Driver_test.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <linux/kernel.h>
#include <linux/poll.h>    /*copy_to_user*/
#include <linux/types.h>    /*size_t*/
#include <linux/cdev.h>
#include <linux/slab.h>
#include <asm/uaccess.h>

MODULE_LICENSE("GPL");      /*内核的模块许可证声明*/

static int global_var=0;
static unsigned int major=250;       /*定义主次设备号*/
static unsigned int minor=0;
static unsigned int devno;
static char *filename="test";          /*定义设备名*/
static struct cdev *mycdev=NULL;

static int mycdevflg=0;
static int devnoflg=0;
static int adddevflg=0;

/* Is device open? Used to prevent multiple access to device */
static int Device_Open = 0;

static ssize_t test_read(struct file *filp, char *buf, size_t len, loff_t *off)
{ /*将内核空间中的global_var  变量复制到用户空间*/
   if(copy_to_user(buf,&global_var,sizeof(int)))
            return -1;
   return sizeof(int);
}
```

```
static ssize_t test_write(struct file *filp, char *buf,size_t len, loff_t *off)
{   /*将用户空间中的指定数据复制到内核空间的global_var 变量*/
   if(copy_from_user(&global_var,buf,sizeof(int)))
     return -1;
 return sizeof(int);
}

static int test_open(struct inode *inodep,struct file *flip)
{   /*该函数通常用来实现对设备的前期初始化，如初始化一些关键寄存器， 使设
    备处于待工作状态。*/
   printk("test open is running!\n");
   if (Device_Open)  /*防止同一设备的多次打开操作*/
          return -EBUSY;
   Device_Open++;
   try_module_get(THIS_MODULE);       /*模块使用计数加1*/
   return 0;
}

static int test_release(struct inode *myinode,struct file *flip)
{   /*  该函数通常用来实现和open操作相反的操作  */
   printk("test release is running!\n");
   Device_Open--;
   module_put(THIS_MODULE);       /*模块使用计数减1*/
   return 0;
}

static struct file_operations test_fops={
     /*  结构体中特定成员赋初值，前面加.     */
     .owner = THIS_MODULE,
     .open = test_open,
     .read = test_read,
     .write = test_write,
     .release = test_release,
};
```

```
static int __init test_init(void)
{    //模块装载函数返回值int，参数void
   int result=-1;
   if(major)
   {       /*  静态设备号申请，申请成功后可以通过cat /proc/devices命令查看  */
           devno=MKDEV(major,minor);    /*将主次设备号,转换为一个dev_t*/
           result=register_chrdev_region(devno,1,filename); /*注册字符设备*/
           devnoflg=1;        /*标识设备号注册成功*/
   }
   else
   {
           result=alloc_chrdev_region(&devno,minor,1,filename); /*动态分配设备号*/
           major=MAJOR(devno);
           devnoflg=1; /*标识设备号注册成功*/
   }
   if(result<0)
   {
           printk("can't register the major num!\n");
           devnoflg=0;        /*标识设备号注册不成功*/
           return result;       /*出错返回,返回值为-1*/
   }

   mycdev=cdev_alloc();    /*分配并返回一个cdev结构*/
   if(mycdev==NULL)
   {       /* cdev结构分配不成功*/
           printk("can't request the memory!\n");
           mycdevflg=0;
   }
   else mycdevflg=1;       /*标识cdev结构分配成功*/

   /*初始化字符设备，并建立cdev和file_operations之间的连接。   */
  /* mycdev->owner=THIS_MODULE;      /* THIS_MODULE相当于this指针，指向驱动所在的模块，用于防止驱动在使用时，模块被卸载  */
   mycdev->ops=&test_fops;*/
   cdev_init(mycdev,&test_fops);
```

```
    /*添加字符设备*/
    result=cdev_add(mycdev,devno,1);
    if(result<0)
    {
        printk("can't add cdev!\n");
        adddevflg=0;
    }
    else    adddevflg=1;     /*标识字符设备添加成功*/

    return 0;
}

static void __exit test_exit(void)
{
    if(adddevflg)
    {   /*若字符设备添加成功，在卸载设备驱动时需删除字符设备*/
        cdev_del(mycdev);
        adddevflg=0;
    }
    if(mycdev)
    {       /*若cdev结构分配成功，在卸载设备驱动时需释放其所占用的内存*/
        kfree(mycdev);
        mycdev=NULL;
    }
    if(devnoflg)
    {       /*若设备号分配成功，在卸载设备驱动时需注销该字符设备*/
        unregister_chrdev_region(devno,1);
        devno=0;
        devnoflg=0;
    }
}

/*声明模块装载和卸载函数*/
module_init(test_init);
module_exit(test_exit);
```

test_app.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>

int main(void)
{
   int fd,num;
  /*O_RDWR:以读写的方式打开文件
    *S_IRUSR:Permits the file's owner to read it.
    *S_IWUSR:Permits the file's owner to write to it.*/
   fd=open("/dev/test",O_RDWR,S_IRUSR|S_IWUSR);

 if(fd != -1)
 {
      read(fd,&num,sizeof(int));    /*从设备中读取数据*/
      printf("the golbal_var is:%d\n",num);

      printf("please input the num written to global_var:\n");
      scanf("%d",&num);
      write(fd,&num,sizeof(int));     /*向设备写数据*/

      read(fd,&num,sizeof(int));      /*从设备中读取数据*/
      printf("the global_var is:%d\n",num);

      close(fd);
   }
   else
          printf("device open failure!!\n");
}
```

Makefile

```
ifeq ($(KERNELRELEASE),)
#定义模块所在目录的变量
PWD := $(shell pwd)
# 定义内核源码所在目录的变量,该目录下的内核源码应该是被移植好的内核源码，
# 并且经过了正确的配置和编译，修改Makefile时需要修改该变量的值。
KERNELDIR ？= /home/linux/workdir/6818/Fs6818/kernel/
# 定义根文件系统的目录，该目录即为NFS挂载的主机目录。模块将被安装在
# 该目录下的lib/modules/<内核版本号>/extra目录下，修改Makefile时需要
# 修改该变量的值。
INSTALLDIR ?= /home/kevin/Workspace/FSC100/rootfs

# 第一个目标，为默认的目标，即执行make modules命令和执行make命令的效果相同。
modules:
    # $(MAKE)相当于make，-C表明进入到一个指定目录进行编译，此时会进入
    # 到内核源码所在的目录，即KERNELDIR所指定的目录进行编译。
    # 进入到内核源码目录进行编译的最主要的效果是KERNELRELEASE变量将
    # 被定义，并且被导出到各个子目录，以便在第二次进入模块所在的目录
    # 进行编译时，ifeq条件不成立。M变量指定了内核源码树外的模块目录，用于
    # 指导编译器从内核源码树目录重新回到模块所在目录进行编译。
    # modules用于指定编译模块，正如make zImage用于编译内核映像一样。
    $(MAKE)  -C  $(KERNELDIR)  M=$(PWD)  modules

# 用于模块的安装，推荐使用该方式，这样可以使用modprobe命令进行模块的装载。
modules_install:
    # INSTALL_MOD_PATH用于指定根文件系统的路径，路径名一定要正确。
    # modules_install用于说明是进行模块安装操作。
    $(MAKE)  -C  $(KERNELDIR)  M=$(PWD) INSTALL_MOD_PATH = $(INSTALLDIR)  modules_install
clean:
    rm -rf *.o *.ko *.mod.c .*.cmd modules.order Module.symvers .tmp_versions

# 第二次进入模块所在的目录进行编译时，由于KERNELRELEASE变量已被定义，
# 所以else条件成立。
```

```
else
# obj-m用于指明相应的文件被编译成模块，正如 obj-y用于指明相应的文件编译进
#  内核映像一样。
      obj-m := fsmod.o

endif
```