ECEN 449 - Lab Report

Lab Number: 7

Lab Title: Built-in Modules

Section Number: 501

Saira Khan

731005607

November 26 2024

TA: Shao-Wei Chu

Introduction:

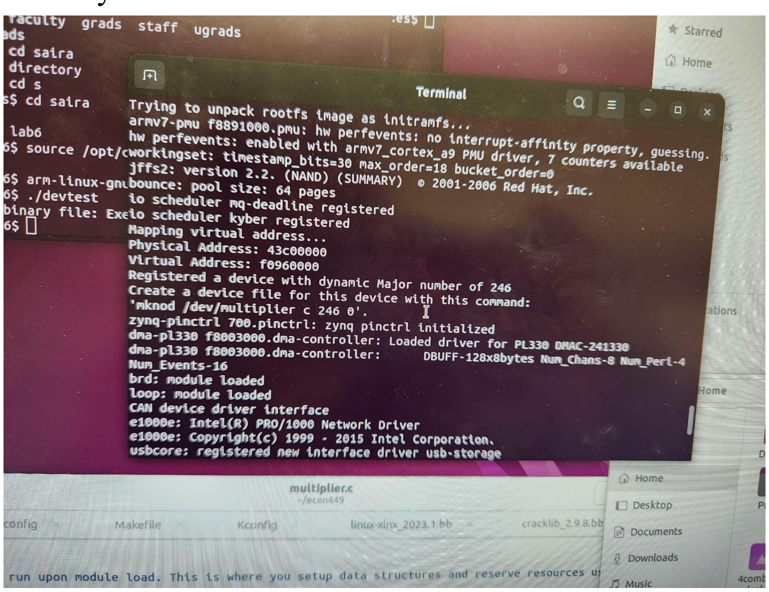
The purpose of this lab was to get familiar with built-in kernel drivers. A built-in kernel was constructed and the multiplier device driver, from the last lab, was added to this kernel in order to have it load during boot.

Procedure:

A built-in kernel is a kernel that automatically loads the device driver into the kernel when the kernel is booted up. There is another way of loading a device driver into a kernel with loadable kernel modules. This is when the module is loaded into the kernel after it is booted up. In order to create a built-in kernel a petalinux project has to be created. Steps from the previous labs were performed to create a petalinux project. A directory for the multiplier_driver was then created and the source files for the multiplier kernel were placed in that. A Makefile and Kconfig file were created and placed in the directory as well. Makefiles are used to dictate which parts of a program need to recompile. For this lab, a Makefile was made and the code put into it ensured that the multiplier driver was built when the configuration multiplier driver was enabled during kernel configuration. Kconfig files are used to select build-time options and to enable/disable features. This Kconfig file allows the kernel build system to include/exclude the multiplier driver based on the user's selection during configuration. The Linux source was then linked to the petalinux project, and the multiplier driver was configured to be built-in to the Linux kernel. The project was then built and tested. Some features were then excluded on the menuconfig from the build. This reduced the size of the image.ub.

Results:

The built-in kernel was successfully made and the device driver was implemented into it correctly.



Conclusion:

From this lab, I learned how to configure a kernel using menuconfig. I became familiar with the way Kconfig files are structured for driver configuration including dependencies, default values, and user interface elements in menuconfig. Going through this lab allowed me the opportunity to see the difference between built-in kernel modules and loadable kernel modules.

Answers to Questions:

a) Some advantages of loadable kernel modules include that they can be loaded and removed without rebooting the system, the modules not in use can be unloaded to free memory, and it is easier to test/debug because it doesn't require a kernel rebuild. Some disadvantages include that it takes time to configure the module after bootup. Some advantages of built-in modules are that the required drivers are always available after boot and being a part of the kernel means that they are less prone to compatibility issues. Some disadvantages include that it increases the size of the kernel and it requires a kernel rebuild to modify or add features.

Appendix:

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


//multiplication peripheral - hardware device designed to perform multiplication operations
//resides in memory-mapped I/O space - can be accessed by writing to or reading from specific memory addresses
//character device driver - kernel module that allows user-space applications to communicate with character-based devices via file-operations (open, read, write)

//driver registers the device with the kernel and dynamically gets a major number (identifier for the driver)
//device file created (/dev/multiplier) - acts as inteface between user-space applications and the driver
//driver responsibilites:
	//read - transfers data from multiplication peripheral to user-space applications
	//write - transfers data from user-space applications to multiplication periphereal
```

//open/close - proper resource management
//driver maps physical memory of multiplication peripheral to virtual address using ioremap - allows kernel to access peripheral registers directly

//ioremap/iounmap
//map/unmap physical memory regions to virtual addresses in the kernel

//register_chrdev/unregister_chrdev
//registers/unregisters a character device with the kernel

//put_user/get_user
//used to transfer data between kernel and user space

//iowrite8/ioread8
//write/read 8-bit data to/from mapped periphereal memory

```
#define PHY_ADDR XPAR_MULTIPLY_0_S00_AXI_BASEADDR //physical address of multiplier
/*size of physical address range for multiple */
#define MEMSIZE XPAR_MULTIPLY_0_S00_AXI_HIGHADDR - XPAR_MULTIPLY_0_S00_AXI_BASEADDR+1

#define DEVICE_NAME "multiplier"
/* 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 Device_Open=0;

void* virt_addr; //virtual address pointing to multiplier
static int Major; /* Major number assigned to our device driver */

/* This structure defines the function pointers to our functions for  opening, closing, reading and writing the device file.  There are  lots of other pointers in this structure which we are not using, see the whole definition in linux/fs.h */

static struct file_operations fops = {

  .read = device_read,
  .write = device_write,
```

```
  .open = device_open,
  .release = device_release};
/* This function is run upon module load. This is where you setup data structures and reserve
resources used by the module. */

//maps physical memory of multiplication peripheral to virtual address using ioremap
//register driver as character device
//virtual mapping to access hardware registers from kernel
//device registration enables user-space programs to use standard file operations to interact
with periphereal

static int __init my_init(void) {
/* Linux kernel's version of printf */
printk(KERN_INFO "Mapping virtual address...\n");

/*map virtual address to multiplier physical address*/
//use ioremap
//PHY_ADDR = physical address of peripheral
//size determined by MEMSIZE
//ioremap creates virtual address mapping - allows driver to access peripheral's registers

virt_addr = ioremap(PHY_ADDR, MEMSIZE);
//msg_ptr = kmalloc
printk("Physical Address: %x\n", PHY_ADDR); //Print physical address
printk("Virtual Address: %x\n", virt_addr); //Print virtual address

/* This function call registers a device and returns a major number associated with it.  Be wary,
the device file could be accessed as soon as you register it, make sure anything you need (ie
buffers ect) are setup _BEFORE_ you register the device.*/

Major = register_chrdev(0, DEVICE_NAME, &fops);

/* Negative values indicate a problem */
if (Major < 0) {
/* Make sure you release any other resources you've already grabbed if you get here so you
don't leave the kernel in a broken state. */
printk(KERN_ALERT "Registering char device failed with %d\n", Major);
//iounmap((void*)virt_addr);
return Major;
} else {
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);
```

```
//a non 0 return means init_module failed; module can't be loaded.

return 0;
}

/* This function is run just prior to the module's removal from the system. You should release
_ALL_ resources used by your module here (otherwise be prepared for a reboot). */
//unmaps virtual address previously mapped using iounmap
//unregisters character device - free major number
//unmaps virtual address space - release kernel memory resources
//cleanup - avoid memory leaks

static void __exit my_exit(void) {
printk(KERN_ALERT "unmapping virtual address space...\n");
unregister_chrdev(Major, DEVICE_NAME);
iounmap((void*)virt_addr);

}

/* Called when a process tries to open the device file*/

 //only one process can open the device at a time

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

{
  printk(KERN_ALERT "This device is opened\n");
  if (Device_Open)
return -EBUSY;
//trak device usage
Device_Open++;
//increment module's reference ount, prevent it from being unloaded while in use
try_module_get(THIS_MODULE);
  return 0;

}

/* Called when a process closes the device file.*/
static int device_release(struct inode *inode, struct file *file)

{

  printk(KERN_ALERT "This device is closed\n");
  Device_Open--;
```

```
  module_put(THIS_MODULE);
return 0;

}



/* Called when a process, which already opened the dev file, attempts to read from it.*/

//transfers bytes from devices mapped memory to user-space buffer using put_user

//reads bytes from memory locations 0 to 11 in peripheral's address range based on request4ed
length
//returns number of bytes successfully read

//put_user function ensure kernel-to-user space data transfer is safe
//reading byte-by-byte with loop makes function good for variable-length user requests

//virt_addr + i - offset from base virtual address of mapped physical address
//loop reads each byte of data from hardware peripheral's address space
//ioread8(virt_addr + i) accesses data at the computed address
//virt_addr + i - iterate through range of memory-mapped addresses within hardware peripheral's
address space to fetch data sequentially

static ssize_t device_read(struct file *file, /* see include/linux/fs.h*/

  char *buffer,      /* buffer to fill with data */
  size_t length,     /* length of the buffer - number of bytes user requesting  */


{

/*Number of bytes actually written to the buffer*/
int bytes_read = 0;
int i;

//iterates for length bytes request by user
//uses ioread8 to read from peripheral memory
//put_user - copy data to user buffer
//track number of bytes read

for(i=0; i<length; i++) {

//put_user(value to transfer, destination pointer)
```

```
//ioread8  - reads 8-bit value from memory-mapped address
//char cast - value read is treated as 8-bit character
//buffer+i = destination - + i = read to next location in user-space buffer sequnetially
//offset passed to virt_addr determines which specific register of multiplier peripheral is being
accessed
//offset 0 = first register = input
//offset 1 = next 32-bit register  - next input
//offset 2 - output
//offset + 3 - MSB

put_user((char)ioread8(virt_addr+i), buffer+i);
bytes_read++;

}


/* Most read functions return the number of bytes put into the buffer*/
return bytes_read;

}



/* This function is called when somebody tries to write into our device file.*/

//transfers data from user-space to peripheral using get_user to fetch data and iowrite8 to write
it to mapped memory
//writes data only to locations 0 to 7

static ssize_t device_write(struct file *file, const char __user * buffer, size_t length, loff_t * offset)

{
int i;
char message;

/* get_user pulls message from userspace into kernel space */
for(i=0; i<length; i++) {
get_user(message, buffer+i);
//virt_addr+i = base address + offset

iowrite8(message, virt_addr+i);

}
```

```
/* Again, return the number of input characters used */
return i;

}

/* These define info that can be displayed by modinfo */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("");
MODULE_DESCRIPTION("labe");

/* Here we define which functions we want to use for initialization and cleanup */
module_init(my_init);
module_exit(my_exit);


#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;
  //open character device file in read/write mode
  int fd = open("/dev/multiplier",O_RDWR);
  int i,j;
  unsigned int read_i;
  unsigned int read_j;
  char input = 0;
  int buffer[3];

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

  while(input != 'q')
  {
```

```
  //iterates over values of i and j from 0 to 16 to test all combinations of numbers withint this
range
    for(i=0; i<=16; i++)
    {
      for(j=0; j<=16; j++)
      {

              //fills buffer array with i and j
              //calls write to send i and j to device
  buffer[0]=i;
buffer[1]=j;
//write 8 bytes, 4 for buffer[0] (i) and 4 for j
write(fd,(char*)&buffer,8);
//read first and second operand (8 byes), and last 4 = multiplication result
        read(fd,(char*)buffer,12);
        read_i=buffer[0];
        read_j=buffer[1];
        result=buffer[2];
        printf("%u * %u = %u ",read_i,read_j,result);

        if(result==(i*j))
          printf("Result Correct!");
        else
          printf("Result Incorrect!");
                      //wait for user input before moving to next iteration

        input = getchar();
      }

    }
  }
  close(fd);
  return 0;
}
```