_rx_if**
Description	Ethernet MAC data receive interface. This interface allows clients to receive packets from the Ethernet MAC.

Functions

Function	**get_index**
Description	Get the index of a given receiver client.
Type	`size_t get_index()`

Function	**packet_ready**

Description	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;`
	`}`

Type	`[[notification]]`
	`slave void packet_ready()`

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

Type	`[[clears_notification]]`
	`void`
	`get_packet(ethernet_packet_info_t &desc,`
	`           char packet[n],`
	`           unsigned n)`

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.

Type	**eth_packet_type_t**
Description	Type representing the type of packet from the MAC.

Values	`ETH_DATA` A packet containing data.
	
	`ETH_IF_STATUS`
	A control packet containing interface status information.
	
	`ETH_OUTGOING_TIMESTAMP_INFO`
	A control packet containing an outgoing timestamp.
	
	`ETH_NO_DATA`
	A packet containing no data.

Type	**ethernet_packet_info_t**
Description	Structure representing a received data or control packet from the Ethernet MAC.

Fields	`eth_packet_type_t type`
	Type representing the type of packet from the MAC.
	
	`unsigned len`
	Length of the received packet in bytes.
	
	`unsigned timestamp`
	The local time the packet was received by the MAC.
	
	`unsigned src_ifnum`
	The index of the MAC interface that received the packet.
	
	`unsigned filter_data`
	A word of user data that was registered with the MAC address filter.

3.7 The Ethernet MAC high-priority data handling interface

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

Type	`void`
	`ethernet_send_hp_packet(streaming chanend c_tx_hp,`
	`char packet[n],`
	`unsigned n,`
	`unsigned ifnum)`

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.

Function	**ethernet_receive_hp_packet**
Description	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.

Type	`void`
	`ethernet_receive_hp_packet(streaming chanend c_rx_hp,`
	`char packet[],`
	`ethernet_packet_info_t &packet_info)`

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.

3.8 Creating a raw MII instance

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

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

Function	**mii**
Description	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.

Type	`void mii(server mii_if i_mii,`
	`         in port p_rxclk,`
	`         in port p_rxer,`
	`         in port p_rxd,`
	`         in port p_rxdv,`
	`         in port p_txclk,`
	`         out port p_txen,`
	`         out port p_txd,`
	`         port p_timing,`
	`         clock rxclk,`
	`         clock txclk,`
	`         static const unsigned rx_bufsize_words)`

Continued on next page

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.

3.9 The MII interface

Type	**mii_if**
Description	Interface allowing access to the MII packet layer.
Functions	

Function	**init**
Description	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).
Type	`mii_info_t init()`
Returns	state structure to use in subsequent calls to send/receive packets.

Continued on next page

Type	mii_if (continued)

Function	get_incoming_packet
Description	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.
Type	`{int * unsafe, size_t, unsigned} get_incoming_packet()`
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.

Function	release_packet
Description	Release a packet back to the MII layer. This function will release a packet back to the MII layer to be used for buffering.
Type	`void release_packet(int *unsafe data)`
Parameters	`data` The pointer to packet to return. This should be the same pointer returned by `get_incoming_packet()`

Function	send_packet
Description	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.
Type	`void send_packet(int *unsafe buf, size_t n)`
Parameters	`buf` The pointer to the packet to be transferred to the MII layer. `n` The number of bytes in the packet to send.

Function	**mii_incoming_packet**

Description	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;`
	`...`

Type	`unsafe void mii_incoming_packet(mii_info_t info)`

Function	**mii_packet_sent**

Description	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;`
	`...`

Type	`unsafe void mii_packet_sent(mii_info_t info)`

Type	**mii_info_t**
Description	Type containing internal state of the mii task. This type contains internal state of the MII tasks. It is given to the application via the `init()` function of the 'mii_if' interface and its main use is to allow eventing on incoming packets via the `mii_incominfg_packet()` function.

3.10 Creating an SMI/MDIO instance

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

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

Function	**smi**
Description	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.

Type	`[[distributable]]`
	`void smi(server interface smi_if i_smi, port p_mdio, port p_mdc)`

Parameters	`i_smi` Client register read/write interface `p_mdio` SMI MDIO port `p_mdc` SMI MDC port

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

Type	`[[distributable]]`
	`void`
	`smi_singleport(server interface smi_if i_smi,`
	`               port p_smi,`
	`               unsigned mdio_bit,`
	`               unsigned mdc_bit)`

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

3.11 The SMI/MDIO PHY interface

Type	**smi_if**
Description	SMI register configuration interface. This interface allows clients to read or write the PHY SMI registers
Functions	

Function	**read_reg**
Description	Read the specified SMI register in the PHY.

Type	`uint16_t read_reg(uint8_t phy_address,`
	`                   uint8_t reg_address)`

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

Function	**write_reg**
Description	Write the specified SMI register in the PHY.

Type	`void write_reg(uint8_t phy_address,`
	`               uint8_t reg_address,`
	`               uint16_t val)`

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

3.12 SMI PHY configuration helper functions

Function	**smi_configure**
Description	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.

Type	`void smi_configure(client smi_if smi,`
	`uint8_t phy_address,`
	`ethernet_speed_t speed_mbps,`
	`smi_autoneg_t auto_neg)`

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

Type	**smi_autoneg_t**
Description	Type representing PHY auto negotiation enable/disable flags.
Values	`SMI_DISABLE_AUTONEG` Enable auto negotiation. `SMI_ENABLE_AUTONEG` Disable auto negotiation.

Function	**smi_set_loopback_mode**
Description	Function to enable loopback mode with the Ethernet PHY.

Type	`void`
	`smi_set_loopback_mode(client smi_if smi,`
	`uint8_t phy_address,`
	`int enable)`

Continued on next page

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

Function	**smi_get_id**
Description	Function to retrieve the PHY manufacturer ID number.
Type	`unsigned smi_get_id(client smi_if smi, uint8_t phy_address)`
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

Function	**smi_phy_is_powered_down**
Description	Function to retrieve the power down status of the PHY.

Type	`unsigned`
	`smi_phy_is_powered_down(client smi_if smi,`
	`                        uint8_t phy_address)`

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

Function	**smi_get_link_state**
Description	Function to retrieve the link up/down status.

Type	`ethernet_link_state_t`
	`smi_get_link_state(client smi_if smi,`
	`                   uint8_t phy_address)`

Continued on next page

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

APPENDIX A - Known Issues

There are no known issues with this library.

APPENDIX B - Ethernet MAC library change log

B.1 3.0.2

- Improve interoperability of PHY speed and link detection via RGMII inter-frame data
- Fix 64-bit alignment of MII lite to prevent crash on XS2

B.2 3.0.1

- Fixed issue with optimisation build flags not being overridden by the module
- Added missing extern declaration for inline interface function send_timed_packet()
- Added ability to override the number of Ethertype filters from the ethernet_conf.h

B.3 3.0.0

- Major rework of structure and API
- Added RGMII Gigabit Ethernet MAC support for xCORE-200