LeWiz LMAC1 Ethernet MAC
Device Driver Specification
As Implemented on the FPGA Design Example

LeWiz Communications, Inc.

"The Wizard of Internet Communications"

November 16, 2020
Revision 1.1

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.

1. Introduction

In the FPGA design example for LMAC1 core, we used the LMAC1 Ethernet core in an FPGA design on a Xilinx ZCU102 development board. The DMA mechanism was implemented. The resulting design of the LMAC1 is a network interface device in the system (LMAC1 NIC). Similar to a network interface device (e.g., NIC) on a server or desktop computer system. A Linux software device driver was developed to control the LMAC1 NIC. This document describes the software driver, APIs and notes the observations we found during the example design implementation.

The main purpose of the driver is to (i) configure the board hardware for the LMAC1 NIC, (ii) register the driver in the Linux kernel and initialize essential software resources, (iii) control the transmit (TX) and receive (RX) of packets from the Ethernet network. Linux has an API used to interact with network interface drivers. This API is used to register the driver structure including pointers to functions that the Linux kernel will call when it needs some operation done by the specific driver. This driver has the ability to be built into the kernel or to be built as a kernel module that can be loaded at runtime.

This spec assumes the reader has some knowledge of writing C code and driver software. Refer to the file `lmac1.c` regarding the code described in this spec.

2. Initialization

There are several layers of initialization.

2.1 Init Probe

Probing is always done by the kernel when:

1. The kernel discovers new hardware.

2. A new kernel module is loaded into the kernel.

Since the LMAC1 driver is non-PCI and it targets an ARM64 Linux, it uses Flat Device Tree's or Device Tree's to describe the hardware node.

At the bottom of the `lmac1.c` file (the main C code of the LMAC1 driver), you'll see the following:

/* Match table for OF platform binding */

```
static const struct of_device_id lmac1_of_match[] = {
	{ .compatible = "lewiz,lmac1", },
	{ /* end of list */ },
};

MODULE_DEVICE_TABLE(of, lmac1_of_match);

static struct platform_driver lmac1_of_driver = {
	.driver = {
	.name = DRIVER_NAME,
	.of_match_table = lmac1_of_match,
	},

	.probe = lmac1_of_probe,
	.remove = lmac1_of_remove,
};
```

```
module_platform_driver(lmac1_of_driver);
```

The `module_platform_driver()` macro expands to code that registers the lmac1_of_driver structure with the kernel. The function contains a list of compatible match entries and two function callbacks for probing and removal of the driver.

The list of compatible strings is used to tell the kernel to filter out any device-tree nodes that do not match our list. Here's a portion of the actual device-tree node of the LMAC1:

```
lmac1@80000000 {
	compatible = "lewiz,lmac1";
	reg = <0x0 0x80000000 0x0 0x1000>;
	mac-address = [00 12 32 ff ff ff];
};
```

As we can see, the compatible strings match. So when the kernel sees this lmac1 node and our lmac1 driver is loaded, it will go ahead and call our registers `probe()` function.

The lmac1 `probe()` function does a series of `platform_get_x` and `devm_x` calls and further probing and initialization. Note that this is only called once during the driver's life time. Either once when the system boots up or once when the driver module is loaded. If you remove and re-load the driver module it is called once per loading.

The probe function will get the base address and interrupt numbers for this particular instance of the LMAC1 (note that, there could be multiple of them on the same system). It then goes on and reads out the clocks.

2.2 Clock, Reset Initialization

The clocks on the ZCU102 board are: `pl-clk0, pl-clk1, si570-user-clk` and the `sfp-clk`. For each clock, we do:

1. `devm_clk_get()` to get a reference to the clock.

2. `clk_prepare_enable()` to tell the kernel to prep the clock because we're about to use it.

3. `clk_set_rate()` to set the frequency of the clock.

The `pl-clk0` and `pl-clk1` are mandatory. The `si570-user-clk` and `sfp-clk` are not mandatory. If they are not mentioned in the device-tree, the driver will simply not configure them and assume these were configured somewhere else in the system (i.e., by the boot-loaders).

Another interesting part here is the use of `reset-gpios` to specify a GPIO that we'll toggle to reset the PL. Note that this is done prior to accessing any of the MAC registers.

There's a call to `of_get_named_gpio_flags()` to get the "`reset-gpios`" and further to `devm_gpio_request_one()` to get the "`pl`" reset the GPIO.

This part is intrusive (it resets the entire programmable logic or PL) so it's optional as well. If you specify the "`reset-gpios`" property in the device-tree, the module will look for the "`pl`" reset button and push it.

Once probe is successfully done, it calls `register_netdev()` to register our network interface specific driver callbacks. At this point, you should be able to see the interface on your system with `ifconfig` or `ip link show`.

2.3 Init Open

When a privileged user on the system does an `ifconfig up` or `ip link set up` on the LMAC1 Ethernet interface, the kernel will call our `lmac1_open()` callback.

At this time, we need to prepare the interface to be able to receive and transmit packets.

In `lmac1_open()`, we:

1. Reset the MAC.

2. Register a callback for interrupt reception via `request_irq()`.

3. Allocate and initialize the DMA descriptor rings. (See DMA Ring section below)

4. Enable interrupts.

5. And call `netif_start_queue()` to tell the kernel we're ready to TX packets.

Similarly, when a user does `ifconfig down` or `ip link set down` on the LMAC1 interface, the kernel will call our `lmac1_close()` callback.

Here we disable interrupts and stop the TX queues.

3. Data Path Functions for TX and RX

3.1 TX - lmac1 start_xmit()

Whenever the kernels protocol stack has generated a packet that it needs to transmit on our Ethernet interface, it will call `lmac1_start_xmit()` callback.

`lmac1_start_xmit()` callback is responsible for mapping the `skb` (socket buffer) onto the DMA ring buffers. Once the packet has been transmitted by the LMAC DMA, we must call `dev_kfree_skb_irq()` or `dev_consume_skb_irq()` depending on the Linux kernel version we're running on.

Since the driver uses interrupts, the transmit sequence is split into two parts.

1. Setup the packet for transmission, `lmac1_start_xmit()`.

2. Complete packet transmission, `lmac1_tx_interrupt()`.

In `lmac1_start_xmit()`, we get the skb and need to prepare it for transmission via the DMA rings. Since the DMA of our Ethernet MAC does not use virtual addresses but physical addresses, we must map our `skb` buffers into physical addresses. This is done by calling `dma_map_single()`.

`dma_map_single()` returns a `dma_addr_t`, the physical equivalence of `skb-data`.

So, for clarification:

`skb->data` - Virtual address for CPU accesses.
`lp->tx.descr[pos].addr` – Physical address for DMA access.

Furthermore, `dma_map_single()` will map and flush any caches that may need to be flushed to avoid incoherence between the CPU caches and main memory. After `dma_map_single()`.

Once we've got the buffers mapped, we can setup the reset of the descriptor. This is what it looks like (with error handling code removed for clarity):

```
lp->tx.descr[pos].addr = dma_map_single(&lp->pdev->dev,
skb->data, skb->len,DMA_TO_DEVICE);

lp->tx.descr[pos].control = len | CTRL_SOF | CTRL_EOF;

lp->tx.descr[pos].status = 0;
```

Once the descriptor is setup, we call `lmac_tail()`. `lmac_tail()` is responsible for updating the DMA's tail pointer, effectively telling the DMA that we've added a new buffer and that it should run.

The variable `lp->tx.in_flight` is incremented and decremented for every packet transmitted and packet transmit completion, respectively. It's useful to keep track of while debugging.

If out TX DMA ring-buffer get full when inserting this packet, we call `netif_stop_queue()`. This tells the kernel that we're out of resources and that it should back-pressure the software stack until we call `netif_wake_queue()` indicating we've regained resources to transmit. This is the Linux Kernels software-based back-pressure or flow-control mechanism.

3.2 TX - lmac1_tx_interrupt()

Once the DMA is ready with transmitting the packets, it will signal completion via an interrupt. When the kernel receives this interrupt it'll call the `lmac1_tx_interrupt()` callback function.

At this point we need to complete any number of packets that are done in the DMA ring-buffers. So, we traverse the ring-buffer and call:

1. `dma_unmap_single()` to unmap the Virt to Phys mapping.

2. `dev_kfree_skb_irq()` or `dev_consume_skb_irq()` to free the skb (or socket buffer).

3. `netif_trans_update()` indicate to the kernel that TX progress has been made.

4. `netif_wake_queue()` is used to tell the kernel that we've regained TX resources.

A few things to note here. In order to not miss interrupt events, we must follow a sequence in a specific order.

1. Read out the interrupt status register of the DMA.

2. ACK the interrupts we've just read out.

3. Process the interrupts we read out in step 1.

So the flow of the code does:

```
tx_sr = readl(lp->base_addr + DMA_M2S_OFFSET + DMA_R_SR);
lmac1_ack_interrupt(lp, DMA_M2S_OFFSET, tx_sr);

if (tx_sr & DMA_SR_IOC_IRQ) {
     /* Do packet completion. */
}
```

3.3 RX - lmac1_rx_interrupt()

At `lmac1_open`, when we initialize the DMA ring-buffers, the TX buffers are basically organized into a linked list of empty buffers. For RX, it's a little different.

We organize the ring-buffer as a linked list but we also pre-allocate an `skb` for each descriptor. This `skb` is allocated with MAX_FRAME_SIZE since we don't know the size of the packet we'll receive until after the fact.

Since the addresses we'll be using in the DMA descriptors need to be physical, we also need to call `dma_map_single`. This is all handled in `lmac_alloc_skb()` and `lmac_alloc_dma_rings()`.

So, the RX path essentially, pre-allocates `skbs`, maps them for direct DMA, pre-initializes the DMA ring-buffers and enables DMA reception and RX interrupts.

Then we wait.

When the kernel receives an RX interrupt, it'll call our `lmac1_rx_interrupt()` callback. At this stage we've got packets in our DMA ring-buffer and we need to extract them and send them up the software stack. So `lmac1_rx_interrupt()` walks the ring-buffer and for any packets that are complete, it calls `lmac_rx_process()` that injects the `skb` into the software stack.

`lmac_rx_process()` does:

1. `dma_unmap_single()`, this is done to synchronize caches.

2. `skb_put(skb, len)`, to update the length of the skb from MAX_FRAME_SIZE to the actual length received from the MAC.

3. `netif_rx()`, to inject the `skb` into the Linux protocol stack.

4. `lmac_alloc_skb()`, to pre-allocate a new `skb` and prepare it for reception.

4. DMA Rings

The DMA rings are a circular linked list of DMA descriptors. These are allocated in `lmac_alloc_dma_rings()` by calling `dma_alloc_coherent()`. A single large block is allocated and then we loop around the array of descriptors updating the next pointers to create a circular list.

For TX it's pretty straight forward. For RX, we pre-allocate `skbs` and prepare each descriptor with a buffer so that the MAC and DMA can received straight into the `skb`.

Whenever a new buffer is updated, one needs to call `lmac_tx_tail()` or `lmac_rx_tail()` to inform the DMA that there's a new "`tail`" buffer.

There are also two helper functions to start the DMA in each direction. This involves updating the CURDESC (current descriptor) register to the start of the DMA ring followed by an update of the tail to the last descriptor.

5. Memory barriers

Since the ARMv8 cores on the Xilinx ZU+ are superscalar and can do instruction reordering, we use the `wmb`() and `rmb`() memory barriers to make sure memory operations have been completed before we continue.

In the following specific example, the descriptor is updated, followed by a write memory barrier to make sure those updates hit memory before the `tail` descriptor gets updated:

```
lp->tx.descr[pos].control = len | CTRL_SOF | CTRL_EOF;

lp->tx.descr[pos].status = 0;

wmb();

lmac_tx_tail(lp, pos);
```

More details on this topic can be read here:
https://www.kernel.org/doc/Documentation/memorybarriers.txt