Perfection in Automation
www.br-automation.com

openMAC Driver (omethlib)
Documentation

Date: December 23, 2013

I Versions

Version	Date	Comment
0.1	2013-12-23	Creation

Table 1: Versions

II License

III Table of Contents

1 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. 1: 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 2 - Application note).

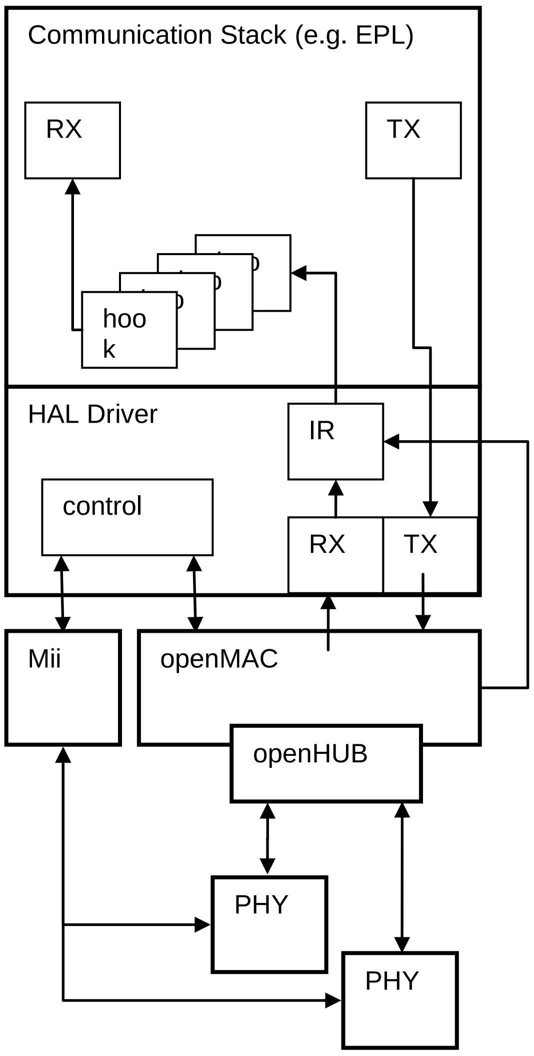

Fig. 1: block diagram, example for HAL usage

1.1 Features

1.1.1 Time stamp

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

1.1.2 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. 1: 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.

1.1.3 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, …)

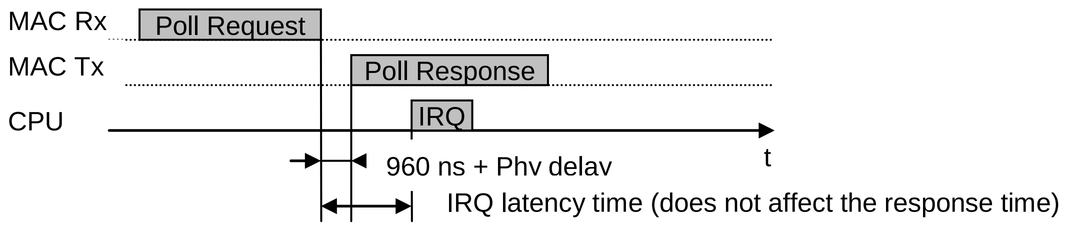

Fig. 2: auto response

1.2 Ethernet driver

1.2.1 Files

Tab. 2: 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.

1.2.2 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. 3: 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.

1.2.3 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.

1.2.3.1 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.

1.2.3.2 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)

1.2.3.3 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. 4.

Tab. 4: 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.

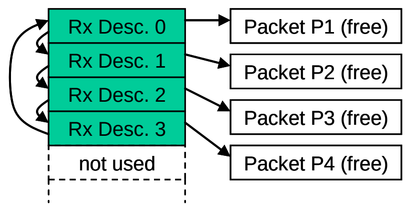

Fig. 3: receive queue after omethCreate with rxBuffers=4

1.2.3.4 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!)

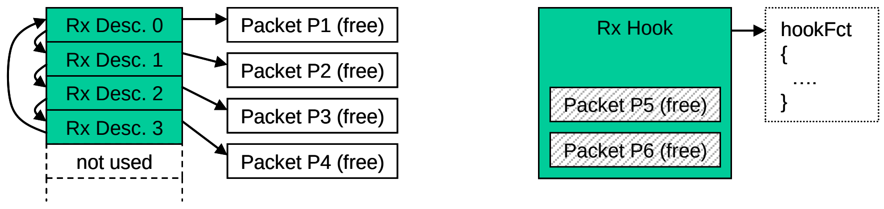

Fig. 4: buffers after omethHookCreate(…, &hookFct, 2)

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

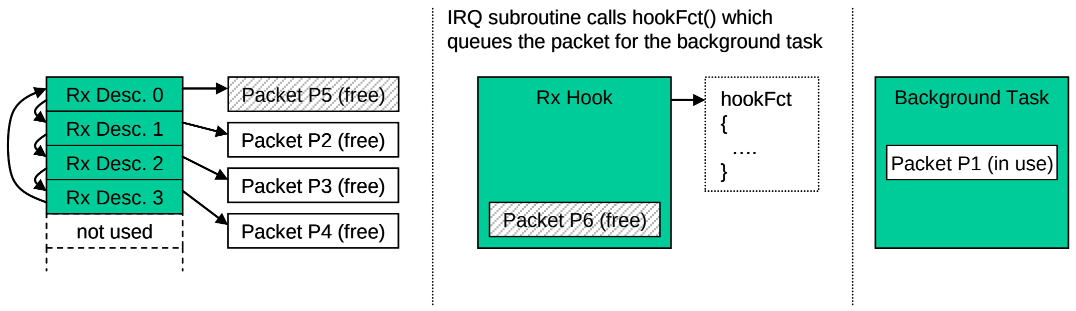

Fig. 5: buffers after a packet was queued by the background task

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).

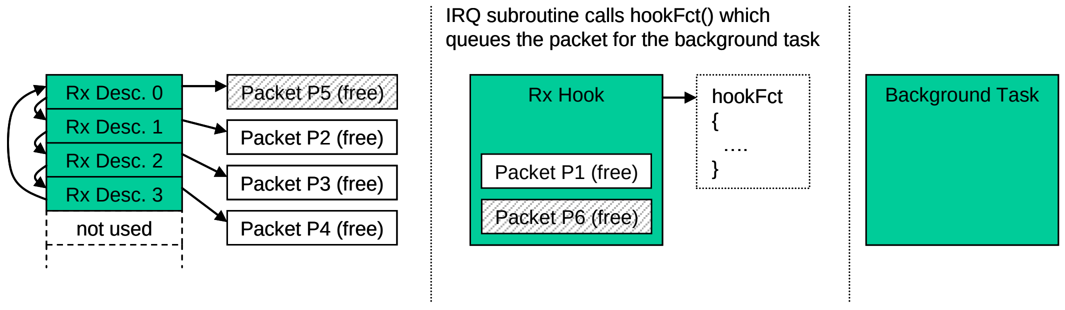

Fig. 6: buffers after a background task has processed packet

1.2.3.5 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.

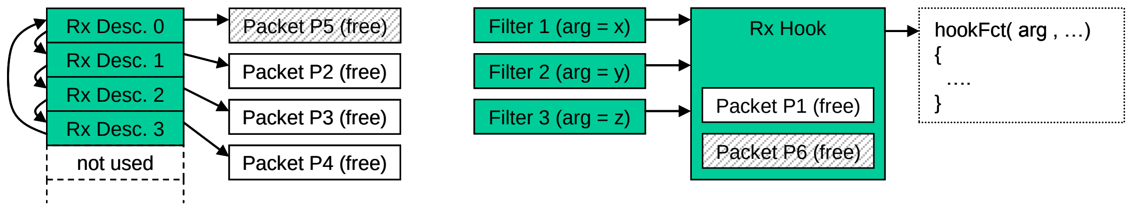

Fig. 7: filters and hooks

1.2.3.6 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.

1.2.3.7 omethStart

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

1.2.3.8 omethStop

Stop Ethernet driver.

1.2.3.9 omethPeriodic

The function must be called by the user cyclically.
It does the update of the PHY-Register image (see 1.2.6.2) 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

1.2.3.10 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

1.2.4 Filter manipulation

1.2.4.1 omethFilterSetPattern

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

1.2.4.2 omethFilterSetArgument

Change the hook argument for an filter created with omethFilterCreate.

1.2.5 Auto response functionality

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

1.2.5.1 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.

1.2.5.2 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!

1.2.5.3 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.

1.2.5.4 omethResponseEnable

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

1.2.6 Other API functions

1.2.6.1 omethGetHandler

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

1.2.6.2 omethPhyInfo

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

1.2.6.3 omethHookSetFunction

Change the call back function of a receive hook.

1.2.6.4 omethGetTimestamp

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

1.2.6.5 omethGetTxBufBase

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

1.2.6.6 omethGetRxBufBase

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

1.2.6.7 omethStatistics

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

1.2.6.8 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.

1.2.7 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. 5: 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;
```

2 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. 8 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.

2.1 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. 9 – the “Communication” and “Application” CPU can be e.g. Nios II (Altera), Microblaze (Xilinx) or any other soft-core processor.

2.2 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. 10.

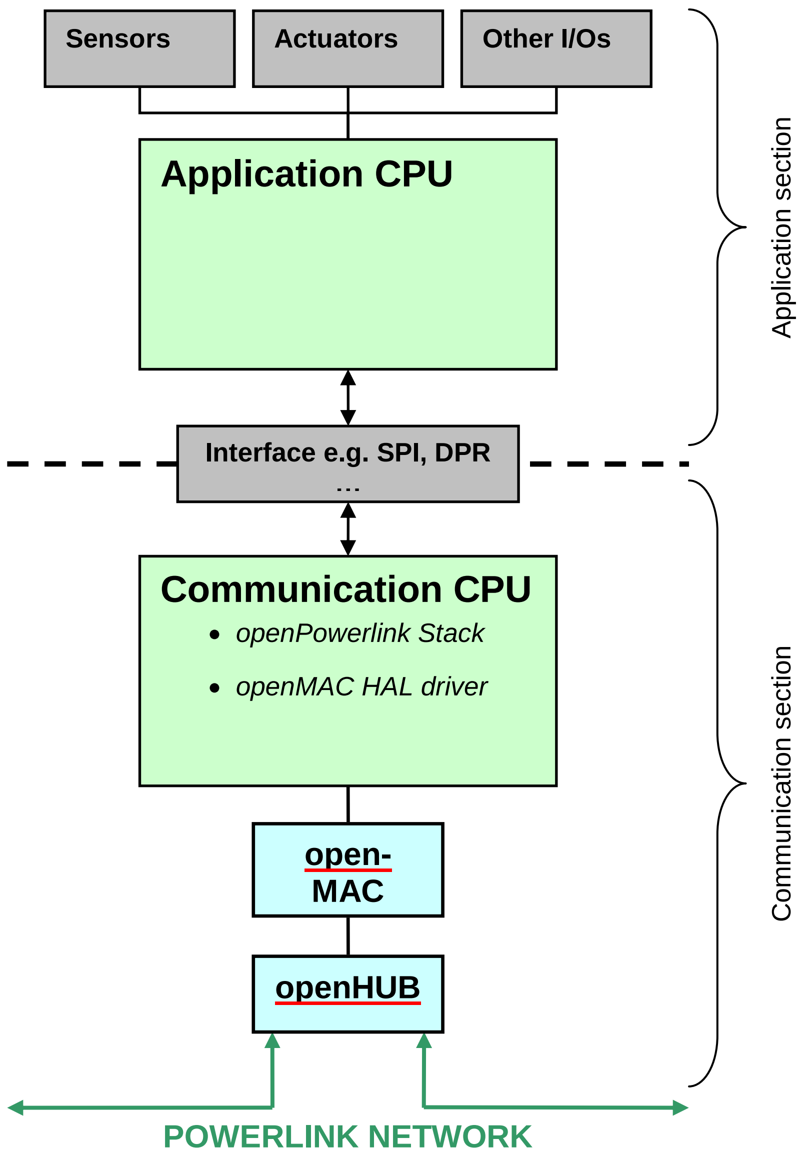

Fig. 8: generic design approach

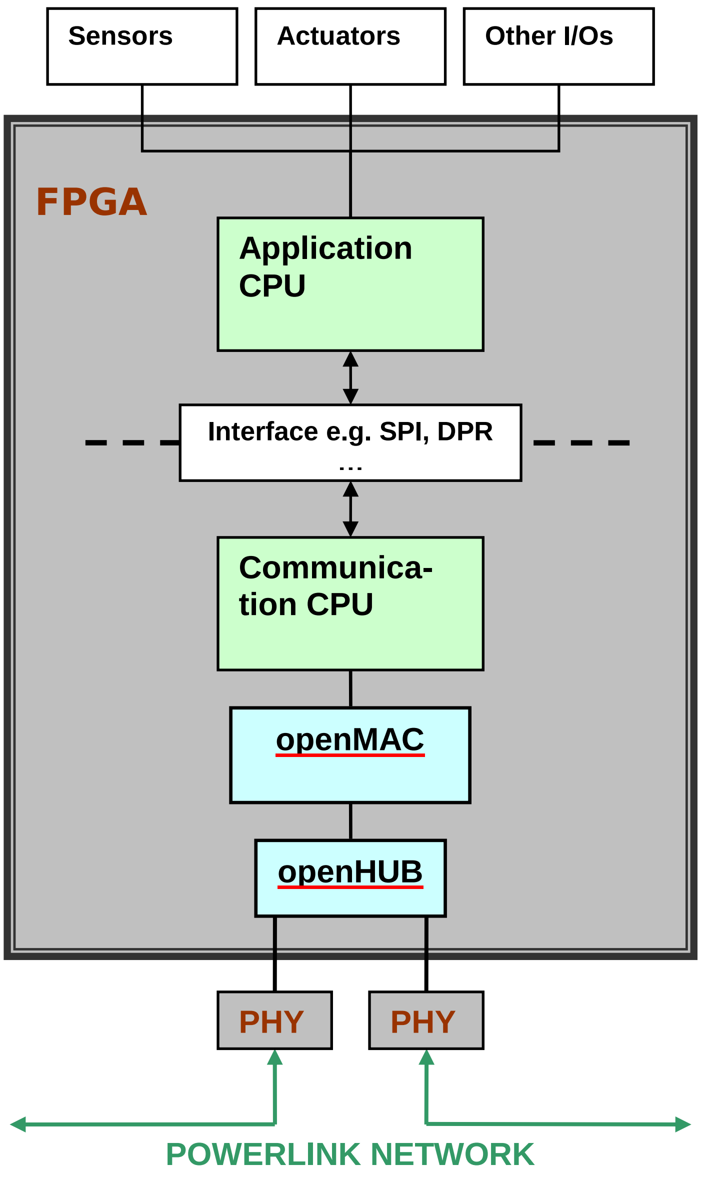

Fig. 9: FPGA design with two soft-core processors

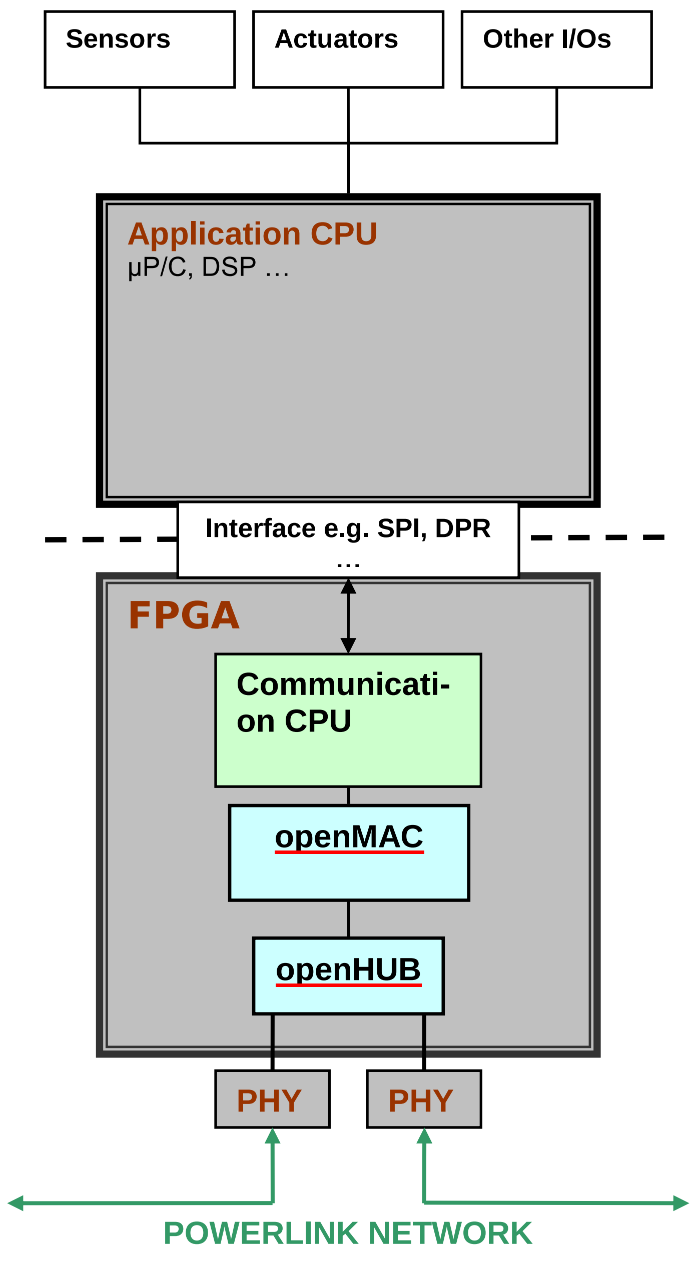

Fig. 10: FPGA design for communication tasks only

3 Figure Index

4 Table Index

5 Index