LeWiz Ethernet MAC
Device Driver Specification

LeWiz Communications, Inc.

"The Wizard of Internet Communications"

March 1, 2019
Revision 1.00

LeWiz can be contacted at: support@lewiz.com
or address: PO Box 9276, San Jose, CA 95157-9276
www.lewiz.com
Author: LeWiz Communications, Inc.

Change Log

Version	Significant Changes
1.00	Release version

Table of Contents

NOTE: This document is intended for SW/HW users using LeWiz MAC IP Core.

1 Introduction

Computer systems have many peripherals attached such as a keyboard, mouse, RS232 port, USB port, printer, video cards, network interface cards (NIC) and others. The user uses these peripherals to communicate with the computer to perform a job. The computer uses device driver to configure and control each peripheral attached to it. The peripheral may require reading or writing to the system memory or access other devices in the system. The device driver allocates system resources for the peripheral to use in accessing memory and other devices in the system.

This document describes an implementation of a device driver for LeWiz's Ethernet MAC controller – LMAC Core1 and LMAC Core2. The LMAC controller makes use of peripheral DMA channels to read packets from system memory for transmitting out to the network. It also uses the DMA channels to write packets received from the network to system memory. For the DMA to read a packet from memory, the software device driver instructs the DMA where to read and how many bytes to read. Likewise, for writing packet to memory, the device driver allocates space in the memory and instructs the DMA where to send received packets to. DMA uses interrupt to notify the system of packet transmitted or received. The device driver also provides this interrupt capability to the LMAC peripheral's DMA channels.

To test the device driver software without actual hardware, LeWiz uses the co-simulation environment developed by Xilinx and worked with Xilinx to integrate LeWiz's MAC controller to the co-sim environment and tested the LMAC device driver and LeWiz's hardware in the emulated Zynq Ultrascale+ FPGA environment. The following discusses both the co-sim environment used in the testing and also specifies the device driver APIs and functionalities.

2 Overview of the co-simulation model

This co-simulation project can be divided in two parts mainly Hardware and Software. To understand the LMAC core, the reader is encouraged to refer the LeWiz LMAC documents on GitHub. (You can search for the LMAC cores on GitHub using "LMAC CORE2", for example). This section gives an overview of the software and a brief description of how the entire co-simulation model works, the flow of packets in each direction (i.e. transmitting path and receiving path) and how the model communicates with the Xilinx emulator.

LeWiz Ethernet MAC transmits packets and receives packets from the network. These packets have a particular format using the Ethernet and TCP/IP protocol standards. In order to transmit, the TCP/IP software stack and the device driver create the packet. The software then informs the LMAC DMA of the available packet for it to fetch the packet and provides to the LMAC controller for transmitting out to the network. After the packet has been transmitted the DMA generates an interrupt to notify the system that the packet has been transmitted. The LMAC controller, if enabled, receives valid packets from the network. The LMAC DMA then sends the received packet into system memory at the space previously allocated by the device driver. After sending in the packet, it causes an interrupt to inform the system that a packet has been received. (The processing of the packets in memory is done by the system's TCP/IP stack and is beyond the scope of this document.) The co-sim environment models these flows both in hardware and software. In testing, LeWiz demonstrated that the co-sim environment successfully transfer data across the Internet.

Figure 1 below shows a block diagram of the co-simulation model.

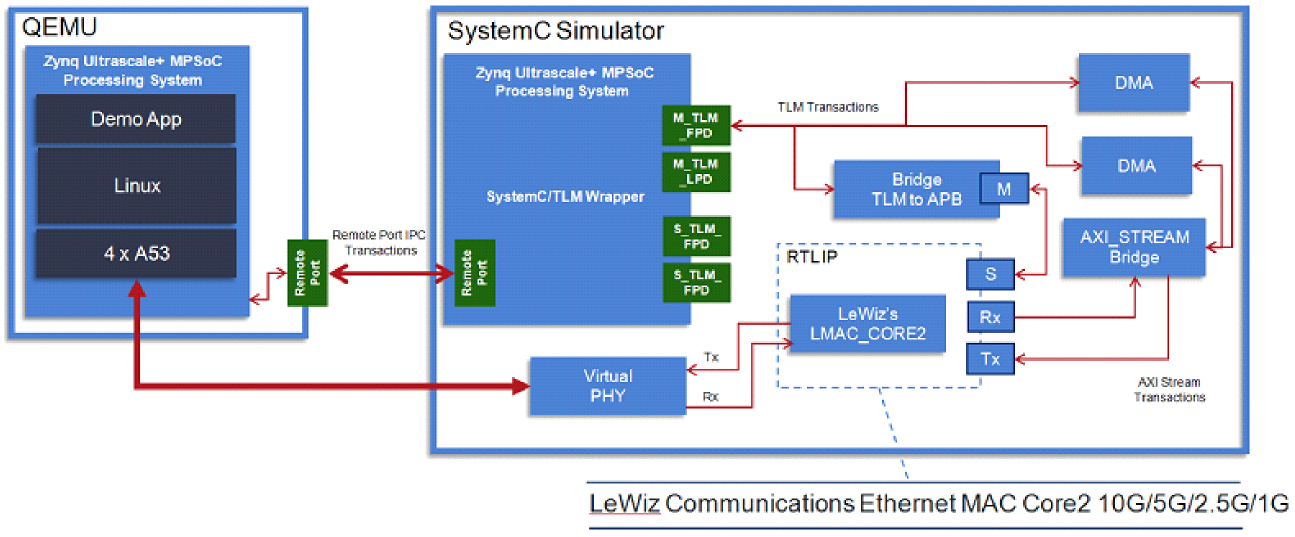

Figure 1. Block diagram of co-simulation model

The model is divided in two parts:

- PL (Programmable Logic on the right side of Figure 1)
- PS (Processor system on the left side of Figure 1)

Programmable logic implements the LMAC core, associated DMA channels and PHY. This was implemented in Verilog and converted to fit into SystemC simulator environment. This part emulates the hardware of the system under test.

Processor system (on the right) implements Zynq Ultrascale+ processing system. The emulator for this part uses QEMU simulator. It models the ARM CPU cores, Linux kernel (runs on top of the ARM CPUs) and associated Zynq PS peripherals.

The PL models are written using Verilog and SystemC TLM 2.0. The LMAC core coded in Verilog. The aim of this part of the project is to verify the functionality of the design on an emulated FPGA environment. The entire model is developed in SystemC simulator. Modules which are available in Verilog were converted from Verilog to SystemC using Verilator tool. Verilator as used in this case is merely a translator rather than a simulator. It converts Verilog code and logic into equivalent C++/SystemC code. These SystemC modules are connected with each other using a SystemC TLM 2.0 (Transaction Level Modeling API).

The block diagram illustrates two major modules: the SystemC simulator and QEMU (emulator). The SystemC simulator comprises of Processor System (PS) wrapper and Programmable Logic (PL). The QEMU is entirely a processor system. QEMU emulates and creates an environment which replicates an actual Zynq Ultrascale+ system.

The SystemC simulator side also has the virtual PHY. This emulates the PHY layer of an Ethernet MAC. In the co-sim model, this PHY is virtually connected to the actual Ethernet port of the Linux host system which runs the entire co-sim environment. The Linux host system is an Intel CPU PC server runs Ubuntu Linux OS. The RTL-IP portions in the block diagram was written in Verilog and converted into equivalent C++/SystemC using Verilator. These modules communicate with the memory via DMA (Direct Memory Access) controlled by the device driver. The interconnection of the modules is done using SystemC TLM 2.0 as shown in Figure1. In the block diagram, modules “AXI_STREAM Bridge”, “Bridge TLM to APB” and also there is a bridge between the virtual PHY and LMAC core called “TLM to XGMII” and “XGMII to TLM”, these were developed in SystemC TLM 2.0.

3 Transmission of packets from LMAC Core

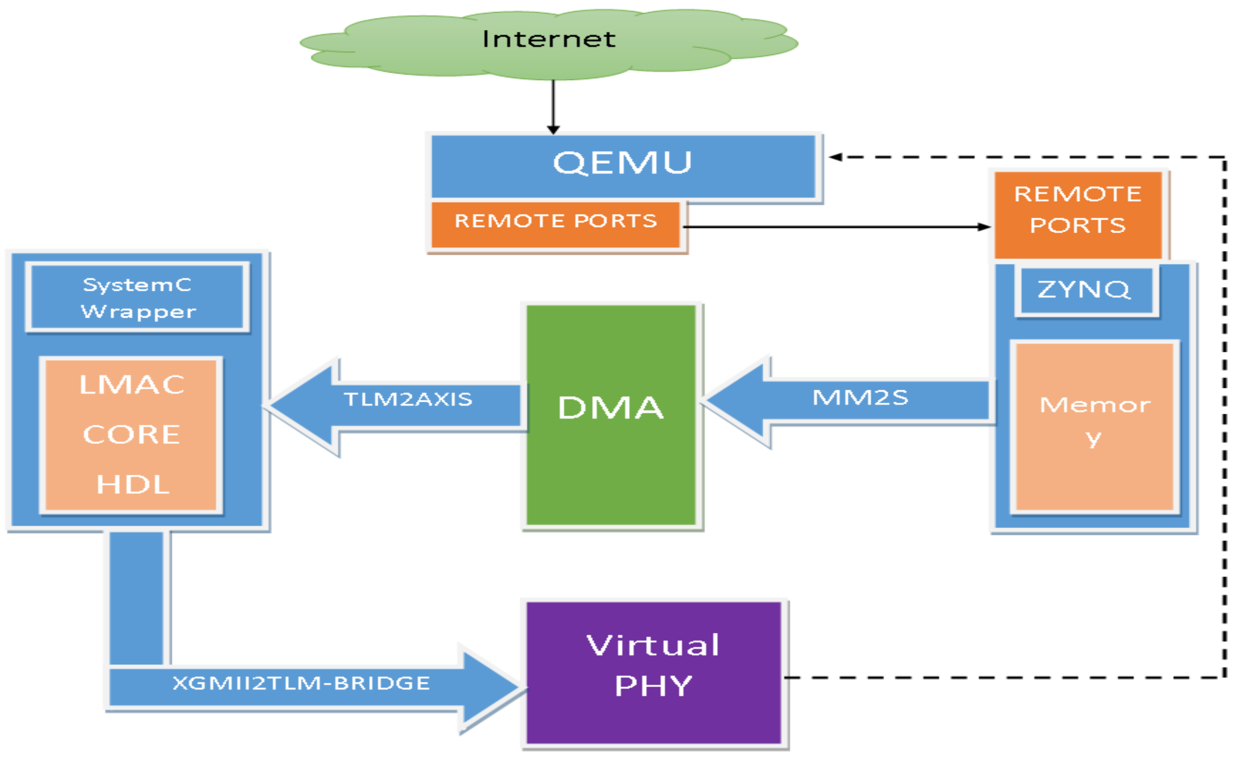

Figure 2. Transmitting path.

The above figure shows the flow of packets when the LMAC is in transmit mode.

The system software on the QEMU side generates the packets and stores them inside the memory. The LMAC DMA driver configures the MM2S DMA (Main memory to stream). The DMA reads packets from the memory and forwards the packets to the LMAC Core for processing. The LMAC appends preamble pattern (D5555555555555FB) at the start of the packet. It also adds error detecting CRC code at the end of the Ethernet frame. The Ethernet frame is then transmitted to the virtual PHY via the XGMII2TLM-BRIDGE. The virtual PHY then sends the data to the network. (In the demo case, the packets are sent to the Linux host Ethernet port [emulating a real Ethernet network connection] which forward to the Internet.)

The flow of operation for transmitting a packet is as follows:

- QEMU side generates packets.
- Packets are pumped into the memory via remote port.
- The driver configures the DMA (in this case MM2S DMA)
- DMA reads packets from the memory.
- Pump the packets into LMAC.

- The DMA generates an interrupt when the transfer completes
- The LMAC adds preamble and CRC code to the packet to form the Ethernet frame.
- Virtual PHY forwards packets to the network

4 Reception of packets by LMAC Core

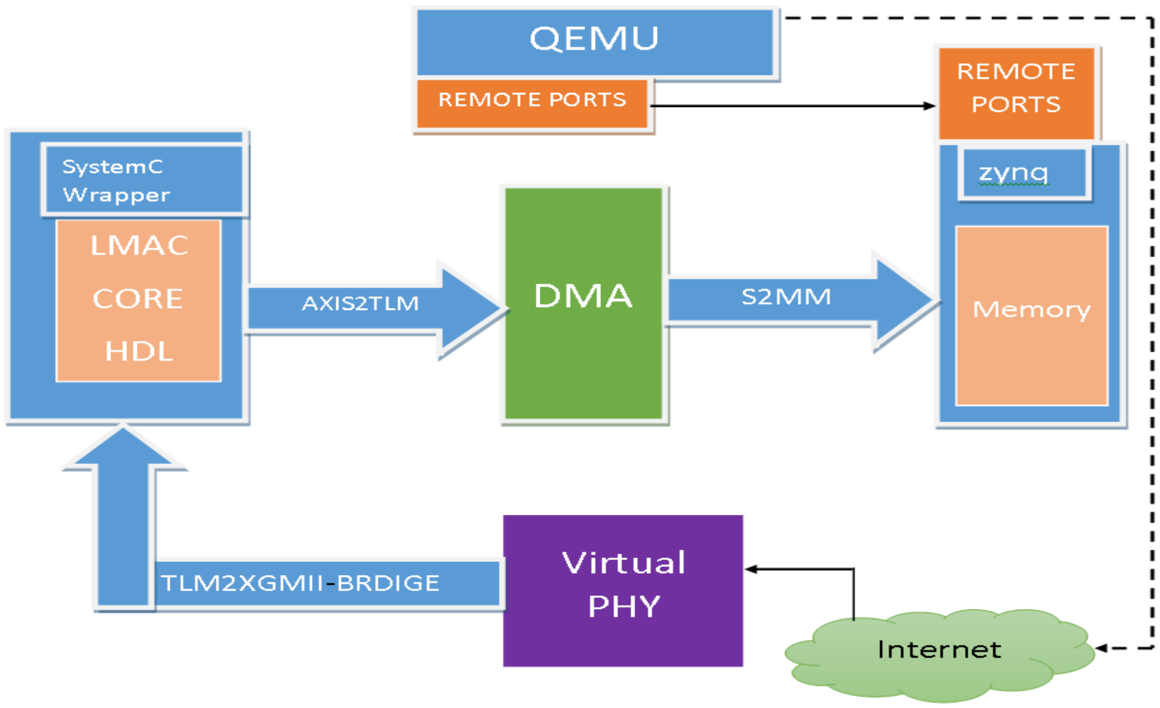

Figure 3. Receiving path.

Above figure shows the flow of packet when the LMAC is receiving.

During receiving operation, Ethernet frames come from the network. (For the co-sim demo, real data packets originate from the Internet and forwarded to the virtual PHY via the QEMU emulator.) The packets are accepted by the virtual PHY. (In the co-sim demo, the virtual PHY adds the preamble and CRC code to the packet from the Internet to form the Ethernet frame.) The Ethernet frames are then sent to the LMAC for processing. The LMAC checks the frames for validity. It also removes the preamble, end of frame bits and checks CRC for any errors. After processing, the remain is a data packet. The LMAC sends those packets to the S2MM DMA (stream to main memory DMA). (Before the packet is received, the S2MM DMA driver pre-configures the DMA with allocated memory address for packet receiving.) The packets are transferred into the memory by the DMA. The DMA generates an interrupt when the transfer of the packet is completed. The system recognizes the interrupt and causes the interrupt handler to process the interrupt. It activates the LMAC driver to receive packets from the memory. The packets are then handed to the TCP/IP software stack for the user application.

The flow of operation for receiving a packet is as follows:

- The virtual PHY receives Ethernet frames from the QEMU.
- The Virtual PHY forwards the frames to the LMAC for processing and extracting the data packets.
- Packets are pumped into the memory by the DMA (in this case S2MM DMA). It’s important to note that the S2MM DMA is preconfigured by the device driver for receiving packets. This way the

hardware can send data to the memory using the DMA without waiting for the driver to configure the DMA.

- After a packet is transferred to memory, the DMA generates an interrupt for the system.
- The system/device driver detects the interrupt and reads the packet from the memory and pushes the packet to the TCP/IP socket's data buffer which is a part of the TCP stack for sending packets to the user application.

5 Code structure of LMAC Device Driver

This section explains the LMAC device driver code. This device driver code is intended to provide a starting point for the LMAC core users. Users should modify it to fit user's system design. The code consists a #define section (for constants and renaming), the data structures and sub-routines. The following specifies each section of the code.

5.1 #Defines

The 'defines' contains information about the registers of the DMA. The device driver code uses the defines to configure the DMA to perform transmit or receiving functions as explained above. All the registers are assigned memory-mapped addresses. Thus, they are accessible as memory locations. The user reads or writes a particular memory address for accessing the specific register. Each bit in the register is also assigned an address offset that the user can use to set or clear that bit.

The word "offset" below means address offset.
The word "stream" means the AXI stream interface.
Each register can be 32 bits.
Also refer to the code lmac2.c for actual assigned values
RX = receive; TX = transmit

• `DMA_M2S_OFFSET:`	Offset to access the main-memory-to-stream DMA For main memory to stream transfers.
• `DMA_S2M_OFFSET:`	Offset to access the stream-to-main-memory DMA For stream to main memory transfers.
• `DMA_CR_RS(00):`	Run/Stop control for controlling the DMA channel. 0 = Stop. Stop or disable the DMA 1 = Run. Start DMA operations. ((00) is the bit location in the register)
• `DMA_CR_RESET(01):`	Soft reset for resetting the DMA core. 0 = Normal operation. 1 = Reset.
• `DMA_CR_KEYHOLE(02):`	Reserved.
• `DMA_CR_CYCLIC_BD(03):`	Reserved.
• `DMA_CR_IOC_IRQ_EN(12):`	Interrupt on Complete (IOC) Enable bit. 0 = IOC interrupt disabled. 1 = IOC interrupt enabled.
• `DMA_SR_HALTED(0x00):`	Reserved.
• `DMA_SR_IDLE(0x01):`	Reserved.
• `DMA_SR_SGINCLD(0x03):`	Reserved.
• `DMA_SR_IOC_IRQ(0x12):`	When set to 1, it generates an interrupt on completion of a transfer.

And if DMA_CR_IOC_IRQ_EN is set then the interrupt is generated.

• `DMA_R_CR:`	MM2S DMA Control Register.
• `DMA_R_SR:`	MM2S DMA Status Register.
• `DMA_R_ADDR:`	MM2S lower 32 bit source (or destination) address.
• `DMA_R_ADDR_MSB:`	MM2S upper 32 bit source (or destination) address.
• `DMA_R_LENGTH:`	Transfer length register (in bytes).

5.2 Data structures in LMAC Driver

Following are the data structures used in the LMAC driver. These data structures used built-in kernel data structures.

`struct net_local` This data structure defines an instance of network interface. It keeps track of a network device that is connected to the TCP/IP stack. It also tracks the base address of the net device. This structure allocates socket buffers for transmit and receive. It also tracks the physical address of the allocated buffers.
`struct net_device_ops lmac2_netdev_ops` This structure maps basic functions that will be called on operations like: open, close, start transmit, check for if timeout occurred while transmitting, polling
`struct platform_driver lmac2_of_driver` Used for registering the device driver in the system
`struct of_device_id lmac2_of_match` A match table

5.3 Functions (Sub-routines) in LMAC Driver

The device driver code is distributed among small sub-routines. Each performs a specific operation.

Following are the functions used in LMAC driver.

- `static inline void lmac2_ack_interrupt (struct net_local *lp, unsigned long offset):`

  This sub-routine acknowledges the interrupt whenever transmit or receive interrupts are generated. The sub-routine writes the `Interrupt-on-complete` field in the status register (DMA_SR_IOC_IRQ).

- `static void lmac2_tx_timeout (struct net_device *dev):`

  The sub-routine is called when the timer runs out on transmit. It resets the timer registers and start again.

- `static void lmac_prepare_rx (struct net_device *device):`

  This subroutine allocates RX space for a new packet to arrive and enables the DMA. It allocates a socket buffer of maximum packet size and maps the buffer to a physical DMA address which will store the packet.

- `static irqreturn_t lmac2_rx_interrupt (int irq, void *dev_id)`

  This sub-routine is called when an RX interrupt is generated. The sub-routine acknowledges the interrupt by reading the packet and preparing for the next receive packet by calling `lmac_prepare_rx()` function.

- `static irqreturn_t lmac2_tx_interrupt (struct sk_buff *skb, struct net_device *dev)`

  This sub-routine is called when a transmit interrupt is generated. The sub-routine acknowledges the interrupt, updates the TX statistic (packet count, byte count), and frees the TX buffer.

- `static int lmac2_start_xmit (struct sk_buff *skb, struct net_device *dev):`

  The sub-routine prepares the DMA to start the transmission of packets by writing valid information to the address registers and the length registers. (DMA_R_ADDR, DMA_R_ADDR_MSB, DMA_R_LENGTH).

- `static int lmac2_open(struct net_device *dev):`

  This sub-routine is called when the user calls the `open()` function to enable the network interface. (This is called when the Linux `ifconfig up` is excuted.) The sub-routine registers the interrupt handler for transmit and receive if there are any. The function allocates the socket buffer to receive packets. Enable the interrupt so that it can read transmit or receive interrupts and call respective handlers.

- `static int lmac2_close(struct net_device *dev):`

  This sub-routine is called when the software disables the network interface or when the system shutdowns. The function disables the DMA interrupts and stops the transmission.

- `static int lmac2_probe(struct platform_device *ofdev):`

  This function is called to confirm that the driver exists and the functionality is correct. It is used to detect and install actual devices.

- `static int lmac2_of_remove (struct platform_device *ofdev):`

  This sub-routine unregisters the device driver.

6 Steps to Compile and test the LMAC Driver

The LMAC device driver is a part of the Linux kernel software. When the kernel is built, it also compiles the device driver. LMAC device driver for Linux kernel can be found under the directory:

/linux/drivers/net/ethernet/lewiz

Here are quick instructions on how to build the Linux kernel.

1. Create a directory named “linux”.
2. Install the ARM64 toolchain
   - “apt-get install gcc-aarch64-linux-gnu”
3. Git clone the repo and switch to posh-lmac-branch.
   - “git clone https://github.com/edgarigl/linux.git”
   - “git checkout –t origin/posh-lmac
4. In the directory with the Image, qemu-run.sh, pl-run.sh scripts you'll find the .config and rootfs tarball. Copy the .config and untar the target-arm64.tgz archive into the linux directory.
   - To untar a file “tar -xzvf /path_of_where_the_tar_file_is_located”
5. Your Linux directory should now have the .config and targetfs.arm64 directory. Now you need to create a file called dev-files. The files should be provided to you by LeWiz. Copy dev-files into the linux directory.
6. Now build the kernel (for Zynq UltraScale+ , use -j8 if CPU has more cores)
   - “make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- -j4”
7. The new Image should now be located in arch/arm64/boot/Image
8. Prepare for demo (to test the driver).
9. Open one `putty` terminal and run `./qemu-run.sh` script
10. Open another `putty` terminal and run `./pl-run.sh` script.