# ECEN 449: Microprocessor System Design
# Department of Electrical and Computer Engineering
# Texas A&M University

Prof. Sunil P. Khatri

Lab exercise created and tested by:
Kushagra Gupta, Cheng-Yen Lee, Abbas Fairouz, Ramu Endluri, He Zhou,
Andrew Douglass and Sunil P. Khatri

# Laboratory Exercise #6

# An Introduction to Character Device Driver Development

## Objective

The purpose of lab this week is to create device drivers in an embedded Linux environment. Device drivers are a part of the operating system kernel which serves as the bridge between user applications and hardware devices. Device drivers facilitate access and sharing of hardware devices under the control of the operating system. You will then extend the capabilities of your kernel module from Lab 5, thereby creating a complete character device driver. To test your multiplication device driver, you will also develop a simple Linux application which utilizes the device driver, providing the same functionality as seen in Lab 3.

## System Overview

The hardware system you will use this week is that which was built in Lab 4. Figure 1 depicts a simplified view of the hardware and software you will create in this lab. Please note the hardware/software boundary in the figure below. Above this boundary, the blocks represent compiled code executing within the ARM processor. Below this boundary, the blocks represent hardware attached to the microprocessor. In our particular case, the hardware is a multiplication peripheral attached to the ARM Processor. Obviously, other hardware/software interactions exist in our system, but Figure 1 focuses on that which you will provide. Also

notice the existence of the kernel-space/user-space boundary. The kernel module represents the character device driver executing in kernel space, while the user application represents the code executing in user space, reading and writing to the device file, '/dev/multiplier'. The device file is not a standard file in the sense that it does not reside on a disk, rather it provides an interface for user applications to interact with the kernel.

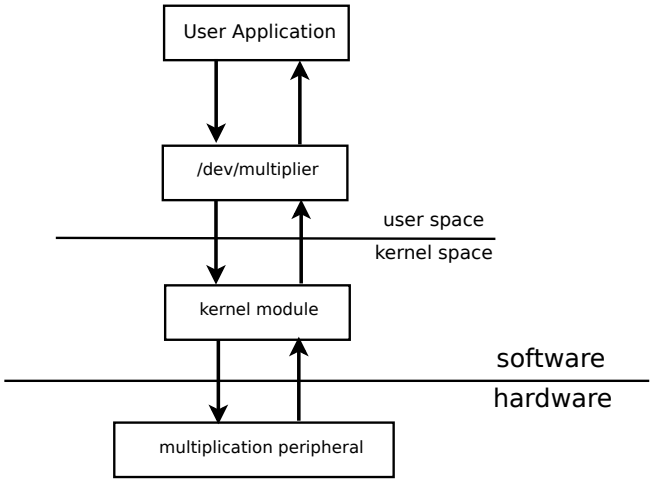


Figure 1: Hardware/Software System Diagram

## Procedure

1. With a functioning kernel module that reads and writes to the multiplication peripheral, you are now ready to create a character device driver (using PetaLinux) that provides applications running in user space access to your multiplication peripheral.

   (a) Insert the USB drive that includes the installation and the project directory of PetaLinux (from Lab 4).

(b) From within the project directory, create a character device driver called 'multiplier.c' (refer to the steps used in Lab 5 for module creation) using the following guidelines:

- Take a moment to examine 'my_chardev.c', 'my_chardev_mem.c' and the appropriate header files within '/home/faculty/shared/ECEN449/module_examples/'. Use the character device driver examples provided in *Linux Device Drivers, 3rd Edition* and the laboratory directory as a starting point for your device driver.
- Use the 'multiply.c' kernel module created in Section 2 of this lab as a baseline for reading and writing to the multiplication peripheral and mapping the multiplication peripheral's physical address to virtual memory.
- Within the initialization routine, you must register your character device driver after the virtual memory mapping. The name of your device should be 'multiplier'. Let Linux dynamically assign your device driver a major number and specify a minor number of 0. Print the major number to the kernel message buffer exactly as done in the examples provided. Be sure to handle device registration errors appropriately. You will be graded on this!
- In the exit routine, unregister the device driver before the virtual memory unmapping.
- For the open and close functions, do nothing except print to the kernel message buffer informing the user when the device is opened and closed.
- Read up on 'put_user' and 'get_user' in *Linux Device Drivers, 3rd Edition* as you will be using these system calls in your custom read and write functions.
- For the read function, your device driver must read bytes 0 through 11 within your peripheral's address range and put them into the user space buffer. Note that one of the parameters for the read function, 'length', specifies the number of bytes the user is requesting. Valid values for this parameter include 0 through 12. Your function should return the number of bytes actually transferred to user space (i.e. into the buffer pointed to by char* buf). You may use 'put_user' to transfer more than 1 byte at a time.
- For the write function, your device driver must copy bytes from the user buffer to kernel space using the system call 'get_user' to do the transfer and the variable 'length' as a specification of how many bytes to transfer. Furthermore, the device driver must write those bytes to the multiplication peripheral . The return value of your write function should be similar to that of your read function, returning the number of bytes successfully written to your multiplication peripheral. Only write to valid memory locations (i.e. 0 through 7) within your peripheral's address space.

(c) Modify 'multiplier.bb' to include any .h files as you did in Lab 5.

(d) Build multiplier module:

```
$ petalinux-build
```

(e) As with 'multiply.ko' from Lab 5, load 'multiplier.ko' into your ZYBO Z7-10 Linux system.

(f) Use 'dmesg' to view the output of the kernel module after it has been loaded. Follow the instructions provided within the provided example kernel modules to create the '/dev/multiplier' device node. Demonstrate your progress to the TA.

2. At this point, we need to create a user application that reads and writes to the device file, '/dev/multiplier', to test our character device driver, and provides the required functionality similar to that seen in Lab 3.

   (a) View the man pages on 'open', 'close', 'read', and 'write' from within the CentOS workstation. You will use these system calls in your application code to read and write to the '/dev/multiplier' device file. Accessing the man pages can be done via the terminal. For example, to view the man pages on 'open', type 'man open' in a terminal window.

   (b) Create lab6 directory and within that create a source file called 'devtest.c' and copy the following starter text into that file:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
  unsigned int result;
  int fd;                          /* file descriptor */
  int i,j;                         /* loop variables */

  char input = 0;

  /*open device file for reading and writing*/
  /*use 'open' to open '/dev/multiplier'*/

  /*handle error opening file*/
  if(fd == -1){
    printf("Failed to open device file!\n");
    return -1;
  }

  while(input != 'q'){ /*continue unless user entered 'q'*/

    for(i=0; i<=16; i++){
      for(j=0; j<=16; j++){

        /*write values to registers using char dev*/
        /* use write to write i and j to peripheral */
        /*read i, j, and result using char dev*/
        /*use read to read from peripheral*/
```

```
            /*print unsigned ints to screen*/
            printf("%u * %u = %u ",read_i,read_j, result);

            /*validate result*/
            if(result == (i*j))
              printf("  Result Correct!");
            else
              printf("  Result Incorrect!");

            /*read from terminal*/
            input=getchar();
          }
        }
      }
      close(fd);
      return 0;
    }
```

(c) Complete the skeleton code provided above using the specified system calls.

(d) Compile 'devtest.c' by executing the following commands in the terminal window under the lab6 directory:

$ source /opt/coe/Xilinx/Vivado/2022.1/settings64.sh
$ arm-linux-gnueabihf-gcc -o devtest devtest.c

(e) Copy the executable file, 'devtest', on to the SD card and execute the application by typing the following in the terminal within the SD card mount point directory:
>./devtest

(f) Examine the output as you hit the 'enter' key from the terminal. Demonstrate your progress to the TA.

## Deliverables

1. [7 points.] Demo the working kernel module and device driver to the TA.

Submit a lab report with the following items:

2. [5 points.] Correct format including an Introduction, Procedure, Results, and Conclusion.
3. [2 points.] The output of the picocom terminal for parts 2 through 4 of lab.
4. [6 points.] Answers to the following questions:

(a) Given that the multiplier hardware uses memory mapped I/O (the processor communicates with it through explicitly mapped physical addresses), why is the ioremap command required?

(b) Do you expect that the overall (wall clock) time to perform a multiplication would be better in part 3 of this lab or in the original Lab 3 implementation? Why?

(c) Contrast the approach in this lab with that of Lab 3. What are the benefits and costs associated with each approach?

(d) Explain why it is important that the device registration is the last thing that is done in the initialization routine of a device driver. Likewise, explain why un-registering a device must happen first in the exit routine of a device driver.