哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2022 春季

课程名称：嵌入式计算

实验名称：虚拟设备驱动程序设计

学生班级：1901105

学生学号：190110509

学生姓名：王铭

评阅教师：

报告成绩：

实验与创新实践教育中心制

2022 年 3 月

一、 实验详细设计

1.1 mydev.c 的实现

（1）加载与卸载设备驱动 mydev_init 函数

宏参数以及设备数据结构如图 1.1，1.2 所示，采用静态定义设备号，其中主设备 mydev_major 为 230，mydev_minor 为 0。

①在 mydev_init 函数中，首先调用 kzalloc 对 mydev_devp 初始化，调用 cdev_init 函数对 cdev 进行初始化。

②由于采用静态定义设备号，故调用 register_chrdev_region 函数申请。

③最后调用 cdev_add 函数完成注册。

④mydev_exit 函数中，调用 cdev_del 函数释放占用的设备号，调用 kfree 释放为 mydev_devp 申请的内存空间。

⑤调用 unregister_chrdev_region 注销设备。

具体代码如图 1.3，1.4 所示。

```
#define mydev_SIZE  0x1000
#define MEM_CLEAR 0x1  /* 清空内存的ioctl命令 */
#define mydev_MAJOR 230  /* 主设备号 */
#define mydev_MINOR 0
static int mydev_major = mydev_MAJOR;
static int mydev_minor = mydev_MINOR;
module_param(mydev_major, int, S_IRUGO);
```

图 1.1

```
struct mydev_dev {
    struct cdev cdev;
    unsigned char mem[mydev_SIZE];  /* 申请4KB大小的内存 */
};

struct mydev_dev *mydev_devp;
```

图 1.2

```
static int __init mydev_init(void)
{
    int ret,err;
    dev_t devno = MKDEV(mydev_major, mydev_minor);

   /* 初始化cdev */
    mydev_devp = kzalloc(sizeof(struct mydev_dev), GFP_KERNEL);
    cdev_init(&mydev_devp->cdev, &mydev_fops);
    mydev_devp->cdev.owner = THIS_MODULE;

   /* 获取字符设备号 */
    if (mydev_major)
        ret = register_chrdev_region(devno, 1, "mydev");
    else {
        ret = alloc_chrdev_region(&devno, 0, 1, "mydev");
        mydev_major = MAJOR(devno);
        mydev_minor = MINOR(devno);
    }
    if (ret < 0)
        return ret;

     /* 注册设备 */
    err = cdev_add(&mydev_devp->cdev, devno, 1);
    if (err)
        printk(KERN_NOTICE "Error %d adding mydev",err);
    return 1;
}
```

图 1.3

```
static void __exit mydev_exit(void)
{
   /* 释放占用的设备号 */
    cdev_del(&mydev_devp->cdev);
    kfree(mydev_devp);
   /* 注销设备 */
    unregister_chrdev_region(MKDEV(mydev_major, mydev_minor), 1);
}
```

图 1.4

（2）读函数 read

①在读设备前，首先判断当前指针是否超过了设备允许的最大值，若超过直接返回 0 表示当前指针已经到了设备末尾。

②判断读完后指针是否超过了设备允许的最大值，若是则更新 size 为最大允许读的字节。

③调用 copy_to_user 函数，将内核中设备的数据传给用户空间的缓存，若失败返回-1。

④更新 ppos，返回读取的字节数。

具体代码如图 1.5 所示。

```
ssize_t mydev_read(struct file *filp, char __user * buf, size_t size,loff_t * ppos){
    long p = *ppos;
    if(p >= mydev_SIZE)
        return 0;
    if(p + size > mydev_SIZE)
        size = mydev_SIZE - p;
    if(copy_to_user(buf,mydev_devp->mem+p,size)){
        return -1;
    }
    *ppos += size;
    return size;
}
```

图 1.5

（3）写函数 write

①write 函数的实现与 read 函数类似，首先判断当前指针是否超过了设备允许的最大值，若超过直接返回 0 表示当前指针已经到了设备末尾。

②判断写完后指针是否超过了设备允许的最大值，若是则更新 size 为最大允许写的字节。

③调用 copy_from_user 函数，将来自用户空间的数据传给设备，若失败返回-1。

④更新 ppos，返回写入的字节数。

具体代码如图 1.6 所示。

```
ssize_t mydev_write(struct file *filp, const char __user * buf,size_t size, loff_t * ppos){
    long p = *ppos;
    if(p >= mydev_SIZE)
        return 0;
    if(p + size > mydev_SIZE)
        size = mydev_SIZE - p;
    if(copy_from_user(mydev_devp->mem+p,buf,size)){
        return -1;
    }
    *ppos += size;
    return size;
}
```

图 1.6

（4）ioctl 函数

仅实现了清空 4KB 缓存的 MEM_CLEAR（0x1）命令功能，调用 memset 函数将 4KB 缓存设置为 0.具体实现如图 1.7 所示。

```
long mydev_ioctl(struct file *filp, unsigned int cmd,unsigned long arg)
 {

    switch (cmd) {
    case MEM_CLEAR:
    //clear mem
        memset(mydev_devp->mem,0,mydev_SIZE);
        break;

     default: /* 不能支持的命令 */
        return -1;
    }
    return 0;
 }
```

图 1.7

（5）llseek 函数

根据输入的命令以及 off 参数设置新的指针位置并返回，用 switch-case 结构实现（本实验只用到了 SEEK_SET），具体实现如图 1.8 所示。

```
loff_t mydev_llseek(struct file *filp, loff_t off,int orig){
    loff_t NewPos=0;
    int offset=off;
    switch(orig){
        case SEEK_SET:    //SEEK_SET代表以文件头为偏移起始值
            NewPos=offset;
        break;
        case SEEK_CUR:    //SEEK_CUP代表以当前位置为偏移起始值
            NewPos=filp->f_pos+offset;
        break;
        case SEEK_END:    //SEEK_END代表以文件结尾为偏移起始值
            NewPos=mydev_SIZE+offset;
        break;
        default:
            return -1;
    }
    if(NewPos<0)
        return -1;
    filp->f_pos=NewPos;
    return NewPos;
}
```

图 1.8

（6）open，release 函数以及 file_operations 实现

open、release 函数直接返回 0，根据上述实现的函数填写 file_operations 结构体即可。具体实现如图 1.9 所示。

```
int mydev_open(struct inode* inode,struct file* filp){
    return 0;
}

int mydev_release(struct inode* inode,struct file* filp){
    return 0;
}


 struct file_operations mydev_fops = {
    .owner = THIS_MODULE,
    .llseek = mydev_llseek,
    .read = mydev_read,
    .write = mydev_write,
    .unlocked_ioctl = mydev_ioctl,
    .open = mydev_open,
    .release = mydev_release
};
```

图 1.9

1.2 mydev_test.c 的实现

①调用 open 函数打开设备，用 write 函数向 mydev 设备文件写入“hello world”，写入后初始化 buf，调用 lseek 将指针移至开始位置（即 0）。
②调用 read 函数，将数据从 mydev 设备中读出并打印，初始化 buf 并调用 lseek 将指针移至开始位置（即 0）。
③调用 ioctl 函数，清空内存，再次读数据并打印。
④调用 close 函数关闭设备文件并退出。
具体实现如图 1.10 所示。

```c
int main()
{
    int fp;
    int i;
    char buf[mydev_SIZE];
    fp = open("/dev/mydev", O_RDWR);
    // TODO: 往mydev设备文件写入数据
    strcpy(buf,"hello world");
    write(fp,buf,sizeof(buf));
    memset(buf,0,mydev_SIZE);
    lseek(fp,0,SEEK_SET);
    // TODO: 从mydev设备读出数据
    i = 0;
    while(read(fp,&buf[i++],1)){
    }
    printf("first read:%s\n",buf);
    memset(buf,0,mydev_SIZE);
    lseek(fp,0,SEEK_SET);
    // TODO: 使用IOCTL的MEM_CLEAR命令清空内存
    ioctl(fp,0x1);
    lseek(fp,0,SEEK_SET);
    // TODO: 再次从mydev设备读出数据
    i = 0;
    while(read(fp,&buf[i++],1)){
    }
    printf("second read:%s\n",buf);
    close(fp);
    return 0;
}
```

图 1.10

二、 实验结果截图及分析

在输入 mknod 命令创建 mydev 文件后，用 echo 命令向设备写入“hello world”，调用 cat 命令查看，成功显示出“hello world”，如图 2.1 所示。

```
/ # mknod /dev/mydev c 230 0
/ # echo "hello world" > /dev/mydev
/ # cat /dev/mydev
hello world
/ #
```

图 2.1

运行 mydev_test 文件，可以看到第一次读出的数据为“hello world”，在调用 ioctl 清空内存后，第二次读出的数据为空，如图 2.2 所示。

```
/ # ./mydev_test
first read:hello world
second read:
/ #
```

图 2.2

三、 实验中遇到的问题及解决方法

1.输入 cat 命令查看写入设备的数据时，出现重复读取 hello world 的问题
解决方法:查看源码发现，源码中在调用 read 函数时，若 ppos 为 0，进入 status 为-1 的状态，在此时存在一个 while 循环，在读到文件末尾时才会结束调用 read 函数，故在 read 函数中添加了 ppos 向后移动，同时增加 if 语句判断界限。

```
    if(p >= mydev_SIZE)
        return 0;
    if(p + size > mydev_SIZE)
        size = mydev_SIZE - p;
```

四、 实验收获和建议

本学期嵌入式计算实验以一种较为简单的模式，通过 qemu 模拟开发板帮助我了解了交叉编译以及设备驱动开发的简单流程，帮助我理解了理论课上所讲的知识，加深了对嵌入式开发的理解。
建议之后如果有可能，还是在实体开发板上进行操作更有助于提高实验的积极性，同时增加实验实现的功能和代码量。