ECEN 449 - Lab Report

Lab Number: 6

Lab Title: An Introduction to Character Device Driver Development

Section Number: 501

Saira Khan

731005607

November 19, 2024

TA: Shao-Wei Chu

Introduction:
The objective of this lab is to create device drivers in a Linux environment. A Linux application was created to test the multiplication device driver as well. A device driver is a kernel module that allows user-space applications to communicate with character-based devices via file operations like open, read, and write functions.

Procedure:
For hardware, this lab included a multiplication peripheral, designed to perform multiplication operations, attached to an ARM processor. The character device driver executes in the kernel space and is represented by the kernel module. The kernel module reads and writes to the multiplication peripheral. The character driver provides applications running in the user space to the multiplication peripheral. The steps taken to create a petalinux project were performed again in this lab. The character device driver was written in C and the petalinux project was built using it. The device driver included an initialization routine where the data structures were setup and resources were reserved. It maps the physical memory of the multiplication peripheral to the virtual address and registers the driver as a character device. The device registration enables user-space programs to use standard file operations to interact with the peripheral. An exit routine was written as well in order to release all resources used by the module. It maps the virtual address previously mapped, frees the major number, and releases kernel memory resources. An open function is included when a process tries to open the device file and tracks device usage. Both the open and close functions inform the user when the device is opened and closed. The read function transfers bytes from devices mapped memory to the user-space buffer. It reads the bytes from memory locations 0 to ll in the peripheral's address range based on the user's requested length. The function returns the number of bytes successfully read and transferred to the user space. A write function is included to transfer data from the user-space to the peripheral. The user application was then written to test the device driver. This application reads and writes to the device file /dev/multiplier. The device file acts as an interface for the user application to interact with the kernel.

Results:
A character device driver was successfully made and tested using a device file. This allowed for the kernel module and user application to interact with each other.

Conclusion:
Through this lab, I was able to continue familiarizing myself with creating and building petalinux projects. As well, I continued to learn how to create device drivers while utilizing C. I learned how to write my own device driver during this lab as well as a user application to test the driver out.

Answers to Questions:

a) The ioremap command is required because it maps the physical address to the virtual address. Kernel modules cannot directly access physical addresses, so they have to work with the virtual address mapped to the physical memory location.
b) I expect that the overall time to perform a multiplication would be better in lab 3 due to the direct connections to the hardware.
c) Lab 3 is able to perform faster because of the direct connections to hardware, but the driver is also limited due to the hardware. For this lab, there are no limitations due to the hardware but it is a slower runtime.
d) It is important that the device registration is the last thing done in the initialization routine because it makes the device visible and accessible to the rest of the kernel. If it is registered too early, then other parts of the system could try to interact with it before the driver is fully initialized. It also ensures that the driver has successfully initialized all its resources. Unregistering a device must happen first in the exit routine preventing the kernel and user-space program from accessing the driver. If it happened later, external entities could try to interact with the device during cleanup. Once the device is unregistered, the driver can release its resources.

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;
}
```