Embedded system 实验三

IO接口驱动编程:LED 灯的控制

171180616 郭成伟

- Embedded system 实验三
  - IO接口驱动编程:LED 灯的控制
    - 171180616 郭成伟
  - 背景介绍
  - 实验平台
  - am3858硬件介绍
  - IO地址映射
    - `ioremap()` 函数与 `iounmap()` 函数
  - LED控制
  - 设备文件操作
    - `mknod()` 函数
    - `register_chrdev()` 函数
    - `unregister_chrdev()` 函数
    - `file_operaions`
  - 接口驱动程序设计
    - 模块初始化
    - 模块卸载
    - `open()` 函数重载
    - `release()` 函数重载
    - `write()` 函数设计
      - `copy_from_user()` 函数
      - 控制信号设计
      - `write()` 函数重构
    - Makefile
    - 模块的安装与卸载
      - `insmod`
      - `rmmod`
    - 用户程序编写
  - 一些问题

背景介绍

字符设备是指在I/O传输过程中以字符为单位进行传输的设备，例如键盘，打印机等。在UNIX系统中，字符设备以特别文件方式在文件目录树中占据位置并拥有相应的结点。

与普通文件类似，我们可以使用`open()`、`close()`、`read()`、`write()`等操作直接对字符设备操作

字符设备以特别文件方式在文件目录树中占据位置并拥有相应的结点。结点中的文件类型指明该文件是字符设备文件。字符设备文件可以使用与普通文件相同的文件操作命令对字符设备文件进行操作，例如打开、关闭、读、写等。本次实验的重点就是构建、重载我们自己的设备文件操作函数。

当一台字符型设备在硬件上与主机相连之后，必须为这台设备创建字符特别文件。操作系统的mknod命令被用来建立设备特别文件。

设备与驱动程序的通信方式依赖于硬件接口。当设备上的数据传输完成时，硬件通过总线发出中断信号导致系统执行一个中断处理程序。中断处理程序与设备驱动程序协同工作完成数据传输的底层控制。

实验平台

- BeagleBone black 开发板一块

am3858硬件介绍

Linux 以模块（`modules`）的形式加载设备，通常一个模块对应一个设备驱动，但开发者可以根据实际工作需要在一个模块中实现不同的驱动程序。驱动程序，创建了一个硬件与硬件，或硬件与软件沟通的接口，经由主板上的总线（`bus`）或其它沟通子系统（`subsystem`）与硬件形成连接的机制，这样的机制使得硬件设备（`device`）上的数据交换成为可能。

设备驱动程序负责将应用程序如读、写等操作正确无误的传递给相关的硬件，并使硬件能够做出正确反映，因此在编写设备驱动时，必须先了解相应的硬件设备的寄存器、I/O 口的物理内存地址参数。

设备驱动在准备好以后可以编译到内核中, 在系统启动时和内核一起启动, 这种方法在嵌入 式 Linux 系统中经常被采用。在开发阶段, 设备驱动的动态加载更为普遍。开发人员不必在调试过程中频繁启动机器就能完成设备驱动的调试工作。

嵌入式处理器片内集成了大量的可编程设备接口, 为构成处理器系统带来了极大的便利。 am335x 实现 4 组 `GPIO` 模块、每组 32 只引脚的输入/输出控制功能，它们可用于信号的输 入/输出、键盘控制以及其他信号捕获中断功能。有些 `GPIO` 的引脚可能与其他功能复用。

在实验班 `mini-USB` 接口附近，有四个可供用户控制的 `LED`，他们来自 `GPIO1` 模块的 `21~24` 引脚。本次实验实现对这几个LED灯接口的设计。

IO地址映射

一般来说，在系统运行时，外设的`I/O`内存资源的物理地址是已知的，由硬件的设计决定。但是CPU通常并没有为这些已知的外设`I/O`内存资源的物理地址预定义虚拟地址范围，驱动程序并不能直接通过物理地址访问`I/O`内存资源，而必须将它们映射到核心虚地址空间内（通过页表），然后才能根据映射所得到的核心虚地址范围，通过访内指令访问这些`I/O`内存资源。

`ioremap()`函数与`iounmap()`函数

```
1 void* ioremap(unsigned long phys_addr, unsigned long size, unsigned long
  flags);
2               //要映射的起始地址                    /映射内存部分大小
```

参数意义:

- `phys_addr`:需要映射的物理起始地址
- `size`：映射的地址大小
- `flags`：要映射的`IO`空间的和权限有关的标志

返回值：返回一个内核的虚拟地址。

在将`I/O`内存资源的物理地址映射成核心虚地址后，理论上讲我们就可以象读写RAM那样直接读写I/O内存资源了。为了保证驱动程序的跨平台的可移植性，我们应该使用Linux中特定的函数来访问`I/O`内存资源，而不应该通过指向核心虚地址的指针来访问。

读写 `I/O` 的函数如下所示(分别有三种规格的函数：`byte`、`word`、`long`，在这里我们不详细列出)：

- `writel()`

  `writel()` 往内存映射的 I/O 空间上**写数据**，`wirtel()` I/O 上写入 32 位数据 (4字节)。

  原型：

```
1 void writel (unsigned char data , unsigned int addr )
```

- `readl()`

  readl() 从内存映射的 I/O 空间上**读数据**,readl 从 I/O 读取 32 位数据 (4 字节)。
  原型：

```
1 unsigned char readl (unsigned int addr )
```

`iounmap()` 函数用于取消 `ioremap()` 所做的映射，原型如下：

```
1 void iounmap(void* addr);
```

查询 `am335x` 的技术手册可以得到 `user LED` 部分的电路图以及控制管脚信息如下：

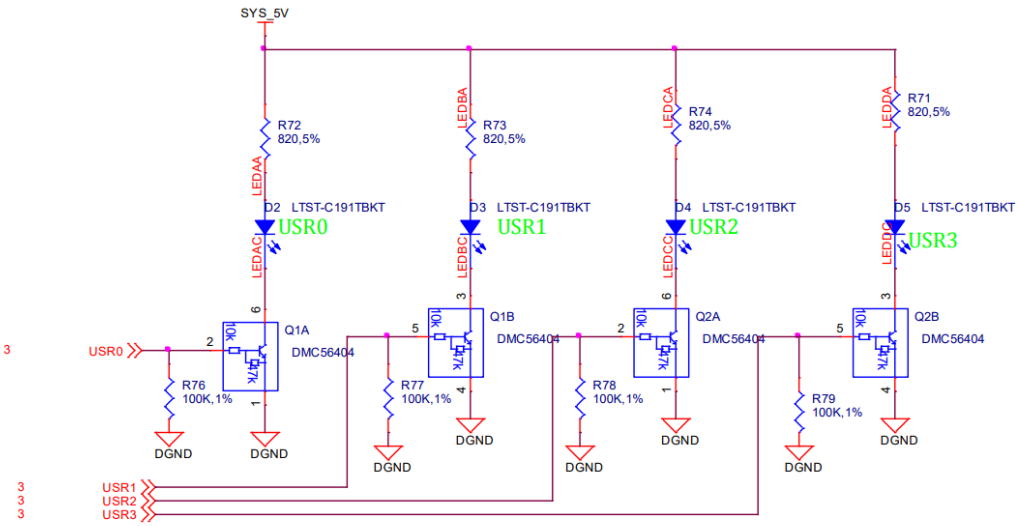


Figure 37. User LEDs

Table 8. User LED Control Signals/Pins

LED	GPIO SIGNAL	PROC PIN
USR0	GPIO1_21	V15
USR1	GPIO1_22	U15
USR2	GPIO1_23	T15
USR3	GPIO1_24	V16

通过电路图可以得到，逻辑电平`1`会使得LEDs 变亮。

查询~的技术手册可以得到控制LED灯的`GPIO1`的物理地址为：

起始地址：`0x4804_C000` 结束地址：`0x4804_CFFF`

该GPIO与实验相关的寄存器地址偏移以及寄存器的基本信息如下：

offset	Acronym Register	Description
0x130h	GPIO_CTRL	Module Control
0x134h	GPIO_OE	Output Enable
0x138h	GPIO_DATAIN	Data Input
0x13Ch	GPIO_DATAOUT	Data Output

LED控制

在完成上述内存映射之后，即可对LED进行控制

在实际操作过程中，发现不能控制第一个灯，第一个灯显示了系统的工作状态。

我们定义一个结构体来保存映射的地址。

```
typedef struct  GPIO_m{
    volatile int *pConf;
    volatile int *pDataIn;
    volatile int *pDataOut;
}gpio_t;
```

在这里我们用了一个关键字：`volatile`。

一个定义为`volatile`的变量是说这变量可能会被意想不到地改变，

这样，编译器就不会去假设这个变量的值了。精确地说就是，优化器在用到这个变量时必须每次都小心地重新读取这个变量的值，而不是使用保存在寄存器里的备份。(在一些情况下，系统会自动将一些经常读取的变量读取到高速缓存中以优化性能，简单来说这会使得数据会出现不一致的情况)

下面是`volatile`变量的几个例子：

1). 并行设备的硬件寄存器（如：状态寄存器）

2). 一个中断服务子程序中会访问到的非自动变量(Non-automatic variables)

3). 多线程应用中被几个任务共享的变量

代码示例如下：

先完成地址映射，再对该地址下的寄存器进行操作。

```
gpio_t *gpio_manger;//初始化一个用于管理的结构体
gpio_manger->pConf =ioremap(GPIO1_OE,4);//映射4个字节大小地址
gpio_manger->pDataOut =ioremap(GPIO1_OUT,4);//映射4个字节大小地址

*(gpio_manger->pConf) &= ~(1<<22); //控制第二个灯
*(gpio_manger->pDataOut) |= (1<<22); //2号灯亮  高电平灯亮
```

设备文件操作

设备文件允许进程同内核中的设备驱动通信，并且通过它们和物理设备通信。Linux 系统中有三类设备文件：字符设备、块设备与网络接口，本次实验实现的就是字符设备 驱动。 字符设备是能够像字节流一样被访问的设备，由字符设备驱动程序来实现这种特性， 它通常至少需要实现 `open`、`close`、`read`、`write` 系统调用。字符设备可以通过文件系统节点来访问，它和普通文件的唯一差别在于，对普通文件的访问可以前后移动指针， 而大多数字符设备是只能顺序访问的数据通道。

`mknod()`函数

`mknod` 设备文件由命令 mknod 创建，并被赋予一个主设备号和一个次设备号，格式如下：

```
mknod device_name c/b MAJOR MINOR -m 666
```

命令参数分析：

- `device name` 设备文件名称， 在这里我们需要指定成 和内核中一致。
- `c/b` 这一位表示设备类型。
  - `b`，即`block`，块文件。系统从块设备中读取数据的时候，直接从内存的buffer中读取数据，而不经过磁盘；
  - `c`，即`character`，字符设备文件。与设备传送数据的时候是以字符的形式传送，一次传送一个字符，比如打印机、终端都是以字符的形式传送数据；
- `MAJOR` 是主设备号；
- `MINOR` 是次设备号；
- `-m` 参数用于指定设备文件访问权限。`666`代表对用户 可读可写

在内核中使用主设备号标识一个设备，次设备号提供给设备驱动使用。在打开一个设备的时候，内核会根据设备的主设备号得到设备驱动，并且把次设备号传递给驱动。linux操作系统中为设备文件编号分配了32位无符号整数，其中前12位是主设备号，后20位为次设备号，所以在向系统申请设备文件时主设备号不好超过`4095`，次设备号不好超过`2^20 -1`。

```
mknod /dev/myled c 245 0 -m 666
                  #字符设备文件
                      # 主设备号为245
                           # 权限为 可读可写
```

`register_chrdev()` 函数

`register_chrdev()` 是注册设备驱动程序的内核函数。

原型：

```
#include <linux.fs.h>
int register_chardev (unsigned int major, const char *name, struct
file_operations *fops)
```

变量说明：

- `major`：主设备号，该值为 0 时，自动运行分配。而实际值不是 0 。
- `name`：设备名称；
- `fops`：file_operations 结构体变量地址(指针)。`fops` 是指向函数指针数组的结构指针，驱动程序的入口函数都包括在这个指针内部。
- 返回值：
  - `major` 值为 0 时，正常注册后，返回分配的主设备号。如果分配失败，返回 `EBUSY` 的负值。
  - 指定 `major` 值后，若有注册的设备，返回 `EBUSY` 的负值 。若正常注册，则返回 0 值。

`file_operaions` 的文件结构将在下面介绍。

`register_chrdev()` 为实现对设备的操作，内核模块需要调用函数 `register_chrdev()` 在注册表中对设备进行注册。注册函数传递一个 `file_operations` 结构，包含了对设备文件的打开、 释放、读、写及 I/O 控制等各种操作的函数指针。这项工作通常在模块加载时完成。 当使用 `register_chrdev()` 函数成功注册一个字符设备后，会在 `/proc/devices` 文 件中显示出设备信息。

`unregister_chrdev()` 函数

`unregister_chrdev()` 是注销设备驱动程序的内核函数。

原型：

```
#include <linux.fs.h>
int unregister_chrdev (unsigned int major, const char *name)
```

变量说明：

- `major`：主设备号。
- `name`：设备文件名称；
- 返回值：

  指定 `major` 值后，若将要注销的 `major` 值并不是注册的设备驱动程序，返回 `EINVAL` 的负值 。正常注销则返回 0值。

卸载时，内核会比较设备驱动名称与设备号是否相同。错误地卸载设备驱动可能带来严重后果，因此在卸载驱动的时候应该对函数返回值做判断。

`file_operaions`

文件操作，`file_operations`结构或者指向这类结构的指针称为`fops`，这个结构中每个字段都必须指向驱动中实现特定操作的函数的函数指针。结构定义在`<linux/fs.h>`中，其中包含一组指针，每个打开的文件和一组函数关联（通过包含指向一个`file_operations`结构的`f_op`字段）。这些操作主要用来实现系统调用，也可以说文件可以认为是一个对象，而操作它的函数是方法。

```
struct file_operations {
    //在这里我们只列了一些我们需要重构的函数

    struct module *owner;//拥有该结构的模块的指针，一般为THIS_MODULES
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ////从设备中同步读取数据
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t
*);
    //向设备发送数据
    int (*open) (struct inode *, struct file *);//打开
    int (*release) (struct inode *, struct file *);//关闭
    ...
};
```

`open()`：在打开设备的时候内核会调用该函数。

`release()`：在用户关闭设备的时候内核会调用该函数。

`read()`：对设备进行读操作的时候内核会调用该函数。

`write()`：对设备进行写操作的时候内核会调用该函数。

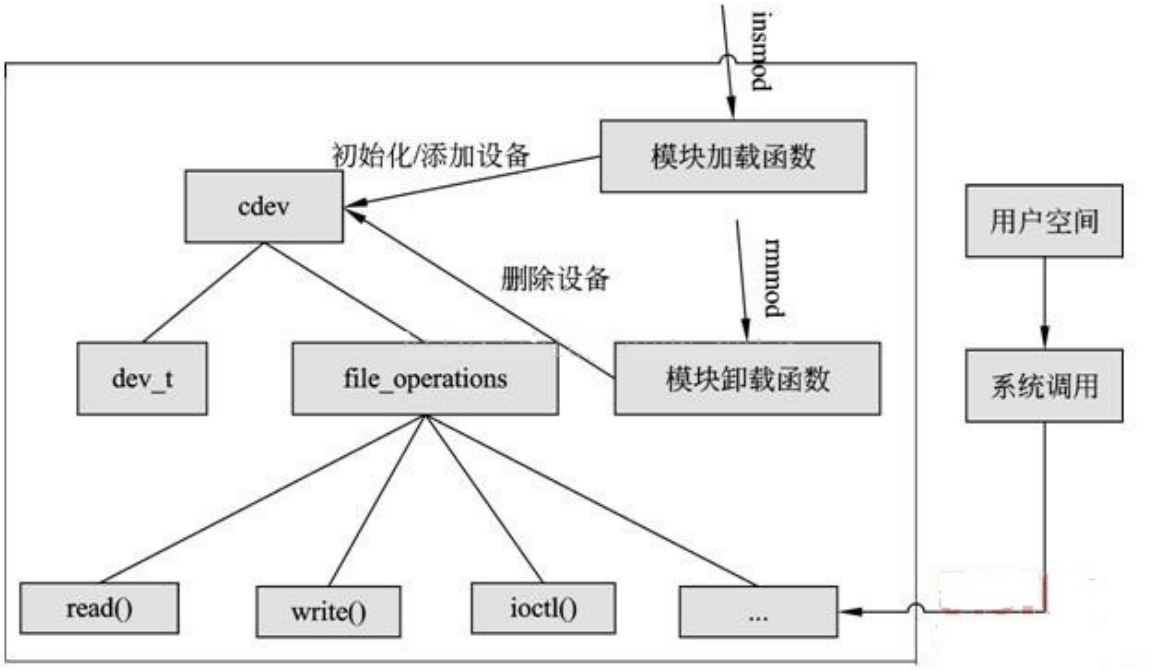
`ioctl()`：该函数用于向设备传递控制信息或者获取设备状态。

本次实验接口编程的一大主要工作即是重载这些函数。

```
struct file_operations myled_fop =
{
    .open = LedOpen,
    .release = LedRelease,
    .read = LedRead,
    .write = LedWrite,
};
```

接口驱动程序设计

接口驱动的逻辑图如下:



本实验由于是控制LED灯,所以不进行`read`函数的重构.

模块初始化

在安装模块的时候，接口驱动自动调用`init`函数，在初始化过程中，我们申请注册设备。

```
int init_module(void){
  if(!(major = register_chrdev(0, "myled", &myled_fop))){
    printk("unable to get device number!\n");
    return -1;
  }
    printk("module has been installed, major dev num is %d\n",major);
    return 0;
}
```

模块卸载

卸载模块时，我们注销设备。

```
void cleanup_module(void){
    if(!(  unregister_chrdev(major, "myled"))){
          printk("myled module removed\n");
        return 0;
    }
    return -EINVAL;
}
```

`open()`函数重载

`_file->private_data`介绍

`struct file`是字符设备驱动相关重要结构。`struct file`代表一个打开的文件描述符，它不是专门给驱动程序使用的，系统中每一个打开的文件在内核中都有一个关联的`struct file`。它由内核在`open`时创建，并传递给在文件上操作的任何函数，直到最后关闭。当文件的所有实例都关闭之后，内核释放这个数据结构。

`UNIX`系统的设计本着一切都是文件的原则。对设备的访问，可以简化成对设备文件的访问。也就是都要经过VFS层，`file`结构体从属于进程的，用来描述某个进程正使用的设备文件，`private_data`用来直接挂简单的字符设备驱动，刚好是一脉相承。 当然我们可以使用全局变量来存储这个结构体。但这会导致一个问题，全局变量是同一段内存空间，这些数据结构如果是全局变量，则一个设备驱动驱动多个相同设备时， 多个相同设备就共用了这些自定义数据结构，会引起冲突。

如果采用`file`结构体那么，不同的进程访问相同设备，用的是不同的`file`结构体实例，文件指针也不一 样。

我们采用`_file->private_data`来存储我们映射的地址。

```
int LedOpen(struct inode *_inode, struct file *_file){
    gpio_t *gpio_manger;
    _file->private_data = kmalloc(sizeof(gpio_t), GFP_KERNEL);
    gpio_manger =(gpio_t *)(_file->private_data);

    gpio_manger->pConf =ioremap(GPIO1_OE,4);
    gpio_manger->pDataIn =ioremap(GPIO1_IN ,4);
    gpio_manger->pDataOut =ioremap(GPIO1_OUT,4);
    printk("LEDOpen succeeded !\n");
    return 0;
}
```

`release()`函数重载

在设备关闭时，我们将所有LED灯关闭，并且解除映射的地址，以及释放掉使用到的内存。

```
#define CON_ALL ~(1<<22|1<<23|1<<24)  //控制三个灯
#define CLOSE_ALL ~(1<<22|1<<23|1<<24)  //控制三个灯灭

int LedRelease(struct inode *_inode, struct file *_file){
    gpio_t *gpio_manger =(gpio_t *)(_file->private_data);

    *(gpio_manger->pConf) &= CON_ALL; //设置控制三个灯
    *(gpio_manger->pDataOut) &= CLOSE_ALL; //控制三个灯全灭
    iounmap(gpio_manger->pConf);//解除映射
    iounmap(gpio_manger->pDataIn);
    iounmap(gpio_manger->pDataOut);
    kfree(gpio_manger);//释放空间
    printk("LEDRelease succeeded !\n");
    return 0;
}
```

`write()` 函数设计

`copy_from_user()`函数

用于将用户空间的数据传送到内核空间。为什么会用到这样一个函数呢？实际上是因为，用户空间的地址与内核空间的地址并不相同，在两个空间之间不能通过直接访问地址来访问数据。

```
unsigned long copy_from_user(void * to, const void __user * from, unsigned long n)
```

`参数to`：内核空间的数据目标地址指针，

`参数from`：用户空间的数据源地址指针，

`参数n`：数据的长度。

返回值：如果数据拷贝成功，则返回0；否则，返回没有拷贝成功的数据字节数。

此函数将from指针指向的用户空间地址开始的连续n个字节的数据产送到to指针指向的内核空间地址

由于内核空间与用户空间的内存不能直接互访，因此借助函数 copy_from_user()完成用户空间到内核空间的复制，完成从内核空间到用户空间的复制。

`copy_from_user`函数内部的实现当然不仅仅拷贝数据，还需要考虑到传入的用户空间地址是否有效，比如地址是不是超出用户空间范围，地址是不是没有对应的物理页面，否则内核就会oops的。不同的架构，该函数的实现不一样。

控制信号设计

我们考虑用一个字节的信号控制一个灯。

具体控制如下：

将表示LED的二进制位左移四位 或上 表示状态的二进制右移四位

```
#define LED1 0x01  //1
#define LED2 0x03  //3
#define LED3 0x07  //7
#define LED4 0x0F  //15
#define ON  0xF0
#define OFF  0x00
```

操作示例：

```
char buffer;
    buffer = (LED2 << 4)|(ON >> 4); //led2亮
```

`write()`函数重构

我们用`copy_from_user`函数把用户写入的数据读到之后，对读到的数据进行处理。读出对应的`LED`和状态

```
        char flag =temp[i];
        flag = (flag >> 4);
        char status = temp[i];
        status =(status << 4);

switch(flag){
            case LED2:
                *(gpio_manger->pConf)  &= ~(1<<22); //控制第2个灯
                if(status==ON){
                    *(gpio_manger->pDataOut) |= (1<<22); //2号灯亮
                }
                else{
                    *(gpio_manger->pDataOut) &= ~(1<<22); //2号灯灭
```

```
                }
                break;
            ...
            //other conditions
```

如此实现对LED灯的控制。

再完成上述函数编写之后，接口驱动程序基本完成。接下来进入编译安装测试阶段。

Makefile

```
obj-m = myio.o
KDIR =/home/171180616/linux
PWD =$(shell pwd)
COMPILE=/opt/armhf-linux-2018.08/bin/arm-none-linux-gnueabi-

ARCH_TYPE=arm

all:
        $(MAKE) CROSS_COMPILE=$(COMPILE) ARCH=$(ARCH_TYPE) -C $(KDIR)
SUBDIRS=$(PWD) modules
clean:
        $(MAKE) CROSS_COMPILE=$(COMPILE) ARCH=$(ARCH_TYPE) -C $(KDIR)
SUBDIRS=$(PWD) clean

```

`obj-m` 表示编译成模块。

`make -C` 选项的作用是指将当前的工作目录转移到指定的目录，即 `KDIR` 目录，程序到 `PWD` 当前目录查找模块源码，将其编译，生成 `.ko` 文件。

执行 `makefile` 之后，我们可以在当前文件夹找到 `myio.ko` 文件。

模块的安装与卸载

`insmod`

用于载入模块至内核之中。

```
insmod myio.ko
```

内核中有许多功能都是模块化开发的，而其中很多模块默认不开启，也就是未加载到内核中。你通过 `insmod` 工具可以将某个内核模块加载到正在运行的内核中，内核就开启了该模块的功能。如此可使 `kernel` 较为精简，进而提高效率，以及保有较大的弹性。这类可载入的模块，通常是设备驱动程序。

`rmmod`

`rmmod` 命令用于卸载模块。

```
rmmod [-as][模块名称...]
#-a   删除所有目前不需要的模块。
#-s   把信息输出至syslog常驻服务，而非终端机界面。
```

执行`rmmod`指令，可删除不需要的模块。

动态加载利用了Linux的module特性，可以在系统启动后用insmod命令添加模块`.ko`，在不需要的时候用`rmmod`命令卸载模块，采用这种动态加载的方式便于驱动程序的调试，同时可以针对产品的功能需求，进行内核的裁剪，将不需要的驱动去除，大大减小了内核的存储容量。

总的说来，将`makefile`生成的`myio.ko`文件导入到开发板上，执行`insmod`命令将模块加载进入内核，再执行`mknod`命令，创建设备文件即可以正常使用该LED设备。

```
insmod myio.ko #调用 驱动程序的 init_moudule函数
mknod /dev/myled c 245 0 -m 777
mkdir -p /lib/modules/4.4.155 #为了之后的卸载进行准备
```

接下来，我们通过用户程序对该设备进行使用。

用户程序编写

```
    int fd;
    fd=open("/dev/myled", O_RDWR); //调用LedOpen 函数
    if(fd<0){
        perror("fd open");
        printf("fb_open is error");
        return -1;
    }

    char buffer;
    buffer =(LED2 << 4)|(ON >> 4); //led2亮
                                    //时间原因在这里 写了一个比较简单的测试版本代码
    write(fd,&buffer,sizeof(buffer));   //调用LedWrite函数

    sleep(2);
    close(fd);//调用LedRelease函数
```

运行该程序之后，可以发现第二个`LED`灯亮两秒之后灭，与预期现象相符。

一些问题

1. 第一次利用 `rmmod` 命令卸载模块时会出现无法卸载的现象： `no such directory`

   ```
   mkdir -p /lib/modules/4.4.155
   ```

   执行上述命令之后，再进行卸载即可。

2. 在安装模块的时候，可能会遇到证书相关的警告，可加上 `MODULE_LICENSE("GPL")`;

   从2.4.10版本内核开始，模块必须通过MODULE_LICENSE宏声明此模块的许可证，否则在加载此模块时，会收到内核被污染 “`kernel tainted`” 的警告。从`linux/module.h`文件中可以看到，被内核接受的有意义的许可证有“`GPL`”，“`GPL v2`”，“`GPL and additional rights`”，“`Dual BSD/GPL`”，“`Dual MPL/GPL`”，“`Proprietary`”。

3. 实验中发现userLED的1号灯并没有办法控制，猜测时因为系统的状态由该灯显示。

至此，第三次实验完成。