ECEN 449: Microprocessor System Design

Lab 7: Built-in Module

Kylan Lewis

UIN: 719001131

ECEN 449 -504

TA: Ashwin Ashokan

Date: 11/20/2020

Introduction:

The purpose of this lab is to learn how to add device drivers into the Linux kernel. Additionally we learned how we can remove certain built in drivers that are unused so that we can reduce the total image size.

Procedure:

Part 1:

1. Create a new folder called "Lab7" and copy the previous lab linux kernel into the folder.

2. Use "make menuconfig" inside the Lab7\linux folder and examine some of the device drivers included in the configuration file.

Part 2:

1. Create a folder inside the linux drivers folder called "multiplier_driver"

2. Inside this folder create a Makefile and Kconfig file then copy the relevant lines from the lab manual to them.

3. Then edit the Makefile and Kconfig files in the linux drivers source folder by referencing the newly created multiplier_driver folder.

4. Generate a new uImage and use BOOT.bin and devicetree.dtb from previous labs to boot the newly configured linux image onto the ZYBO board.

Part 3:

1. Go back and disable the Networking support, Multimedia support and Sound Card support in menuconfig then generate a new uImage.

Results:

Functional multiplier

```
zynq> cd /mnt/
zynq> ls
BOOT.bin                     ramdisk8M.image
MISC                         test
System Volume Information    uImage
devicetree.dtb               uramdisk.image.gz
modules
zynq> cd modules/
zynq> ./devtest
This device is opened
0 * 0 = 0 Result Correct!
0 * 1 = 0 Result Correct!
0 * 2 = 0 Result Correct!
0 * 3 = 0 Result Correct!
0 * 4 = 0 Result Correct!
0 * 5 = 0 Result Correct!
0 * 6 = 0 Result Correct!
0 * 7 = 0 Result Correct!
0 * 8 = 0 Result Correct!
0 * 9 = 0 Result Correct!
0 * 10 = 0 Result Correct!
0 * 11 = 0 Result Correct!
0 * 12 = 0 Result Correct!
0 * 13 = 0 Result Correct!
0 * 14 = 0 Result Correct!
0 * 15 = 0 Result Correct!
0 * 16 = 0 Result Correct!
1 * 0 = 0 Result Correct!
1 * 1 = 1 Result Correct!
1 * 2 = 2 Result Correct!
1 * 3 = 3 Result Correct!
1 * 4 = 4 Result Correct!
1 * 5 = 5 Result Correct!
1 * 6 = 6 Result Correct!
1 * 7 = 7 Result Correct!
1 * 8 = 8 Result Correct!
1 * 9 = 9 Result Correct!
1 * 10 = 10 Result Correct!
```

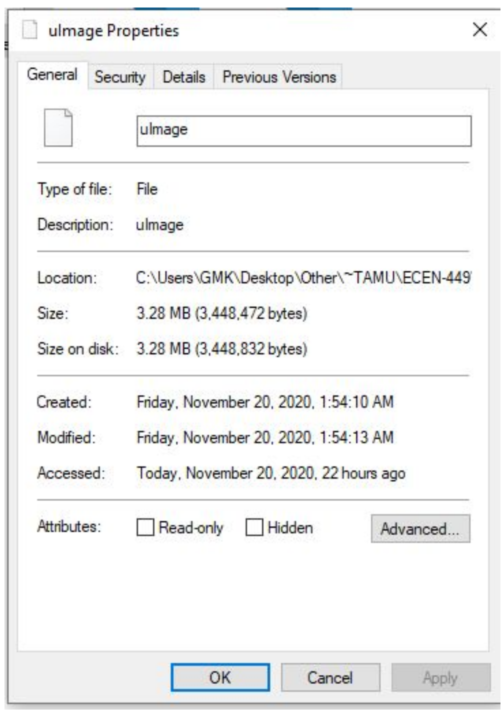
Loaded during boot

```
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ehci-pci: EHCI PCI platform driver
zynq-dr e0002000.ps7-usb: Unable to init USB phy, missing?
usbcore: registered new interface driver usb-storage
mousedev: PS/2 mouse device common for all mice
i2c /dev entries driver
Xilinx Zynq CpuIdle Driver started
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
sdhci-pltfm: SDHCI platform and OF driver helper
sdhci-arasan e0100000.ps7-sdio: No vmmc regulator found
sdhci-arasan e0100000.ps7-sdio: No vqmmc regulator found
mmc0: SDHCI controller on e0100000.ps7-sdio [e0100000.ps7-sdio
] using ADMA
ledtrig-cpu: registered to indicate activity on CPUs
usbcore: registered new interface driver usbhid
usbhid: USB HID core driver
Mapping virtual address...
Physical Address: 43c00000
Virtual Address: 608e0000
Registered a device with dynamic Major number of 245
Create a device file for this device with this command:
'mknod /dev/multiplier c 245 0'.
TCP: cubic registered
NET: Registered protocol family 17
can: controller area network core (rev 20120528 abi 9)
NET: Registered protocol family 29
can: raw protocol (rev 20120528)
can: broadcast manager protocol (rev 20120528 t)
can: netlink gateway (rev 20130117) max_hops=1
zynq_pm_ioremap: no compatible node found for 'xlnx,zynq-ddrc-
a05'
zynq_pm_late_init: Unable to map DDRC IO memory.
Registering SWP/SWPB emulation handler
drivers/rtc/hctosys.c: unable to open rtc device (rtc0)
ALSA device list:
  No soundcards found.
mmc0: new high speed SDHC card at address aaaa
mmcblk0: mmc0:aaaa SL16G 14.8 GiB
 mmcblk0: p1
RAMDISK: gzip image found at block 0
EXT2-fs (ram0): warning: mounting unchecked fs, running e2fsck
 is recommended
VFS: Mounted root (ext2 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 212K (40627000 - 4065c000)
Starting rcS...
++ Mounting filesystem
++ Setting up mdev
++ Starting telnet daemon
++ Starting http daemon
++ Starting ftp daemon
++ Starting dropbear (ssh) daemon
random: dropbear urandom read with 1 bits of entropy available
rcS Complete
zynq>
```

New uImage size

```
[kylanlewis]@apollo3 ~/ECEN_449/Lab7b/linux-3.14> (08:06:36 11/20/20)
:: make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- UIMAGE_LOADADDR=0x8000 uImage
  CHK     include/config/kernel.release
  CHK     include/generated/uapi/linux/version.h
  CHK     include/generated/utsrelease.h
make[1]: `include/generated/mach-types.h' is up to date.
  CALL    scripts/checksyscalls.sh
  CHK     include/generated/compile.h
  CHK     kernel/config_data.h
  Kernel: arch/arm/boot/Image is ready
  Kernel: arch/arm/boot/zImage is ready
  UIMAGE  arch/arm/boot/uImage
Image Name:   Linux-3.18.0-xilinx
Created:      Fri Nov 20 08:07:09 2020
Image Type:   ARM Linux Kernel Image (uncompressed)
Data Size:    2317632 Bytes = 2263.31 kB = 2.21 MB
Load Address: 00008000
Entry Point:  00008000
  Image arch/arm/boot/uImage is ready
```

Old uImage size



Post-Lab Questions :

1. What are the advantages and disadvantages of loadable kernel modules and built-in modules?

   Some advantages of loadable kernel modules are that it can help reduce the size of the kernel, but the disadvantage is that it takes a lot longer to configure the new device module. On the other hand, some advantages of built-in kernel modules are that it speeds things up and devices are ready faster after bootup, but the disadvantage is that it takes a lot more space on the kernel image.

Conclusion:

I certainly learned a lot about device drivers and configuring them in this lab. Additionally this lab was good insight into what you should look for and the kinds of decisions that go into configuring built-in and loadable kernel modules. It's essentially a trade off of time (speed) and memory with this situation.

Code:

Multiplier.c

```
#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 */

/*from xparameters.h*/
#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
*);

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. */

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
    virt_addr = ioremap(PHY_ADDR, MEMSIZE);
    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);
    }
```

```c
    //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). */
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, like "cat
 * /dev/my_chardev".  Link to this function placed in file operations
 * structure for our device file.
 */
static int device_open(struct inode *inode, struct file *file)
{
  printk(KERN_ALERT "This device is 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 "This device is closed\n");
  return 0;
}

/*
 * Called when a process, which already opened the dev file, attempts
 * to read from it.
 */
static ssize_t device_read(struct file *filp, /* see include/linux/fs.h*/
               char *buffer,      /* buffer to fill with
                         data */
               size_t length,     /* length of the
                         buffer  */
```

```
                loff_t * offset)
{
    /*
    * Number of bytes actually written to the buffer
    */
    int bytes_read = 0;
    int i;

    for(i=0; i<length; i++) {
        put_user(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.
 */
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);
        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("ECEN449 Kylan Lewis");
MODULE_DESCRIPTION("Simple multiplier module");

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