CS 194-24 Lab 4: Device Drivers

Palmer Dabbelt

April 18, 2014

Contents

For this assignment you will be writing a driver for the ETH194, an extended version of a PCI clone of the venerable NE2000 Ethernet card. You can find datasheets for both the ETH194 and the card that it's based (the RTL28029AS) on Piazza, or on the course website in the handouts section.

The ETH194 is a plug-and-play PCI device. If you are unfamiliar with PCI devices, then I suggest you read up on Linux's PCI documentation. Thanks to TLDP, a good overview can be found online:

`http://tldp.org/LDP/tlk/dd/pci.html`

1 ETH194 References

In addition to the data sheets, there is an existing Linux driver for the NE2000/RTL28029AS. These are included in your Linux sources distribution here:

`linux/drivers/net/ethernet/8390/ne2k-pci.{h,c}`

As usual, the Linux source code is a significantly better documentation resource then Realtek's datasheet. If you pull the latest source code, you'll find that there is also a QEMU implementation of the ETH194 Ethernet controller located here:

`qemu/hw/eth194.{h,c}`

This controller provides exactly the implementation of the ETH194 that your QEMU will be using for this project.

In order to add support for the ETH194 to Linux it's suggested that you read both of the provided data sheets, find out what the differences are (hint, they're mostly inside the DMA interface) and create a modified NE2000 driver that works with the ETH194. A significant amount of the NE2000 functionality lives inside `lib8390.c`, so you'll probably need to pull at least some of that functionality inside of your new driver.

2 ETH194+

The provided Ethernet controller is functional, but not particularly high performance. After you get a basic working driver for the ETH194 Ethernet controller we're going to have you extend the ETH194 with a new feature, creating the ETH194+. You'll produce modifications to the provided ETH194 datasheet that describes these changes, along with a QEMU implementation of the ETH194+ and a Linux driver that works with your changes.

If you look closely at the ETH194 you'll find that it works by chaining a linked list of frame buffers together. When it receives a packet it fills the current entry and then walks the list to the next buffer. Effectively all this driver does is take the serial stream in from Ethernet, split it up into frames, and alert the host when data is available.

Your ETH194+ design should sort Ethernet frames into separate transmit/receive queues by destination/source MAC addresses. Due to memory limitations, there will be a fixed number of queues. You will need to design a hashing scheme to multiplex a set of IP flows onto a particular queue.

This modification will require changes at the device level (i.e in QEMU) and at the driver level. In order to realize the performance advantage of this design, we would need a NUMA-capable system and an extensive set of changes to the driver's memory allocation scheme [1]. This is outside of the scope of this lab. Focus on making a clean kernel/hardware interface, don't worry too much about performance, and keep your driver simple.

[1] On NUMA-capable systems it's desirable to DMA frames into buffers which are *physically closer* to the CPU handling the corresponding IP flow.

3 Tasks

As usual, there a number of somewhat independent tasks you'll have to do for this assignment.

3.1 ETH194 Linux Driver (20 pts)

This is fairly simple to describe: you'll want a working ETH194 Linux device driver.

3.2 Test Harness (10 pts)

Testing your ETH194 Linux driver is going to be a bit different than testing your other code because you'll need to write Ethernet packets directly in order to expose many of the interesting edge cases in the device driver. In order to do this, you can start up two QEMU instances, using Linux's built-in packet generator to generate Ethernet packets to send to the machine that runs your ETH194 driver.

3.3 ETH194+ Datasheet (10 pts)

We're going to have you document your additions to the ETH194 Ethernet controller by modifying the provided datasheet. The datasheet you produce should be sufficient to fully implement all of your improvements.

This modified datasheet is designed to help your group members independently implement their changes to both QEMU and Linux. As this is really part of the design phase, a preliminary datasheet will be due along with your design document.

3.4 ETH194+ QEMU Driver (10 pts)

This section consists of a QEMU driver for your ETH194+ Ethernet controller. Ideally your ETH194+ datasheet will provide sufficient documentation to write this, but in reality it'll probably be tightly coupled with your ETH194+ Linux implementation.

3.5 ETH194+ Linux Driver (10 pts)

This section consists of a Linux driver for your ETH194+ Ethernet controller. Ideally your ETH194+ datasheet will provide sufficient documentation to write this, but in reality it'll probably be tightly coupled with your ETH194+ QEMU implementation.

4 Implementation Notes

These implementation notes cover material discussed in past sections, and common pitfalls encountered during office hours. It is by no means a complete set of notes. The expectation is that your design document will flesh out details that these notes gloss over.

1. For the ETH194+, you will need to add at least one register to your device. See the `ioport_read` and `ioport_write` functions in the `eth194.c` QEMU controller for a place to start.

2. Your device will need access to DMA-capable memory. You should use a SLAB allocator (see `kmem_cache_alloc`) along with the `__GFP_DMA` flag to allocate Ethernet frame buffers. You will also need to convert the virtual addresses your kernel handles to the physical addresses your device needs (see `virt_to_phys`).

3. Although `block_output` is not called in an interrupt context, it may be pre-empted at any point (due to SMP or normal scheduling behavior). If you are tracking changes to the state of your device in a 'shadow state' hosted by your driver, you must ensure that these updates are enclosed within a critical section.

4. Since this is an Ethernet device, you won't have to send buffers larger than the link MTU (hard-coded as 1514 bytes).

5. `block_input` *is* called in an interrupt context (see `__ei_interrupt` in `lib8390.c`), specifically, when a DMA receive operation has completed. Your safest option here is to use `spin_lock_irqsave`, update your 'shadow state', and defer the DMA buffer → `sk_buff->data` copy to a tasklet (which is scheduled later in some non-interrupt context).

6. It should never* be necessary to update the CURR register from your driver after the first time you initialize it [2]. The way to accomplish this is to inspect the device flags ('df') and host flags ('hf') of all frame buffers (starting from the head of a linked list) on each interrupt. These flags will indicate whether you must deallocate a buffer, update the 'nphy' field of a buffer, etc.

7. It is incorrect to update the device's CURW register if it is in the middle of transmitting a frame. This operation is not atomic, and may have the effect of pre-empting a pending device operation. You must maintain a bit of state (let's call it `is_txing`) that tracks whether the device is currently transmitting or not. You will need to handle the remote DMA complete ('RDC') and transfer complete ('TX') IRQs to implement this bit. You may also need to steal/mutilate code in `lib8390.c` to make this work.

8. You may update the CURW register if `!is_txing`. Otherwise, you must update the 'nphy' field of of the last buffer on the transmit queue to point to your new buffer.

5 Schedule

We'll be following the same schedule for this assignment as for previous assignments in this class: the project is 3 weeks long, with a design document due on Thursday, 9pm of the first week. Please note that a preliminary ETH194+ datasheet is due along with your design document.

We recommend completing the bulk of this lab before dead week begins. We'll hold extra office hours to help you complete the lab as soon as possible.

5.1 Design document

Here's what we're looking for in your design document:

1. A description of how the ETH194+ will sort ethernet frames.

2. A description of how the ETH194+ QEMU controller will differ from the existing controller.

3. A datasheet for your ETH194+.

5.2 Checkpoint 1

1. Design document

2. A preliminary implementation of an ETH194 driver in Linux. You'll need to:

[2] The only exception is when the device runs out of receive buffers, resulting in an overflow interrupt you must handle. This is an unlikely scenario.

(a) Add your driver to the source tree and make it compile.

(b) Boot QEMU using the ETH194 network card and verify that your driver is able to find it (hint, check dmesg).

(c) Run `pktgen` against your card and verify that the `block_input` method is called.

(d) Run `pktgen` on top of the ETH194 and verify that the `block_output` method is called.

5.3 Checkpoint 2

1. A complete ETH194 driver, and basic tests which cover all of its functionality
2. A preliminary implementation of an ETH194+ QEMU controller.
3. A preliminary implementation of an ETH194+ driver. See the Checkpoint 1 ETH194 driver requirements for what we expect.

5.4 Checkpoint 3

1. A complete ETH194+ driver, and basic tests which cover all of its functionality
2. A complete ETH194+ QEMU controller