I/O 接口驱动程序实验报告

戴文谦
141180014

目录

1 实验环境介绍

1.1 硬件环境

硬件开发环境即为嵌入式实验系统开发板 x210v3。本次实验是实现 I/O 接口驱动程序，故最重要的硬件资源是开发板提供的 GPIO 接口及周边外设电路。本次实验用到的硬件资源如下：

- 板载三盏 LED 灯及周边电路
- 板载六个控制按键及周边电路
- 核心处理器：S5PV210，主频 1GHz，含多个输入/输出/功能型 I/O[1]

[1]本实验用到 GPJ 型、GPH 型

- 内存：512MB
- RS–232 接口 (UART2)
- RJ45 接口 (有线以太网 DM9000CEP)
- 功能按键 (开机/复位)

1.2 软件环境

软件环境主机端主要有以下配置：

- Ubuntu 桌面环境（包含 minicom 工具）。
- 位于主机上的 busybox 1.20.2[1]，作为嵌入式系统使用的 nfs 文件系统[2]
- Linux 内核 3.0.8 版本
- 针对 ARM 的交叉编译工具链 2016.08 版本，并加入环境变量中[3]

在嵌入式开发板 x210v3 端，厂商预装了底层的 bootloader 和 Android4.0.4 系统。开发板的 Bootloader 环境在嵌入式系统内核与文件系统实验中已经配置好网络地址和 nfs 文件系统启动的环境变量，可在本次实验中直接使用。本次实验与开发板板载 FLASH 芯片中的 Android 系统无关。

2 实验原理说明

2.1 内核模块的相关概念

模块是指运行时向内核中添加的代码。每个模块由目标代码组成，可以由 insmod 程序动态链接到正在运行的内核，也可以由 rmmod 程序解除链接。内核模块化一方面可以在运行时动态的加入新功能而无需重新编译内核，为系统提供了极强扩展性；另一方面还意味着可以减小内核的体积，待到用相应功能时再加载内核模块，具有很强的灵活性。在编译 Linux 时，通过 menuconfig 命令可选择配置的内核模块中就包含文件系统、内存管理、网络功能、驱动程序等不同类型的模块。

内核模块不同于应用程序。模块运行在内核空间中，而应用程序运行在用户空间中，因此模块具有对系统极大的操作权限。对于应用程序，打开后单独执行一个任务，而内核模块则是通过入口函数预先注册自己，成为系统调用。在用户程序或内核运行需要用到该模块时再通过系统调用来使用模块提供的功能。由于内核模块以类似动态库的形式存在，分析编译的过程可知它仅被链接到内核，只能调用内核导出的函数，不能使用一般标准 C 库的函数。

内核模块的编译也与一般应用程序不同，需要相应版本内核的源码，并需要指定专门的内核模块编译命令嵌套调用内核源码下的 Makefile。

[1] 安装地址为/srv/nfs4/nfslinux

[2] 已配置了开机自动挂载主机 nfs 文件夹至/mnt

[3] 包含 GNU 工具：gcc compiler，binutil，C library，C header

2.2 设备驱动程序的原理

设备驱动程序属于内核模块的一种。概括来说，设备驱动程序提供了用户对硬件的操作方法，是应用和实际设备之间的一个软件层。用户在用户空间通过一组标准化的对硬件的调用来通知内核，而内核则用其中的设备驱动模块来将用户的操作映射到实际对硬件的特定操作上。

对于 linux 操作系统，在 VFS 文件系统的机制下，有相当多的功能都抽象成文件，操作相应的功能即需要打开、操作相应的文件，因而对于用户空间的应用程序，linux 系统提供操作硬件的方法是基于文件的，即利用文件来代表硬件：在应用中通过设备文件名打开设备后，文件描述符对应上相应的模块，其它操作再通过文件描述符对应找到文件的操作方式，对设备进行操作。

Linux 操作系统将设备分为三种设备：字符设备、块设备和网络接口，其中前两种在操作系统中都以特殊文件的形式存在。对于字符设备和块设备，在用户空间对硬件的操作实际上等同于用户对设备文件的操作。在内核层面，用作设备驱动功能的模块可以调用针对字符设备和块设备的专用内核函数来实现对设备注册等功能。为分析设备驱动程序的原理，此处以实验涉及的字符设备为例，应分析以下几个内核文件：[1]

```
linux-2.6.35.6/include/linux/fs.h
linux-2.6.35.6/fs/char_dev.c
```

```
struct file {
    /* 指明对文件的操作方式 */
    const struct file_operations    *f_op;
    /* 提供了一个可以保存设备信息对应内存空间的指针 */
    void                *private_data;
};

struct file_operations {
    /* 指明文件操作方式属于哪个模块，用户层面打开该种设备文件时即对应到该模块 */
    struct module *owner;
    /* 用户对于设备文件、设备文件描述符的操作会由以下函数指针指向的函数来执行 *
     * file *，inode*均为指向打开设备文件的相应参数，___user*为由用户空间传入 *
     * 的数据指针。对于read和write函数，第三个参数为传入数据的字节数，第四个 *
     * 参数，指明用户在设备文件中进行存取操作的位置                    */
    ssize_t (*read) (struct file *, char ___user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char ___user *, size_t, loff_t *);
    /* ioctl指针在3.0.8内核版本中取消，不影响用户的调用但影响驱动程序的编写 *
     * 倒数第二个参数是传入的控制字，最后一个参数是指向传入内容内存的指针 */
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

[1]以 2.6.35 版本的内核文件来分析，3.0.8 版本中有少量改动，将在注释中说明

```
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*release) (struct inode *, struct file *);
};
```

注册字符设备和注销字符设备的函数为：

```
static inline int register_chrdev(unsigned int major, const char *name,
                  const struct file_operations *fops);
static inline void unregister_chrdev(unsigned int major, const char *name);
```

上两个函数实际上会进一步调用 `char_dev.c` 文件中的函数。阅读该文件可知，注册字符设备的函数会将指定的操作方式对应上相应的字符设备主设备号并为主设备号分配 0-255 作为可用次设备号，因而驱动程序在将自身注册后，在用户程序中打开字符设备就会通过主设备号来寻找已经注册的对应的文件操作；如果选择主设备号 0，该函数将返回自动分配的可用的设备号，如果选择其他主设备号，返回 0 表示分配成功。注销字符设备的函数做的操作则相反。

在 `<linux/ioctl.h>` 中还提供了构造 ioctl 控制字的宏定义：`_IOW`，`_IOR`，`_IOWR` 等，方便根据数据传输的大小、方向、传输的设备类型定义出不同的控制字。

在驱动程序进行具体操作时，常常需要和用户空间传入的参数进行交互，但用户空间传入的内存空间指针不能在内核空间中使用，反之亦然，故需要专门在两种空间之间拷贝内存内容的函数。在 `<asm/uaccess.h>` 中提供了以下内核函数：

```
unsigned long copy_to_user(void * to, const void *from, unsigned long count);
unsigned long copy_from_user(void * to, const void *from, unsigned long count);
```

2.3 接口电路与 GPIO 口介绍

实验中的驱动程序的操作对象是板载的三个 LED 灯和六个非固定功能按键，这些硬件都连接核心处理器 S5PV210 的 GPIO 接口，因此可以通过控制 GPIO 接口来控制这些硬件。

为了研究该驱动什么信号，驱动的信号应该在哪些 GPIO 口输出，应该查阅嵌入式开发板的电路图。涉及 LED 和按键部分的电路如图1所示：

本次实验中涉及的的几个 GPIO 口特性和控制方法可以在三星提供的 S5PV210 用户手册中找到，在此进行介绍：[1]

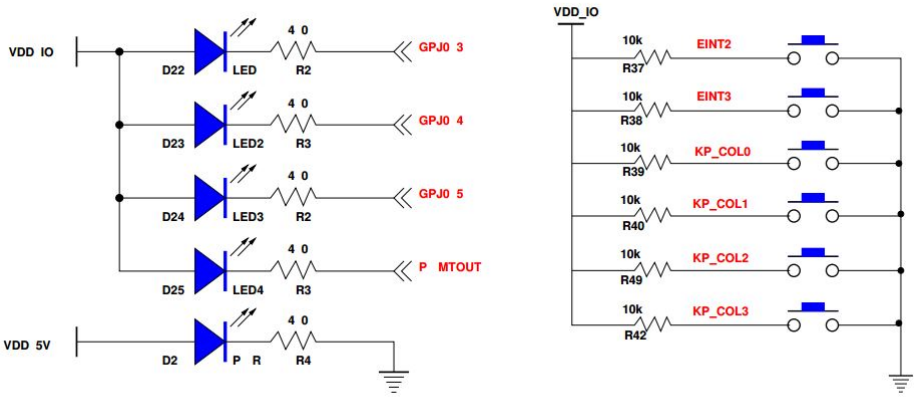
寄存器	地址	功能描述
GPJ0CON	0xE020_0240	GPJ0 配置寄存器
GPJ0DAT	0xE020_0244	GPJ0 数据寄存器
GPJ0PUD	0xE020_0248	GPJ0 上拉/下拉寄存器
GPH0CON	0xE020_0C00	GPH0 配置寄存器
GPH0DAT	0xE020_0C04	GPH0 数据寄存器
GPH0PUD	0xE020_0C08	GPH0 上拉/下拉寄存器

[1]只介绍和实验相关功能

寄存器	地址	功能描述
GPH2CON	0xE020_0C40	GPH2 配置寄存器
GPH2DAT	0xE020_0C44	GPH2 数据寄存器
GPH2PUD	0xE020_0C48	GPH2 上拉/下拉寄存器

寄存器	位域	功能描述	初始状态
GPJ0CON[7]	[31:28]	0000 = Input 0001 = Output	0000
GPJ0CON[6]	[27:24]	0000 = Input 0001 = Output	0000
GPJ0CON[5]	[23:20]	0000 = Input 0001 = Output	0000
GPJ0CON[4]	[19:16]	0000 = Input 0001 = Output	0000
GPJ0CON[3]	[15:12]	0000 = Input 0001 = Output	0000
GPJ0CON[2]	[11:8]	0000 = Input 0001 = Output	0000
GPJ0CON[1]	[7:4]	0000 = Input 0001 = Output	0000
GPJ0CON[0]	[3:0]	0000 = Input 0001 = Output	0000
GPJ0DAT	[7:0]	端口配置为输入时为读入引脚电平，配置为输出时为写出引脚电平，配置为功能引脚时寄存器值不确定	0x00
GPJ0PUD[n]	[2n+1:2n] n=0-7	00 = 关闭上拉、下拉 01 = 打开下拉 10 = 打开上拉 11 = 保留	0x5555
GPH0CON[2]	[11:8]	0000 = Input 0001 = Output 1111 = EXT_INT[2]	0000
GPH0CON[3]	[15:12]	0000 = Input 0001 = Output 1111 = EXT_INT[3]	0000
GPH0DAT	[7:0]	端口配置为输入时为读入引脚电平，配置为输出时为写出引脚电平，配置为功能引脚时寄存器值不确定	0x00
GPH0PUD[n]	[2n+1:2n]	00 = 关闭上拉、下拉	0x5555

寄存器	位域	功能描述	初始状态
	n=0-7	01 = 打开下拉 10 = 打开上拉 11 = 保留	
GPH2CON[0]	[3:0]	0000 = Input 0001 = Output 0011 = KP_COL[0]	0000
GPH2CON[1]	[7:4]	0000 = Input 0001 = Output 0011 = KP_COL[1]	0000
GPH2CON[2]	[11:8]	0000 = Input 0001 = Output 0011 = KP_COL[2]	0000
GPH2CON[3]	[15:12]	0000 = Input 0001 = Output 0011 = KP_COL[3]	0000
GPH2DAT	[7:0]	端口配置为输入时为读入引脚电平，配置为输出时为写出引脚电平，配置为功能引脚时寄存器值不确定	0x00
GPH2PUD[n]	[2n+1:2n] n=0-7	00 = 关闭上拉、下拉 01 = 打开下拉 10 = 打开上拉 11 = 保留	0x5555



(a) LED 控制信号 (b) 按键控制信号

图 1: LED 和按键部分的电路

3 实验过程与内容

本次实验的文件存放在 ~/dwq/github/kernel_learn/driver 下，利用 git 进行版本控制，具体细节不再赘述。[1]

3.1 内核模块与用户程序共同生成的方法

在实验原理中提到，内核模块的生成与用户程序的生成方式有所不同，故此次实验采取的方式是使用一个单独的命令 make program 生成用户程序，其它部分利用内核模块生成的规范格式来完成，直接输入 make 就可以生成内核模块。下为本实验用到的 Makefile，以说明程序的编译结构。每次生成新程序后程序后只需在终端下输入 make cp 即可拷贝内核模块与用户程序到开发板。

```
MODULE = xled.o xkeypad.o
CC = arm-linux-gcc
PROGRAM = ledread ledwrite ledioctl keypadread mixtest
KMODULE = xled.ko xkeypad.ko
ifneq ($(KERNELRELEASE),)
obj-m    := $(MODULE)
else
KDIR    := /home/student/dwq/linux-3.0.8/
PWD    := $(shell pwd)
default:
    $(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
endif

program : $(PROGRAM)
ledread : ledread.o
    $(CC) -o $@ $<
...
%.o : %.c
    $(CC) -c  $<

.PHONY : clean cp
clean :
    rm -f *.o *.ko
cp :
    cp $(PROGRAM) /srv/nfs4/driver
    cp $(KMODULE) /srv/nfs4/driver
```

实际上，除 .ko 文件外内核模块编译还会生成其它一些文件，这也与一般用户程序不同。

[1]报告中由于篇幅、排版、后续优化等原因对原始代码有一定删减改动

3.2 LED 灯设备驱动实现与测试

经过实验发现，在目标机上动态加载的模块若使用自动生成设备号的方式，首个模块生成的主设备号为 250，后续生成的主设备号默认递减。故此处在主机操作，在 nfs 文件系统的根目录上建立 LED 灯设备文件后开始实验：

```
$ mknod /srv/nfs4/nfslinux/xled c 250 0 -m 666
```

首先对于需要用到的一些常数和宏进行定义：

```
/* GPIO.h */
//定义一些用于ioctl方法的控制字
#define GPIO_IOC 233
#define GPIOSET _IOW(GPIO_IOC, 0, gpio_info)
#define GPIOGET _IOR(GPIO_IOC, 0, gpio_info)
#define GPIORW _IOWR(GPIO_IOC, 0, gpio_info)
#define XLEDBLANK _IOW(GPIO_IOC, 1, int)
//定义GPIO操作用到的寄存器物理地址
#define GPJ0CON 0xE0200240
...
#define GPH2PUD 0xE0200C48

#define ALLIOIN 0x00000000
#define ALLIOOUT 0x11111111
//提高程序可读性的置位宏
#define SETBIT(n) (1 << n)
//实际上是由后续实验测定得到的按键与键盘read方法读出值的映射关系
#define LEFTBUTTON SETBIT(4)
#define DOWNBUTTON SETBIT(5)
#define UPBUTTON SETBIT(0)
#define RIGHTBUTTON SETBIT(1)
#define BACKBUTTON SETBIT(2)
#define MENUBUTTON SETBIT(3)
//记录寄存器物理地址映射的虚拟地址
//程序不做操作的情况下读取类寄存器内容会随状态而变，故变量应加上volatile关键字
typedef struct gpio_struct{
    volatile int *pCtrl[4];
    volatile int *pData[4];
    volatile int *pPut[4];
}gpio_t;
//GPIOSET，GPIOGET控制字传输的结构，用于读取或改变寄存器状态
typedef struct gpio_struct_info{
```

```
    int Ctrl;
    int Data;
    int Put;
}gpio_info;
```

对于 LED 灯驱动，将仔细注释代码以说明驱动程序的细节。可以成功运行的驱动程序代码如下：

```
/* xled.h */
//根据电路图和端口功能表定义led用到的端口(GPJ0[3]–[5])
#define XLEDIO 0x00111000;
#define XLEDSIGN (1 << 3) | (1 << 4) | (1 << 5)
...
struct file_operations xled_fop ={
    .owner = THIS_MODULE,
    .open = xled_open,
    .release = xled_release,
    .read = xled_read,
    .write = xled_write,
    //3.0内核中没有ioctl指针，一般以unlocked_ioctl代替
    //compat_ioctl是在64位系统下运行32位程序时才会用到，嵌入式中不考虑
    .unlocked_ioctl = xled_ioctl
};
/* xled.c */
static dev_t major;
//入口函数，加载模块时调用，负责注册字符设备
int init_module(void){
    if (!( major = register_chrdev(0, "xled", &xled_fop))){
        printk("unable to get device number!\n");
        return -1;
    }
    printk("xled module installed, major device number is %u\n", major);
    return 0;
}
//卸载模块或加载模块失败时调用
void cleanup_module(void){
    unregister_chrdev(major, "xled");
    printk("xled module removed\n");
}
//打开设备文件，一般在此函数中对设备进行初始化
int xled_open(struct inode *inode, struct file *filp ){
```

```
    gpio_t *gpio;
    //利用文件结构体提供的指针保存指向设备信息的指针
    filp->private_data = kmalloc(sizeof(gpio_t), GFP_KERNEL);
    gpio = (gpio_t *)(filp->private_data);
    //将物理地址映射到内核空间虚拟地址并保存，且设置I/O口状态
    gpio->pCtrl[0] = ioremap(GPJ0CON, 4);
    gpio->pData[0] = ioremap(GPJ0DAT, 4);
    *(gpio->pCtrl[0]) = XLEDIO;
    //增加模块调用次数
    try_module_get(THIS_MODULE);
    return 0;
}
//仅在最后一次close设备文件时调用，为测试程序方便，关闭设备时不还原led灯全灭状态
int xled_release(struct inode *inode, struct file *filp){
    gpio_t *gpio = (gpio_t *)filp->private_data;
    //解除映射并释放占用的内存空间
    iounmap(gpio->pCtrl[0]);
    iounmap(gpio->pData[0]);
    kfree(gpio);
    //减少模块调用次数
    module_put(THIS_MODULE);
    return 0;
}
//字符设备read方法和write方法都可以是堵塞的，可以通过cat和echo命令在终端下调用方法
//led灯的读写操作本身都很简单，不需要堵塞，如有需要则人为设计消息队列以实现堵塞
//三个led灯的状态少，写入方法强调不改变其它GPIO位状态，读取时直接丢弃其它位数据
//由于led灯是共阳极驱动，低电平对应led灯亮，不过读写方法均以1表示led灯亮即低电平
//led灯完全由用户输入决定状态，一个正确的工作模式是读取到的数据即是上次写入的数据
//以上思路在按键的操作上也有体现
ssize_t xled_read(struct file *filp, char *buff, size_t count, loff_t *f_pos){
    gpio_t *gpio = (gpio_t *)filp->private_data;
    char c = (char)*(gpio->pData[0]);
    c &= ~(XLEDSIGN);
    copy_to_user(buff, &c, 1);
    //对于该简单设备的读操作无论如何都能成功读出1字节
    return 1;
}
ssize_t xled_write(struct file *filp, char *buff, size_t count, loff_t *f_pos){
    gpio_t *gpio = (gpio_t *)filp->private_data;
    char c,tmp;
```

```
    copy_from_user(&c, buff, 1);
    c = ~c;
    c &= (XLEDSIGN);
    tmp = *(gpio->pData[0]);
    tmp &= ~(XLEDSIGN);
    *(gpio->pData[0]) = tmp;
    *(gpio->pData[0]) |= c;
    return 1;
}
//ioctl方法一般用于改变设备状态，不过led灯状态少且设备具体化后gpio口可调整空间很小
long xled_ioctl(struct file *filp , unsigned int cmd, unsigned long arg){
    gpio_t *gpio_p = (gpio_t *)filp->private_data;
    gpio_info gpio;
    gpio.Data = *(gpio_p->pData[0]);
    gpio.Ctrl = *(gpio_p->pCtrl[0]);
    switch (cmd){
    //设想是改变、获取整个io口的状态，但这种设计太过简单且是否合理还需要探讨
    case GPIOSET:
        copy_from_user(&gpio, (void *)arg, sizeof(gpio));
        *(gpio_p->pCtrl[0]) = gpio.Ctrl;
        *(gpio_p->pData[0]) = gpio.Data;break;
    case GPIOGET:
        copy_to_user((void *)arg, &gpio, sizeof(gpio));break;
    //关闭所有LED灯
    case XLEDBLANK:
        *(gpio_p->pCtrl[0]) = XLEDIO;
        *(gpio_p->pData[0]) |= XLEDSIGN;break;
        default:return -ENOTTY;break;
    }
    return 0;
}
//注明一些模块信息，若不注明编译器会报警告
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("xLED DRIVER");
MODULE_AUTHOR("141180014");
```

将模块拷贝到目标机后，执行以下命令加载、卸载模块，有以下实验现象：

```
# cd /mnt/driver
# insmod xled.ko
xled module installed, major device number is 250
```

```
# rmmod xled
xled module removed
```

由于 busybox 的配置选项，导致第一次利用 `rmmod` 命令卸载模块时会出现无法卸载的现象：

```
# rmmod xled
rmmod: chdir(/lib/modules): No such file or directory
rmmod: chdir(3.0.8): No such file or directory
```

解决方法为在主机下执行以下操作后即可在开发板上卸载模块：

```
$ cd /srv/nfs4/nfslinux/lib
$ mkdir modules
$ cd modules
$ mkdir 3.0.8
```

设计了以下测试程序对 led 驱动程序的功能进行检测：[1]

1. `ledwrite`：向 led 写入 0x18，间隔两秒后再向 led 写入 0x20，并分别打印在终端。实验现象是先亮起左侧两盏 led 灯，两秒后，改为亮最右一盏灯，与程序预期效果一致，终端显示也与先前描述一致。

2. `ledread`：读取 led 端口情况，间隔两秒后再次读取，并分别打印在终端。在灯全灭时，终端两次显示均为 `read = 0x0`，在执行 `ledwrite` 后测试，两次打印均为`read = 0x20`，与预期效果一致。

3. `ledioctl`：利用 ioctl 方法读取 gpio 口的实际寄存器值，两秒后再将三个 led 对应 gpio 口的值根据格式置 0，最后再过两秒写入 `XLEDBLANK` 关键字。在执行 `ledwrite` 后测试，终端第一次显示 `GPJ0 = 0x18`，第二次显示 `xled light on` 并开启三个 led 灯，最后显示 `xled blank` 并将所有 led 灯熄灭，完全通过测试。

3.3 按键设备驱动实现与测试

在 nfs 文件系统的根目录上建立按键设备文件后开始实验：[2]

```
$ mknod /srv/nfs4/nfslinux/xled c 249 0 -m 666
```

阅读三星手册，发现虽然 S5PV210 的部分 GPIO 口原生支持矩阵键盘功能，但实验板的电路决定了无法开启该功能[3]，连接按键的 GPIO 也不一定具备中断功能，故所有连接按键的 GPIO 口都应该配置为常规的电平输入口，通过查询方式来获得按键状态。可以成功运行的驱动程序代码如下：[4]

[1] 代码较为简单不再赘述

[2] 默认已经加载了 led 设备模块，因此会自动分配新设备号 249

[3] 具有 KP_ROW 功能的行线与按键无连接，开启了功能也无意义

[4] 为节省篇幅，与 LED 灯驱动程序类似处省略

```
/* xkeypad.h */
//指定用到（即需上拉）的IO口
#define KEYPADH0PUT SETBIT(5) | SETBIT(7)
#define KEYPADH2PUT SETBIT(1) | SETBIT(3) | SETBIT(5) | SETBIT(7)
//规定读出值的有效位，其它位的读出值将舍弃
#define KEYPADH0_MASK 0x0C
#define KEYPADH2_MASK 0x0F
//忽略函数声明
//对于查询法使用的键盘，只有open、read、release、ioctl方法有意义
//由于时间关系，没有规划一个较好的ioctl方法，故直接取消了该方法
struct file_operations xkeypad_fop ={
    .owner = THIS_MODULE,
    .open = xkeypad_open,
    . release  = xkeypad_release,
    .read = xkeypad_read,
};
/* xkeypad.c */
int xkeypad_open(struct inode *inode, struct file *filp){
     filp ->private_data = kmalloc(sizeof(gpio_t), GFP_KERNEL);
    gpio_t *gpio = (gpio_t *)(filp->private_data);
    gpio->pCtrl[0] = ioremap(GPH0CON, 4);
    gpio->pData[0] = ioremap(GPH0DAT, 4);
    gpio->pPut[0] = ioremap(GPH0PUD, 4);
    gpio->pCtrl[2] = ioremap(GPH2CON, 4);
     ...
    *(gpio->pCtrl[0]) = ALLIOIN;
    *(gpio->pPut[0]) = KEYPADH0PUT;
    *(gpio->pCtrl[2]) = ALLIOIN;
    *(gpio->pPut[2]) = KEYPADH2PUT;
     ...
}
ssize_t xkeypad_read(struct file *filp, char *buff, size_t count,
loff_t *f_pos){
    gpio_t *gpio = (gpio_t *)filp->private_data;
    //读出功能的意义设定为低6位为按键是否按下，按下为1，剩余2位直接为0
    char keyh0 = ~(char)*(gpio->pData[0]) & KEYPADH0_MASK;
    char keyh2 = ~(char)*(gpio->pData[2]) & KEYPADH2_MASK;
    char c = (keyh0 << 2) | keyh2;
    copy_to_user(buff, &c, 1);
    return 1;
```

}

将模块拷贝到目标机后，执行以下命令加载、卸载模块，有以下实验现象：

```
# cd /mnt/driver
# insmod xkeypad.ko
xkeypad module installed, major device number is 249
# rmmod xkeypad
xkeypad module removed
```

执行间隔两秒读一次键盘状态，共读取两次的简单测试程序 keypadread，代码较为简单不再赘述。在没有按下任何按键的情况下应该在终端中显示如下实验结果：

```
read = 0x0
read = 0x0
```

由于版图厂家在电路图1上只标注了 KP_COL，为了成功进行该实验，实际上花去了相当多时间寻找正确对应的 GPIO 口。根据三星手册上具有 KP_COL 功能的 GPIO 口，尝试 GPJ1，GPJ2 口在配置寄存器后的效果，发现无论如何按键，读取值都相同，显然不起作用。在根据实际电路图1正确配置上拉电阻后，也都只能读出 0x00。在仔细阅读三星手册中关于键盘功能的使用和 GPIO 口的功能配置后，发现 GPH2 口通过配置寄存器也有 KP_COL 的功能，故而猜测键盘电路连接的 KP_COL[0]-[3] 实际上是 GPH2[0]-[3]。重新配置寄存器后，按下不同的按键会有不同的读取值，证明了猜测是正确的。[1]

确定了调用的按键应该被 GPH2 控制后，由于电路图上的按键并未像电路板上进行标号，运行测试程序并逐个按下按键。测试结果是正确的，证明按键驱动程序正常工作。通过多次实验可以得到，按下 LEFT 键读出 0x10，对应 GPH0[2]；按下 DOWN 键读出 0x20，对应 GPH0[3]；按下 UP 键读出 0x1, 对应 GPH2[0]，按下 RIGHT 键读出 0x2，对应 GPH2[1]，按下 BACK 键读出 0x4，对应 GPH2[2]，按下 MENU 按键读出 0x8，对应 GPH2[3]，为下一步设计综合测试程序提供了一个硬件映射图。

3.4 用户程序的设计

设计一个综合测试程序 mixtest，通过两个设备文件同时调用 LED 灯和按键设备，主程序分别调用亮灯函数和识别按键的函数来完成通过按键控制 LED 灯的效果。代码如下所示：

```
/* mixtest.c */
#include "GPIO.h"
#define LED_NUM 3
enum Buttons {UP, LEFT, DOWN, RIGHT, BACK, MENU};
//亮灯函数，输入0不亮灯，输入"灯数+1"亮所有灯，输入灯号亮指定灯
void lightled(int fd, int state){
    char led = 0x38;
```

[1]手册上，与前两个寄存器的区别是分别属于序列 pad0 和 pad1，且 GPH 类型端口不可休眠

```
    if(state == LED_NUM + 1) write(fd, &led, 1);
    else if(state == 0) ioctl(fd, XLEDBLANK, 0);
    else{
        led = SETBIT(state + 2);
        write(fd, &led, 1);
    }
}
//识别按键（含消抖功能）函数
int keyscan(int fd){
    int ret = -1;
    static int key = -1;
    static int key_state = 0;
    char key_press;
    read(fd, &key_press, 1);
    switch(key_state){
        //按键每次检测都要延时，减少检测次数的同时可以消抖
        case 0:
            if(key_press) key_state = 1;
            usleep(10000);break;
        //延时后若还能检测到按键电平则说明是有效按键
        case 1:
            key_state = 2;
            switch(key_press){
                case 0:
                    key_state = 0;break;
                case UPBUTTON:
                    key = UP;break;
                ...
                case DOWNBUTTON:
                    key = DOWN;break;
                default:
                    key_state = 0;break;
            }
        //确认按键后，检测到按键抬起后再响应按下的键并重置按键状态
        case 2:
            if(!key_press){
                key_state = 0;
                ret = key;
                key = -1;
            }
```

```
            break;
    }
    return ret;
}
int main(int argc, char *argv[]){
    int fdled = open("/xled", O_RDWR);
    int fdkey = open("/xkeypad", O_RDWR);
    int key;
    int led_state = 0;
    int light_sign = LED_NUM;
    ioctl(fdled, XLEDBLANK, 0);
    //采用非中断法控制键盘，只能利用条件循环不断查询键盘
    while(1){
        key = keyscan(fdkey);
        if((key == MENU))
            break;
        else{
            switch(key){
            case UP:
                light_sign = LED_NUM;
                lightled(fdled, led_state);break;
            case LEFT:
                led_state = ((led_state - 2 + LED_NUM) % LED_NUM) + 1;
                printf("state=%d\n",led_state);
                lightled(fdled, led_state & light_sign);break;
            case DOWN:
                light_sign = 0;
                lightled(fdled, 0);
                break;
            case RIGHT:
                led_state = ((led_state + LED_NUM) % LED_NUM) + 1;
                printf("state=%d\n",led_state);
                lightled(fdled, led_state & light_sign);break;
            case BACK:
                lightled(fdled, LED_NUM + 1);break;
            default:break;
            }
        }
    }
    ioctl(fdled, XLEDBLANK, 0);
```

```
    close(fdkey);
    close(fdled);
    return 0;
}
```

经多次测试和修改代码后，该程序可以正常运行。加载两个设备驱动模块后运行程序，结果和程序逻辑一致：以按键按下后抬起作为一次按键过程。按下 MENU 按键，LED 灯全部熄灭并退出程序；按下 BACK 按键，LED 灯会全部亮起；按下 DOWN 按键，LED 灯会全部熄灭；按 LEFT 和 RIGHT 按键可以向左、向右选择 LED 灯，在选择 1、2、3 号灯中循环并在终端打印出选择了哪一个灯；按 UP 键会只点亮选择的灯。在这个程序中用到了驱动程序提供的所有功能，执行成功也即说明了两个驱动程序都工作正常。

4 实验讨论与小结

4.1 部分实验问题讨论

* 驱动程序的结构设计

在这次的实验中对于驱动程序的结构设计有两种思路：一种是将两种设备看做完全无关，完全独立成主设备号不同的设备，调用不同的驱动程序进行处理。另一种思路是将两种设备完全视作 GPIO 口设备，分配不同的次设备号，以相同逻辑进行处理，也可以通过软件结构的方式实现两套不同的逻辑，但有部分共通的内部函数、描述结构等。

实际上这次实验中，LED 灯、按键两种设备由于结构都很简单，除了都用到了 GPIO 口，实质的操作模式是改变 GPIO 口对应的 3-6 种寄存器以外，并没有其他的相似之处。对于单次嵌入式系统开发实验中外设不多且用途很单一的情况，以上文方法即第一种思路分别实现 LED 灯、按键驱动，两种驱动都有各自方便的表示方式，这是一种简明易懂，实现容易，不需要在软件结构上花太多心思也可行的方案，而采取第二种思路有或者出现设计过度、开发周期长问题，或者出现用户需要了解 GPIO 与设备之间深入关系的问题。对于外设功能丰富，需要节约使用主设备号，驱动结构复杂内部耦合性也较高的，可以给用户提供的较好的用户程序或示例等情况下，还是应该优先采用整体来看更加完备的第二种思路。

4.2 实验小结

本次实验探究的是 I/O 接口驱动程序。在理清文件-文件描述符-文件操作-底层设备这条关系之前，确实走了一些弯路，理解后，再阅读了一些驱动程序的代码，对于整个驱动程序的实现，原理都已经比较了解，顺利实现了两套简单的设备驱动程序并以此开发了实际应用的测试程序。实际上，模块化的驱动程序和 frambuffer 都是 linux 系统提供的一种抽象-实现机制，因此再回头对比图形用户接口实验后，对 frambuffer 设备和设备驱动程序本身都有了更进一步的理解。同时，本次实验中对于驱动程序结构和实现的思考很多都源自于对 linux 源代码的一些查阅和 linux 自身提供的一些抽象机制。

本次实验还是在嵌入式开发中第一次遇到了需要详细查阅嵌入式处理器手册、开发板电路图的情况，最后在多次实验和查阅手册互补后成功排除对于寄存器的不明确之处，算是收获了一些实际开发的锻炼经验，也再次意识到了嵌入式开发具有的复杂性。