# Lab Report 6
# Introduction to Linux Device Drivers

Christopher Padilla

ECEN 449 - 504

Purpose/Introduction

- The purpose of this lab is to guide students through the process of running a simple character device driver using a Linux kernel module on the FPGA. It will build off the last lab, where first we will need to get a Linux kernel module running on the board. The module will be used to initiate a device driver for user interaction with our multiplication peripheral from lab 3.

Procedure

- We will once again use the contents of lab 4 which includes the Linux kernel source and boot Linux on the board to read and write files to it and test out the mount operation on the SD card as we did in lab 5; we will, however, add the functionality for this module to read from and write to the multiplication peripheral.
- In part 1, all the reading and writing was done directly within our init_module() function. Part 2 is where we add the file operations to read and write (this is where the user can interact directly with the character device).
- The final part was to create a means of testing our character device using many multiplications and calculating new products every time we receive a user input.

Results

- After cross-compiling and multiplier kernel module. I was able to see the correct results of the multiplication (of the numbers 7 and 2 that I hardcoded), as well as the physical and virtual base addresses of the multiplication peripheral.
- Then I cross-compiled the kernel module in conjunction with the character device driver I was able to register my device using the dynamically allocated major number.
- The last part was to develop a way for user interaction with the device by means of the character device driver. This part required much debugging; the most time-consuming (and annoyingly simple) bug was that the while loop provided in the example driver did not allow for me to read more than one byte in our device_read() function. Switching loop to a for loop allowed me to read every byte successfully, and the correct results were then displayed on the PICOCOM terminal.

Conclusion

- In this lab, I was able to get a simple Linux kernel module and character device driver up and running on the ZYBO board. I was guided through the process of creating a device driver whose purpose was to display the result of multiplying several numbers together. Using user input, the driver knew when to continue to read and write to the multiplication peripheral.
- Even though the purpose of this lab was pretty simple, I now feel more comfortable working with modules to extend the kernel's functionality by adding device drivers.

Questions

- Using ioremap() was required to map the physical address of the multiplication peripheral to a virtual address. Virtual memory is necessary because this way we can assume that our peripheral's addresses (that we read and write to) are located contiguously in memory. This may not be the case for the physical address.
- The original hardware in lab 3 was mapped directly to the ARM processor. Since there is no operating system in the way, our original lab 3 implementation may perform quicker.

It is important to note, however, that the user interface may not be as friendly as our current implementation.

- Our original lab 3 implementation may be harder to interact with; Linux device drivers make it easier for the user to interact with the hardware, so while it may be minimally slower, it is much easier to work with from a user aspect. Reading and writing to the device is much easier to do in lab 6 than lab 3. The only negative is that in lab 6, with Linux as a middleman, it may perform a bit slower.
- The idea is that we want to get everything we need set up *before* we initialize our character device driver. For example, we don't want to perform initialization before we map the virtual address space because we are missing a component that allows our device driver to perform properly. Likewise, in the exit routine, we want to unregister our device *first* so that it remains uninterrupted by the unmapping/cleaning up of resources while the device is still connected, since the device may still be using the components like memory allocation.