Perfection in Automation
www.br-automation.com

openMAC & Components
Documentation

Date: September 7, 2011

I Versions

Version	Date	Comment	Edited by
1.0	--/04/2009	Creation	Zelenka Joerg
1.1	--/05/2009	Phy Management Core, Register structure	Zelenka Joerg
1.2	--/09/2009	Completely reworked chapter 1 – 3 Deleted chapter 4 (Ethernet driver), added chapter Application note	Zelenka Joerg
1.3	09/02/2010	Changed template	Zelenka Joerg
1.4	15/06/2010	Changes in Avalon bus interface (2.6) Added Packet buffer management (3.2.3) Added Get TX/RX buffer base (3.2.7.5/6)	Zelenka Joerg
1.5	19/08/2010	Changed Requirements to Features (2.1) Added features (2.1) Added generics “TxDel” and “TxSyncOn” (Tab. 8) Avalon slave interface mapping	Zelenka Joerg
1.6	07/09/2011	Removed Avalon bus specific content (moved to POWERLINK IP-core documentation) Changed openMAC block diagram	Zelenka Joerg

Table 1: Versions

II License

III Table of Contents

1 Introduction

1.1 About this document

This document describes structure and function of the openMAC, openFILTER, openHUB and its HAL driver (HAL ... Hardware Abstraction Layer).

Important:

This documentation only describes the IP-Core openMAC from a generic view. If you require further information for the specific usage with Altera Nios II or Xilinx Microblaze, please refer to the corresponding documentation of the POWERLINK IP-Core!

Using the specific features of this MAC allow very short response times in EPL networks.
In chapter 4 there are given some examples to show how to integrate the whole openMAC-solution into common applications.

1.2 About openMAC

This chapter gives an overview on the MAC's features and itself. At Fig. 1 you can see the openMAC's structure (OpenMAC.vhd), ports to a processor bus and to the Reduced Media Independent Interface (RMII). The Mac includes a full transmitter (TX) and receiver (RX) – therefore its full-duplex ability is enabled. All frames will be fetched or stored from or into a memory via the included DMA. The Mac's descriptors are located in a Dual Port Ram (DPR, 16/16) to set the frame buffer's locations. This memory is accessible over a memory mapped slave interface.
To enable the hardware acceleration ability, a packet filter is included. This component compares the first 31 Bytes of a received packet and starts a transmission of a defined frame from the memory after the Inter Package Gap (960ns, 100Mbps) or a configured delay. It is possible to configure 16 different packet filters and set for every filter the "Auto-Response Feature".
Every received and transmitted packet gets a time stamp from a 32 Bit Counter and can be read from the descriptor-memory. The Counter value can be used to generate time-dependent interrupts.
The openMAC is equipped with one RMII and can be connected to an external Phy directly. Optionally a three-port (or bigger) hub can be used (OpenHUB.vhd) to attach more Phys.
Every used Phy can be configured and monitored via the Serial Management Interface with the Phy Management Interface Core (OpenMAC_PHYMI.vhd). The HAL driver automatically detects the connected Phys on the SMI bus-line.
It is recommended to use the provided distortion filter (OpenFILTER.vhd) to lock out noise from the connected network. This core should be positioned between the Mac and the Phy or between the Mac and the Hub.

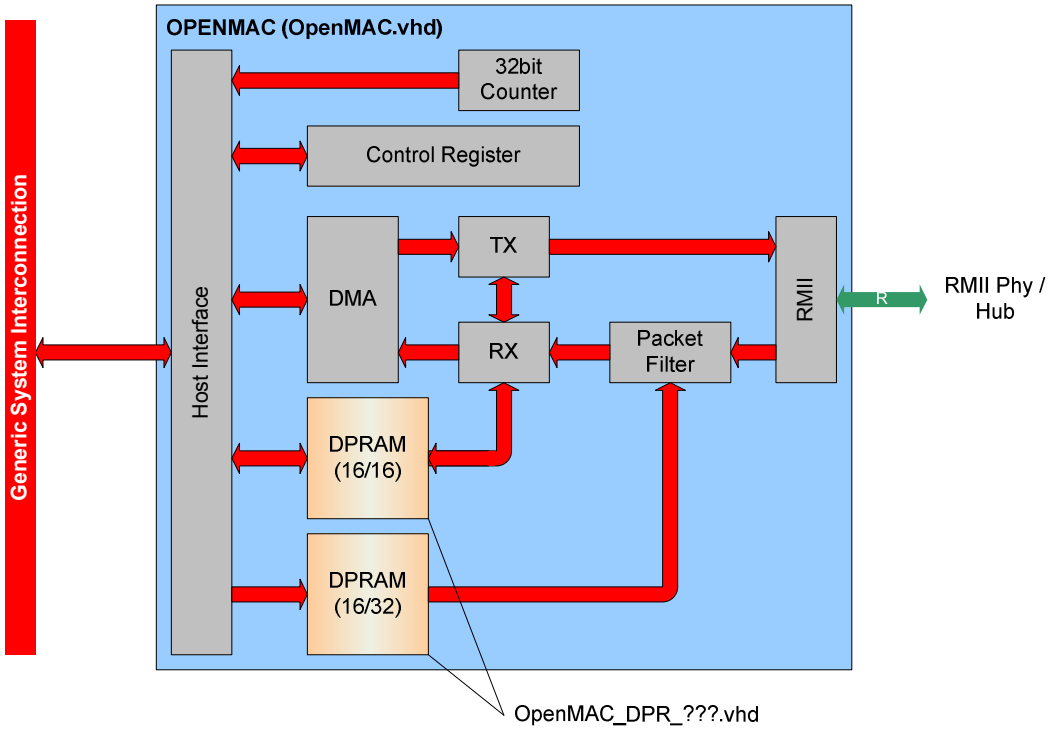


Fig. 1: Block diagram openMAC

2 IP-Cores

This chapter should give an overview on the free IP-Cores
OpenMAC.vhd (openMAC, 2.2)
OpenMAC_PHYMI.vhd (Phy Management, 2.3)
OpenHUB.vhd (openHub, 2.4)
OpenMAC_DPR_???.vhd (Dual Port RAM for MAC, depends on your chip-architecture Altera/Xilinx)

2.1 Features

The following requirements are fulfilled by the IP-Cores:

- sparse resources in FPGA
- full- and half- duplex 100Mbps MAC
- RMII (reduced media independent interface)
- 3-port hub (openHUB)
- 16 RX filters with the ability to start a transmission
- First 31 bytes are filtered by the RX filters
- auto-response feature
- low through-put time
- 32bit free-running timer for time-stamps (RX and TX) and IRQ generation
- Distortion filter (openFILTER)

The following features are added:

- IPG of auto-response packets are adjustable (necessary for Poll-Response Chaining)
- FPGA-internal TX/RX packet buffer (Altera Nios II systems only)
- Timer-triggered packet transmission (beneficial for low SoC jitter by POWERLINK MN)

2.2 OpenMAC

2.2.1 Interface

The openMAC's interface to the CPU includes descriptors, one status and one command register. It is important to connect the Mac's Clk port to a 50MHz Clk-signal.

2.2.1.1 Descriptor block

Every descriptor includes information about the received- or (to be) transmitted packets, like the address to the buffer, the packets length and some status flags. The TX- and RX-descriptors are located in the MAC's dual-port-RAM, which enables fast access (without CPU-control) by the MAC with the RMII Ref-CLK (50MHz). This block has a size of 512 Bytes and is divided into TX- and RX-descriptors (refer to Tab. 1). The openMAC supports 16 TX- and 16 RX-descriptors.

Tab. 1: DPR TX- and RX-descriptor block

Ram_Base + 0x5fc	Tx Desc. 15	Time Stamp	
Ram_Base + 0x5f8		Tx Start time	
Ram_Base + 0x5f4		Frame Pointer	
Ram_Base + 0x5f0		Tx Status	Tx Frame length
...			
Ram_Base + 0x50C	Tx Desc. 0	Time Stamp	
Ram_Base + 0x508		Tx Start time	
Ram_Base + 0x504		Frame Pointer	
Ram_Base + 0x500		Tx Status	Tx Frame length
Ram_Base + 0x4FC	Rx Desc. 15	Time Stamp	
Ram_Base + 0x4F8		-	
Ram_Base + 0x4F4		Frame Pointer	
Ram_Base + 0x4F0		Rx Status	Rx Frame length
....			
Ram_Base + 0x40C	Rx Desc. 0	Time Stamp	
Ram_Base + 0x408		-	
Ram_Base + 0x404		Frame Pointer	
Ram_Base + 0x400		Rx Status	Rx Frame length

The TX start time (in TX descriptor) is used to start a transmission at a given time (32bit MAC time).

Tab. 2: TX-descriptor status register

15	14	13	12	11	10	9	8	7	6	5	4	3..0
-	Start Time	-	Del	-	Written	Last	Owner	-	-	-	-	Tx Collision Count

Start Time … If this bit is set to one, TX starts at the given time.
Del … frame is delayed by the value in Tx Start time (plus the IPG)
Written ... Is one, if transmission is finished and the MAC is ready.
Last ... Is one, if this descriptor is the last one in list (of the 16 descriptors).
Owner ... Is one, if MAC is the owner of this descriptor. Otherwise (zero) the MAC has no access.
Tx Collisions Count ... Includes the MAC's tries on sending a packet.

Tab. 3: RX-descriptor status register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	Align Error	Hub-Port		Last	Owner	Filter Number				Noise Error	Preamb Error	Over Size	Crc Error

Align Error … Is one, if the Ethernet frame ending was not aligned to a byte.

Last, Owner ... refer to Tab. 2: TX-descriptor status register
Filter Number ... links to the filter, which passes through the packet
Hub-Port … If the openHub is connected to the Mac, then these 2 Bits represents the Hub's port, where the packet was received.
Noise Error ... Is one, if the MAC notices invalid signals on the RX data line (Crs_Dv was asserted shorter than 64 Bits!)
Preamb Error ... Is one, if the preamble was not correct.
Over Size ... Is one, if the received frame was cut, because the buffer was smaller than the message.
Crc Error ... Is one, if a check sum error occurs.
The Hub-Port value can be as follows:
1 to 3 … packet was received on this port (should not be the internal port to the Mac)
0 … error / no hub connected

The available buffer length of a RX-descriptor may be set to maximal 1518 Bytes (Basic Ethernet) or 2048, because of the receive counter. It should be set to an even number and before using this descriptor. The RX-length includes the CRC-field (4 Bytes), so the CPU needs to subtract by itself.
If the RX-length was exceeded (e.g. length was set to 512 Bytes, but a packet with 1024 Bytes was received) the status flag OvMax (in the RX-descriptor's status register) will be set. The odd data won't be written to the RAM, but the MAC performs a CRC control and sets the CRC-Error bit if necessary.
The application needs to pay attention on the TX-packet-length. In case of Tx Frame Length shorter than 60 Bytes, the MAC sends an invalid frame with the set length. Thus, the application must set the minimum length of 60 Bytes and set the padding bytes to zero.
The frame pointer (valid for TX and RX) needs to point on a word address (word alignment!).
After transmission the TX frame length is set to the sent bytes or to the byte number, when the collision occurs (e.g. collision with the first byte => Col = 0).
The time stamp register will be set by starting the TX preamble or receiving the SOF (start of frame). The value's source is a free-running 32 bit counter (50MHz). When collisions are allowed the time of the last try is recorded in the time stamp.
In the TX-descriptor status register the bits 8 and 9 (and also 11 to 15) will be set to zero, after successfully sending a frame. When writing on the descriptor the bit "Written" will be set. So it is possible to recognize the TX-packets end independent of the Owner bit.

2.2.1.2 TX/RX control/status registers

TX control/status registers (all addresses access to the same register with different functions)

Tab. 4: TX control/status registers

Register Base + 0 Rd Read TX Status

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	Half	-	Int. Pending	Run	-	Tx Idle	-	Descriptor Pointer

Register Base + 0 Wr TX Control

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	Half	-	-	Run	-	-	-	-

Register Base + 2 Wr Set Bit sets Bits to one

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	Half	-	-	Run	-	-	-	-

Register Base + 4 Wr Clr Bit clears Bits to zero

15	14	13	12	11..9	8	7	6	5	4	3..0
IE	-	Half	-	-	Dec Int	Run	-	-	-	-

Register Base + 6 Wr set TX descriptor pointer

15	14	13 … 8	7	6	5	4	3..0
-	Set IPG	Inter Package Gap / 20ns	-	-	-	-	Descriptor Pointer

Assert “Set IPG” bit when setting the IPG Value!

RX control/status registers (all addresses access to the same register with different functions)

Tab. 5: RX control/status registers

Register Base + 8 Rd Read RX Status

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	-	-	Int. Pending	Run	-	Rx idle	Lost	Descriptor Pointer

Register Base + 8 Wr RX Control

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	-	-	-	Run	-	-	-	-

Register Base + 10 Wr Set Bit sets bits to one

15	14	13	12	11..8	7	6	5	4	3..0
IE	-	-	-	-	Run	-	-	-	-

Register Base + 12 Wr Clear Bit clears bits to zero

15	14	13	12	11..9	8	7	6	5	4	3..0
IE	-	-	-	-	Dec Int	Run	-	-	Lost	-

Register Base + 14 Wr set RX descriptor pointer

15	14	13	12	11..8	7	6	5	4	3..0
-	-	-	-	-	-	-	-	-	Descriptor Pointer

The Descriptor Pointer may only be set when the Run bit is set to zero!
To set the MAC into Half-Duplex-Mode the Half bit should be set to one. In this mode collisions are possible.
An interrupt (TX) occurs when the first transmission try was successful or 16 collisions happened. With setting the Run bit to one, the transmitter and receiver are enabled.
The IE bit should be set to one to enable interrupts. When an interrupt occurs the Int Pending field will be incremented (per interrupt). To acknowledge the interrupt, the CPU needs to access via Clear Bit and set Dec Int. this will decrement the Interrupt Pending field. The interrupt will be set until the Int Pending field is zero or the IE bit is cleared. This logic makes sure to not lose any interrupt or execute it twice.
After the Interrupt was acknowledged, the descriptor pointer will be set to the next descriptor, which may be owned by the MAC (to send the packet or copy the next received frame into its buffer).
The bit Lost will be set, when a frame was received, but the Run bit was set to zero or there was no descriptor for the MAC available (every Owner was zero).
The idle bits (TX and RX) are set to one, if there is no transmission or reception active.

2.2.2 Receive filters

The openMAC contains 16 different filters for filtering the incoming packets. The filter is able to compare the first 31 Bytes after the Start of Frame Delimiter (SFD).
The filtering is done with a value and a mask and will be calculated in the following way:

```
Match = ( rxdata[30..0] xor filterValue[30..0] ) and filterMask[30..0]
```

When Match is zero, the packet fits to the filter; otherwise the packet will be checked with the next filter.
When the packet does not match with any given filter, the packet won't be received by the MAC.

Tab. 6: filter structure

	Addr + 0	Addr + 1
	31..24 (+0)	23..16 (+1)
Ram_Base + 0x1FE		Command 15
Ram_Base + 0x1FC	Maske_15 [30]	Wert_15 [30]
….		
	Maske_15 [n]	Wert_15 [n]
…..		
Ram_Base + 0x3C0	Maske_15 [0]	Wert_15 [0]
….		
Ram_Base + 0x04E		Command 1
Ram_Base + 0x04C	Maske_1[30]	Wert_1 [30]
….		
	Maske_1 [n]	Wert_1 [n]
…..		
Ram_Base + 0x040	Maske_1 [0]	Wert_1 [0]
Ram_Base + 0x03E		Command 0
Ram_Base + 0x03C	Maske_0 [30]	Wert_0 [30]
….		
	Maske_0 [n]	Wert_0 [n]
Ram_Base + 0x002	Maske_0 [1]	Wert_0 [1]
Ram_Base + 0x000	Maske_0 [0]	Wert_0 [0]

The MAC is using the RMII Interface, so per Clock only two bits are transmitted. Therefore the filter needs to be executed within 4 clocks. In one clock 4 filters will be compared with the received packet.
The processing starts after the 31st received byte with filter 0. After the first match, the filter number will be set in the RX-descriptor (refer to Tab. 3), if there is a free RX-buffer.
It is impossible to read the filter image directly, so the CPU needs to read from the copy in the memory.
The filter is realized with a 16x32 bit RAM block. The 16 bit site is connected to the openMAC's core and the 32 bit site is connected to interpretation logic.
When the received message fits to the filter the command register (refer to Tab. 7) will be interpreted.

Tab. 7: filter command register (0…15)

7	6	5	4	3..0
Tx_Enable	Filter On	-	-	Tx Desc. #

Filter On is set to 1, if the filter value and mask is valid.

When the filter's Tx_Enable bit is set, the packet of the descriptor (Tx Desc. #, bit 3 to 0) will be sent immediately, when its owner bit (refer to Tab. 2) is set to one (MAC is the owner); otherwise there won't be a transmission.
Examples:
When the filter should pass through broadcasts, the first six bytes need to be 0xFF of the filter-value and mask.
When all filter-mask bytes are set to 0x00 and the Filter On bit is set, then the filter will pass through all received packets to the MAC. If this filter is the first entry, all packets will be executed.

2.2.3 Timing

The openMAC has one slave-interface that is connected to the internal DPRs (filter and descriptors) and to the control/status registers. The Mac's DMA (direct memory access) has access over a simple master-interface. It is crucial to consider the following timing requirements!
The following figures were taken with Altera Quartus II Signal Tap and edited afterwards. X stands for don't care or invalid!

2.2.3.1 Slave interface

When reading from openMAC the output data is valid after one cycle (refer to Fig. 2).
When performing a write instruction all signals (address, data, write, byte enable) can be asserted synchronously with Sel_Ram/Sel_Cont (refer to Fig. 3).

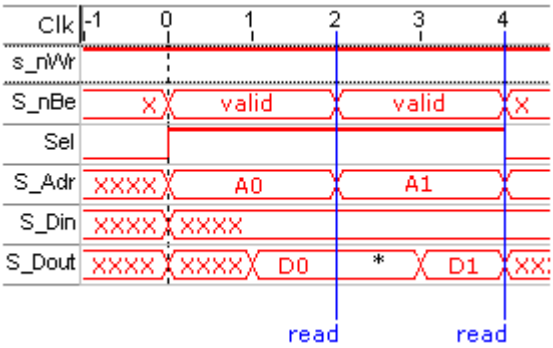


Fig. 2: read from openMAC
*) second read: one cycle S_Dout is data from last read

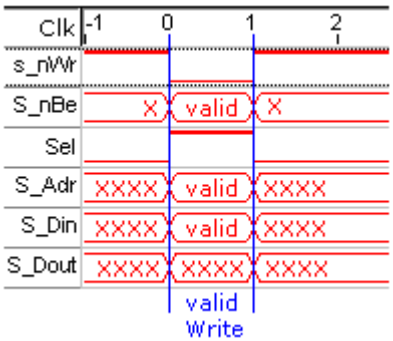


Fig. 3: write to openMAC

CAUTION:

The write cycle's length should be only one clock, otherwise more IRQs (refer to Tab. 4 and Tab. 5 – "dec Int") will be acknowledged within one write instruction!

2.2.3.2 Master interface (DMA)

The DMA starts the access with asserting the Dma_Req signal, Dma_Rw (Read = 1, Write = 0), Dma_Addr (word addresses!) and Dma_Dout (Write access). This condition is held until the Dma_Ack signal is set.
When the DMA performs read commands (send packet) the data (Dma_Din) should be valid one cycle after Dma_Ack was set. So, Dma_Ack should be asserted only one cycle. Please refer to Fig. 4!
In case of DMA write accesses (receive packet) the data (Dma_Dout) is valid when the Dma_Req signal is set. The data is held until the next DMA Request follows. Please refer to Fig. 5!
If using the Mac in half-duplex mode, every 8th cycle (50MHz) will be a new DMA write/read access (the first DMA Read Request starts earlier – therefore it is longer). If the Mac is set to full-duplex mode, every 4th cycle a new request starts when simultaneous read and write is necessary (send packet and receive packet at the same time).

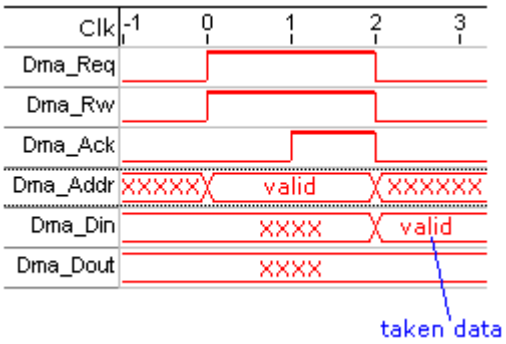


Fig. 4: DMA read access

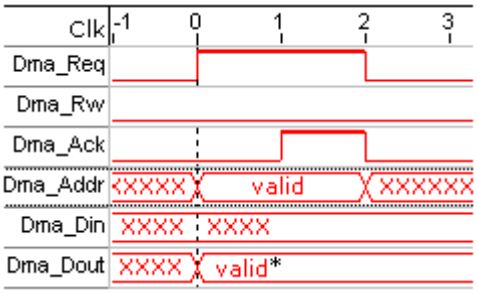


Fig. 5: DMA write access

*) out data valid till next Dma_Req

2.2.4 Port/generic map

Tab. 8: generic map openMAC

Generic	Description
HighAdr	The highest address bit in the Dma_Addr vector. Note: The DMA can only address 1GB, because in case of Auto Response Packets the 30th and 31st bits will be masked alternately with zero or one (refer to 3.2.5.2).
Timer	Enables the internal Mac Timer for using time stamp feature, time-dependent IRQ generation, timer-triggered TX (TxSyncOn) and auto-response delay (TxDel)
TxSyncOn	Enables the time-triggered packet transmission (HAL: omethStartTime) Caution: needs the Timer!

TxDel	Enables to adjust the IPG of auto-response packets (adjustable for every filter) Caution: needs the Timer!
Simulate	Is necessary to simulate the openMAC

Tab. 9: port map openMAC

Port	Description
nRes	Low active synchronous reset signal
Clk	The Clock signal for the MAC. Must be 50MHz (±25ppm) and synchronous to the Phys RefClk.
S_nWr	Low active write signal
Sel_Ram	Selects the filter and descriptor memory range (DPR)
Sel_Cont	Selects the Mac's status and control registers for TX and RX
S_nBe	Low active Byte Enable vector. S_nBe(1) = 0 selects s_Din(16 downto 8)
S_Adr	Word address
S_Din	In data
S_Dout	Out data
nTx_Int	Low active TX interrupt. Is '1' as long as no IRQ is pending.
nRx_Int	Low active RX interrupt. Is '1' as long as no IRQ is pending.
nTx_BegInt	Not yet supported.
Dma_Req	DMA access request signal
Dma_Rw	Read DMA access (TX packet) when signal is '1'. Otherwise Write access (RX packet)
Dma_Ack	DMA acknowledge signal
Dma_Addr	DMA address vector
Dma_Dout	DMA out data
Dma_Din	DMA in data
rRx_Dat	RMII receive half-nibble
rCrs_Dv	RMII Carrier Sense / Data Valid
rTx_Dat	RMII transmit half-nibble
rTx_En	RMII Transmit Enable
Hub_Rx	Connected to the openHub (2.4) to detect the hub port of the RX packet
Mac_Zeit	Outputs the free running 32bit MAC Timer

2.3 Phy management core

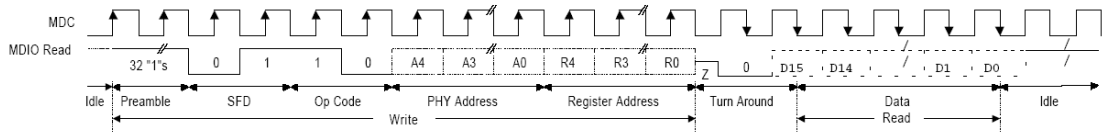
The Physical Interface (Phy) is identified, configured and controlled with SMI (serial management interface, IEEE 802.3). It is possible to address 32 different Phys.

2.3.1 Serial management interface (SMI)

The transmission is done over a two-wire-connection with a clock wire and a pull-up data line. So every Phy's SMI can be connected to one line (Attention: Phys' Hardware Address must be different to each other!). The data exchange starts with 32 bits preamble (PRE) and two start bits (ST). The operation code (OP) decides a write or read command. The next 10 bits includes the (unique) Phy's address (PHYAD) and the register address (REGAD). After that, there are two bits for turn around (TA) and then the date will be transmitted (16 bit, MSBit first).

	PRE	ST	OP	PHYAD	REGAD	TA	DATA	IDLE
READ	1…1	01	10	AAAAA	RRRRR	Z0	DDDDDDDDDDDDDDDD	Z
WRITE	1…1	01	01	AAAAA	RRRRR	10	DDDDDDDDDDDDDDDD	Z

Management Interface - Read Frame Structure



Management Interface - Write Frame Structure

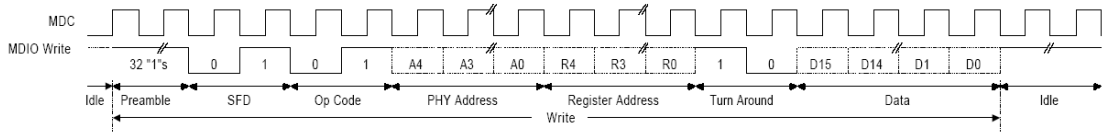


Fig. 6: Serial Management Interface

To get further information about the SMI protocol, please refer to http://www.smsc.com/main/anpdf/an79.pdf.

2.3.2 Interface

Before the Core writes data to the Phy, the data register, the Phy Address and Register Number need to be set to the desired values. To start a transmission set the control register to the value 0x5002 (write command) or 0x6000 (read command), masked with Phy Address and Register Number.

e.g. send to Phy 0x15 (1 0101_{bin}) Mii-Register 0x01 (0 0001_{bin}): 0x5002 OR 0x0A84 = 0x5A86

Tab. 10: Mii-core status registers (RegBase + 0) (read)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	nRst	-	-	-	-	-	-	Busy

nRst … the Reset-Signal state (0 … reset)
Busy … If a transmission (SMI) is active, the Busy flag is 1.

Tab. 11: Mii-core control registers (RegBase + 0) (write)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	01 Write 10 Read		Phy Address [4..0]					Register Number [4..0]					1 Write 0 Read	0

Tab. 12: Mii-core data register (RegBase + 2) (read/write)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MII Read /Write Data [15..0]															

Tab. 13: Mii-core reset command register (RegBase + 4) (write)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	-	nRst	-	-	-	-	-	-	-

nRst … Controls the nReset pin of the Phy(s) (0 … reset, 1 … active)

2.3.3 Timing

The Mii Core's slave interface has the same timing requirements like the openMAC's slave interface (refer to 2.2.3.1).

2.3.4 Port map

Tab. 14: port map Mii-core

Port	Description
Clk	50MHz Clock signal
nRst	Low active synchronous reset
Addr	Word address
Sel	Select signal
nBe	Low active Byte Enable vector. nBe(1) = 0 selects Data_In(16 downto 8)
nWr	Low active write signal
Data_In	Data in
Data_Out	Data out
Mii_Clk	Clock signal to the Phys' serial management interface
Mii_Di	Input signal from the Phys' serial management interface
Mii_Do	Output signal to the Phys' serial management interface
Mii_Doe	Output enable signal for tri-state buffer ('0' = output, '1' = input)
nResetOut	Low active reset signal to Phys

2.4 OpenHUB

The openHUB can be used to connect the openMAC to more than one Ethernet port (Phy). The ports nRst and Clk can be connected to the Phy's RMII Reference Clk (50MHz) and Reset.
The Generic Value "Ports" needs to be set to the number of needed ports. Its default and minimum value is 3, values like 0 or 1 are useless.
The following ports need to be connected to the Phys / MAC:
RxDv, RxDat0, RxDat1, TxEn, TxDat0 and TxDat1
The following ports can be left open
internPort ... Sets the port connected to the MAC. Default value is 1
TransmitMask ... Can enable / disable the associated port by one / zero. Its default value is all ones.
ReceivePort ... Outputs the active port (0 = inactive Hub, 1...n = Ports)
The openHUB + openFILTER create the following delays: Mac to Phys = 3 Cycles // Phys to Mac = 12 Cycles

2.4.1 Port/generic map

Tab. 15: generic map openHUB

Generic	Description
Ports	Sets the amount of hub-ports. Should be 3 in case of a 2-port-hub (one port is reserved for Mac)

Tab. 16: port map openHUB

Port	Description
nRst	Low active synchronous reset
Clk	50MHz RMII Clock
RxDv	RMII Carrier Sense / Data Valid from every Port
RxDat0/1	RMII RX Data from every Port
TxEn	RMII TX Enable to every Port
TxDat0/1	RMII TX Data to every Port
internPort	Should be set to the port connected to the Mac
TransmitMask	Enable/disable the Hub's ports ('1' = enabled, '0' = disabled)
ReceivePort	Outputs the port number of the currently received packet (0 = idle) Connect this signal to the Mac's Hub_Rx Port (convert to std_logic_vector!)

2.5 OpenFILTER

The openFILTER core is used to avoid invalid Crs_Dv signals propagating to the MAC. If only one Phy is used, openFILTER has to be inserted between MAC and Phy. If two Phys are connected to the MAC via the openHUB, every Phy should be equipped with openFILTER. This approach prevents distortion propagation across the whole network.
The signals nRst and Clk should be connected to the RMII Ref_Clk and Rst. The nCheckShortFrames signal is low active.

Important:

The openFILTER is an optional (but recommended) addition to openMAC. However, please don't confuse openFILTER with the packet filter functionality (2.2.2) of openMAC. OpenFILTER does not observes the content of Ethernet packets!

2.5.1 Port map

Tab. 17: port map openFILTER

Port	Description
nRst	Low active synchronous reset
Clk	50MHz RMII Clk
nCheckShortFrames	Low active control signal. Set to '0'
RxDvIn	RMII Receive Carrier Sense / Data Valid (from Phy or Hub)
RxDatIn	RMII Receive Data (from Phy or Hub)
RxDvOut	RMII Receive Carrier Sense / Data Valid (to Mac)
RxDatOut	RMII Receive Data (to Mac)
TxEnIn	RMII Transmit Enable (from Mac)
TxDatIn	RMII Transmit Data (from Mac)
TxEnOut	RMII Transmit Enable (to Phy or Hub)
TxDatOut	RMII Transmit Data (to Phy or Hub)

3 HAL Ethernet driver

The HAL driver (=Hardware Abstraction Layer) controls the openMAC- and Mii Core, and can be used for every Ethernet application (TCP/IP, EPL, raw Ethernet …). In Fig. 7: block diagram, example for HAL usage you can see how the HAL driver can be included into an application (also have a look into chapter 4 - Application note).

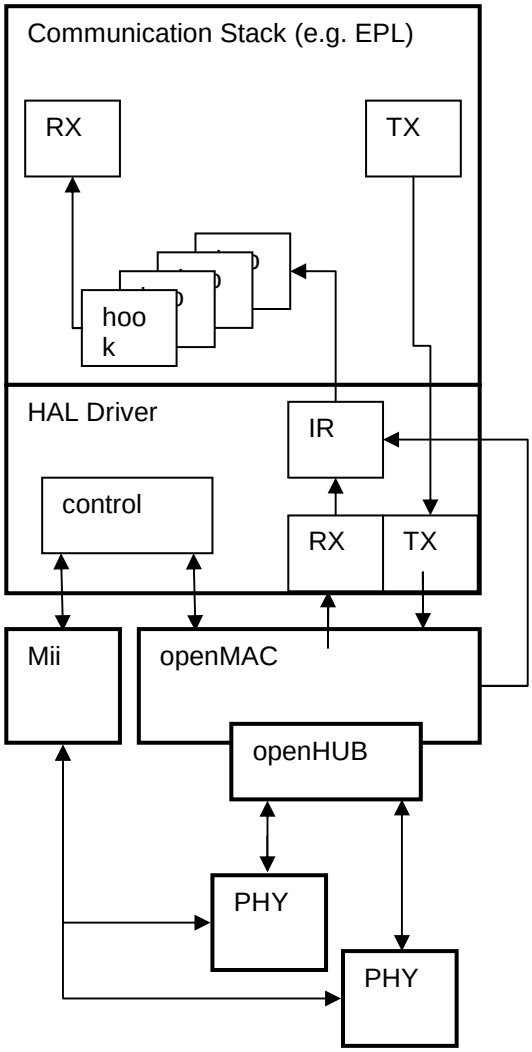


Fig. 7: block diagram, example for HAL usage

3.1 Features

3.1.1 Time stamp

For each received and sent packet a 32 Bit / 20ns timestamp is generated.

3.1.2 Packet filters

The MAC supports 16 packet filters with 31 byte filter and 31 byte mask for each filter.

The filters can be used to…

- determine the type of received frames (like the address-filter on conventional MACs)

- transmit a dedicated packet directly after receiving a packet with defined format (with the defined inter-frame gap of 960 ns)
- transmit a packet from the transmit queue directly after a packet with defined format

Tab. 18: filter example

Byte	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	30
Mask	00	00	00	00	00	00	00	00	00	00	00	00	FF	FF	FF	00	...	00
Value	Xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	xx	88	AB	01	xx	...	xx

Offset 12/13 = 0x88AB … Ethertype = EPL V2
Offset 14 = 0x01 … Message ID = Start Of Cyclic

Each incoming packet will be compared with all 16 filters (In the order as the filters were installed by the application).
Only the bits will be compared where the respective mask bit is set. If a matching filter was found, a specific action will be performed.

3.1.3 Auto response

Triggered by a defined incoming packet the MAC can automatically transmit a packet after the minimum frame gap of 960 ns.
This gives a constant and very short response time to EPL frames which require an immediate response (PollRequest, IdentRequest, …)

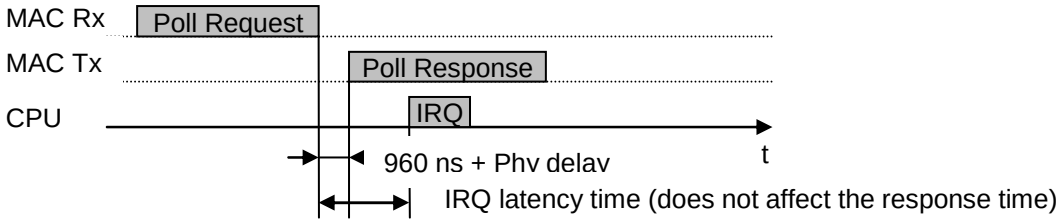


Fig. 8: auto response

3.2 Ethernet driver

3.2.1 Files

Tab. 19: HAL Ethernet driver files

File	Description
omethLib.c	Driver Functions (not to be changed by the user)
omethLib.h	Defines and Declarations (not to be changed by the user)
omethLib_target.h	Target specific defines OMETH_HW_MODE Defines the type of the used CPU, mode 0/1 are general modes for big/little endian systems OMETH_MAKE_NONCACHABLE(ptr) Defines a function to make a pointer 'non-cacheable'. The file omethLib.c will use the defined function for non-cacheable access to the FPGA registers and to the receive packets. If the CPU has no data cache no function has to be defined: #define OMETH_MAKE_NONCACHABLE(ptr) (ptr) !! Attention !! Transmit packets are managed by the user. Therefore the user has to make sure that packets passed to a transmit function are non-cacheable.

3.2.2 IRQ subroutines

The user has to connect the Ethernet driver to the IRQ system of the CPU. Generally the RX IRQ should have the higher priority than the TX IRQ. Every received packet that matches a filter generates a RX interrupt!
Every transmitted packet (sent with omethTransmit & Co or auto response) generates a TX interrupt! If the TxIrqHandler is executing an "auto response" occurred interrupt, the user's free function **won't be called**. If the TxIrqHandler is executing an "user sent" occurred interrupt (sent with omethTransmit & Co) the user's free **function is called**.

Tab. 20: IRQ handlers

omethRxIrqHandler **omethTxIrqHandler**	To be used if the IRQ is generated by only 1 MAC and the used system is able to pass an user argument to the IRQ routine. As argument the handle to the driver instance (returned by omethCreate) has to be passed to the IRQ routine.
omethRxIrqHandlerMux **omethTxIrqHandlerMux**	To be used if the IRQ is generated by more than 1 MAC or the system does not support arguments to IRQ subroutines.
omethRxTxIrqHandler	Has to be used if one IRQ is used for RX and TX. As argument the handle to the driver instance (returned by omethCreate) has to be passed to the IRQ routine. Furthermore the priority (sequence of execution) has to be set.

3.2.3 Basic API functions

At least this set of basic API functions is required to receive and transmit frames on the Ethernet interface. To implement Ethernet Powerlink also the extended API functions are required.

3.2.3.1 omethMiiControl

Function to activate or reset the Ethernet Phys.
(Per default the Phy nReset pin is 0 … Phys are in reset state)
Before initializing the Ethernet driver (omethCreate) the application has to make sure the used Phys are active and available. Depending on the type of Phy after activation it may take some time until the Mii interface of the Phys is ready. This delay must be implemented by the user.

3.2.3.2 omethInit

The function has to be called at start up of the system. All driver internal variables and instance references will be cleared.
(The function will not 'free' any previously allocated resources)

3.2.3.3 omethCreate

omethCreate initializes an Ethernet driver instance.
The parameters are passed by a structure which will be copied to internal memory (the same structure can be used to initialize other instances later) – refer to Tab. 21.

Tab. 21: instance configuration type „ometh_config_typ"

Element	Description / Value
macType	OMETH_MAC_TYPE_01

Mode	Combination of OMETH_MODE_HALFDUPLEX OMETH_MODE_FULLDUPLEX OMETH_MODE_100 OMETH_MODE_DIS_AUTO_NEG OMETH_MODE_PHY_LIST (enables the elements phyCount and phyList) OMETH_MODE_SET_RES_IPG
Adapter	The user defines the adapter number for the Ethernet interface. Each instance has to be initialized with an unique number.
pRamBase	Pointer to MAC Descriptor/Filter Registers
pRegBase	Pointer to MAC Control Registers
pBufBase	Pointer to MAC internal packet buffer (on-chip memory)
rxBuffers	Number of used RX buffers (up to 16)
rxMtu	Maximum MTU of incoming packets. Packets exceeding this size will be discarded. Maximum Packet Size = MTU + 18 (DstMAC, SrcMAC, Ethertype, CRC)
pPhyBase	Pointer to phy management port: All phys on this port are assigned to this driver instance, except the element mode contains OMETH_MODE_PHY_LIST, in this case only the listed phys will be assigned to this instance. This dedicated phyList is required if the same pPhyBase is used for phys which do not belong to this driver instance.
phyCount	Only valid if mode contains OMETH_MODE_PHY_LIST: Number of valid entries in phyList
phyList	Only valid if mode contains OMETH_MODE_PHY_LIST: Array with up to 8 phy addresses of the phys which are assigned to this driver instance.
responseIpg	Inter Package Gap (available when mode contains OMETH_MODE_SET_RES_IPG): This value changes the Mac's IPG and should be given in [ns]. Values lower than 140ns will result in IPG = 140ns

For each used RX buffer a packet will be allocated. The next incoming frame matching one of the installed filters will be stored to the next free packet in the queue.

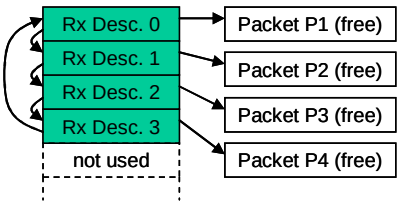


Fig. 9: receive queue after omethCreate with rxBuffers=4

3.2.3.4 omethHookCreate

To be able to receive packets the user has to create hooks (call backs) which will be called by the IRQ subroutine.
If 'maxPending=0' the hook is not allowed to queue the received packet. The processing of the packet data has to be done directly in the hook. After terminating the hook function the received packet will be passed back to the receive queue.
If 'maxPending > 0' the hook allocates its own pool of packets.
When receiving a packet which belongs to the hook the IRQ subroutine will pass the packet to the hook function.
The hook function can decide whether it wants to queue the packet (return 0) or do pass the packet directly back to the receive queue (return -1).
If the hook has queued the packet the next free packet of the hook's packet pool will be used to replace the current descriptor in the receive queue (Because all descriptors of the receive queue have to be linked to free packets at all time!)

Fig. 10: buffers after omethHookCreate(…, &hookFct, 2)

The following example shows what happens if a packet for this RX hook is received and the hook function has been queued the packet:

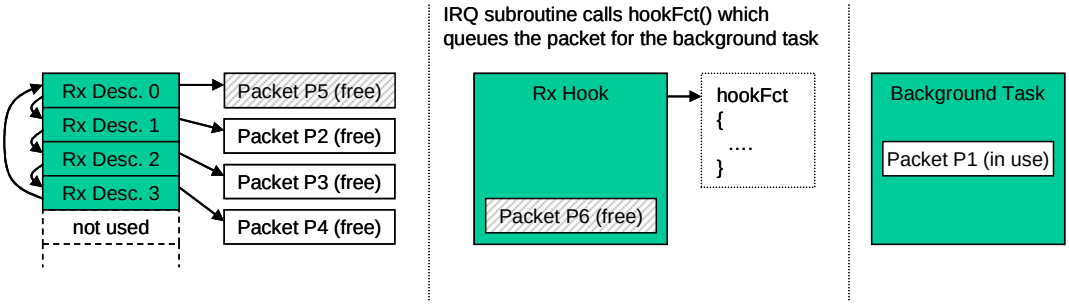


Fig. 11: buffers after a packet was queued by the background task

If the RX hook does not have free packets anymore the hook function will not be passed to the background task, the packet remains in the RX descriptor for the next received packet.
This ensures that the RX queue is always complete.
Even if the background task of one hook does not free the packets (because of an application error or because of leak of CPU performance) the other hooks still will get packets.
After processing the packet the background task has to free the packet (pass back to hook).

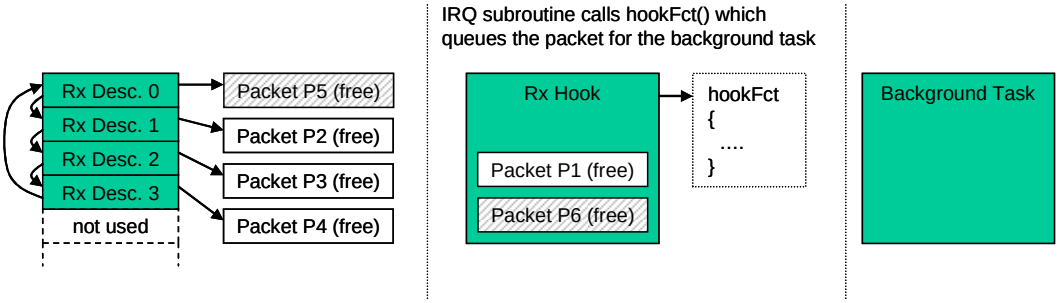


Fig. 12: buffers after a background task has processed packet

3.2.3.5 omethFilterCreate

A hook can receive packets matching one ore more packet filters. With omethFilterCreate the user has to define these filters.
The parameter 'arg' will be passed to the hook to allow a efficient demultiplexing if more than 1 filters are linked to the same hook.

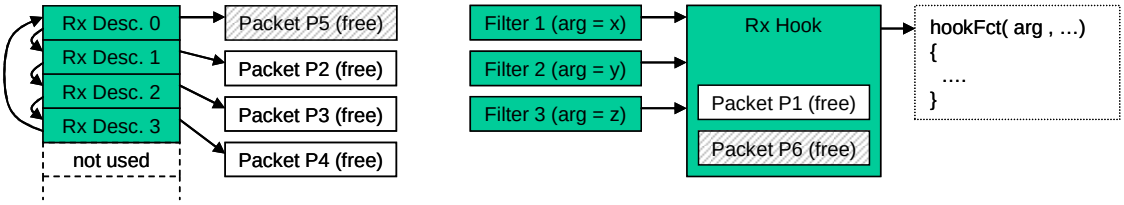


Fig. 13: filters and hooks

3.2.3.6 omethTransmit

Transmits a packet to the send queue of the MAC. The packet will be sent as soon as possible to the network.
The user has to pass a function pointer to a free-function. A packet which was passed to the send queue must not be changed until the driver has released the packet by calling the free-function.
!! Attention !!
If the CPU uses data cache the application has to make sure the packet is really transferred to the memory before calling omethTransmit.

3.2.3.7 omethStart

Start Ethernet driver. (Make sure the IRQ system is initialized before starting)

3.2.3.8 omethStop

Stop Ethernet driver.

3.2.3.9 omethPeriodic

The function must be called by the user cyclically.
It does the update of the PHY-Register image (see 3.2.6.2) and controls the Full/Half duplex mode of the MAC depending on the connected communication partner.
Every call the function will retrieve maximum 1 phy register.
Example:
The function is called every 5 ms on a system with 2 interfaces and 4 Phys on each interface.
Total Phy ports : 8
Total Phy registers : 64 (the first 8 registers of each phy will be read)
The update cycle of the function omethPeriodic will be 5ms x 64 = 320 ms

3.2.3.10 omethDestroy

Stops the Ethernet driver and releases all allocated resources. Also resources allocated with omethHookCreate, omethFilterCreate, … will be released.
After omethDestroy the interface can be initialized again with omethCreate

3.2.4 Filter manipulation

3.2.4.1 omethFilterSetPattern

Change filter pattern (mask and value) of an filter created with omethFilterCreate.

3.2.4.2 omethFilterSetArgument

Change the hook argument for an filter created with omethFilterCreate.

3.2.5 Auto response functionality

The following functions are required to handle the openMAC auto response feature.

3.2.5.1 omethResponseInit

The function has to be called before omethStart and prepares a defined filter entry for auto response frames.
The response frame will be finally activated at the first call of omethResponseSet.

3.2.5.2 omethResponseSet

Passes a packet to the Ethernet driver which will be sent to the network if a received frame matches the given filter.
After passing the packet to the driver, the packet has to be considered as 'locked', the user is not allowed to change it until the driver passes the packet back to the user.
The return value can have the following values:

- OMETH_INVALID_PACKET : An error occurred (invalid filter passed, invalid packet passed)
- 0 : Function call successful
- > 0 : The driver passes a packet back to the user, the packet can be reused

The packet pointer will be masked alternately with 0x80000000, 0x40000000 or 0x00000000. So, the TX packets should be located in memory space e.g. 0x00000000 - 0x3FFFFFFF!

3.2.5.3 omethResponseDisable

Disables the auto response feature. The user can continue passing packets to the driver with omethResponseSet and will still get call backs on the given filter, only the frame transmission is disabled.

3.2.5.4 omethResponseEnable

Enable the auto response feature. The last packet passed with omethResponseSet will be activated for auto response.

3.2.6 Other API functions

3.2.6.1 omethGetHandler

Retrieves the instance handle of an interface created with omethCreate based on the given adapter number.

3.2.6.2 omethPhyInfo

Retrieves the pointer to a PHY-Register image of the driver instance.

3.2.6.3 omethHookSetFunction

Change the call back function of a receive hook.

3.2.6.4 omethGetTimestamp

Get timestamp of a received packet (Resolution: 20ns)

3.2.6.5 omethGetTxBufBase

This function returns the MAC-internal TX buffer base address.

3.2.6.6 omethGetRxBufBase

This function returns the MAC-internal RX buffer base address.

3.2.6.7 omethStatistics

Retrieve the pointer to the statistics structure of the driver instance.

3.2.6.8 omethSetSCNM

Sets the SCNM (Slot Communication Network Mode).
By default all frames sent with omethTransmit will be sent to the network as soon as possible. The MAC runs in collision mode, if collisions occur the MAC will repeat the packet.
The parameter hFilter can have the following values:

- 0: The MAC is set to collision mode (default mode after omethCreate)
- valid filter handle: The MAC will send packets passed with omethTransmit only after an incoming frame matches the given filter. In case of collisions the MAC will NOT retry the transmission.
- OMETH_INVALID_FILTER: The send queue is disabled, packets passed with omethTransmit will be queued but will not be sent to the network.

3.2.7 Ethernet packet structure

The Ethernet packets for (to transmit to the network) and from the openMAC (to receive from the network) have a structure defined in the omethlib.h file (ometh_packet_typ). It is recommended to use this type to build Ethernet frames in your application. Before sending the packet with omethTransmit or setting an auto-response frame with omethResponseSet, the length member need to be set to the Ethernet frame's length excluding the CRC field!

Tab. 22: Ethernet packet struct

Member	Description
length (4 bytes)	The packets length without the checksum in bytes. e.g. smallest Ethernet packet with crc is 64 bytes => set length to 60 bytes
dstMac (6 bytes)	destination MAC address
srcMac (6 bytes)	source MAC address
Ethertype (2 bytes)	Ethernet type / length field
minData (min. 46 bytes)	data field – the minimum length is 46 bytes (header is 14 bytes)
checkSum (4 bytes)	packet's checksum field Don't care this field; the MAC does it for the application.

```
//********** packet structure for ethernet frames ***********************
typedef struct
{
   unsigned long  length;             // frame length excluding checksum

   struct ometh_packet_data_typ
   {
    unsigned char dstMac[6];
    unsigned char srcMac[6];
    unsigned char ethertype[2];
    unsigned char minData[46];// minimum number of data bytes for a standard EthFrame
    unsigned char checkSum[4];
   }data;
}ometh_packet_typ;
```

4 Application note

In order to show how the openMAC and its components (Hub, Filter, Drivers …) are used in an open-Powerlink Controlled Node (Powerlink Slave), this chapter was inserted. It is recommended to follow this reference to achieve high performance, because of very low response delays when using the openMAC. For this reason the best way is to add a second CPU which processes application data and leave one CPU for running the openPowerlink communication-stack. In Fig. 14 a generic design is shown of a “Two-Processor System” using the openMAC (incl. openFILTER, openHUB …) as an interface to the Powerlink network. The “Communication” CPU is used to control the MAC and execute the whole openPowerlink Stack. Information can be exchanged through an interface (e.g. SPI, DPR, PCI, Mailbox, Mutex …). Sensors, actuators and other I/Os can be connected to the second CPU to complete a Powerlink Controlled Node.

4.1 FPGA design with two soft-core CPUs

If an FPGA (Field Programmable Gate Array) is used it is possible to integrate a Two-Processor System into one chip to save circuit board area. Chip manufacturer (like Altera and Xilinx) provide Tool Chains which can handle easily Multi-Processor applications.
The openMAC was designed to save area and chip resources, but the actual requirement depends on the application! Refer to Fig. 15 – the “Communication” and “Application” CPU can be e.g. Nios II (Altera), Microblaze (Xilinx) or any other soft-core processor.

4.2 FPGA for communication tasks only

If it is not possible to integrate the whole application into one chip, an external µprocessor/controller can be connected to the FPGA including the “Communication” soft-core processor and openMAC. Refer to Fig. 16.

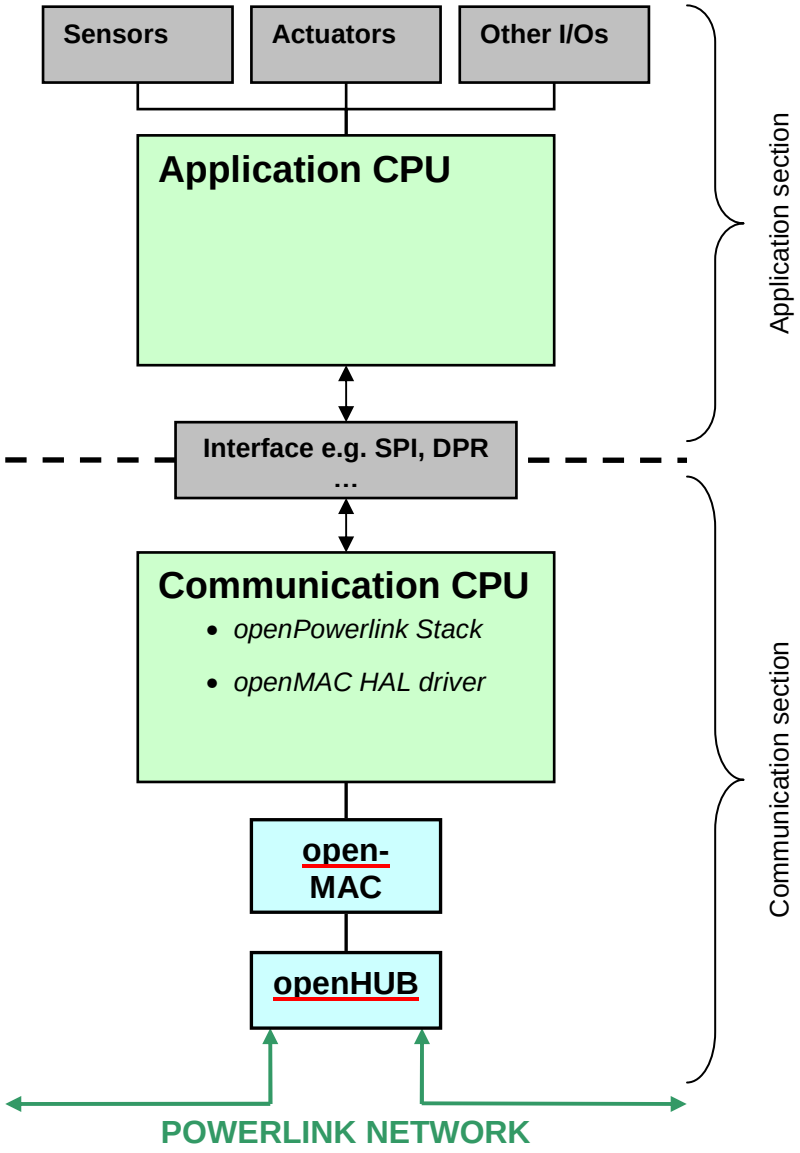


Fig. 14: generic design approach

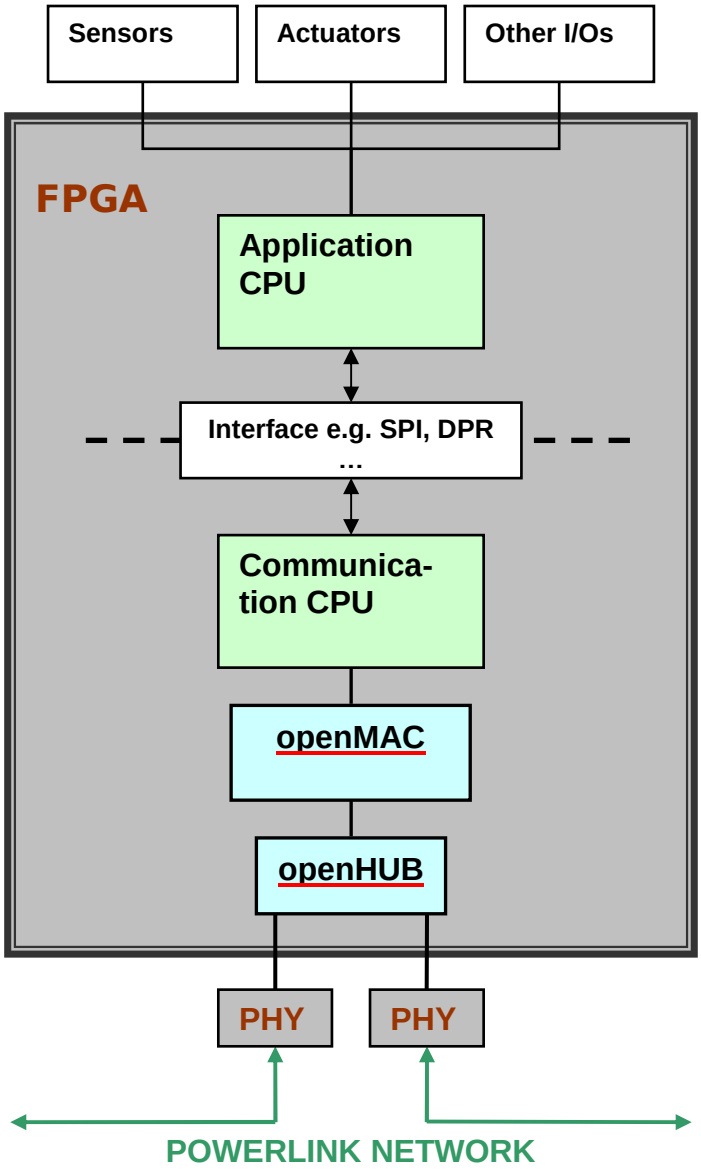


Fig. 15: FPGA design with two soft-core processors

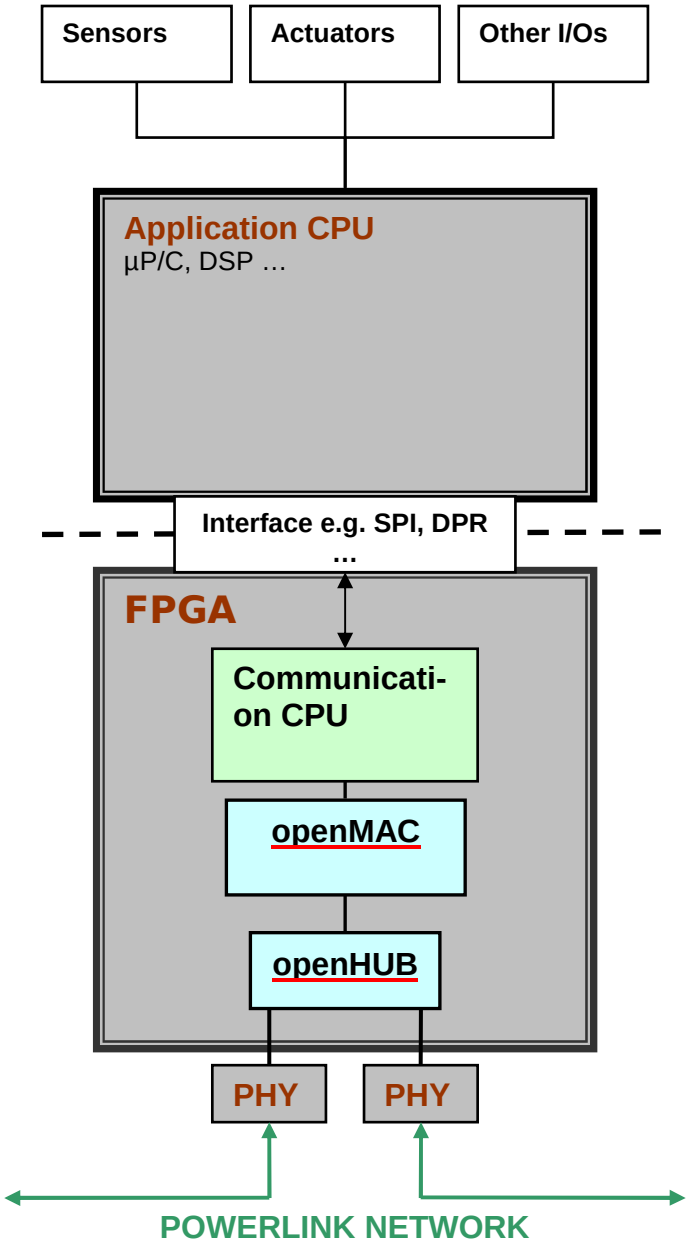


Fig. 16: FPGA design for communication tasks only

5 Figure Index

6 Table Index

7 Index