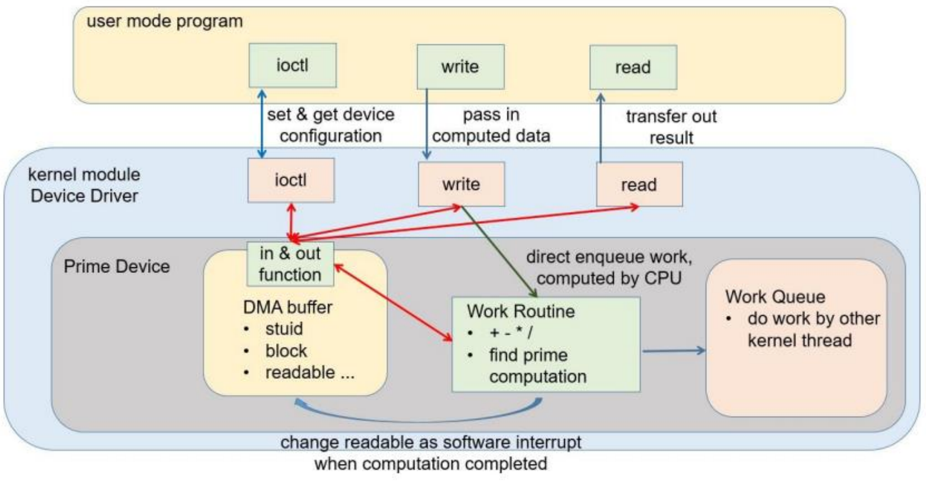
Report

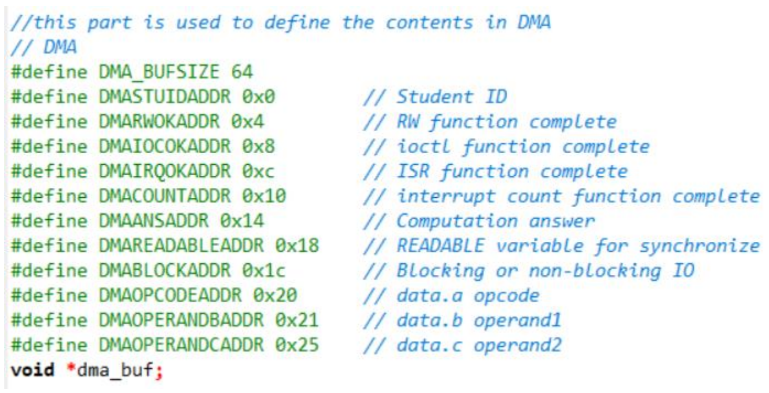
Li Huayue 118010138

1 The thought of design:

After reading the requirements of the assignment, my whole thought is to make a prime device in Linux and implement file operations in kernel module to control this device. I need to finish the functions which are shown in the template. What is more is that I also create some functions of my own to help to achieve these functions. The global view of this project is shown as follows:

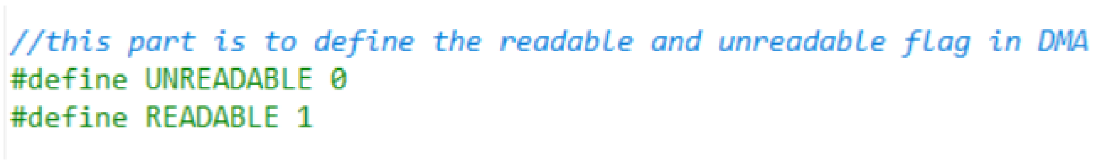


After observation, I find the whole device can be divided into two parts: the DMA part and the work part. In the DMA part, all the space of the DMA has been created and allocated in the template which is shown as follows:



```
//this part is used to define the contents in DMA
// DMA
#define DMA_BUFSIZE 64
#define DMASTUIDADDR 0x0        // Student ID
#define DMARWOKADDR 0x4         // RW function complete
#define DMAIOCOKADDR 0x8        // ioctl function complete
#define DMAIRQOKADDR 0xc        // ISR function complete
#define DMACOUNTADDR 0x10       // interrupt count function complete
#define DMAANSADDR 0x14         // Computation answer
#define DMAREADABLEADDR 0x18    // READABLE variable for synchronize
#define DMABLOCKADDR 0x1c       // Blocking or non-blocking IO
#define DMAOPCODEADDR 0x20      // data.a opcode
#define DMAOPERANDBADDR 0x21    // data.b operand1
#define DMAOPERANDCADDR 0x25    // data.c operand2
void *dma_buf;
```

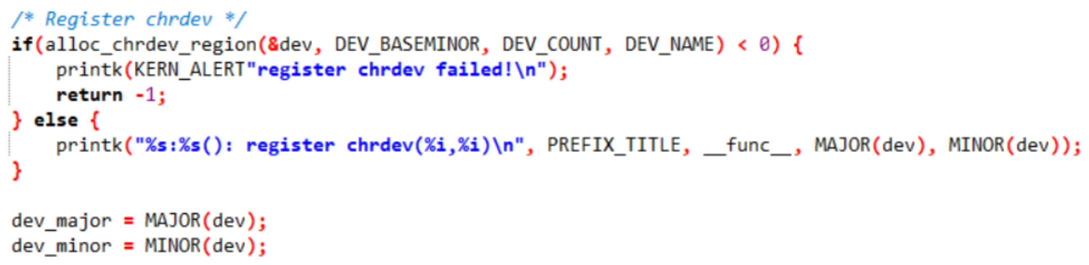
In the readable flag part of DMA, I define two numbers to define "readable" and "unreadable" which is shown as follows:



```
//this part is to define the readable and unreadable flag in DMA
#define UNREADABLE 0
#define READABLE 1
```

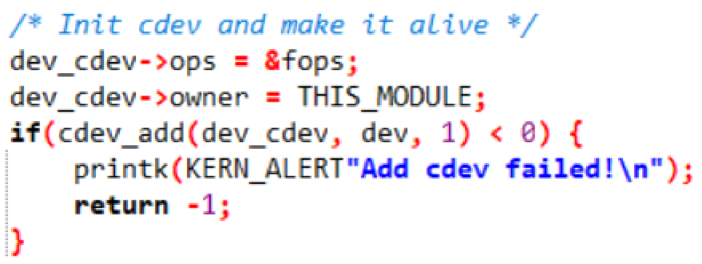
The work part is used to finish the work put into the device one by one and do corresponding operations. Now I will introduce every part of my project following the requirements of this assignment.

I use a function called "*__init init_modules*" to initial the modules and register a character device in it which is shown as follows:



```
/* Register chrdev */
if(alloc_chrdev_region(&dev, DEV_BASEMINOR, DEV_COUNT, DEV_NAME) < 0) {
    printk(KERN_ALERT"register chrdev failed!\n");
    return -1;
} else {
    printk("%s:%s(): register chrdev(%i,%i)\n", PREFIX_TITLE, __func__, MAJOR(dev), MINOR(dev));
}

dev_major = MAJOR(dev);
dev_minor = MINOR(dev);
```

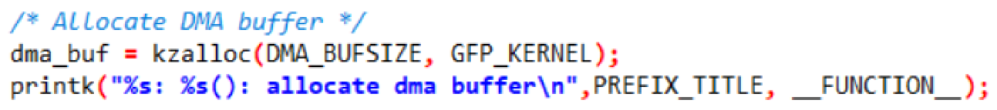
You can see that my function shows whether it succeeds to register a character device or not. After that, I initialize a *cdev* and add it to make it alive which is shown as follows:



```
/* Init cdev and make it alive */
dev_cdev->ops = &fops;
dev_cdev->owner = THIS_MODULE;
if(cdev_add(dev_cdev, dev, 1) < 0) {
    printk(KERN_ALERT"Add cdev failed!\n");
    return -1;
}
```

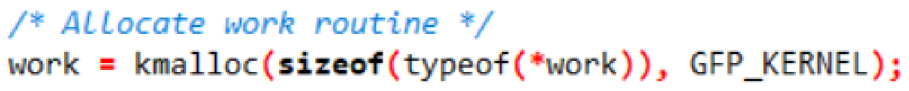
And the program can also show whether it succeeds to initialize a *cdev* or not.

Then I allocate DMA buffer which is shown as follows:



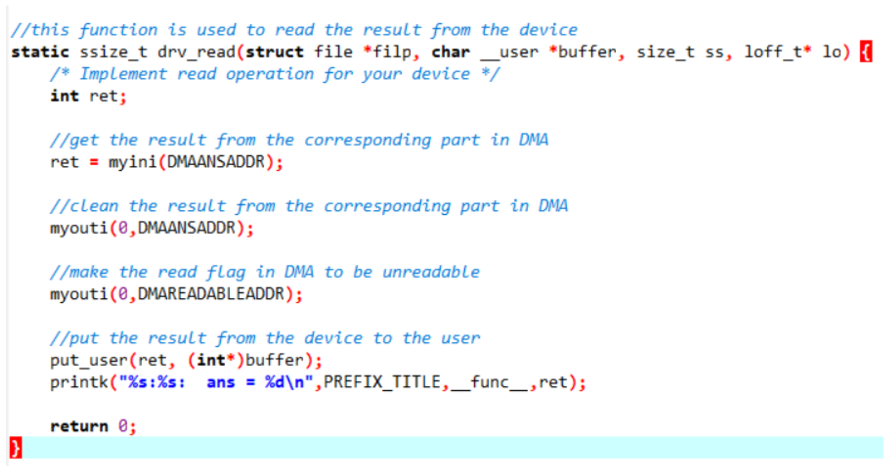
```
/* Allocate DMA buffer */
dma_buf = kzalloc(DMA_BUFSIZE, GFP_KERNEL);
printk("%s: %s(): allocate dma buffer\n",PREFIX_TITLE, __FUNCTION__);
```

Finally, I allocate the work routine which is shown as follows:



```
/* Allocate work routine */
work = kmalloc(sizeof(typeof(*work)), GFP_KERNEL);
```

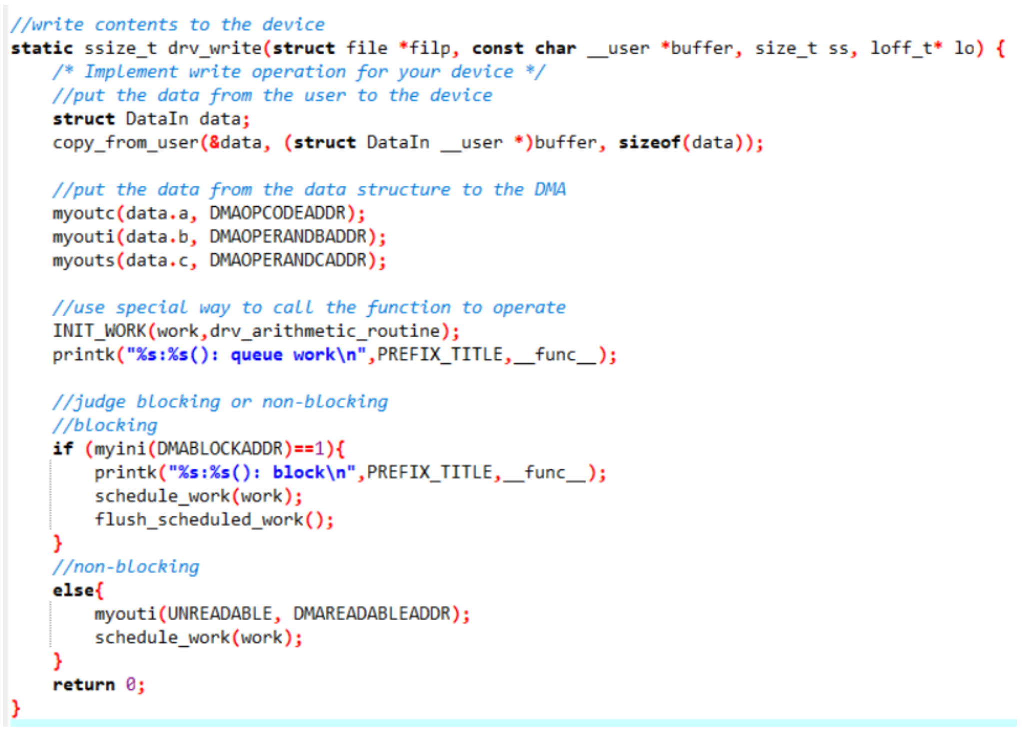
That is the whole work in the “*_init init_modules*” function. Now I will introduce the implementation of read operation for my device. When the user calls the read operation, the program will call the “*drv_read*” function which is shown as follows:



```
//this function is used to read the result from the device
static ssize_t drv_read(struct file *filp, char __user *buffer, size_t ss, loff_t* lo) {
    /* Implement read operation for your device */
    int ret;

    //get the result from the corresponding part in DMA
    ret = myini(DMAANSADDR);

    //clean the result from the corresponding part in DMA
    myouti(0,DMAANSADDR);

    //make the read flag in DMA to be unreadable
    myouti(0,DMAREADABLEADDR);

    //put the result from the device to the user
    put_user(ret, (int*)buffer);
    printk("%s:%s:  ans = %d\n",PREFIX_TITLE,__func__,ret);

    return 0;
}
```

According to the hints in tut, my read function will get the result from the corresponding part in DMA, then clean the result from the corresponding part in DMA, then make the read flag in DMA to be unreadable and finally put the result from the device to the user. That is the whole process of read operation. Now I will introduce my thought of design to implement the write operation for my device.

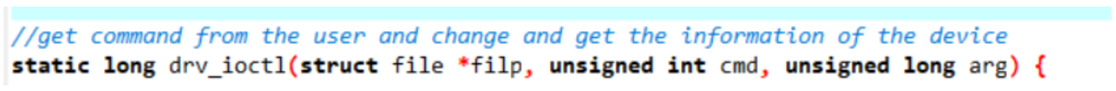
The write operation is a little complex because there are two kinds of writing for the device which are blocking and non-blocking. When the user calls the write operation, the program will call the “*drv_write*” function which is shown as follows:



```c
//write contents to the device
static ssize_t drv_write(struct file *filp, const char __user *buffer, size_t ss, loff_t* lo) {
    /* Implement write operation for your device */
    //put the data from the user to the device
    struct DataIn data;
    copy_from_user(&data, (struct DataIn __user *)buffer, sizeof(data));

    //put the data from the data structure to the DMA
    myoutc(data.a, DMAOPCODEADDR);
    myouti(data.b, DMAOPERANDBADDR);
    myouts(data.c, DMAOPERANDCADDR);

    //use special way to call the function to operate
    INIT_WORK(work,drv_arithmetic_routine);
    printk("%s:%s(): queue work\n",PREFIX_TITLE,__func__);

    //judge blocking or non-blocking
    //blocking
    if (myini(DMABLOCKADDR)==1){
        printk("%s:%s(): block\n",PREFIX_TITLE,__func__);
        schedule_work(work);
        flush_scheduled_work();
    }
    //non-blocking
    else{
        myouti(UNREADABLE, DMAREADABLEADDR);
        schedule_work(work);
    }
    return 0;
}
```

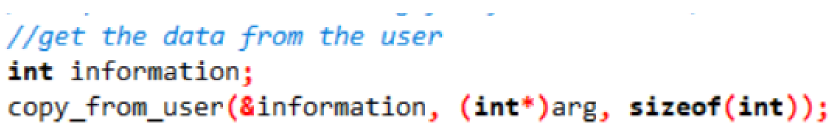
According to the hints in tut, my work function will put the data from the user to the device and store them in a data structure (the data contains two number and one operation), put the data from the data structure to the corresponding positions in the DMA. Then the program will use special way to call the function to arithmetic operate. Finally, the program will judge the status of the device is blocking or non-blocking and then judge whether to put the work into work queue or not. What you should pay attention is that when the status of the device is non-blocking, we need to avoid conflicts between work, so we need to set the readable flag in the DMA to be unreadable and it will recover in latter program. That is the whole process of mine to implement the write operation for my device. Now I will introduce my design to implement $ioctl$ setting for my device.

I think the $ioctl$ setting for my device is the most important part of this project because the $ioctl$ setting can control the status of the whole device. When the user calls the *ioctl* setting, the program will call the "*drv_ioctl*" function which is shown as follows:



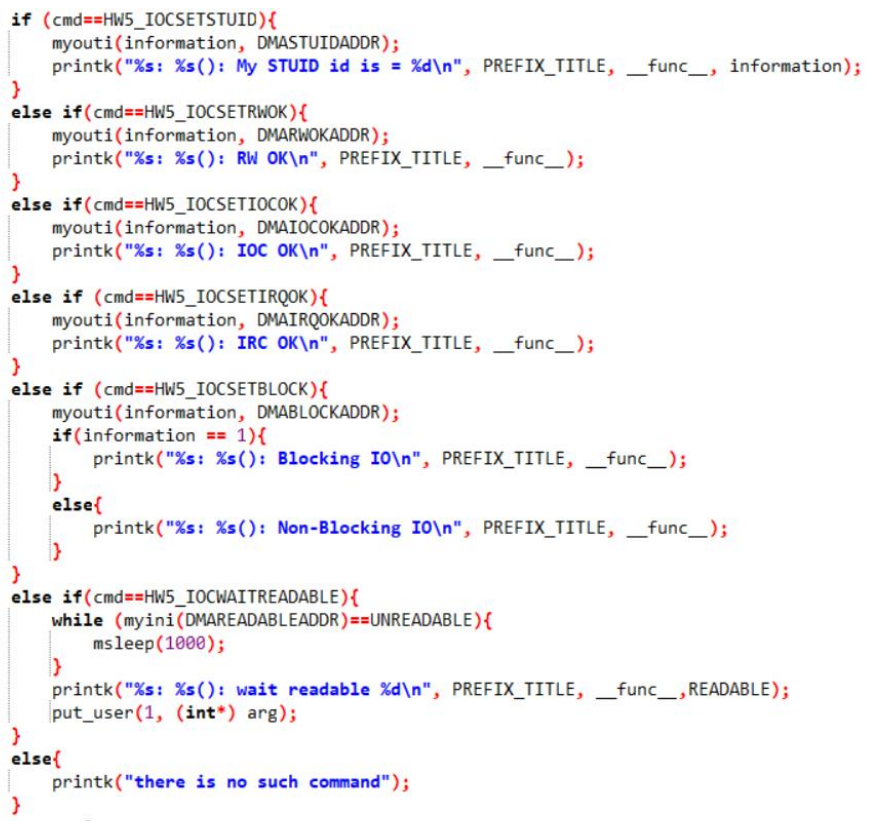
```
//get command from the user and change and get the information of the device
static long drv_ioctl(struct file *filp, unsigned int cmd, unsigned long arg) {
```

According to the hints in tut, the program will first get the data from the user which is shown as follows:



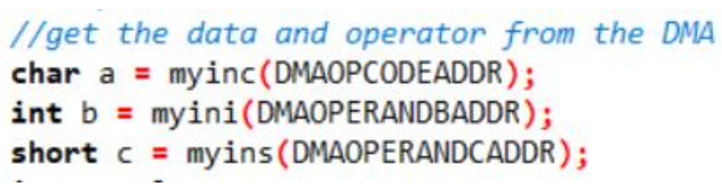
```
//get the data from the user
int information;
copy_from_user(&information, (int*)arg, sizeof(int));
```

Then the program will judge the types of the command and do corresponding work. I use a *if − else if* structure to achieve this function which is shown as follows:



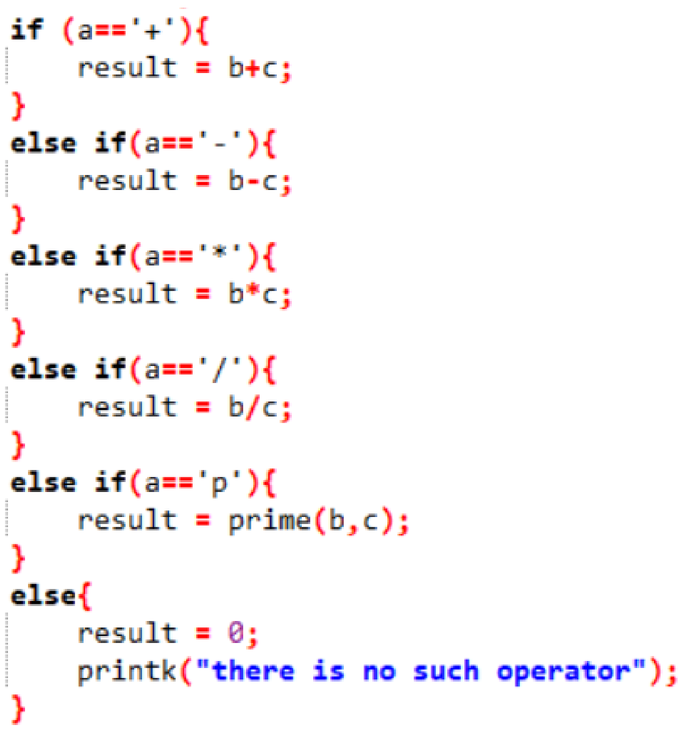
```
if (cmd==HW5_IOCSETSTUID){
    myouti(information, DMASTUIDADDR);
    printk("%s: %s(): My STUID id is = %d\n", PREFIX_TITLE, __func__, information);
}
else if(cmd==HW5_IOCSETRWOK){
    myouti(information, DMARWOKADDR);
    printk("%s: %s(): RW OK\n", PREFIX_TITLE, __func__);
}
else if(cmd==HW5_IOCSETIOCOK){
    myouti(information, DMAIOCOKADDR);
    printk("%s: %s(): IOC OK\n", PREFIX_TITLE, __func__);
}
else if (cmd==HW5_IOCSETIRQOK){
    myouti(information, DMAIRQOKADDR);
    printk("%s: %s(): IRC OK\n", PREFIX_TITLE, __func__);
}
else if (cmd==HW5_IOCSETBLOCK){
    myouti(information, DMABLOCKADDR);
    if(information == 1){
        printk("%s: %s(): Blocking IO\n", PREFIX_TITLE, __func__);
    }
    else{
        printk("%s: %s(): Non-Blocking IO\n", PREFIX_TITLE, __func__);
    }
}
else if(cmd==HW5_IOCWAITREADABLE){
    while (myini(DMAREADABLEADDR)==UNREADABLE){
        msleep(1000);
    }
    printk("%s: %s(): wait readable %d\n", PREFIX_TITLE, __func__,READABLE);
    put_user(1, (int*) arg);
}
else{
    printk("there is no such command");
}
```

What you should pay attention is that when the command is the *HW5_IOCWAITREADABLE*, the program will wait until the readable flag in the DMA becomes readable and then continue to do the following work. After finishing the function every time, the status of the device will change and the user can get the status of the device easily. That is the whole thought of mine to design to implement the $ioctl$ setting for my device. Now I will introduce my thought of design to implement the arithmetic routine for my device. The arithmetic routine function is used to do the corresponding operations and will be called in the write function. According to the hints in tut, this function will first get the data and operator from the DMA which is shown as follows:



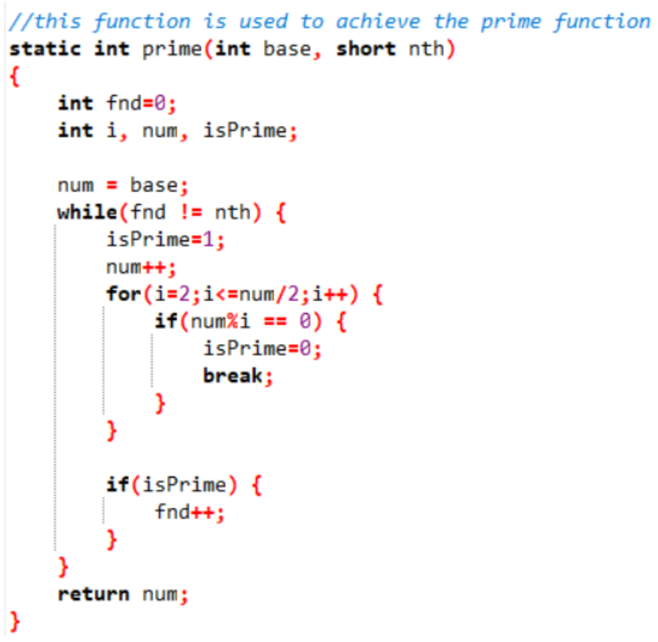
```
//get the data and operator from the DMA
char a = myinc(DMAOPCODEADDR);
int b = myini(DMAOPERANDBADDR);
short c = myins(DMAOPERANDCADDR);
```

Then it will judge the meaning of the operator and then do the corresponding operation:



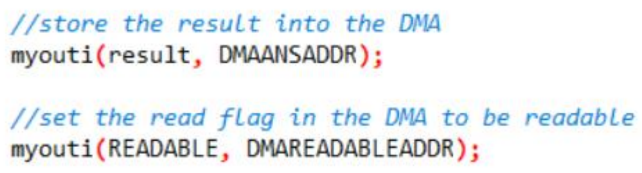
```
if (a=='+'){
    result = b+c;
}
else if(a=='-'){
    result = b-c;
}
else if(a=='*'){
    result = b*c;
}
else if(a=='/'){
    result = b/c;
}
else if(a=='p'){
    result = prime(b,c);
}
else{
    result = 0;
    printk("there is no such operator");
}
```

The prime operation is to find the $c-th$ prime number bigger than b. I create a function called *“prime”* to achieve this function which is shown as follows:



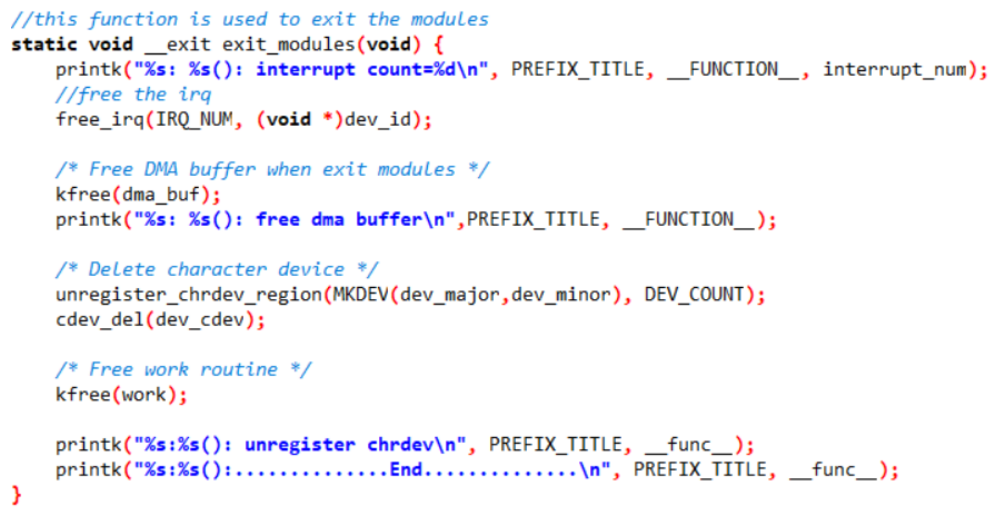
```
//this function is used to achieve the prime function
static int prime(int base, short nth)
{
    int fnd=0;
    int i, num, isPrime;

    num = base;
    while(fnd != nth) {
        isPrime=1;
        num++;
        for(i=2;i<=num/2;i++) {
            if(num%i == 0) {
                isPrime=0;
                break;
            }
        }

        if(isPrime) {
            fnd++;
        }
    }
    return num;
}
```

Then the program will store the result into the DMA and set the read flag in the DMA to be readable to continue the following work:



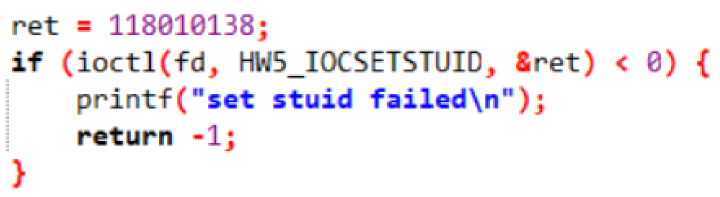
```
//store the result into the DMA
myouti(result, DMAANSADDR);

//set the read flag in the DMA to be readable
myouti(READABLE, DMAREADABLEADDR);
```

That is the whole process of mine to design to implement the arithmetic routine for my device. Now I will introduce the thought of design to implement to complete module exit functions which is shown as follows:



```
//this function is used to exit the modules
static void __exit exit_modules(void) {
    printk("%s: %s(): interrupt count=%d\n", PREFIX_TITLE, __FUNCTION__, interrupt_num);
    //free the irq
    free_irq(IRQ_NUM, (void *)dev_id);

    /* Free DMA buffer when exit modules */
    kfree(dma_buf);
    printk("%s: %s(): free dma buffer\n",PREFIX_TITLE, __FUNCTION__);

    /* Delete character device */
    unregister_chrdev_region(MKDEV(dev_major,dev_minor), DEV_COUNT);
    cdev_del(dev_cdev);

    /* Free work routine */
    kfree(work);

    printk("%s:%s(): unregister chrdev\n", PREFIX_TITLE, __func__);
    printk("%s:%s():..............End..............\n", PREFIX_TITLE, __func__);
}
```

According to the hints in tut, the program will exit the modules after finishing all the work. The program will first free the DMA buffer when exit modules and then delete the character device, finally free the work routine. That is the whole process of mine to design to implement to complete the module exit functions. Then for the own student ID part, I update my student ID in the test case and make it be printed in kernel *ioctl* which is shown as follows:



```
ret = 118010138;
if (ioctl(fd, HW5_IOCSETSTUID, &ret) < 0) {
    printf("set stuid failed\n");
    return -1;
}
```

Finally, I run the test cases to check the write and read operations and find the output of mine is the same as the demo output. That is the whole process of my thought of design.

2 The problems I met in this assignment and the solutions:

(1) The first and biggest problem I met is that I am not familiar with the program in Linux, so I feel a little confused to know the template of this assignment. I read the tut ppt many times and search a lot of resources in the Internet. I get familiar with the template and requirements of this assignment little by little. Finally, this problem is solved.

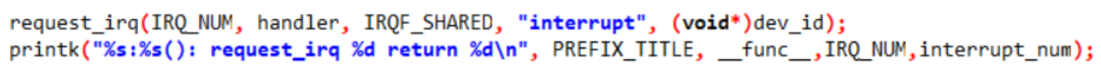
(2) The second problem is that I find the non-blocking write may meet the case that the program wants to read but the work of write has not finished which means conflicts. I feel a little confused to handle this case. After observation, I find that the right of

read is controlled by the readable flag in the DMA. Therefore, I try to set the readable flag to be unreadable and recover it when the work of write finish. Finally, this problem is solved.

(3) The third problem is the use of Linux functions like *get_user*. When I use these functions to get data from the user to the device. I find that this function can not be used for the *DataIn* structure. After reading the ppt of tut, I try to use other Linux functions like *copy_from_user* and it works. Finally, this problem is solved.

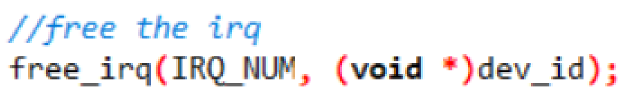
3 Bonus:

The bonus part is to count the interrupt times of input device like keyboard. According to the hints in the introduction and tut ppt. I use "*request_irq*()" in "*module_init*" to add an ISR into an IRQ number's action list which is shown as follows:



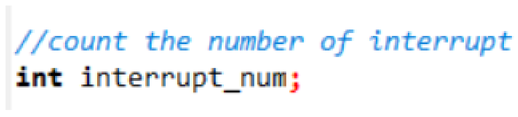
```
request_irq(IRQ_NUM, handler, IRQF_SHARED, "interrupt", (void*)dev_id);
printk("%s:%s(): request_irq %d return %d\n", PREFIX_TITLE, __func__,IRQ_NUM,interrupt_num);
```

Then I use "*free_irq*()" when *module_exit* which is shown as follows:

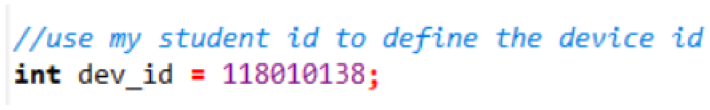


```
//free the irq
free_irq(IRQ_NUM, (void *)dev_id);
```

I use a global variable "*interrupt_num*" to count the interrupt times and use my student ID to define the device ID which is shown as follows:

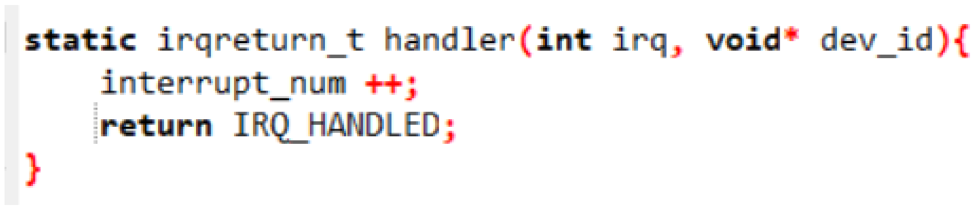


```
//count the number of interrupt
int interrupt_num;
```

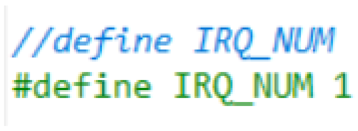


```
//use my student id to define the device id
int dev_id = 118010138;
```

I use a function "*handler*" to add the interrupt number when the keyboard is clicked. And the *IRQ_NUM* is defined to be 1 which is shown as follows:



```
static irqreturn_t handler(int irq, void* dev_id){
    interrupt_num ++;
    return IRQ_HANDLED;
}
```



```
//define IRQ_NUM
#define IRQ_NUM 1
```

Finally, the program will count the interrupt times of input device and print the number in the terminal. That is the whole thought of mine to implement the bonus part.

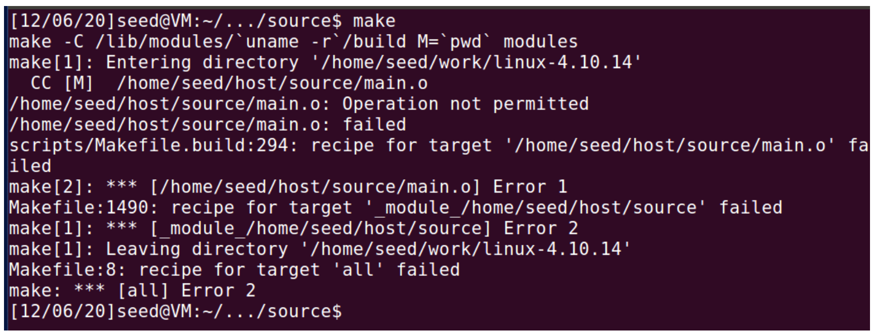
4 The steps to execute the program:

The execution steps of running the program in corresponding environment are shown as follows:

(1) Open the terminal in the *"source"* folder in Linux.

(2) Enter "make" to compile the program.

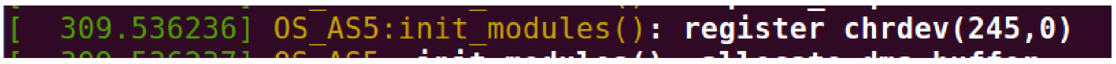
What you should pay attention is that when you enter "make" in the terminal, it may show the error which is shown as follows:



```
[12/06/20]seed@VM:~/.../source$ make
make -C /lib/modules/`uname -r`/build M=`pwd` modules
make[1]: Entering directory '/home/seed/work/linux-4.10.14'
  CC [M]  /home/seed/host/source/main.o
/home/seed/host/source/main.o: Operation not permitted
/home/seed/host/source/main.o: failed
scripts/Makefile.build:294: recipe for target '/home/seed/host/source/main.o' fa
iled
make[2]: *** [/home/seed/host/source/main.o] Error 1
Makefile:1490: recipe for target '_module_/home/seed/host/source' failed
make[1]: *** [_module_/home/seed/host/source] Error 2
make[1]: Leaving directory '/home/seed/work/linux-4.10.14'
Makefile:8: recipe for target 'all' failed
make: *** [all] Error 2
[12/06/20]seed@VM:~/.../source$
```

This error is caused by the Linux and has nothing to do with this program and it does not matter. You can enter the "make" again and it will work.

(3) Enter *dmesg* to get the number of the major device and the minor device.



```
[  309.536236] OS_AS5:init_modules(): register chrdev(245,0)
```

(4) Enter “$sudo\ ./mkdev.sh\ major\_num\ minor\_num$” (for example, “$sudo\ ./mkdev.sh$ 245 0”)

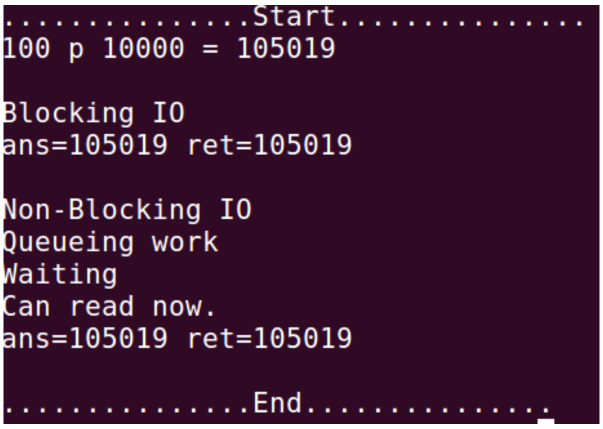
(5) Enter “$./test$”

(6) Enter “$make\ clean$”

(7) Enter “$sudo\ ./rmdev.sh$”

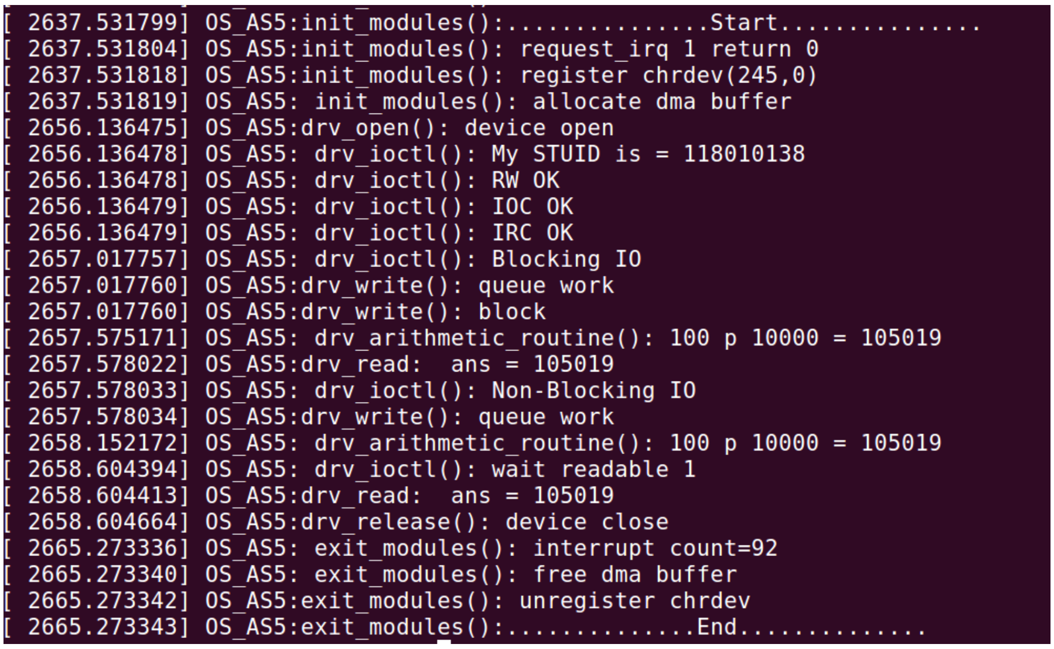
5 The screenshot of the program output:

(1) The user mode output:



```
...............Start...............
100 p 10000 = 105019

Blocking IO
ans=105019 ret=105019

Non-Blocking IO
Queueing work
Waiting
Can read now.
ans=105019 ret=105019

...............End...............
```

(2) The kernel mode output:



```
[ 2637.531799] OS_AS5:init_modules():..............Start..............
[ 2637.531804] OS_AS5:init_modules(): request_irq 1 return 0
[ 2637.531818] OS_AS5:init_modules(): register chrdev(245,0)
[ 2637.531819] OS_AS5: init_modules(): allocate dma buffer
[ 2656.136475] OS_AS5:drv_open(): device open
[ 2656.136478] OS_AS5: drv_ioctl(): My STUID is = 118010138
[ 2656.136478] OS_AS5: drv_ioctl(): RW OK
[ 2656.136479] OS_AS5: drv_ioctl(): IOC OK
[ 2656.136479] OS_AS5: drv_ioctl(): IRC OK
[ 2657.017757] OS_AS5: drv_ioctl(): Blocking IO
[ 2657.017760] OS_AS5:drv_write(): queue work
[ 2657.017760] OS_AS5:drv_write(): block
[ 2657.575171] OS_AS5: drv_arithmetic_routine(): 100 p 10000 = 105019
[ 2657.578022] OS_AS5:drv_read:  ans = 105019
[ 2657.578033] OS_AS5: drv_ioctl(): Non-Blocking IO
[ 2657.578034] OS_AS5:drv_write(): queue work
[ 2658.152172] OS_AS5: drv_arithmetic_routine(): 100 p 10000 = 105019
[ 2658.604394] OS_AS5: drv_ioctl(): wait readable 1
[ 2658.604413] OS_AS5:drv_read:  ans = 105019
[ 2658.604664] OS_AS5:drv_release(): device close
[ 2665.273336] OS_AS5: exit_modules(): interrupt count=92
[ 2665.273340] OS_AS5: exit_modules(): free dma buffer
[ 2665.273342] OS_AS5:exit_modules(): unregister chrdev
[ 2665.273343] OS_AS5:exit_modules():.............End.............
```

6 What did I learn from this assignment:

In this assignment, I have learnt how to make a prime device in Linux and implement file operations in kernel module to control device. I have tried my best to finish this assignment and in this process I get familiar with Linux little by little. I also learn the kernel programming and the structure of the prime device. All in all, I have learnt a lot in this assignment.