KEY驱动程序编写

实现步骤:

1.确定使用的IO口并准备硬件

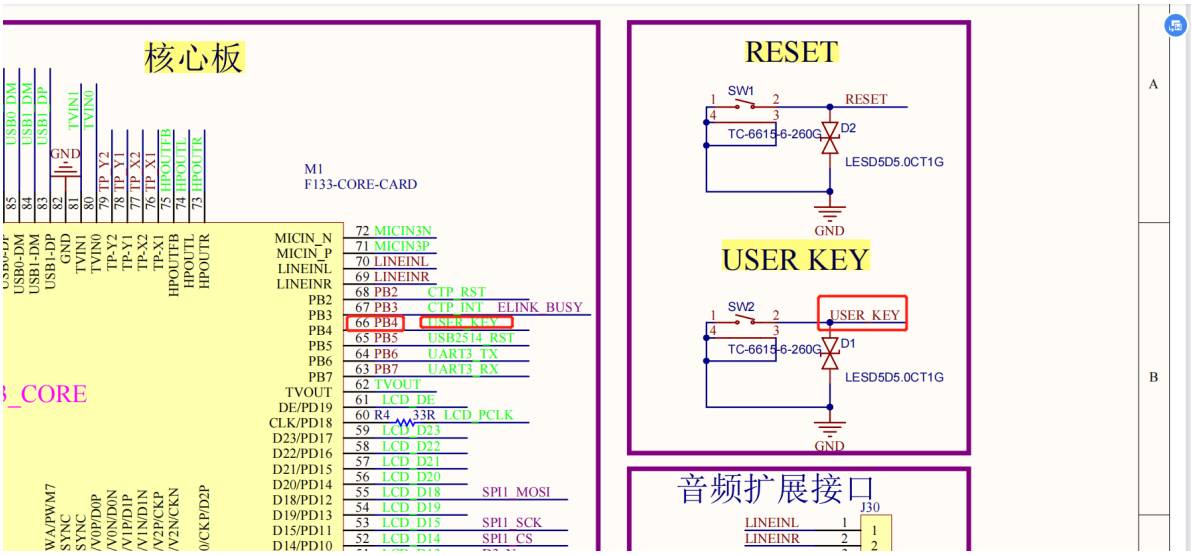
2.修改设备树

3.裁剪LINUX内核，编译并烧写

4.用户按键驱动编写

1.确定使用的IO口并准备硬件

开发板上带有IO留给用户按键KEY，PB4作为本次现象的引脚，为第66号引脚，这次不需要飞线，也是该文档中的第一个真正意义上的驱动程序编写



2.修改设备树

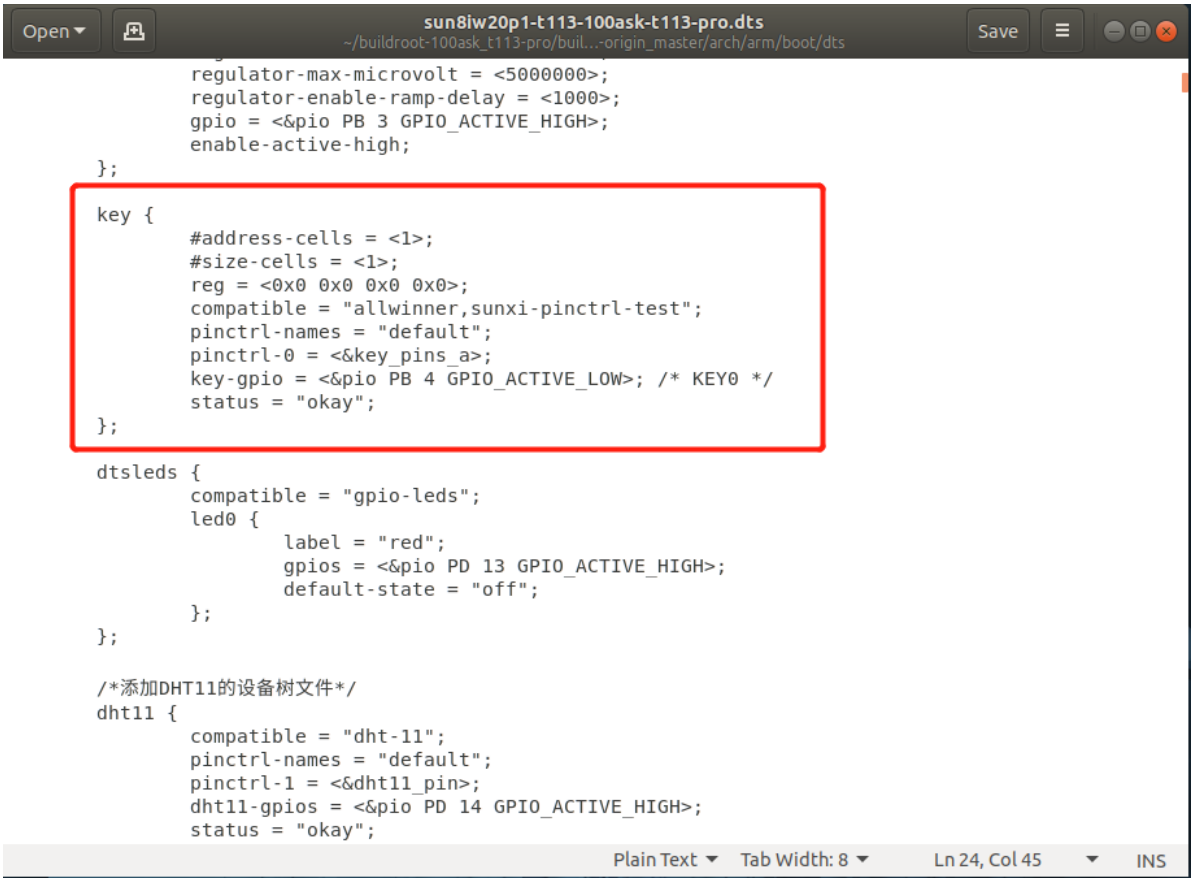
2.1.设备树修改实操:

```
cd /buildroot-100ask_t113-pro/buildroot/output/build/linux-
origin_master/arch/arm/boot/dts

sudo gedit sun8iw20p1-t113-100ask-t113-pro.dts
```

代码块1:

在根设备节点下添加挂载key设备节点:

```
key { /*定义了一个名称为"key"的节点。*/
        #address-cells = <1>; /*此节点使用一个地址单元。*/
        #size-cells = <1>;  /*此节点使用一个尺寸单元。*/
        reg = <0x0 0x0 0x0 0x0>;/*此节点的物理地址范围为0x0到0x0，因为所有的值都为0。*/
        compatible = "allwinner,sunxi-pinctrl-test"; /*此节点
与"Allwinner"和"sunxi-pinctrl-test"兼容。*/
        pinctrl-names = "default"; /*此节点的默认引脚控制列表名称为"default"。*/
        pinctrl-0 = <&key_pins_a>; /*此节点的引脚控制列表使用了名称为"key_pins_a"的引脚
控制列表。*/
        key-gpio = <&pio PB 4 GPIO_ACTIVE_LOW>; /*按键连接到了引脚PB4，GPIO的电平为低
电平(GPIO_ACTIVE_LOW)，并将此引脚标记为KEY0。*/
        status = "okay"; /*此节点的状态为正常（"okay"）*/
    };
```



代码块2：

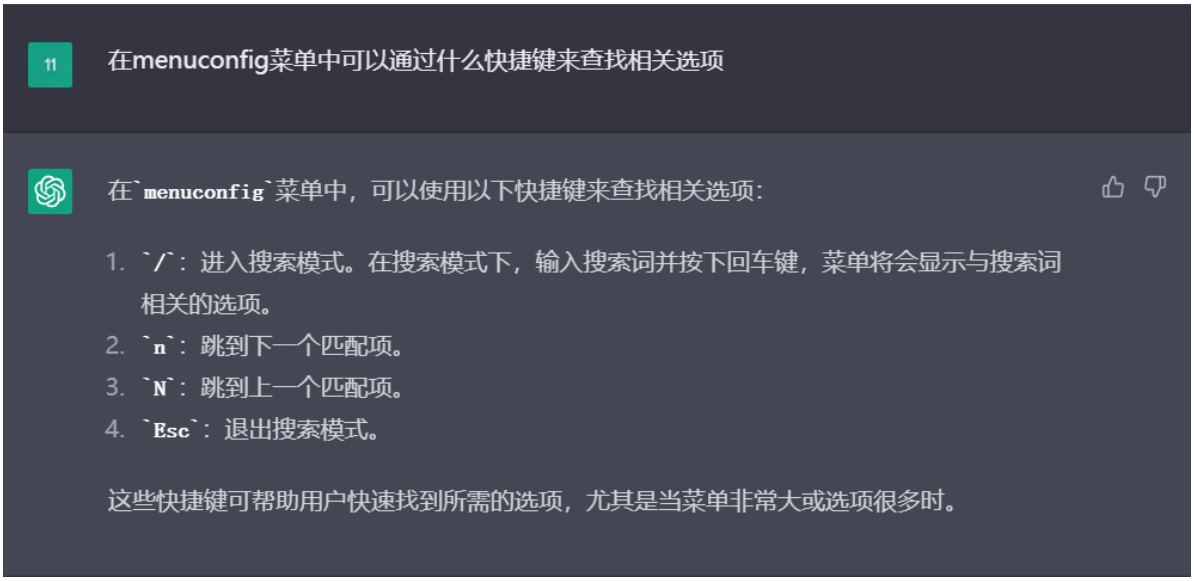
```
key_pins_a: userkey {
        allwinner,pins = "PB4" ;
    };/*定义一个GPIO引脚（GPIO pin）并将其命名为“userkey”，该引脚被映射到Allwinner SoC的PB4引脚。*/
```

3.裁剪LINUX内核，编译并烧写

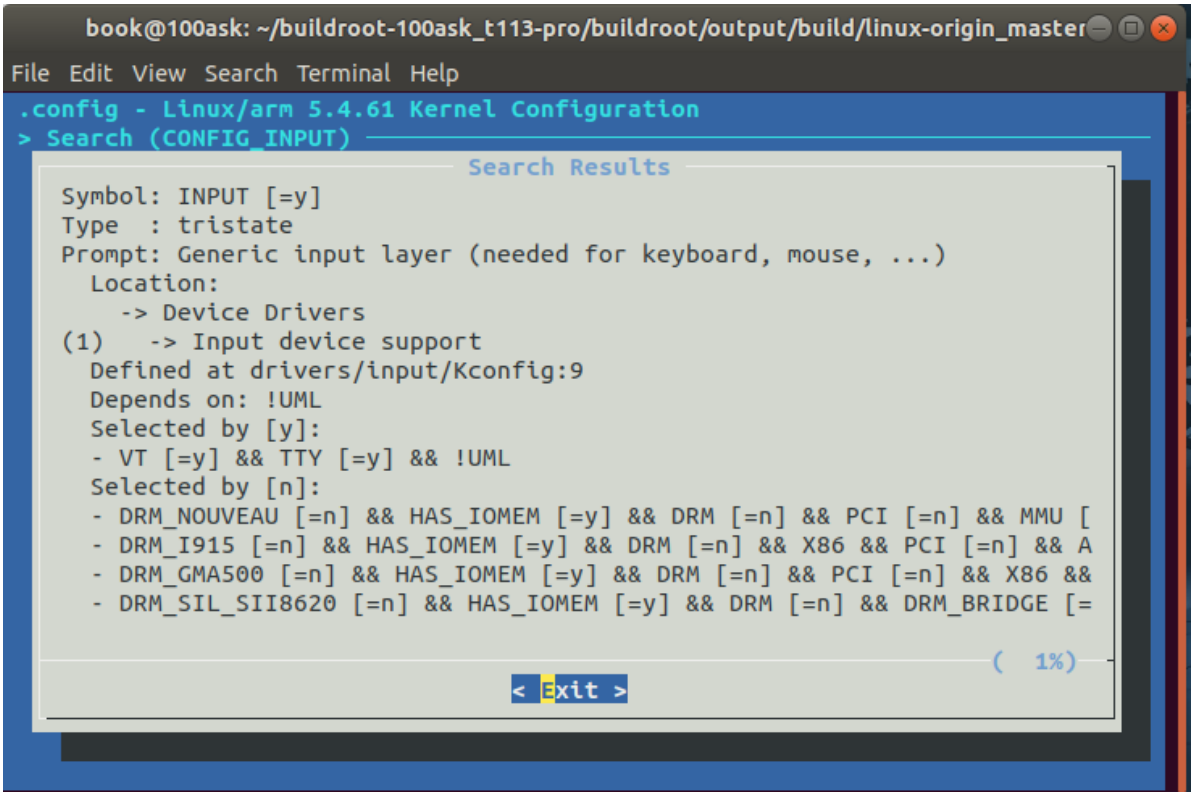
3.1.裁剪内核

按键驱动所需裁剪内核内容：

检查方法：



一般官方给的buildroot工具链框架中已经将这些内核裁剪进去了，不需要再进行操作。



3.2.编译内核

进入到/buildroot目录下，输入以下两条命令即可在/output/images目录下找到生成的sd卡镜像，该命令用到了buildroot框架。

```
make linux-rebuild V=1
make V=1
```

3.3.烧写SD卡

不演示

4.用户按键驱动编写

对于第一个驱动程序，我感觉需要详细一点介绍什么是“驱动”

4.1.最基本的linux驱动框架

其实Linux驱动开发说白了就是Linux驱动框架再加上单片机寄存器编程，理论上你只要会单片机你就能写驱动，在这里以一个最经典的案例介绍一个Linux驱动程序应该包含什么，每个部分分别有什么用，由什么组成?

一个驱动程序编写的具体流程如下：

/* 1. 确定主设备号*/

/* 2. 定义file_operations结构体*/

/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/

/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个**入口函数***/

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个**出口函数***/

/* 7. 其他完善：提供设备信息，自动创建设备节点*/

hello_drv.c逐帧解析

/* 0.头文件*/

```
#include <linux/module.h>

#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>
```

为什么是这些头文件？我也不知道，因为这个驱动程序是借用另一个驱动程序改的，但某种程度来讲，头文件只要包含就行，不管是不是真的需要，编译器只会将被引用的头文件中的定义转换为实际代码，并将其链接到可执行文件中，不会直接导致可执行文件大小变大。若是缺了头文件则通过`man`手册查找即可

/* 1. 确定主设备号*/

```
static int major = 0;
```

将主设备号设置为0意味着将该设备标识为第一个设备驱动程序。这是一个特殊情况，因为主设备号为0通常被保留用于非常基本的设备驱动程序（例如null，random等）或者动态分配设备号时的一个标记。

/* 2. 定义file_operations结构体*/

```
static struct file_operations hello_drv = {
    .owner   = THIS_MODULE,
    .open    = hello_drv_open,
    .read    = hello_drv_read,
    .write   = hello_drv_write,
    .release = hello_drv_close,
};
```

.owner   = THIS_MODULE功能：指定设备驱动程序的拥有者。其实就是设备驱动程序模块的指针,THIS_MODULE是一个宏定义，用于获取当前模块的指针，这样在内核卸载模块时可以检查驱动程序是否正在使用，并在模块卸载前释放所有相关资源。

```
.open    = hello_drv_open,
.read    = hello_drv_read,
.write   = hello_drv_write,
.release = hello_drv_close,
```

四大定式，类比文件操作

/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */

```
//驱动读函数
static ssize_t hello_drv_read (struct file *file, char __user *buf, size_t size,
loff_t *offset)
{
    int err;
    err = copy_to_user(buf, kernel_buf, MIN(1024, size));
    return MIN(1024, size);
}
//驱动写函数
static ssize_t hello_drv_write (struct file *file, const char __user *buf,
size_t size, loff_t *offset)
{
    int err;
    err = copy_from_user(kernel_buf, buf, MIN(1024, size));
    return MIN(1024, size);
}
//驱动打开函数
static int hello_drv_open (struct inode *node, struct file *file)
{
    return 0;
}
//驱动关闭函数
static int hello_drv_close (struct inode *node, struct file *file)
{
    return 0;
}
```

太简单，都不知道怎么解释好，读函数功能其实就是从kernel_buf内核缓冲区中复制数据到用户缓冲区变量buf中，写函数则是反过来。
唯一可说的点感觉是函数的那些参数为什么这样设计，

对于读写函数：

`struct file *file`：表示被操作的文件，其中包含了与文件相关的信息，例如文件描述符、文件操作相关的函数指针等等。其实就是传入文件描述符(fd)的

`char __user *buf`：表示用户空间中的缓冲区，用于接收读取到的数据，`__user` 表示该指针所指向的内存空间是用户空间的。

`size_t size`：表示欲读取的数据字节数。

`loff_t *offset`：表示当前文件指针的位置，一般情况下是指当前读写位置的偏移量。

对于打开关闭函数：

`struct inode *node`：表示与打开的文件相关联的 `inode` 节点，包含了与文件相关的元数据信息，例如文件的访问权限、文件的大小等等。

`struct file *file`：表示被操作的文件，其中包含了与文件相关的信息，例如文件描述符、文件操作相关的函数指针等等。

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/

/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个**入口函数***/

```
static int __init hello_init(void)
{
    int err;
    //1.注册设备
    major = register_chrdev(0, "hello", &hello_drv);
    //2.创建设备类
    hello_class = class_create(THIS_MODULE, "hello_class");
    //3.检查设备类
    err = PTR_ERR(hello_class);
    if (IS_ERR(hello_class)) {
        unregister_chrdev(major, "hello");
        return -1;
    }
    //4.创建设备节点
    device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello");

    return 0;
}
```

1.register_chrdev() 函数用于注册字符设备，并返回分配给设备的主设备号。函数原型如下：
`int register_chrdev(unsigned int major, const char *name, const struct file_operations *fops);`
major参数表示设备请求的主设备号，如果设置为 0，表示让系统自动分配主设备号。
name参数表示设备的名称。
fops参数表示设备的操作函数，包括读、写、打开、关闭等等（file_operations）

2.class_create() 函数用于创建一个新的设备类，它返回一个 struct class 结构体指针。函数原型如下：
`struct class *class_create(struct module *owner, const char *name);`
owner参数表示该类所属的内核模块，一般使用 THIS_MODULE 表示当前模块。
name参数表示该类的名称。

3.PTR_ERR() 和 IS_ERR() 函数用于检查指针是否为错误指针。如果指针是错误指针，可以使用 PTR_ERR() 函数将指针转换为错误码，然后使用 return 语句返回错误码

```
    if (IS_ERR(hello_class)) {//如果hello_class类指针错误
        unregister_chrdev(major, "hello");//注销hello设备
        return -1;//不正常结束
    }
```

4.unregister_chrdev() 函数用于注销字符设备。函数原型如下：

```
void unregister_chrdev(unsigned int major, const char *name);
```

major参数表示需要注销的设备的主设备号。

name参数表示需要注销的设备的名称。

5.device_create() 函数用于创建一个新的设备节点，并将其与设备文件进行关联。函数原型如下：

```
struct device *device_create(struct class *class, struct device *parent, dev_t
devt, void *drvdata, const char *fmt, ...);
```

class参数表示该设备节点所属的设备类。

parent参数表示该设备节点的父节点。

devt参数表示该设备节点的设备号，一般使用 MKDEV() 宏来创建设备号。参数为主设备号和次设备号

drvdata指向设备私有数据的指针，一般不需要使用。

fmt设备节点名称的格式字符串。

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个**出口函数***/

```
static void __exit hello_exit(void)
{
    device_destroy(hello_class, MKDEV(major, 0));
    class_destroy(hello_class);
    unregister_chrdev(major, "hello");
}
```

其实感觉都没必要解释这段了，怎么创建，就怎么销毁，反着来

/* 7. 其他完善：提供设备信息，自动创建设备节点*/

```
module_init(hello_init);//指定内核模块加载时要执行的初始化函数，即 hello_init() 函数。
module_exit(hello_exit);//指定内核模块卸载时要执行的退出函数，即 hello_exit() 函数。
MODULE_LICENSE("GPL");//指定内核模块的许可证，GPL 表示该模块遵循 GPL 许可证。
```

hello_drv.c(完整)

```
/*0.头文件，这里是直接借用了一个写好的驱动程序案例，但也没关系，某种程度上将，头文件也应该是后面
缺什么补什么*/
#include <linux/module.h>

#include <linux/fs.h>
#include <linux/errno.h>
#include <linux/miscdevice.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/mutex.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/stat.h>
#include <linux/init.h>
#include <linux/device.h>
```

```c
#include <linux/tty.h>
#include <linux/kmod.h>
#include <linux/gfp.h>

//驱动程序需要向内核注册自己以便与设备进行通信。设备号是在设备驱动程序中注册设备时使用的一个标识符。设备号由主设备号和次设备号组成，其中主设备号用于标识设备驱动程序，次设备号用于标识特定的设备实例。
/* 1. 确定主设备号*/
static int major = 0;
static char kernel_buf[1024];
static struct class *hello_class;

#define MIN(a, b) (a < b ? a : b)

/* 3. 实现对应的open/read/write等函数，填入file_operations结构体*/
static ssize_t hello_drv_read (struct file *file, char __user *buf, size_t
size, loff_t *offset)
{
    int err;
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    err = copy_to_user(buf, kernel_buf, MIN(1024, size));
    return MIN(1024, size);
}

static ssize_t hello_drv_write (struct file *file, const char __user *buf,
size_t size, loff_t *offset)
{
    int err;
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    err = copy_from_user(kernel_buf, buf, MIN(1024, size));
    return MIN(1024, size);
}

static int hello_drv_open (struct inode *node, struct file *file)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    return 0;
}

static int hello_drv_close (struct inode *node, struct file *file)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    return 0;
}

/* 2. 定义自己的file_operations结构体*/
static struct file_operations hello_drv = {
    .owner   = THIS_MODULE,
    .open    = hello_drv_open,
    .read    = hello_drv_read,
    .write   = hello_drv_write,
    .release = hello_drv_close,
};

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/
```

```c
/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个入口函数 */
static int __init hello_init(void)
{
    int err;


    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    major = register_chrdev(0, "hello", &hello_drv);


    hello_class = class_create(THIS_MODULE, "hello_class");
    err = PTR_ERR(hello_class);
    if (IS_ERR(hello_class)) {
        printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
        unregister_chrdev(major, "hello");
        return -1;
    }

    device_create(hello_class, NULL, MKDEV(major, 0), NULL, "hello");

    return 0;

}

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个出口函数*/
static void __exit hello_exit(void)
{
    printk("%s %s line %d\n", __FILE__, __FUNCTION__, __LINE__);
    device_destroy(hello_class, MKDEV(major, 0));
    class_destroy(hello_class);
    unregister_chrdev(major, "hello");
}


/* 7. 其他完善：提供设备信息，自动创建设备节点*/

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

hello_drv_test.c

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


/*
```

```
 * ./hello_drv_test -w abc
 * ./hello_drv_test -r
   */

int main(int argc, char **argv)
{
    int fd;
    char buf[1024];
    int len;


    /* 1. 判断参数 */
    if (argc < 2)
    {
        printf("Usage: %s -w <string>\n", argv[0]);
        printf("       %s -r\n", argv[0]);
        return -1;
    }

    /* 2. 打开文件 */
    fd = open("/dev/hello", O_RDWR);
    if (fd == -1)
    {
        printf("can not open file /dev/hello\n");
        return -1;
    }

    /* 3. 写文件或读文件 */
    if ((0 == strcmp(argv[1], "-w")) && (argc == 3))
    {
        len = strlen(argv[2]) + 1;
        len = len < 1024 ? len : 1024;
        write(fd, argv[2], len);
    }
    else
    {
        len = read(fd, buf, 1024);
        buf[1023] = '\0';
        printf("APP read : %s\n", buf);
    }

    close(fd);

    return 0;

}
```

4.2.用户按键驱动程序

对于用户按键驱动而言，还是这七步：

/* 1. 确定主设备号*/

/* 2. 定义file_operations结构体*/

/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/

/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个**入口函数***/

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个**出口函数***/

/* 7. 其他完善：提供设备信息，自动创建设备节点*/

key_drv.c逐帧解析

/* 1. 确定主设备号*/

```
/* key设备结构体 */
struct key_dev{
    dev_t devid;                /* 设备号*///设备号，由主设备号和次设备号组成，用于唯一标识一个设备。
    struct cdev cdev;           /* cdev*///字符设备结构体，用于管理字符设备驱动程序。
    struct class *class;        /* 类*///设备类，用于管理设备驱动程序，可用于在用户空间创建设备文件。
    struct device *device;      /* 设备*///设备，用于表示一个具体的设备实例，可用于向用户空间导出设备信息。
    int major;                  /* 主设备号*///主设备号，用于与驱动程序关联。
    int minor;                  /* 次设备号*///次设备号，用于标识同一类型的不同设备。
    struct device_node  *nd;    /* 设备节点*///设备节点，用于表示设备在设备树中的节点。
    int key_gpio;               /* key所使用的GPIO编号*///key所使用的GPIO编号，用于读取按键输入状态。
    atomic_t keyvalue;          /* 按键值*///按键值，用于存储读取到的按键状态。
};

struct key_dev keydev;          /* key设备*///创建设备结构体

其实很多时候框架，记住就行，不需要太多理解
```

/* 2. 定义file_operations结构体*/

```
/* 设备操作函数 */
static struct file_operations key_fops = {
    .owner = THIS_MODULE,
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release =  key_release,
};

这个没什么好说的吧？几乎和hello一致了
```

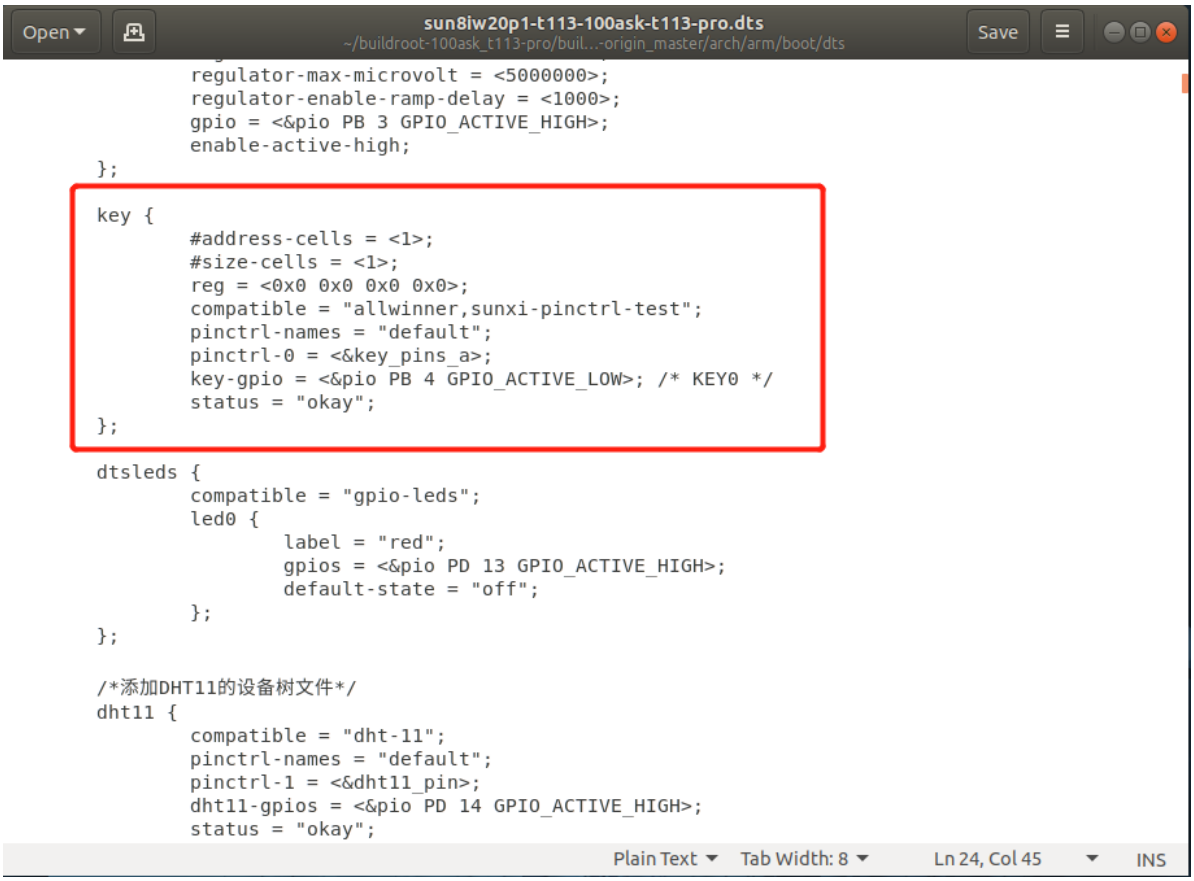
/* 3. 实现对应的open/read/write等函数，填入file_operations结构体 */

3.1.GPIO初始化函数，被用于open函数中调用

```
/*
 * @description : 初始化按键IO，open函数打开驱动的时候
 *                初始化按键所使用的GPIO引脚。
 * @param       : 无
```

```
 * @return      : 无
 */
static int keyio_init(void)
{
    //通过设备树查找 /key 节点，并将其赋值给结构体 keydev 中的成员变量 nd。如果查找失败，返回错误码 -EINVAL。
    keydev.nd = of_find_node_by_path("/key");
    if (keydev.nd== NULL) {
        return -EINVAL;
    }

    //在节点 keydev.nd (key设备节点)中查找名为 key-gpio (包含了GPIO信息)的属性，将其对应的GPIO编号赋值给结构体 keydev 中的成员变量 key_gpio。如果查找失败或者该属性对应的GPIO编号小于0，则返回错误码 -EINVAL
    keydev.key_gpio = of_get_named_gpio(keydev.nd ,"key-gpio", 0);
    if (keydev.key_gpio < 0) {
        printk("can't get key0\r\n");
        return -EINVAL;
    }
    printk("key_gpio=%d\r\n", keydev.key_gpio);

    /* 初始化key所使用的IO */
    gpio_request(keydev.key_gpio, "key0");  /* 请求IO *///请求使用 keydev.key_gpio 对应的GPIO，并将其命名为 key0
    gpio_direction_input(keydev.key_gpio);  /* 设置为输入 *///GPIO配置为输入模式。
    return 0;
}
```

这段代码结合设备树内容来看就很清晰了



3.2.open函数设置私有数据并初始化按键GPIO

```
/*
 * @description     : 打开设备
 * @param - inode   : 传递给驱动的inode
 * @param - filp    : 设备文件，file结构体有个叫做private_data的成员变量
 *                    一般在open的时候将private_data指向设备结构体。
 * @return          : 0 成功;其他 失败
 */
static int key_open(struct inode *inode, struct file *filp)
{
    int ret = 0;
    filp->private_data = &keydev;   /* 设置私有数据 *///将filp->private_data设置为指向keydev结构体的指针，这样在设备驱动程序的其他函数中，可以通过filp->private_data访问keydev结构体中的数据。

    ret = keyio_init();             /* 初始化按键IO */
    if (ret < 0) {
        return ret;
    }

    return 0;
}
```

3.3.read函数

```
/*
 * @description     : 从设备读取数据
 * @param - filp    : 要打开的设备文件(文件描述符)
 * @param - buf     : 返回给用户空间的数据缓冲区
 * @param - cnt     : 要读取的数据长度
 * @param - offt    : 相对于文件首地址的偏移
 * @return          : 读取的字节数，如果为负值，表示读取失败
 */
static ssize_t key_read(struct file *filp, char __user *buf, size_t cnt, loff_t *offt)
{
    int ret = 0;
    int value;
    struct key_dev *dev = filp->private_data;

    if (gpio_get_value(dev->key_gpio) == 0) {       /* key0按下 */
        while(!gpio_get_value(dev->key_gpio));      /* 等待按键释放 */
        atomic_set(&dev->keyvalue, KEY0VALUE);
    } else {
        atomic_set(&dev->keyvalue, INVAKEY);        /* 无效的按键值 */
    }

    value = atomic_read(&dev->keyvalue);
    ret = copy_to_user(buf, &value, sizeof(value));
    return ret;

}

//struct key_dev *dev = filp->private_data;
```

这句代码创建了个struct key_dev类型的指针变量，用来指向filp->private_date，然后这个filp->private_data，
//filp->private_data = &keydev;
又在open函数(调用read必调用open)，取了keydev变量的地址，也就是dev = keydev。

//copy_to_user(buf, &value, sizeof(value));即把value的地址，扔给buf这个指针，后面调用read函数，按键值就保存到了这个传入的buf指针变量中

3.4.write函数

```
/*
 * @description     : 向设备写数据
 * @param - filp    : 设备文件，表示打开的文件描述符
 * @param - buf     : 要写给设备写入的数据
 * @param - cnt     : 要写入的数据长度
 * @param - offt    : 相对于文件首地址的偏移
 * @return          : 写入的字节数，如果为负值，表示写入失败
 */
static ssize_t key_write(struct file *filp, const char __user *buf, size_t cnt,
loff_t *offt)
{
    return 0;
}
```

3.5.release函数

```
/*
 * @description     : 关闭/释放设备
 * @param - filp    : 要关闭的设备文件(文件描述符)
 * @return          : 0 成功;其他 失败
 */
static int key_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

/* 4. 把file_operations结构体告诉内核：注册驱动程序*/

/* 5. 谁来注册驱动程序啊？得有一个入口函数：安装驱动程序时就会去调用这个**入口函数***/

```
/*
 * @description : 驱动入口函数
 * @param       : 无
 * @return      : 无
 */

/*
*按键入口流程：
*1.初始化原子变量：因为按键检测需要一直占用CPU不能被其他进程打断。
*2.注册设备(*)，创建主设备号(这里可以采取，若无主设备号分配，则自动向系统申请一个主设备号)
*3.初始化cdev，初始化字符设备(cdev)结构体(keydev.cdev)，并将字符设备的操作函数集(key_fops)与字符设备结构体关联起来。
*4.创建类(*)
*5.创建设备(*)
```

```
*/
static int __init mykey_init(void)
{
    /* 初始化原子变量 */
    atomic_set(&keydev.keyvalue, INVAKEY);

    /* 注册字符设备驱动 */
    /* 1、注册设备，创建主设备号 */
    if (keydev.major) {     /*  定义了设备号 */
        keydev.devid = MKDEV(keydev.major, 0);
        register_chrdev_region(keydev.devid, KEY_CNT, KEY_NAME);
    } else {                /* 没有定义设备号 */
        alloc_chrdev_region(&keydev.devid, 0, KEY_CNT, KEY_NAME);   /* 申请设备号
*/
        keydev.major = MAJOR(keydev.devid); /* 获取分配号的主设备号 */
        keydev.minor = MINOR(keydev.devid); /* 获取分配号的次设备号 */
    }

    /* 2、初始化cdev */
    keydev.cdev.owner = THIS_MODULE;
    cdev_init(&keydev.cdev, &key_fops);//初始化字符设备(cdev)结构体(keydev.cdev)，并
将字符设备的操作函数集(key_fops)与字符设备结构体关联起来。

    /* 3、添加一个cdev */
    cdev_add(&keydev.cdev, keydev.devid, KEY_CNT);

    /* 4、创建类 */
    keydev.class = class_create(THIS_MODULE, KEY_NAME);
    if (IS_ERR(keydev.class)) {
        return PTR_ERR(keydev.class);
    }

    /* 5、创建设备 */
    keydev.device = device_create(keydev.class, NULL, keydev.devid, NULL,
KEY_NAME);
    if (IS_ERR(keydev.device)) {
        return PTR_ERR(keydev.device);
    }

    return 0;

}

1.atomic_set(&keydev.keyvalue, INVAKEY);//用于将keydev.keyvalue作为原子变量，并且初始
化它的值为INVAKEY，原子操作在多线程并发的情境下，不会被打断，也就是按键按下会一直占用CPU，检测
按键松开。
2.struct cdev { //cdev结构图如下
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
} __randomize_layout;
```

/* 6. 有入口函数就应该有出口函数：卸载驱动程序时，就会去调这个**出口函数***/

```
/*
 * @description : 驱动出口函数
 * @param       : 无
 * @return      : 无
 */
static void __exit mykey_exit(void)
{
    /* 注销字符设备驱动 */
    gpio_free(keydev.key_gpio);
    cdev_del(&keydev.cdev);/*  删除cdev */
    unregister_chrdev_region(keydev.devid, KEY_CNT); /* 注销设备号 */

    device_destroy(keydev.class, keydev.devid); /* 销毁设备 */
    class_destroy(keydev.class); /* 销毁类 */

}

//这个没啥好说的，看看API
```

/* 7. 其他完善：提供设备信息，自动创建设备节点*/

```
module_init(mykey_init);
module_exit(mykey_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("z");
```

key_drv.c(完整)

```
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/delay.h>
#include <linux/ide.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/errno.h>
#include <linux/gpio.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/of_gpio.h>
#include <linux/semaphore.h>
#include <asm/mach/map.h>
#include <asm/uaccess.h>
#include <asm/io.h>

#define KEY_CNT         1       /* 设备号个数     */
#define KEY_NAME        "key"   /* 名字         */

/* 定义按键值 */
#define KEY0VALUE       0XF0    /* 按键值       */
#define INVAKEY         0X00    /* 无效的按键值   */
```

```
/* key设备结构体 */
struct key_dev{
    dev_t devid;            /* 设备号    */
    struct cdev cdev;       /* cdev      */
    struct class *class;    /* 类        */
    struct device *device;  /* 设备      */
    int major;              /* 主设备号    */
    int minor;              /* 次设备号    */
    struct device_node  *nd; /* 设备节点 */
    int key_gpio;           /* key所使用的GPIO编号         */
    atomic_t keyvalue;      /* 按键值       */
};

struct key_dev keydev;      /* key设备 */

/*

 * @description : 初始化按键IO，open函数打开驱动的时候

 * 初始化按键所使用的GPIO引脚。

 * @param       : 无

 * @return      : 无
   */
   static int keyio_init(void)
   {
   keydev.nd = of_find_node_by_path("/key");
   if (keydev.nd== NULL) {
    return -EINVAL;
   }

   keydev.key_gpio = of_get_named_gpio(keydev.nd ,"key-gpio", 0);
   if (keydev.key_gpio < 0) {
    printk("can't get key0\r\n");
    return -EINVAL;
   }
   printk("key_gpio=%d\r\n", keydev.key_gpio);

   /* 初始化key所使用的IO */
   gpio_request(keydev.key_gpio, "key0");   /* 请求IO */
   gpio_direction_input(keydev.key_gpio);   /* 设置为输入 */
   return 0;
   }

/*

 * @description     : 打开设备

 * @param - inode   : 传递给驱动的inode

 * @param - filp    : 设备文件，file结构体有个叫做private_data的成员变量

 * 一般在open的时候将private_data指向设备结构体。
```

```
 * @return          : 0 成功;其他 失败
   */
   static int key_open(struct inode *inode, struct file *filp)
   {
   int ret = 0;
   filp->private_data = &keydev;    /* 设置私有数据 */

   ret = keyio_init();              /* 初始化按键IO */
   if (ret < 0) {
    return ret;
   }

   return 0;
   }

/*

 * @description     : 从设备读取数据

 * @param - filp    : 要打开的设备文件(文件描述符)

 * @param - buf     : 返回给用户空间的数据缓冲区

 * @param - cnt     : 要读取的数据长度

 * @param - offt    : 相对于文件首地址的偏移

 * @return          : 读取的字节数，如果为负值，表示读取失败
   */
   static ssize_t key_read(struct file *filp, char __user *buf, size_t cnt,
loff_t *offt)
   {
   int ret = 0;
   int value;
   struct key_dev *dev = filp->private_data;

   if (gpio_get_value(dev->key_gpio) == 0) {         /* key0按下 */
    while(!gpio_get_value(dev->key_gpio));      /* 等待按键释放 */
    atomic_set(&dev->keyvalue, KEY0VALUE);
   } else {
    atomic_set(&dev->keyvalue, INVAKEY);        /* 无效的按键值 */
   }

   value = atomic_read(&dev->keyvalue);
   ret = copy_to_user(buf, &value, sizeof(value));
   return ret;
   }

/*

 * @description     : 向设备写数据
 * @param - filp    : 设备文件，表示打开的文件描述符
 * @param - buf     : 要写给设备写入的数据
 * @param - cnt     : 要写入的数据长度
```

```
 * @param - offt    : 相对于文件首地址的偏移
 * @return          : 写入的字节数，如果为负值，表示写入失败
   */
   static ssize_t key_write(struct file *filp, const char __user *buf, size_t
cnt, loff_t *offt)
   {
   return 0;
   }

/*

 * @description     : 关闭/释放设备
 * @param - filp    : 要关闭的设备文件(文件描述符)
 * @return          : 0 成功;其他 失败
   */
   static int key_release(struct inode *inode, struct file *filp)
   {
   return 0;
   }

/* 设备操作函数 */
static struct file_operations key_fops = {
    .owner = THIS_MODULE,
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release =  key_release,
};

/*

 * @description : 驱动入口函数

 * @param       : 无

 * @return      : 无
   */
   static int __init mykey_init(void)
   {
   /* 初始化原子变量 */
   atomic_set(&keydev.keyvalue, INVAKEY);

   /* 注册字符设备驱动 */
   /* 1、创建设备号 */
   if (keydev.major) {      /*  定义了设备号 */
    keydev.devid = MKDEV(keydev.major, 0);
    register_chrdev_region(keydev.devid, KEY_CNT, KEY_NAME);
   } else {                      /* 没有定义设备号 */
    alloc_chrdev_region(&keydev.devid, 0, KEY_CNT, KEY_NAME);   /* 申请设备号 */
    keydev.major = MAJOR(keydev.devid); /* 获取分配号的主设备号 */
    keydev.minor = MINOR(keydev.devid); /* 获取分配号的次设备号 */
   }

   /* 2、初始化cdev */
   keydev.cdev.owner = THIS_MODULE;
```

```
    cdev_init(&keydev.cdev, &key_fops);

    /* 3、添加一个cdev */
    cdev_add(&keydev.cdev, keydev.devid, KEY_CNT);

    /* 4、创建类 */
    keydev.class = class_create(THIS_MODULE, KEY_NAME);
    if (IS_ERR(keydev.class)) {
     return PTR_ERR(keydev.class);
    }

    /* 5、创建设备 */
    keydev.device = device_create(keydev.class, NULL, keydev.devid, NULL,
KEY_NAME);
    if (IS_ERR(keydev.device)) {
     return PTR_ERR(keydev.device);
    }

    return 0;
    }

/*

 * @description : 驱动出口函数

 * @param       : 无

 * @return      : 无
   */
    static void __exit mykey_exit(void)
    {
    /* 注销字符设备驱动 */
    gpio_free(keydev.key_gpio);
    cdev_del(&keydev.cdev);/*  删除cdev */
    unregister_chrdev_region(keydev.devid, KEY_CNT); /* 注销设备号 */

    device_destroy(keydev.class, keydev.devid);
    class_destroy(keydev.class);
    }

module_init(mykey_init);
module_exit(mykey_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("z");
```

5.用户按键应用程序

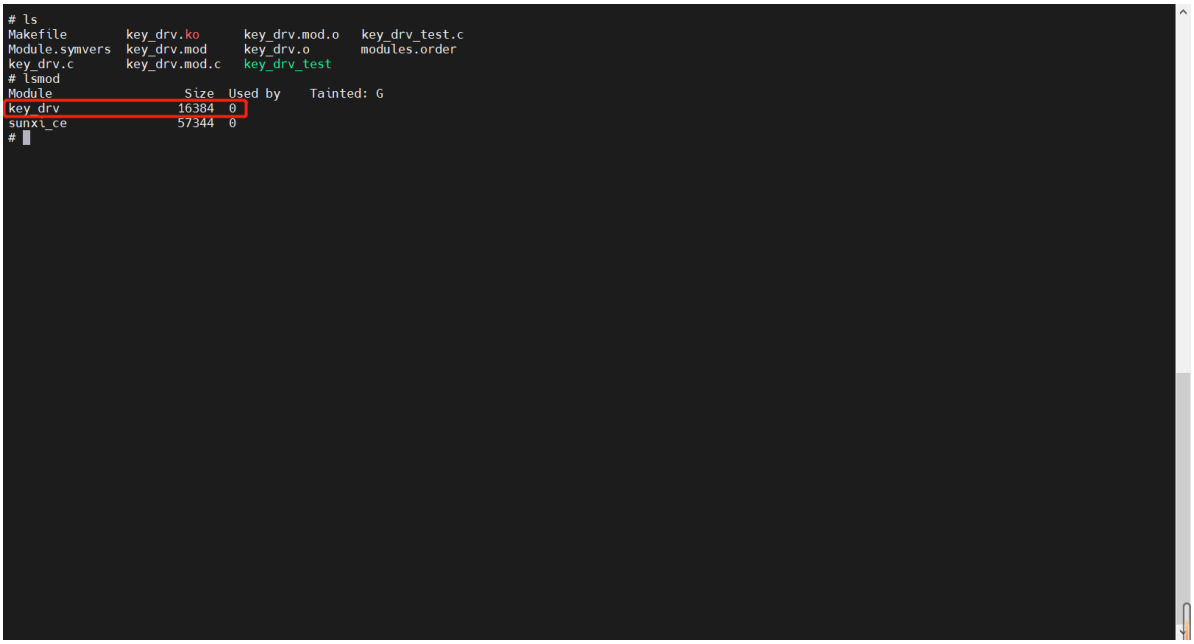
```
#include "stdio.h"
#include "unistd.h"
#include "sys/types.h"
#include "sys/stat.h"
#include "fcntl.h"
#include "stdlib.h"
```

```c
#include "string.h"

/* 定义按键值 */
#define KEY0VALUE   0XF0
#define INVAKEY     0X00


/*
 * @description     : main主程序
 * @param - argc    : argv数组元素个数
 * @param - argv    : 具体参数
 * @return          : 0 成功;其他 失败
 */

int main(int argc, char *argv[])
{
    int fd, ret;
    char *filename;
    int keyvalue;


    if(argc != 2){
        printf("Error Usage!\r\n");
        return -1;
    }

    filename = argv[1];

    /* 打开key驱动 */
    fd = open(filename, O_RDWR);
    if(fd < 0){
        printf("file %s open failed!\r\n", argv[1]);
        return -1;
    }

    /* 循环读取按键值数据！ */
    while(1) {
        read(fd, &keyvalue, sizeof(keyvalue));
        if (keyvalue == KEY0VALUE) {    /* KEY0 */
            printf("KEY0 Press, value = %#X\r\n", keyvalue);    /* 按下 */
        }
    }

    ret= close(fd); /* 关闭文件 */
    if(ret < 0){
        printf("file %s close failed!\r\n", argv[1]);
        return -1;
    }
    return 0;

}
```

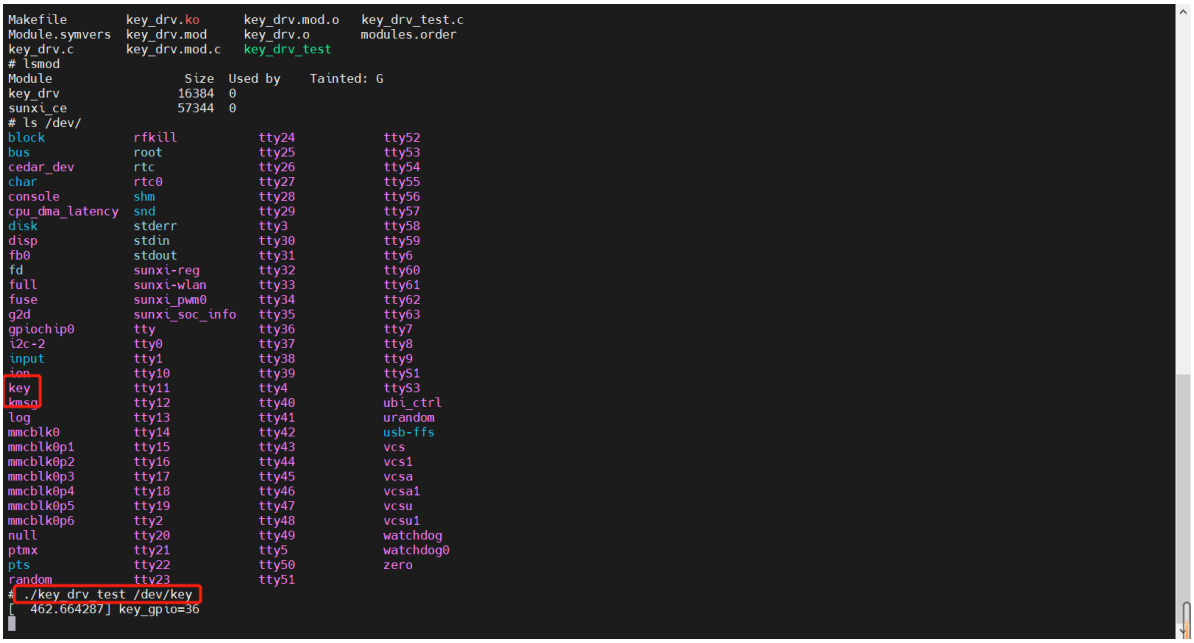
装载驱动程序验证

```
insmod key_drv.ko

lsmod
```

现象：



打开key设备



按下按键，打印按键按下信息，按键值为0xf0，为应用代码中自己定义的宏

```
fb0             stdout          tty31           tty6
fd              sunxi-reg       tty32           tty60
full            sunxi-wlan      tty33           tty61
fuse            sunxi_pwm0      tty34           tty62
g2d             sunxi_soc_info  tty35           tty63
gpiochip0       tty             tty36           tty7
i2c-2           tty0            tty37           tty8
input           tty1            tty38           tty9
ion             tty10           tty39           ttyS1
key             tty11           tty4            ttyS3
kmsg            tty12           tty40           ubi_ctrl
log             tty13           tty41           urandom
mmcblk0         tty14           tty42           usb-ffs
mmcblk0p1       tty15           tty43           vcs
mmcblk0p2       tty16           tty44           vcs1
mmcblk0p3       tty17           tty45           vcsa
mmcblk0p4       tty18           tty46           vcsa1
mmcblk0p5       tty19           tty47           vcsu
mmcblk0p6       tty2            tty48           vcsu1
null            tty20           tty49           watchdog
ptmx            tty21           tty5            watchdog0
pts             tty22           tty50           zero
random          tty23           tty51
# ./key_drv_test /dev/key
[  462.664287] key_gpio=36
clear
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
KEY0 Press, value = 0XF0
```

写个按键点灯的应用：

6.input子系统