CS 503 Spring 2014

Lab 4: A device Driver for an Intel PRO/100 Ethernet Adapter

Due: Sunday, 05/04/14, 11:59 PM

---

1. Overview

Now that you alredy have a baisc Xinu Operating System, no self respecting OS should go without a network stack. In this the lab you are going to write a driver for a network interface card and use the minimal network stack provided to test your driver. The card we are going to work with is based on the Intel 82545EM chip, also known as a member of E100 or PRO100 series.

Writing a driver requires in-depth knowledge of the hardware and the interface presented to the software. Understanding the hardware is a non-trivial task unless the manufacturer provides detailed manuals. Luckily for us, Intel has provided a very good set of manuals for the 82545EM, and you will have to get very well acquainted with them.

When you read about the hardware, it is important to know that chip vendors reuse parts across multiple designs. There are three conceptual pieces: the phy (physical layer), the mac (the media access controle, and the controller (bus and host interface). Thus, it is possible to find the a given controller chip used with either a copper wire or optical fiber phy layer. When you initialize the interface, there may be steps related to the the underlying mac or phy as well as generic steps that apply to any underlying layers.

For this lab assignment, we are very lucky to be using real hardware and have appropriate lab environment to gain invaluable hand-on experience with low level programming skills. However, how to debug the driver code is a big challenge you will encounter. Before devising a wise way to ease the developing process, you may need to rely on kprintf to get the value of corresponding register and figure out what is going on with the low level hardware.

Here is the instruction on how to set up your working environment:
You can download the clean version of source code here
You will implement your code in five files: include/e1000.h, device/eth/ethInit.c, device/eth/82545EMinit.c, device/eth/e1000Read.c and device/eth/e1000Write.c
To compile Xinu, ssh to xinuXX machine, under /compile dir, type:

```
$ make
```

This will generate a Xinu image, xinu.iso in your compile dir.
To run it, under Xinu /compile dir, type:

```
$ bash ./run-vxinu.sh -p [port#]
```

The port number will be assigned to you.

This will create a virtual environment for you and Xinu will boot up inside the enviroment automatically.

---

2. Readings

Chapter 14: Device Independent Input and Output
Chapter 16: DMA Devices And Drivers (Ethernet)
Intel 82545Fast Ethernet Controller Webpage
Intel 8255x 10/100 Mbps Ethernet Controller Family Open Source Software Developer Manual
82545EM Fast Ethernet PCI Controller Datasheet
A similar lab assignment in MIT based on a hardware emulator
You can also look up online for some driver code of e1000 for linux, they are similiar as the one we will implement for this lab.

Do not try to get all details in your first pass. It is more important to get a high level pictures of how the NIC chip is organized and what is needed to create a device driver.

When you read the open source developer manual in depth, glance over Section 4 to learn about the PCI interface. However, very close attention should be paid to Section 6 and Section 8 as Section 6 deals with the Software Interface and Secion 8 provides programming recommendations. In fact, most everything you need is in Section 6. Use the datasheet solely as a reference if you find the developer manual vague.

---

3. PCI Interface

The Intel PRO/100 S Ethernet Adapter is a PCI device, which means it plugs into a PCI bus on the motherboard. A PCI bus has address, data, and interrupt lines, and allows the CPU to communicate with devices and allows devices to read and write memory (i.e., perform DMA transfers without using the CPU). A PCI device needs to be discovered and initialized before it is ready to be used for I/O. Discovery is the process of searching the PCI bus and looking for attached devices. Initialization is the process of allocating I/O and memory space as well as negotiating the IRQ line for the device to use. By default PCI devices support "Plug and Play". When a PC is powered on, device independent software (usually the system BIOS) determines present devices, builds an address map, and assigns non-conflicting resources to those devices. The device independent software accomplishes the initialization task by writing to the PCI configuration space of each individual PCI device.

Each device on a PCI bus is uniquely identified by a pair of dwords: (Device ID, Vendor ID). The hardware we will be using has the Device ID 0x100F and the Vendor ID 0x8086.

Because we will be using a late-binding mechanism, the first step your driver needs to take involves identifing the device from PCI bus and obtaining the index of the device in the PCI map. After that, your driver can obtain information, such as I/O port base address, interrupt line number, and PCI device status, from the corresponding PCI configuration space. The driver can issue commands to control the device's ability to generate and respond to CPU commands. In this lab, you will need to enable PCI I/O access because Xinu operates in real memory mode and doesn't support memory-mapped access to the device.

The x86 version of Xinu we will give you for the lab already contains a set of system functions that your driver can use to communicate over a PCI bus and control devices on the bus. Some of the functions are implemented in assembly language because they require special hardware instructions and operate like

low-level system calls. You can find the list of functions in `pci.h` under the `system` subdirectory.

To achieve the above functionalities, you should write the following function:

*devcall ethInit (struct dentry *devptr)*

1. Initailize the ether tab.
2. Check if we can find the PCI device using (Device ID, Vendor ID). If we can't, return SYSERR.
3. If we can, then set up the device tab for theEthernet Adapter. i.e. set the function pointers of devtab to let dvread point to e1000Read.
4. Start up the ethernet device.

Each device on the PCI has a bus address known as an *I/O port base*. Be sure that you obtain a valid I/O port base for the device before continuing! Each I/O port number is 16 bits, which means the value ranges from 0 through 0xffff.

After obtaining the I/O port base address, you must gain the MAC address (also known as station address) from EEPROM. The MAC address is used as the source address in outgoing Ethernet packets, and used by the hwardware to recognize the destination address during packet reception. The Serial EEPROM, a serial input and output device, stores configuration data (such as the Ethernet MAC address) for the NIC. Driver software may read or write to the EEPROM by accessing the EEPROM port in the 82545EM. The next task for you is to read sections about the EEPROM in the developers manual and write the code to load the MAC address out of the EEPROM (see the 82545EM datasheet for the exact layout of the EEPROM).

4. I/O Port Access

The x86 version of Xinu contains a set of system call functions your driver will use to communicate with the device. The functions perform port-mapped I/O (i.e, transfer values over the bus to a specified bus address). The functions are named inX or outX, where X is a suffix that denotes the size of a transter: 'b' for a byte, 'w' for word, and 'l' for long (double-word). As the names imply inX functions input values from the device and outX functions send a value out to the device.

Please implement e1000_io_writel and e1000_io_readl in e1000.h using inX and outX.

For example, to reset the hardware, your driver must access the System Control Block (SCB) on the NIC. The dystem call appears in the code like this:
outl(ethptr->iobase + 0x08, 0x0000).

Consult the developers manual to make sure you match the correct outX or inX system calls to the width of the registers you intend to access. If you use the wrong width outX or inX system calls, the behavior of the hardware is undefined (it may appear to work, but you find incorrect behavior later).

Also, as mentioned in the lecture, there are a couple of places where the documents specify that each I/O access operation must delay for a certain amount of time before continuing. The driver cannot use the

sleep function. Instead, delay must be handled by a buys wait. A set of busy-wait delay functions can be found in system/i386.h.

Once we know how to use port-mapped I/O, you will be ready to write code that can control the hardware. The 82545EM is controlled through a set of Control/Status Registers (CSR), which are described in Section 6.3 of the developer's manual. The CSR is a 64 byte long data value that can be accessed by reading or writing individual pieces starting from the I/O base address that you found (see the above). You will be working exclusively with the first 24 bytes of the CSR for this lab. The first 8 of the 24 bytes correspond to the System Control Block (SCB). The SCB plays a major role in controlling the device.

5. Reset and initialize

It is common for devices to be reset to a consistent state before ready to use. For our case, even though Intel PRO/100 is internally fully reset on power up, it still needs to be fully reset after a warm reboot. Thus, it is recommended that a driver issue a Port Software Reset command before accessing the device.

After resetting the hardware to a known state, a sequence of initialization procedures must be applied to prepare the hardware. The specific PHY must be detected and initialized. An instantiation of the PHY connects a link layer device (often called a MAC) to a physical medium such as an optical fiber or copper cable. To ease the task, the PHY for the hardware has already be identified and your driver can skip this step.

Section 6.2 and Section 8 in the developer manual talks more about reseting and initialization of the device.

Please implement *status _82545EMInit(struct ether *ethptr)* accordingly.

6. The Command Unit (CU), Receive Unit (RU) and DMA Rings

The 82545EM uses device registers and shared memory to communicate with the host CPU. The driver controls the device by writing and reading data to and from the shared items. The shared items are divided into three parts: the Control/Status Registers (CSR), the Command Block List (CBL), and the Receive Frame Area (RFA). The CSR physically resides on the LAN controller and can only be accessed by port-mapped I/O commands (see above); the rest of the shared itesm reside in system (host) memory. Driver software controls the state of the Command Unit (CU) and Receive Unit (RU) by writing commands to the SCB (for example, the driver can decide whether an item is active, suspended or idle). Like any device that performs DMA transfers, the 82545EM incorporates small processors, which are known as co-processors. The CU and RU are co-processors on the 82545EM; they operate independently of each other. The CU is responsible for executing control commands sent by the driver. For example, a driver can send a command to the CU to configure the device or a command to transmit a packet. The RU's job is to handle packet ereception. The device hardware makes the status of the CU and RU available to the driver by posting the status in the SCB Status word; when the status changes, the device generates an interrupt. The SCB also holds pointers to a linked list of action commands called the CBL and a linked list of receive resources called the RFA. The driver communicates with the CU and RU

co-processors through the CSR and a pair of DMA rings. You may want to look at Section 6.4 to find more details.

A DMA ring is a set of buffers allocated in main memory and chained together by pointers. This ring is usually a circular singly-linked list where the pointers are physical addresses of the next buffer in the ring (read Chapter 16 of the textbook). Using DMA rings allow the driver to specify multiple packet buffers for the hardware to use. There are separate DMA rings for packet transmission and packet reception.

7. Receive Operation

To receive packets, a DMA ring must be constructed in the memory. The DMA ring, which is also called Receive Frame Area (RFA) in Intel speak, is composed by a set of Receive Frame Descriptors (RFD). The structure of the RFD is described in Section 6.4.3.1.2. In the simplified memory model, an RFD is composed of a header followed by a contiguous region of memory that can hold an entire maximum length packet (we use 2048 in Xinu). Consult the developer manual for details. Linking your RFDs in a ring so that the device can use the DMA ring statically to avoid the overhead of dynamically adding new RFDs to the existing ring. We recommand you to allocate a receiving ring with 32 RFDs first. Also, you will have to load the DMA ring address to the RU Base Register during initialization.

Roughly speaking, upon receiving a packet from the network, the Receive Unit (RU) copies that packet into the current RFD, marks that RFD as valid, and follows the link to the next RFD and raise an interrupt to notify the CPU. Setting FR bit in SCB Status indicates the RU has finished receiving a frame and a new packet is available to be picked up. You need to write code that handles receive interrupts in a timely manner.

Here is the break down of the functions you need to implement receiving.

*local void _82545EM_configure_rx(struct ether *ethptr):*
1. Disable receiver while configuring.
2. Set the Receive Delay Timer Register
3. Setup the hardware Rx Head and Tail Descriptor Pointers.
4. Disable Receive Checksum Offload for IPv4, TCP and UDP.
5. Set up the RCTL register and enable receiver.

*devcall e1000Read(struct dentry *devptr ,void *buf, uint32 len):*
1. Wait for a packet to arrive.
2. Find out where to pick up the packet.
3. Verify the package and pick up the packet.
4. Clear up the descriptor and the buffer.
5. Add newly reclaimed descriptor to the ring.
6. Advance the head pointing to the next ring descriptor which will be ready to be picked up.

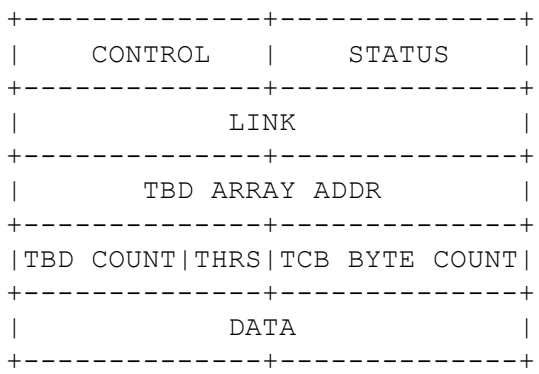
8. Transmit Operation

In order to tell the CU to transmit a packet, a DMA output ring ring must be constructed in the memory

so that the CU knows where to find the packets. Section 6.4 describes the precise format of various Command Blocks (CB). For your lab, our interest will be on the Transmit Command Block (TCB) and the Specific Action Command Block (individual address setup and configure).

Figure 14 in the developer's manual shows a generic CB. The DMA ring of the CBs is called a Command Block List (CBL) in Intel speak.

A TCB is a special CB that can hold a packet the driver wants to send. The fields of a TCB are covered in detail in Section 6.4.2.5. An individual TCB looks like this:



As in a Receiving DMA ring, you need to link a set of TCBs into a ring data structure. Pre-allocate a ring with 32 TCBs and load the DMA ring address to the CU Base Register during initialization. Consult the developer manual to find the appropriate interrupt to use and write code to handle it in a timerly manner.

Here is the break down of the functions you need to implement transmit.

*local void _82545EM_configure_tx(struct ether *ethptr):*
1. Set the transmit descriptor write-back policy.
2. Program the Transmit Control Register.
3. Set the default values for the Tx Inter Packet Gap timer.
4. Set the Tx Interrupt Delay register.
5. Setup the HW Tx Head and Tail descriptor pointers.
6. Set the Transmit Control Register.

*devcall e1000Write(struct dentry *devptr ,void *buf, uint32 len):*
1. Verify Ethernet interface is up and arguments are valid.
2. Check the length, If padding of short packet is enabled, the value in TX descriptor length feild should be not less than 17 bytes
3. Wait for a free ring slot.
4. Find the tail of the ring to insert packet.
5. Copy packet to transmit ring buffer.
6. Insert transmitting command and length.
7. Add descriptor by advancing the tail pointer.
8. Advance the ring tail pointing to the next available ring descriptor.

9. The Organization of a Device Driver and Interrupt Handling

Typically, a driver is divided into two halves: the upper-half and lower-half. The two halves use incoming and outgoing buffers to communicate with each other. We suggest you to read Chapter 15 and Chapter 16 of the textbook in depth to get more hints about how to organize the driver functions.

10. Testing

After building your own driver, another problem right comes to your mind is that how to verify your driver is working correctly. We have already provide a minimum network stack. Using ARP protocal to test your driver may be a good start. After verrify that ARP works, you can then move on to more advanced testings, like TCP and UDP.

11. Debuging

To test/ debug the lab, you can use the network trace file. It is called vxinu.pcap and saved under /compile dir. In order to see the details about the Internet traffic coming in and out of xinu, type:
`$ tcpdump -r vxinu.pcap -v -vv -x`
It shows you all the details of the incoming and outgoing packages of the system.

Also, we provided a simple script to help you test your driver. It is called test.py, you may run it:

`$ test.py [port#] [number of packets you want to send]`

`The port number should be the same as the one you used in the run-vxinu command. This script will send UDP packets to the` actual Xinu machines. You may look into netin.c to learn how Xinu handles UDP packets.

Turn-in Instructions

Electronic turn-in instructions:

1. Go to `xinu-14spring-lab4-x86/compile` directory and do `make clean`.
2. Go to the directory of which your `xinu-14spring-lab4-x86` directory is a subdirectory (NOTE: please do not rename `xinu-14spring-lab4-x86`, or any of its subdirectories.) E.g., if `/homes/abc/xinu-14spring-lab4-x86` is your directory structure, go to `/homes/abc`
3. Type in the following command: **`turnin -c cs503 -p lab4 xinu-14spring-lab4-x86`**
4. To check turned in files type command: **`turnin -c cs503 -p lab4 -v`**

Notes

1. Please use xinu-console to connect the backends. Start from vxinu101 to vxinu140, all those backends are enquiped with the NICs we need for this lab.

2. Please start with a the copy of `xinu-14spring-lab4-x86` and use your RAM disk implementation temporarily.
3. ALL debugging output should be turned off before you submit your code.
4. No system call functionality should be provided in `xinu-14spring-lab4-x86/sys/main.c` file. It will be replaced with a different `main.c` during testing.
5. Please place a short README describing what you have done in your submitted tarball.i
6. Submissions with only partial output should contain a **detailed** README of what works, and design decisions adopted.