ECE 373 Assignment #3
Spring 2021

Part 1: Digging for Details

One of the tasks of a device driver developer is to gather information about the hardware to be controlled – e.g. what I/O ports are connected to which pins, how many microseconds it takes to reset the chip, or what bit pattern is needed to enable a temperature sensor. In the past, we had to dig around in dark shelves for thick paperback device handbooks – you can probably still find some of these around the Engineering labs and offices. Nowadays, though, the internet is often a great resource for much of this information. These questions will require you to do some searching online for useful device information.

1. **Atomic Motherboards (go boom)**

The computer box we're using in the lab is an AIMB-212 from Advantech. Since the box is locked down, you can't look at the insides to see what it has, so you need some product information. Use the internet to find a datasheet, a User Manual, or some other documentation for this little computer. Download it for your reference, and review it.

a) what is the audio device?

b) what device is the GPIO connected to?

c) how many network (LAN) devices are on the motherboard and what are they?

d) how many total serial ports does the box support, inside and out?

2. **Network Noodling**

Since the experimental boxes are not connected to the university network, the network devices are going unused (one port is plugged into the CAT network, but that's only for us to rebuild the boxes if Atomic lives up to its name – it can be disconnected). Later in this assignment we will explore how to play with them, but first we need some information about how they work. Find the datasheet for the LAN device connected through PCIe (**not through the integrated chip's LCI**), and download it for your reference. The Pin Interface chapter tells how to physically connect to the chip. The Driver Programming Interface chapter tells about the memory-mapped registers – what are their addresses and what bit patterns are used to do things. From the datasheet:

a) What pins control the LEDs?

b) What address offset is the Device Control Register?

c) What bit in the Device Control Register will force a reset of the network chip?

3. **Winken, Blinken, and Nod**

Without any other wires or connectors needed, we can use the LEDs on the network devices as our little toys. There are many ways to set up the LEDs for blinking and for indicating various states in the chip, so the description is spread across several pages.

a) What register (name and address) controls the LEDs?

b) What bit pattern should you use to turn off LED1?

c) What bit pattern should you use to make LED2 blink?

Part 2: Make it Blink

It's time to start driving hardware. Here we put together what you've just found with your code from Homework Assignment #2 and information from the PCI lecture. Roll all that together and we can start blinking LED's on our VM's.

Note: This part of the assignment should be done in your VirtualBox VM, since we don't have physical access to the labs this term. Make sure your network device is the e1000 device for your virtual machine. And since you are using a VM that doesn't have actual LEDs to watch, you can try using the ledmon user program that tries to watch the virtual LEDs for you:
http://web.cecs.pdx.edu/~peterw/ece373/ledmon

4. **Confirmation the char device works:**

The first part of this assignment deals with getting the character device framework running. Use your kernel code and userspace program from Homework #2. Make sure you can open the character device, read from it, and write to it. At this point, it should still read from and write to the syscall_var integer.

5. **Register a PCI driver**:

The next piece is to register the PCI device. You'll need to look at lspci to find the device ID for the device in question (hint: **lspci -n** is more helpful for this...).

Attach the device by implementing the .probe() function callback for PCI devices. Make sure to map the BAR for register access, and store the physical address of the BAR for use later.

It's highly recommended to refer to code we looked at in class to figure out what to do here...and there isn't much needed to be done (double hint: the e1000e driver isn't a bad place to peek). Don't forget to properly clean up the device with a .remove() routine.

You'll also need to figure out which header file to include. It's a good idea to look at the header where

the various pci_request_* calls are defined, using that helpful web tool we keep referring to.

Finally, hook up the PCI driver into your init_module() routine, and subsequent unregister in your exit_module(). What happens in dmesg when you load your driver? **NOTE: Make sure to unbind e1000e before doing this! Otherwise your driver will get loaded but not have any device to connect to because the e1000e driver has already claimed it. Refer to lecture notes on how to bind/unbind.**

6. Hook up file operations

Now you have a nice and shiny PCI driver that loads and does nothing. Let's do something with it:

> Modify your read() implementation so it will read the LED register, using the register information you found above in problem 3. Return the value of the register to the supplied userspace buffer (remember copy_to_user?).
>
> Modify your write() implementation so it will write the LED register with the value supplied in the write() system call. Make sure to at least verify the data is somewhat valid...(make sure it's a 32-bit number).

7. The Home stretch!

Now your device driver is hooked up and ready to accept inputs. First unbind the device from the e1000e driver. Modify your tiny C program in userspace that opens the character device you created (it's ok to hard-code your /dev file here), and have it read the current value of the LED register and display it. Then have your C program write a value to make LED0 turn on. After writing the value, read the value back out and display it. Sleep for 2 seconds, and write a value to disable the LED. Remove your driver, then bind the device back to e1000e.

You are done. Phew!

Make sure your module unloads successfully, and properly cleans up ALL resources you allocated in the driver (chardev, dev_t, pci resources, etc.).

What to turn in:

1. Answers to the questions in Part 1

2. Source code to your kernel module, plus your kernel module Makefile.

3. Your userspace program. This does not require a Makefile, just the source code.

4. A typescript of you binding and unbinding the e1000e driver, and your userspace program running through part 7 of this assignment.

Turn these materials in via Github for Classroom, or the D2L dropbox, before **11pm on Wednesday, 28-April-2021**.