ECEN 449 - Lab Report

Lab Number: 7

Lab Title: Linux Kernel – Built In Modules

Section Number: 511

Student's Name: Kris Gavvala

Student's UIN: 929000158

Date: 4/29/2025

TA: Gautham Nemani

Introduction:

This week's lab centered on building a Linux OS using the Petalinux SDK with a custom driver built into the kernel. We also learned how to disable unnecessary drivers to create a kernel with a reduced memory size.

Procedure:

1. Create a Petalinux project in the tmp folder of the Linux machine
2. Configure it with the multiply hardware XSA file created in lab 3
3. Run Petalinux-config -c kernel to configure the kernel
4. Download the Xilinx Linux kernel and add it to the Petalinux project
5. Edit the kernel and add the files for multiply driver (multiplier.c and header files)
6. Create a Makefile in the multiply driver directory that compiles the multiply driver object into the kernel
7. Create a Kconfig file in the multiplier directory and add dependencies and configuration parameters.
8. In the kernel drivers directory add lines in the makefile and Kconfig files to specify the location of the multiplier config file and to add the multiplier driver directory to the kernel build.
9. After adding the kernel to the build, run petalinux-config and navigate the menu to specify the external linux kernel that was added to the build.
10. Build the petalinux project
11. Obtain the image.ub file after the build completes and load it onto a SD card along with the boot.bin and boot.scr files from the previous lab 4, and the devtest executable to test the driver.
12. Boot the operating system on the fpga and see the output after executing devtest.
13. To build an OS with a reduced size kernel image, repeat the steps above, but after #8 go to the config menu and disable the network device support, multimedia support, and soundcard support drivers. Then build it and boot on the fpga.

Results:

Upon booting the OS and running devtest, the picocom terminal showed the output from the multiplication performed in devtest.c using the multiplication driver. This showed all combinations of the numbers 1-16 multiplied by each other. When the kernel was rebuilt with the three drivers disabled, the kernel image size shrank from 18 MB to 16.5 MB.

Conclusion:

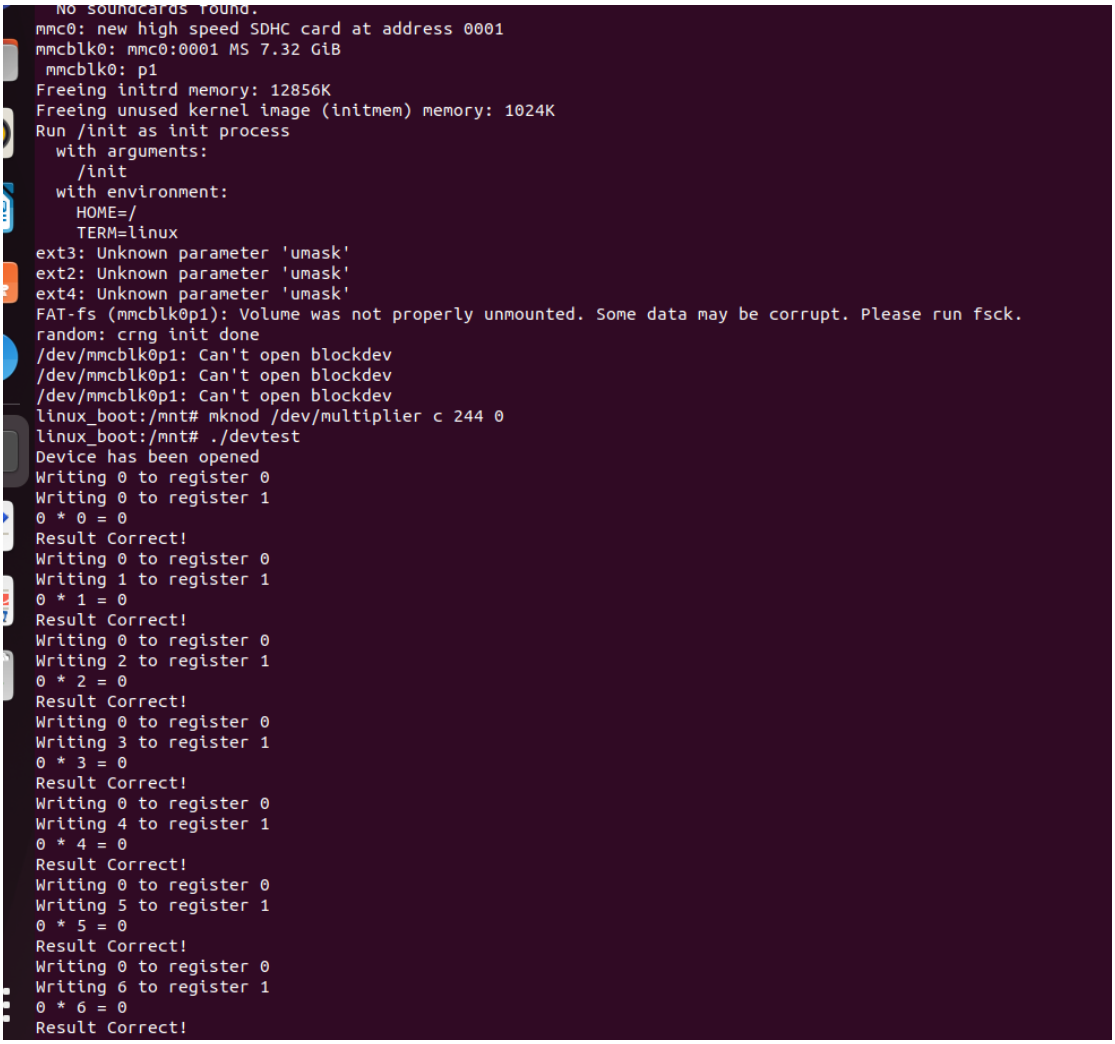
This lab showed an example of how to create a driver by building the driver directly into the kernel space. This is useful because the driver is available immediately at boot time and there is no need insert a .ko file with the insmod command. We also learned how disable drivers to create a leaner kernel image using the petalinux SDK.

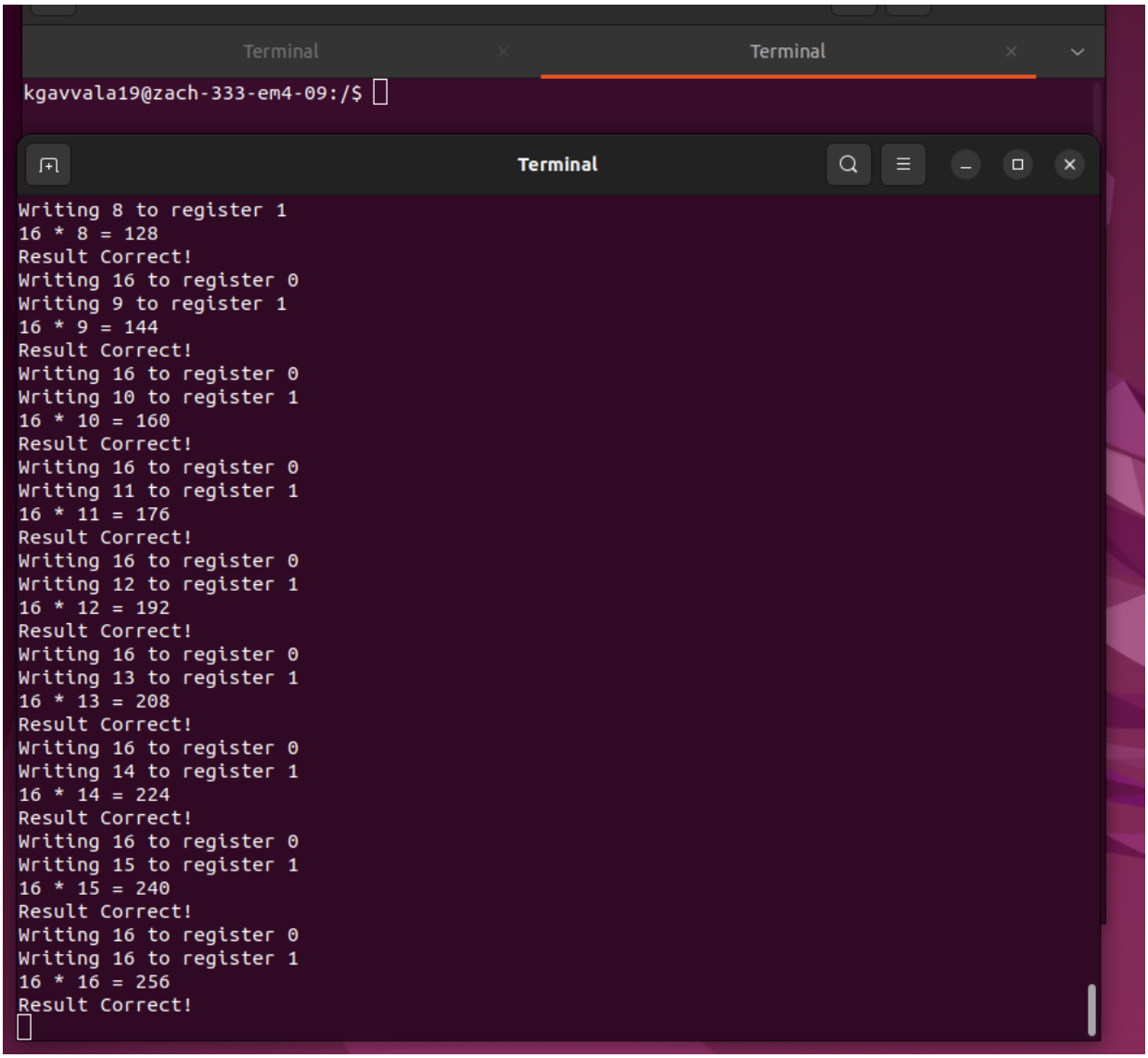
Post-lab Deliverables:

What are the advantage and disadvantages of loadable kernel modules and built-in modules?

ANSWER: Built-in kernel modules can be advantageous when you want to reduce boot time since you don't need to load the module with insmod before booting the system. Loadable kernel modules are more customizable and don't require you to rebuild the whole kernel every time you make a change to the driver. If you don't want to load certain modules you have the option with loadable modules to load as-needed which gives you flexibility with memory usage.

Picocom terminal:




Terminal
Terminal
kgavvala19@zach-333-em4-09:/$
Terminal
Writing 8 to register 1
16 * 8 = 128
Result Correct!
Writing 16 to register 0
Writing 9 to register 1
16 * 9 = 144
Result Correct!
Writing 16 to register 0
Writing 10 to register 1
16 * 10 = 160
Result Correct!
Writing 16 to register 0
Writing 11 to register 1
16 * 11 = 176
Result Correct!
Writing 16 to register 0
Writing 12 to register 1
16 * 12 = 192
Result Correct!
Writing 16 to register 0
Writing 13 to register 1
16 * 13 = 208
Result Correct!
Writing 16 to register 0
Writing 14 to register 1
16 * 14 = 224
Result Correct!
Writing 16 to register 0
Writing 15 to register 1
16 * 15 = 240
Result Correct!
Writing 16 to register 0
Writing 16 to register 1
16 * 16 = 256
Result Correct!

Appendix:

Devtest.c:

```
#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, i_read, j_read;
   int fd;   // file descriptor
   int i, j; // input variables
   char input = 0;
   unsigned int *inputs = (unsigned int *)malloc(2 * sizeof(int));
   char *outbuf = (char *)malloc(3 * sizeof(int));  //allocate dynamic memory
   int *charToint; // used to convert from char* to int* buffer
   // open device file for reading and writing
   fd = open("/dev/multiplier", O_RDWR);

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

```
while (input != 'q')
{ // continue until user enters 'q'
   for (i = 0; i <= 16; i++)
   {
      for (j = 0; j <= 16; j++)
      {
         inputs[0] = i;
         inputs[1] = j;
         char *buffer = (char *)inputs; // convert to char* buffer
         // write the values to the device file

         write(fd, buffer, 2 * sizeof(int));  //read to device file
         read(fd, outbuf, 3 * sizeof(int));
         // read the result from the device file
         // convert the char* buffer to int* buffer
         charToint = (int *)outbuf;
         i_read = charToint[0];
         j_read = charToint[1];
         result = charToint[2];

         printf("%u * %u = %u\n", i_read, j_read, result);
         if (result == (i * j))
         {
            printf("correct\n");
         }
         else
         {
            printf(“incorrect\n");
         }
```

```
            }
        }
        input = getchar();
    }
    close(fd);  //close device file
    free(inputs);  //free memory
    free(outbuf);
    return 0;
}
```

Multiplier.c

```
#include <linux/module.h>      /* Needed by all modules */
#include <linux/moduleparam.h> /* Needed for module parameters */
#include <linux/kernel.h>      /* Needed for printk and KERN_* */
#include <linux/init.h>        /* Need for __init macros  */
#include <linux/fs.h>          /* Provides file ops structure */
#include <linux/sched.h>       /* Provides access to the "current" process
task structure */
#include <linux/slab.h>        //needed for kmalloc() and kfree()
#include <asm/io.h>            //needed for IO reads and writes
#include <asm/uaccess.h>       // Provides utilities to bring user space

#include "xparameters.h" //needed for physical address of the multiplier

#define DEVICE_NAME "multiplier"
#define BUF_LEN 80
```

```
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR // physical address of multiplier

// size of physical address range for multiply

#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR + 1


/* Function prototypes, so we can setup the function pointers for dev

file access correctly. */

int init_module(void);

void cleanup_module(void);

static int device_open(struct inode *, struct file *);

static int device_release(struct inode *, struct file *);

static ssize_t device_read(struct file *, char *, size_t, loff_t *);

static ssize_t device_write(struct file *, const char *, size_t, loff_t *);


static int Major;

/ Major number assigned to our device driver * /

   static int Device_Open = 0; // Flag to signify open device


void *virt_addr; // virtual address pointing to multiplier


static struct file_operations fops = {

   .read = device_read,

   .write = device_write,

   .open = device_open,

   .release = device_release};


// initialize the module

// called when the module is loaded

static int __init my_init(void)
```

```
{
   printk(KERN_INFO "Mapping virutal address...\n");
   // map virtual address to multiplier physical address//use ioremap
   virt_addr = ioremap(PHY_ADDR, MEMSIZE);
   printk("Physical Address: 0x%x\n", PHY_ADDR);
   printk("Virtual Address: 0x%x\n", virt_addr);

   Major = register_chrdev(0, DEVICE_NAME, &fops); // dynamic allocation

   /* Negative values indicate a problem */
   if (Major < 0)
   {
      printk(KERN_ALERT "Registering char device failed with %d\n", Major);
      return Major;
   }

   printk(KERN_INFO "Registered a device with dynamic Major number of %d\n", Major);
   printk(KERN_INFO "Create a device file for this device with this command:\n'mknod /dev/%s c
%d 0'.\n", DEVICE_NAME, Major);

   return 0; /* success */
}

// called when the module is unloaded
// called when the module is removed
static void __exit my_cleanup(void)
{
   // Unregister the device
   unregister_chrdev(Major, DEVICE_NAME);
   printk(KERN_ALERT "unmapping virtual address space...\n");
```

```
    iounmap((void *)virt_addr);
}

// called when a process opens the device file
static int device_open(struct inode *inode, struct file *file)
{
    printk(KERN_ALERT "Device has been opened\n");
    return 0;
}

// called when a process closes the device file
static int device_release(struct inode *inode, struct file *file)
{
    printk(KERN_ALERT "Device has been closed\n");
    return 0;
}

// called when a process reads from the device file
static ssize_t device_read(struct file *filp,
                           char *buffer,
                           size_t length,
                           loff_t *offset)
{

    // bytes read from the buffer
    int bytes_read = 0;
    // allocating kernel buffer
    int *kBuff = (int *)kmalloc(length * sizeof(int), GFP_KERNEL);
    kBuff[0] = ioread32(virt_addr);
    kBuff[1] = ioread32(virt_addr + 4);
```

```
    kBuff[2] = ioread32(virt_addr + 8);
    char *kBuffer = (char *)kBuff; // bytes written one at a time

    int i;
    for (i = 0; i < length; i++)
    {                               // read the buffer one byte at a time
        put_user(*(kBuffer++), buffer++); // char is one byte
        // put_user copies the data from kernel space to user space
        // buffer is the user space buffer
        // kBuffer is the kernel space buffer

        bytes_read++;
    }

    kfree(kBuff);

    // printk("bytes_read: %d\n", bytes_read);
    return bytes_read;
}

static ssize_t
device_write(struct file *filp, const char *buff, size_t len, loff_t *off)
{
    char *kBuff = (char *)kmalloc((len + 1) * sizeof(char), GFP_KERNEL);
    /* use kBuff to write one
        byte at a time from user buffer*/
    int i;
    for (i = 0; i < len; i++)
    {
        get_user(kBuff[i], buff++);
```

```
    }

    kBuff[len] = '\0';
    /* Convert kBuff to int* to write to
       multiply*/
    int *writeBuffer = (int *)kBuff;

    // write to register 0
    printk(KERN_INFO "Writing %d to register 0\n", writeBuffer[0]);
    iowrite32(writeBuffer[0], virt_addr + 0); // base address + offset

    // write to register 1
    printk(KERN_INFO "Writing %d to register 1\n", writeBuffer[1]);
    iowrite32(writeBuffer[1], virt_addr + 4);

    kfree(writeBuffer);
    return i; // number of bytes written
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Kris Gavvala");
MODULE_DESCRIPTION("Multiplier Device Driver");

module_init(my_init);
module_exit(my_cleanup);
```

Multiplier.bb

```
SUMMARY = "Recipe for  build an external multiplier Linux kernel module"
SECTION = "PETALINUX/modules"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"

inherit module

INHIBIT_PACKAGE_STRIP = "1"

SRC_URI = "file://Makefile \
        file://multiplier.c \
            file://COPYING \
            file://xparameters.h \
            file://xparameters_ps.h \
        "

S = "${WORKDIR}"

# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.
```