网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔,笔记很详细,供大家参考。

也许有错漏,请自行分辨。

目录

1 概念介绍

Sysfs:

LINUX2.6 内核开发了全新的设备模型。它采用 sysfs 文件系统，其类似于 proc 文件系统，用于将系统中设备组织成层次结构，并向用户模式程序提供详细的内核数据结构信息。

内核对象机制：

Kobject:

LINUX2.6 引入的新的设备管理机制。通过这个数据结构使所有设备在底层都具有统一的接口。Kobject 提供基本的对象管理，是构成 LINUX2.6 设备模型的核心结构，其与 sysfs 文件系统紧密关联，每个在内核中注册的 kobject 对象都对应于 sysfs 文件系统中的一个目录。Kobject 通常通过 kset 组织成层次化的结构，kset 是具有相同类型的 kobject 的集合。

设备模型结构（/include/linux/device.h）：

1，devices-设备结构：
2，drivers-驱动结构：
3，buses-总线结构：
4，classes-设备类结构：

目录	主要内容	目录	主要内容
/drivers/char	字符型设备驱动	/drivers/media	视频采集、广播、数字电视设备
/drivers/block	块设备驱动	/drivers/base	一切驱动基本函数
/drivers/net	网络设备驱动	/drivers/usb	USB 设备驱动
/drivers/video	显示相关驱动、控制台设备、启动 LOGO	/drivers/mtd	MTD 设备驱动，包括 FLASH 驱动
/drivers/mmc	MMC/SD 卡驱动	/drivers/serial	串口设备驱动

给驱动模块提供参数：

内核代码 include/linux/module.h 中定义的宏 MODULE_PARM(var,type) 用于向模块传递命令行参数。var 为接受参数值的变量名，type 为采取如下格式的字符串[min[-max]]{b,h,i,l,s}。min 及 max 用于表示当参数为数组类型时，允许输入的数组元素的个数范围；b：byte；h：short；i：int；l：long；s：string。在装载内核模块时，用户可以向模块传递一些参数：

```
insmod modname var=value
```

如果用户未指定参数，var 将使用模块内定义的默认值。

在内核中加入新驱动：

将驱动编译进内核。

若将 pxa_smbus.c 添加到内核：

1，先将其复制到 /drivers/char 目录，更改该目录下的 Kconfig,增加：

```
config SMBUS_PMIC
    tristate "SMBUS Driver for PMIC"
    depends on ARCH_PXA || ARCH_SA1100
    default y

    help
```

2,在该目录下的 Makefile 中增添下行：

```
obj-$(CONFIG_SMBUS_PMIC)+=pxa_smbus.o
```

3，进入源代码目录，执行 make menuconfig:

进入源代码目录，执行 make menuconfig 后，选择 character devices 项，进入图 1.1 所示的界面，在图中选项前如果为<*>，表示模块被编译进内核；如果为<M>，表示编译成可加载模块；如果是< >则表示不编译。如果选择<*>，用 make zImage 就可以了。如果选择<M>，则必须使用 make 命令，生成 ko 文件。

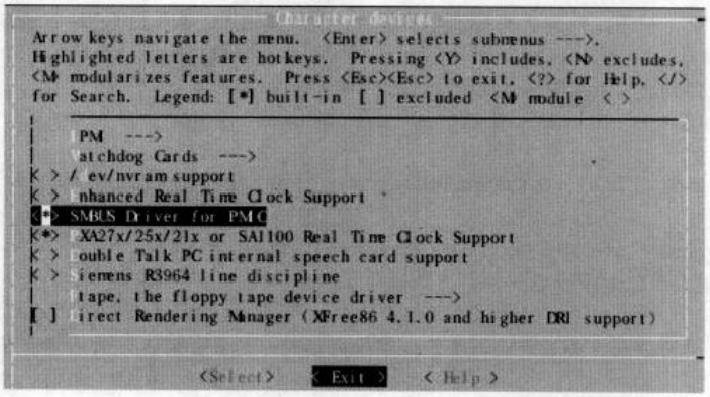
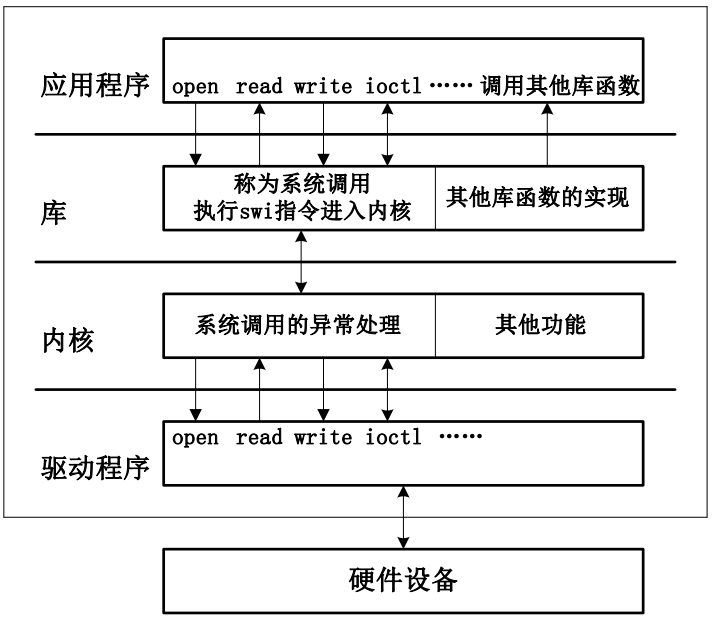


图 1.1 在内核中增加新驱动

驱动程序原理图：



2.字符设备驱动 LED 驱动程序_编写编译：

应用程序：open　　read　　write
驱动程序：led_open　　led_read　　led_write
（最终依赖驱动程序框架来从应用程序对应到驱动程序）

“应用进程” 和 “驱动程序”如何联系：

内核定义了一个 struct file_operations 结构体，这个结构的每一个成员的名字都对应着一个系统调用。

用户进程利用系统调用在对设备文件进行诸如读写操作时，系统调用通过设备文件的主设备号找到相应的设备驱动程序，然后读取这个数据结构相应的函数指针，接着将控制权交给该函数。

```
struct file_operations {
  struct module *owner;
  //模块所有者指针，一般初始化为 THIS_MODULES
  loff_t (*llseek) (struct file *, loff_t, int);
  //用来修改文件当前的读写位置，返回新位置。loff_t 为一个"长偏移量"
  ssize_t (*read) (struct file *, char _user *, size_t, loff_t *);
  //同步读取函数。读取成功返回读取的字节数。设置为 NULL，调用时返回-EINVAL
  ssize_t (*aio_read) (struct kiocb *, char _user *, size_t, loff_t);
  //异步读取操作，为 NULL 时全部通过 read 处理
  ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *);
  //同步写入函数
  ssize_t (*aio_write) (struct kiocb *, const char _user *, size_t, loff_t);
  //异步写入操作
  int (*readdir) (struct file *, void *, filldir_t);
  // 仅用于读取目录，对于设备文件，该字段为 NULL
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  //判断目前是否可以对设备进行读写操作。字段为空时，设备会被认为既可读也可写
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  // 向设备发送 I/O 控制命令的函数。不设置入口点，返回-ENOTTY
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  // 不使用 BLK 的文件系统，将使用此种函数指针代替 ioctl
  long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
  // 在 64 位系统上，32 位的 ioctl 调用，将使用此函数指针代替
  int (*mmap) (struct file *, struct vm_area_struct *);
```

```
    // 用于请求将设备内存映射到进程地址空间。如果无此方法，将访问-ENODEV
    int (*open) (struct inode *, struct file *);
    // 打开设备的函数，如果为空，设备的打开操作永远成功，但系统不会通知驱动程序
    int (*flush) (struct file *);
    // 进程在关闭设备文件描述符副本时调用，执行未完成的操作
    int (*release) (struct inode *, struct file *);
    // file 结构释放时，将调用此指针函数，若 release 与 open 相同可设置为 NULL
    int (*fsync) (struct file *, struct dentry *, int datasync);
    // 刷新待处理的数据，如果驱动程序没有实现，fsync 调用将返回-EINVAL
    int (*aio_fsync) (struct kiocb *, int datasync);
    // 异步的 fsync 函数
    int (*fasync) (int, struct file *, int);
    // 通知设备 FASYNC 标志发生变化，如果设备不支持异步通知，该字段可以为 NULL
    int (*lock) (struct file *, int, struct file_lock *);
    // 实现文件锁，设备驱动常不去实现此 lock
    ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    //实现进行涉及多个内存区域的单次读或写操作
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    // 实现 sendfile 调用的读取部分，将数据从一个文件描述符移到另一个
```

```
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    // 实现 sendfile 调用的另一部分，内核调用将其数据发送到对应文件，每次一个数据页
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
    unsigned long, unsigned long);
    // 在进程地址空间找到一个合适的位置，以便将底层设备中的内存段映射到该位置
    int (*check_flags)(int);
    // 允许模块检查传递给 fcntl(F_SETEL...)调用的标志
    int (*dir_notify)(struct file *filp, unsigned long arg);
    // 应用程序使用 fcntl 来请求目录改变通知时，调用该方法。仅对文件系统有效，驱动程序不必实现
    int (*flock) (struct file *, int, struct file_lock *);
    // 实现文件锁
};
```

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
```

```
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
};
```

在上面的成员函数指针的形参中，`struct file` 表示一个打开的文件。`Struct inode` 表示一个磁盘上的具体文件。

Struct file 数据结构如下：

```
struct file {
    struct list_head f_list;
    struct dentry   *f_dentry;
    struct vfsmount *f_vfsmnt;
    struct file_operations *f_op;
    atomic_t f_count;
    unsigned int f_flags;
    mode_t f_mode;
    int f_error;
    loff_t f_pos;
    struct fown_struct f_owner;
    unsigned int f_uid, f_gid;
    struct file_ra_state f_ra;
    unsigned long f_version;
    void *f_security;
    void *private_data;
    struct address_space *f_mapping;
};
```

成员	含义
f_mode	标识文件的读写权限，它通过 FMODE_READ 和 FMODE_WRITE 位来标示文件是否可读，可写
f_pos	当前读写位置，类型为 loff_t，是 64 位的数
f_flag	文件标志，主要用于进行阻塞/非阻塞型操作时检查。如 O_RDONLY，O_NONBLOCK，O_SYNC 等，驱动程序为了支持非阻塞型操作需要检查这个标志
f_op	文件操作接口函数，在驱动程序中实现
private_data	可以存放任何私有数据。一般用它指向已经分配的数据，在内核销毁 file 结构前要在 release 方法中释放内存
f_dentry	文件对应的目录项结构，一般在驱动中用 filp->f_dentry->d_inode 访问索引节点时用到它
f_vfsmnt	虚拟文件系统挂载点
f_ra	跟踪上次文件操作状态的结构指针

一，内核和驱动过程：

1,最终是要写出驱动程序中的：led_open　　led_read　　led_write 等函数。

```
static int first_drv_open (struct inode * inode, struct file * file)
{
    return 0;
}

static ssize_t first_drv_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
    return 0;
}
```

其中的结构：struct inode 和 struct file

Struct inode:

2,接着要将这些函数告诉内核。

从上到下找到这个 LED 的（如上图），用它前，要让内核知道有 LED 这个东西.

① .定义一个结构体告诉内核。file_operations 结构，填充这个结构。

```
/*
 * NOTE:
 * read, write, poll, fsync, readv, writev, unlocked_ioctl and compat_ioctl
 * can be called without the big kernel lock held in all filesystems.
 */
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
```

```
        long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long,
unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
        ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t, unsigned int);
        ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t, unsigned int);
};
```

用户程序中有什么接口，这个 file_operations 结构中有相应功能的成员。

```
struct file_operations firsr_drv_fops;
```

②有了这个结构后，将这个定义的结构告诉内核。用一个函数 register_chrdev() 告诉内核，即“注册”

```
int register_chrdev(unsigned int major, const char *name,
                    const struct file_operations *fops) 术语为“注册驱动程序”，就是告诉内核
```

参 1：主设备号。
参 2：名字，可以随便写。
参 3：定义的结构（将这个结构告诉内核）

```
root@book-desktop:/work/nfs_root/first_fs# ls /dev/ -l
total 0
crw-rw----+ 1 root audio    14,  12 2011-03-06 11:11 adsp
```

C 表示字符设备。主设备号 14，次设备号 12.

② .上面的注册驱动的函数由谁调用："驱动入口函数"来调用 register_chardev()，有不同的入口函数。

```
int first_drv_init(void)
{
    register_chrdev(major, "first_drv", &first_drv_fops);
    return 0;
}
```

上面的“first_drv_init”即为“驱动入口函数”。
有第 1 个入口函数，第 2 个入口函数，不同的驱动程序有不同的“驱动入口函数”。这些驱动入口函数有不同名字，

内核通过修饰不同的入口函数让内核知道注册的驱动函数是由哪个“入口函数”调用。

```
asmlinkage long sys_poll(struct pollfd __user *ufds, unsigned int nfds,
                         long timeout_msecs)
{
        s64 timeout_jiffies;
```

③ ."入口函数"得修饰。一个 宏“module_init”来定义。

```
modul                e_init(first_drv_init);
```

module_init 就是定义一个结构体，这个结构体中有一个函数指针，指向“入口函数”。
当安装一个驱动程序时，内核回自动找到这样一个结构体，调用里面的“函数指针”，“入口函数”

“入口函数”就将结构“file_operations first_drv_fops”告诉内核。

二，注册驱动函数细节：

①，内核如何找到相应的“file_operations”结构：设备类型 + 主设备号

```
crw-rw----  1 root tty      7,   4  2月 24 2012 vcs4
```

（上面主设备号为 7， 次设备号为 4 。）
应用程序--->
打开一个设备文件 open("dev/xxx"),或 read,write.打开 xxx 有属性：c 字符型设备。
major(主设备号)
Mior (次设备号)

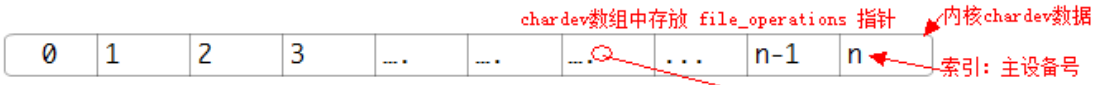

```
register_chrdev(major,"first_drv",&first_drv_fops);
```

register_chrdev 注册驱动程序时，参数有：
参 1，“major”：主设备号
参 2，“名字”：驱动的名字（随便写）。
“file_operations first_drv_fops 结构体(read、write、open 等成员)”

应用程序如何最终找到 register_chrdev 注册的东西（file_operations first_drv_fops 结构），具体：
应用程序 open("dev/xxx") 如何打开结构体 first_drv_fops 中的 open 成员？
用“设备类型->字符类型（这里为字符类型）”+“主设备号”
VFS 系统就是根据打开的文件"open("dev/xxx")中的 xxx，"里面的属性（设备类型+主设备号），根

据这两个属性，就能找到注册（register_chrdev）进去的“file_operations”结构（first_drv_fops）。

②.register_chrdev 最简单的实现方法是：（register_chrdev 的作用）



内核中的“chardev”数组存以“主设备号major”为索引，存放着“file_operations”结构。

在一个内核数组（chardev）中，以“主设备号”major为索引，找到某一项，在这一项中把“file_operations”结构（这里是“first_drv_fops”）填充进去（挂进去）。

总结过程：驱动程序和应用程序的联系

从应用层看：

1 首先,APP 用 Open（“dev/xx”,O_RDWR）打开设备文件后，会得到此设备文件的属性，知道属性中的“设备类型”和“主设备号”。
2 然后 VFS 层通过“设备类型”（如字符设备）去找内核中的“chardev”这个数组。再通过从APP 中

得到的“主设备号”以此为索引从内核的“chardev”数组中知道相应的“file_operations”结构。这个结构
是驱动程序“register_chardev”注册到内核的，这样从索引又找到了它。这个结构中有相应的“读”“写”
成员中，这些成员就对应着硬件的读和硬件的写操作等待如（led_read,led_write）。

（VFS 层工作过程？）

从驱动程序看：如操作 LED

1 首先实现 LED 的读写（led_read,led_write）.
2 然后定义一个“file_operation”结构（操作这个 LED 用），比如此结构中的 Open 成员就指向“led_open”，结构中的 write 就指向对硬件 LED 的操作“led_write”等。
3 再然后，在写的驱动入口函数中，用“register_chardev”把上面定义的“file_operation”

结构体放到内核的“chardev”字符设备数组中去，是放到“chardev”字符设备数组下标为“major”主
设备号编号处。
三.注册是用 register_chardev把对相应硬件的操作定义的“file_operations”结构通过注册时定义的形参“主设备号”放到内核的字符

设备数组中以“主设备号”为索引下标的数组空间去了。

```
int first_drv_init(void)        随意取一个驱动名字
{
    register_chrdev(111, "first_drv", &first_drv_fops); // 注 册 , 告 诉 内 核
    return 0;    给的一个主设备号          操作某硬件相关的 file_operations 结构
}
```

那么要卸载这个驱动程序时，就要把这个硬件对应的“file_operations”结构从内核字符设备chardev数

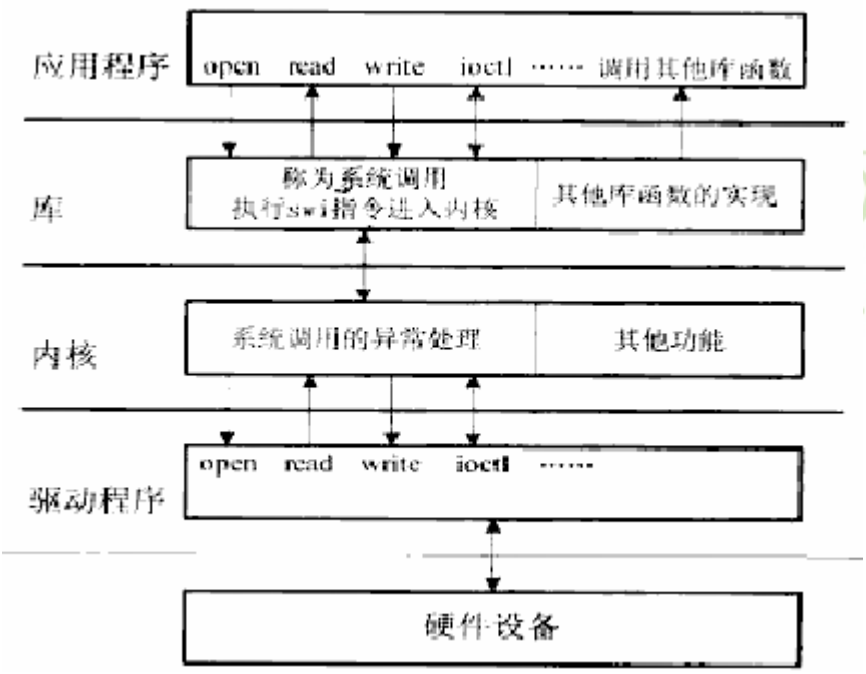
组中取出来（不挂接此硬件 file_operations 结构的地址）。这个过程用“unregister_chrdev”实现。

```
void first_drv_exit(void)
{
    unregister_chrdev(111, "first_drv"); // 卸 载
}
```

主设备号　　驱动名字

只有两个参数，major 和名字。删除驱动是将结构 file_operations 从内核“字符设备数组”中拖出来。
如何让内核知道这个“first_drv_exit”普通函数，也是修饰一下：module_exit();
所谓修饰就是用一个宏来定义一个结构体,结构体中有某个成员“函数指针”指向它。
当卸载驱动程序时，内核就会自动去调用这个结构体中的这个“函数指针”成员。

四，写一个简单的字符驱动程序：



1，一个简单的框架：

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/delay.h>
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <asm/io.h>
#include <asm/arch/regs-gpio.h>
#include <asm/hardware.h>

static int first_drv_open(struct inode *inode,struct file *file)
{
    printk(KERN_ALERT,"first_drv_open\n");
    return 0;
}
static ssize_t first_drv_write(struct file *file,const char __user *buf,size_t count,loff_t *ppos)
{
    printk(KERN_ALERT,"first_drv_write\n");
    return 0;
}

static struct file_operations first_drv_fops = {
    .open = first_drv_open,
    .write = first_drv_write,
};

static int first_drv_init(void)
{
    register_chrdev(111,"first_drv",&first_drv_fops);
    return 0;
}

void first_drv_exit(void)
{
    unregister_chrdev(111,"first_drv");
}

module_init(first_drv_init);
module_exit(first_drv_exit);
MODULE_LICENSE("GPL");
```

以前这里犯了两个错。
1，要加
MODULE_LICENSE("GPL");
2,函数前要加“static”.

Makefile:

```
# 内核要先编译好。
KERN_DIR = /work/system/linux-2.6.22.6

# -C 是转到后面的目录'$(KERN_DIR)' ，使用这个目录下的 Makefile 来编译。
# M 是指当前目录是什么。
# modules 是目标。
all:
    make -C $(KERN_DIR) M=`pwd` modules

clean:
    make -C $(KERN_DIR) M=`pwd` modules clean
    rm -rf modules.order

obj-m   += first_drv.o
```

```
ian:/work/ARM/nfs_root/romfs/my_drivers_test # make
make -C /work/ARM/system/linux-2.6.22.6 M=`pwd` modules
make[1]: 进入目录“/work/ARM/system/linux-2.6.22.6”
  CC [M]  /work/ARM/nfs_root/romfs/my_drivers_test/myled.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/ARM/nfs_root/romfs/my_drivers_test/myled.mod.o
  LD [M]  /work/ARM/nfs_root/romfs/my_drivers_test/myled.ko
make[1]: 离开目录“/work/ARM/system/linux-2.6.22.6”
```

```
# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 29 fb
 90 mtd
 99 ppdev
111 first_drv
116 alsa
128 ptm
136 pts
180 usb
```

```
book@book-desktop:/work/nfs_root/romfs/my_test$ make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/nfs_root/romfs/my_test/first_drv.o
/work/nfs_root/romfs/my_test/first_drv.c: In function `first_drv_open':
/work/nfs_root/romfs/my_test/first_drv.c:16: warning: implicit declaration of function `printf'
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: "puts" [/work/nfs_root/romfs/my_test/first_drv.ko] undefined!
  CC      /work/nfs_root/romfs/my_test/first_drv.mod.o
  LD [M]  /work/nfs_root/romfs/my_test/first_drv.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
```

```
# insmod first_drv.ko
first_drv: module license 'unspecified' taints kernel.   没有许可信息。
first_drv: Unknown symbol puts
insmod: cannot insert 'first_drv.ko': Unknown symbol in module (-1): No such file or directory
```

因为一个函数前没有加“static”所以一直出现下面的错误：

```
# insmod first_drv.ko
first_drv: Unknown symbol puts
insmod: cannot insert 'first_drv.ko': Unknown symbol in module (-1): No such file or directory
# dmesg -c
first_drv: Unknown symbol puts
#
```

```
static int first_drv_init(void)
{
    register_chrdev(111,"first_drv",&first_drv_fops);
    return 0;
}
```

刚才上面的没有加“static”。

```
first_drv: module license 'unspecified' taints kernel.
first_drv: Unknown symbol puts
first_drv: Unknown symbol puts
first_drv: Unknown symbol puts
first_drv: Unknown symbol puts
first_drv: Unknown symbol puts
first_drv: Unknown symbol puts
#
# insmod first_drv.ko
first_drv: Unknown symbol puts
insmod: cannot insert 'first_drv.ko': Unknown symbol in module (-1): No such file or director
# dmesg -c
first_drv: Unknown symbol puts                    上面是出错信息。
#
# insmod first_drv.ko
#                    在函数前都加了“static”后驱动模块加载成功。
```

驱动程序中设备号：

自动分配主设备号
手工分配主设备号

应用程序设备节点：

手动通过主设备号创建设备节点 mknod /dev/??? 设备类型 主设备号 次设备号
自动创建设备节点。
应用程序中有一个"udev"机制，对于 busybox 来说是 mdev。

注册一个驱动程序后，会在 /sys/目录下生成相关设备的硬件信息。mdev程序会根据这些sys目录下提供的信息自动创建设备节点。所以要在驱动程序中提供这些sys目录下的信息。

系统信息要先创建类，再然后在类下面创建设备。创建设备信息时，会提供主设备号和名字：

```
/drivers_test/1_led # cat /etc/init.d/rcS
#mount -t proc none /proc
mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
/drivers_test/1_led #
```

内核中有设备加载或卸载掉时，就会通过“/proc/sys/kernel/hotplug”所指示的应用程序（上面是/sbin/mdev）去自动加载或卸载设备节点。

```
leddrv_class = class_create (THIS_MODULE, "led"); /* 在sys目录下创建led设备类目录 */
```

第一个参数指定类的所有者是哪个模块，第二个参数指定类名。

```
struct class_device *class_device_create(
                        struct class        *cls,       设备类
                        struct class_device *parent,
                        dev_t               devt,
                        struct device       *device,    类下的设备
                        const char          *fmt, ...)  类下设备的名字
```

```
leddrv_class_devs = class_device_create (leddrv_class, NULL, MKDEV(major, 0), NULL, DEV_NAME);
```

第一个参数指定所要创建的设备所从属的类，第二个参数是这个设备的父设备，如果没有就指定为NULL，第三个参数是设备号，第四个参数是设备名称，第五个参数是从设备号。

将*device对应的逻辑设备(class_device)添加到*cls所代表的设备类中。在*cls所在目录下，建立代表逻辑设备的目录。

完善点亮 LED：

要点亮 LED 灯：

1.写好框架。

2.完善硬件操作：

一，分析：完善硬件操作

①.看原理图，确定引脚。

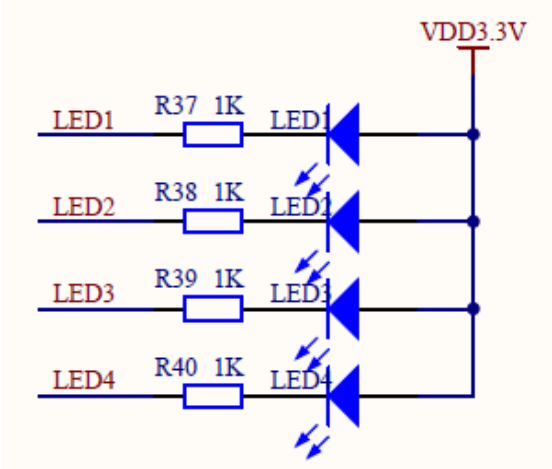
②.看 2440 的芯片手册。查如何操作引脚。

③.写代码。

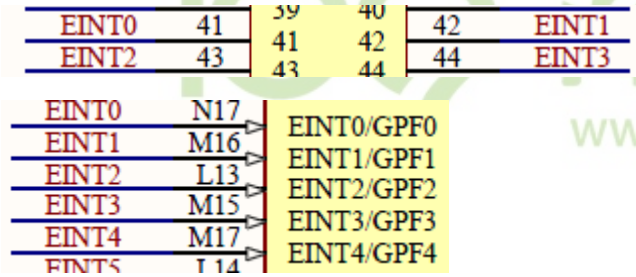
写单片机时是直接操作那个物理地址。
驱动程序中不能直接操作物理地址了，是操作虚拟地址。用 ioremap 函数来将物理地址映射成虚拟地址。

先查原理图：

1， 先看主板的原理图：MOTHERBOARD V3.pdf



再看核心板的原理图：



从原理图上看，GPF0~3 引脚输出 0 ，LED 会亮。
1.查原理图看到三个 LED 灯接的引脚是 GPF4~6,则再看 2440 芯片手册操作相应的寄存器。
在芯片手册中搜索 GPFCON 寄存器。

GPFCON	Bit	Description	
GPF7	[15:14] 两位对应一个I/O口	00 = Input 00为输入 10 = EINT[7] 10是第二功能	01 = Output 01为输出 11 = Reserved 11是没有被使用的。
GPF6	[13:12]	00 = Input 10 = EINT[6]	01 = Output 11 = Reserved
GPF5	[11:10]	00 = Input 10 = EINT[5]	01 = Output 11 = Reserved
GPF4	[9:8]	00 = Input 10 = EINT[4]	01 = Output 11 = Reserved
GPF3	[7:6]	00 = Input 10 = EINT[3]	01 = Output 11 = Reserved
GPF2	[5:4]	00 = Input 10 = EINT2]	01 = Output 11 = Reserved
GPF1	[3:2]	00 = Input 10 = EINT[1]	01 = Output 11 = Reserved
GPF0	[1:0]	00 = Input 00为输入 10 = EINT[0] 10为中断	01 = Output 11 = Reserved

GPF可作为外部中断来使用。对应了外部中断的0到7（EINT[0]--EINT[7]）或设置成 10 就当成了外部中断管脚来使用了。

要驱动LED就要配置这4个管脚作为输出，使能上拉。往这里送0或1的数据。就可以控制LED亮或灭。

GPF有7个I/O口，每个I/O都有三组寄存器GPFCON,GPFDAT,GPFUP。

2.先将 GPF0~3 配置成 Output 输出引脚。

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configures the pins of port F 配置输入或输出再或第二功能	0x0
GPFDAT	0x56000054	R/W	The data register for port F 是管脚的数据	Undef.
GPFUP	0x56000058	R/W	Pull-up disable register for port F 设置是否使用上拉功能。	0x000
Reserved	0x5600005c	–	0是只带上拉，1指不上拉。	–

GPFCON寄存器的物理地址

3.操作 GPFCON 这个寄存器的物理地址是：0x5600 0050 再操作 GPFDAT 这个寄存器。即可。

二，写代码：硬件上的操作：

配置引脚：操作 GPFCON 寄存器。（放到 open 函数中做）
设置：设备 GPFDAT 寄存器，让引脚输出高、低电平。（放到 wirte 函数中做）

1，映射物理地址：

```
volatile unsigned long *gpfcon = NULL;
volatile unsigned long *gpfdat = NULL;
```

先定义两个变量，和寄存器相对应。他们分别的物理地址要分别映射成虚拟地址。
在入口函数中映射（避免打开一次要映射一次）。

用 ioremap (开始地址，结束大小)

```
gpfcon = (volatile unsigned long *) ioremap(0x5600050,16);
gpfdat = gpfcon + 1;
```

先映射GPFCON为虚拟地址。F 寄存为 GPFCON:0x56000050, GPFDAT:0x56000054, GPFUP:0x56000058, Reserved:0x5600005c 共4个，2440为32位位CPU，所以4个4字节为 16字节，故这里映射16字节。

Gpfdat = gpfcon + 1; 这里指针的加 1 是以上面“unsinged long”为单位的。

0x56000050 这是：

Register	Address
GPFCON	0x56000050
GPFDAT	0x56000054
GPFUP	0x56000058
Reserved	0x5600005c

GPFCON 寄存器的地址，后面 16 是 16 字节，这里刚好有 4 个寄存器，每个 4 字节，所以就定成 16 字节了。
这里加 1 不是加了 4 字节，指针的操作是以 指针 所指向的长度为单位的，这里的指针是：“(volatile unsigned long *)”
在入口函数中用 ioreamp() ，则在出口函数中要用 iounmap(),将建立的映射去掉。
iounmap(gpfcon)
以上就是物理地址与虚拟地址的映射，以后要操作寄存器时，就用 gpfcon 和 gpfdat 两个指针来操作。
这两个指针指向的是虚拟地址。

```
int major;
int first_drv_init(void) //这 是 入 口 函 数
{
    major = register_chrdev(0,"first_drv",&first_drv_fops);
    firstdrv_class = class_create(THIS_MODULE, "firstdrv"); //先 创 建 一 个 类 。
    firstdrv_class_dev = class_device_create(firstdrv_class, NULL, MKDEV(major, 0), NULL, "

    gpfcon = (volatile unsigned long *) ioremap(0x5600050,16);
    gpfdat = gpfcon + 1;

    return 0;
}

void first_drv_exit(void)  出口函数
{
    unregister_chrdev(major,"first_drv" );

    class_device_unregister(firstdrv_class_dev);
    class_destroy(firstdrv_class);

    iounmap(gpfcon);
}
```

2，设置引脚为输出引脚：先清零，再或上“01”。

之后以为配置成“输出引脚”，是原理图上 LED 一端接了高电平，只有 2440 这端输出低电平，才有电流通过。
设置成输出引脚：

GPF3	[7:6]	00 = Input 10 = EINT[3]	01 = Output 11 = Reserved
GPF2	[5:4]	00 = Input 10 = EINT2]	01 = Output 11 = Reserved
GPF1	[3:2]	00 = Input 10 = EINT[1]	01 = Output 11 = Reserved
GPF0	[1:0]	00 = Input 10 = EINT[0]	01 = Output 11 = Reserved

GPF0~3 先清零，再或上 1 。就是输出引脚了。在“first_drv_open”中初始化 GPFCON 引脚电平为输出。

```
/* 配置 GPF0~3 为输出引脚 */
gpfcon &= ~((0x3<<0*2) | (0x3<<1*2) | (0x3<<2*2) | (0x3<<3*2)); /* 先清0 */
gpfcon |= ((0x1<<0*2) | (0x1<<1*2) | (0x1<<2*2) | (0x1<<3*2));  /* 再或上1为输出 */
```

gpfcon 本身就代表了 GPFCON 寄存的地址（虚拟地址）：
gpfcon = gpfcon & (~((0x3<<0*2) | (0x3<<1*2) | (0x3<<2*2) | (0x3<<3*2)));
“0x3”二进制为“11”
GPF0 占 bit0~1，则要将它清 0 则为：
~(0x3<<0)

GPF1 占 bit2~3，则要清 0：
~（0x3<<2）

GPF2 占 bit4~5,清 0 则为：
~（0x3<<4）
等等。

3，将用户空间的数据传到内核空间：
val 如何传进来，就是 *buf ,应用程序中调用的时候:

```
int main(int argc, char **argv)
{
    int fd;
    int val = 1;
    fd = open("/dev/xyz, O_RDWR);
    if(fd<0)
        printf("cat't open! \n");

    write(fd, &val, 4); &val，相当于 first_drv_write中的__user *buf;
                        4这个长度相当于 first_drv_write中的 size_t count
    return 0;
}
```

```
static ssize_t first_drv_write(struct file *file, const char __user *buf, size_t count, loff_t * ppos))
```

这个值便是 应用程序 传进来的 buf 和 count.用一个函数来取这些应用程序传过来的值。

从用户空间到内核空间的数据传递函数 copy_from_user()。
从内核空间向用户空间传递数据用函数 copy_to_user();

```
static inline unsigned long __must_check copy_from_user(void *to, const void __user *from, unsigned l
{
    if (access_ok(VERIFY_READ, from, n))
        n = __copy_from_user(to, from, n);
    else /* security hole - plug it */
        memzero(to, n);
    return n;
}
```

将传来的 buf 拷贝到 val（定义的）空间，拷贝的长度是 count（应用程序传进来的）。

接着根据这个值判断：如果这个值 val==1,打开 LED（输出低电平）。否则关掉 LED 灯（输出高电平）。
打开和关闭 LED，就是操作寄存器。

GPFDAT	Bit	Description
GPF[7:0]	[7:0]	When the port is configured as an input port, the corresponding bit is the pin state. When the port is configured as an output port, the pin state is the same as the corresponding bit. When the port is configured as functional pin, the undefined value will be read.

```
static int first_drv_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
    int val;
    printk("first_drv_write\n");
    copy_from_user (&val, buf, count);

    if (val == 1)
    {
        /* 点灯 : 输出低电平 */
        *gpfdat &= ~((1<<0) | (1<<1) | (1<<2) | (1<<3));
    }
    else
    {
        /* 灭灯 : 输出高电平 */
        *gpfdat |= (1<<0) | (1<<1) | (1<<2) | (1<<3);
    }
    return 0;
}
```

当传进来的值 val 为 1，依原理图就要设置 GPF0~3 这些引脚输出 0.那就是清位。要关 LED 时，就使引脚输出高电平，就或上某一位。这里 val = 1,是通过 copy_from_user()将"first_drv_write()"中的用户空间的 buf 值传进的。另一种方法是用 ioctl（）。驱动测试程序 firstdrv_test 打开 /dev/led_test 设备节点时，就会通过设备节点的主设备号进入到内核的 file_operations 结构数组中找到与这个驱动的主设备号相关的结构体，进而关联到 sys_open 等函数上去。

```
if (strcmp(argv[1], "on") == 0) {
    val = 1;
} else {
    val = 0;
}

write(fd, &val, 4); /* 将&val中前4字节写入到fd关联的文件内，1个 int 占4字节。*/
```

上面是测试程序，打开'on'时，val=1,从 write(fd,&val,4); 中将这个 val=1 写到 fd 关联的设备节点“/dev/led_tset”，进而通过它的主设备号进入到内核，关联到：

```
/drivers_2.6.22/1_led # ./firstdrv_test on
first_drv_open
first_drv_write
/drivers_2.6.22/1_led # ./firstdrv_test off
first_drv_open
first_drv_write
/drivers_2.6.22/1_led #
```

```
static struct file_operations first_drv_fops = {
    .owner = THIS_MODULE,
    .open = first_dev_open,
    .write = first_drv_write,
};
```

First_drv_fops 结构中去，write 关联到“int first_drv_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)”，这样，val=1 这个值就传给了“const char __user *buf”。可以看到这是一个用户空间的数据，接着通过“copy_from_user”将这个用户空间的值拷贝到内核空间。

以上程序就已经能同时打开或关闭 4 个 LED 了。
要分别点亮 4 个 LED 时，可以通过程序测试程序中的 write()不同的 val 值对应不同的 file_operations 中的写操作。还有一个方法是利用“次设备号”。

总结：写一个驱动程序。

1.首先写驱动框架。
2.结硬件的操作。
看原理图--看芯片手册--写代码
和单片机差不多，区别在于：
单片机中直接使用物理地址。
驱动程序中得用物理地址映射后的函数地址。
要单独控制某一个 LED 灯。

```
static int first_drv_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
    int val;
    printk("first_drv_write\n");
    copy_from_user (&val, buf, count);

    if (val == 1)
    {
        /* 点灯 : 输出低电平 */
        *gpfdat &= ~((1<<0) | (1<<1) | (1<<2) | (1<<3));
    }
    else
    {
        /* 灭灯 : 输出高电平 */
        *gpfdat |= (1<<0) | (1<<1) | (1<<2) | (1<<3);
    }
    return 0;
}
```

用户空间，通过关联到设备节点文件的fd传进来的val值（write(fd,val,4)）根据这个值的不同，copy_from_user后从用户空间拷贝到内核空间的值不同而if...else。

1 种方法是解析传进来的值，如传进来 1 点亮 LED1,传进来 2 val == 2,则点亮 LED2.根据传进

来的值执行不同动作。

2 种方法：设备节点

```
/drivers_2.6.22/1_led # ls /dev/led_test -l
crw-rw----    1 0        0         252,    0 Jan  1 05:31 /dev/led_test
/drivers_2.6.22/1_led #
```

主设备号是帮我们在内核的 chardrv 数组中找到那一项，再根据这一项找到 file_operatios 结构：

```
static struct file_operations first_drv_fops = {
    .owner = THIS_MODULE,
    .open = first_dev_open,
    .write = first_drv_write,
};
```

但是上面的 次设备号 是留下来给我们用的，由我们管理。

```
int minor = MINOR(inode->i_rdev); //MINOR(inode->i_cdev);
```

用 MINOR 取出次设备号。据不同次设备号来操作。次设备的操作完全由自己定义。
看“myleds.c”这个驱动程序。


```
static int s3c24xx_leds_open(struct inode *inode, struct file *file)
{
    int minor = MINOR(inode->i_rdev); //MINOR(inode->i_cdev);
        minor取出次设备号
    switch(minor)
    {
        case 0: /* /dev/leds */
        {
            //配 置 3引 脚 为 输 出
            //s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF4_OUTP);
            GPFCON &= ~(0x3<<(4*2));
            GPFCON |= (1<<(4*2));

            //s3c2410_gpio_cfgpin(S3C2410_GPF5, S3C2410_GPF5_OUTP);
            GPFCON &= ~(0x3<<(5*2));
            GPFCON |= (1<<(5*2));
```

下面是内核提供的使用虚拟地址操作方式：封装好的。

```
case 1: /* /dev/led1 */
{
    s3c2410_gpio_cfgpin(S3C2410_GPF4, S3C2410_GPF4_OUTP);
    s3c2410_gpio_setpin(S3C2410_GPF4, 0);

    down(&leds_lock);
    leds_status &= ~(1<<0);
    up(&leds_lock);

    break;
}
```

```
# ls /dev/led* -l
crw-rw----    1 0        0         231,   1 Jan  1 01:51 /dev/led1
crw-rw----    1 0        0         231,   2 Jan  1 01:51 /dev/led2
crw-rw----    1 0        0         231,   3 Jan  1 01:51 /dev/led3
crw-rw----    1 0        0         231,   0 Jan  1 01:51 /dev/leds
#
```

```
    leds_class_devs[0] = class_device_create(leds_class, NULL, MKDEV(LED_MAJOR, 0), NULL, "leds");
    /*上面设备为leds,主设备号为“LED_MAJOR",次设备号为 0 */
    for (minor = 1; minor < 4; minor++)
    {  /*下面循环创建次设备,次设备的名字："led%d",minor 。若minor为 1 ,则名字是led1*/
        leds_class_devs[minor] = class_device_create(leds_class, NULL, MKDEV(LED_MAJOR, minor), NULL,
"led%d", minor);
```

3 查询方式获取按键值：

```
HH:/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key # make
make -C /work/arm/system/linux-2.6.22.6 M=`pwd` modules
make[1]: 进入目录“/work/arm/system/linux-2.6.22.6”
  CC [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second.mod.o
  LD [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second.ko
make[1]: 离开目录“/work/arm/system/linux-2.6.22.6”
```

```
/drivers_2.6.22/2_key # insmod second.ko
/drivers_2.6.22/2_key # lsmod
second 1996 0 - Live 0xbf000000
```

```
/drivers_2.6.22/2_key # lsmod
second 1996 0 - Live 0xbf000000
/drivers_2.6.22/2_key # cat /proc/devices |grep second
252 second_drv
/drivers_2.6.22/2_key # ls /dev/buttuons -l 主设备号和次设备号
crw-rw----    1 0        0         252,  10 Jan  1 04:44 /dev/buttuons
/drivers_2.6.22/2_key #
```

```
/* 要自动生成设备节点，则要为 MKDEV(主设备号，次设备号) */
second_drv_class_dev = class_device_create (second_drv_class, NULL, MKDEV(major, 10), NULL, "buttuons");
```


一，驱动框架实现：

```
一，驱动框架：
 1，先定义 file_operations 结构体，其中有对设备的打开，读，和写的操作函数。
 2，分别定义相关的操作函数。
 3，定义好对设备的操作函数的结构体(file_operations)后，将其注册到内核的
    file_operations 结构数组中。此设置的主设备号为此结构在数组中的下标。
 4，定义出口函数：卸载注册到内核中的设备相关资源。
 5，修饰 入口 和 出口 函数。
 6，给系统提供更多的内核信息，在 sys 目录下提供设备的相关信息。应用程序 udev 可以据此自动创建设备节点.
    创建一个class设备类，在此类下创建设备。
```

```
#include <linux/kernel.h> //内核头文件，含有一些内核常用函数的原形定义。
#include <linux/module.h> //最基本的头文件，支持动态添加和卸载模块。Hello World驱动要这一个文件就可以。
#include <linux/fs.h>     //包含了文件操作相关struct的定义，例如struct file_operations。
#include <linux/device.h> //包含 class_creat() 等函数定义。
#include <linux/init.h>
#include <linux/delay.h>
#include <asm/uaccess.h>  //包含了copy_to_user、copy_from_user等内核访问用户进程内存地址的函数定义。
#include <asm/irq.h>
#include <asm/io.h>       //包含了ioremap、iowrite等内核访问IO内存等函数的定义。
#include <asm/arch/regs-gpio.h>
#include <asm/hardware.h>

#define DEV_NAME     "second_drv"
```

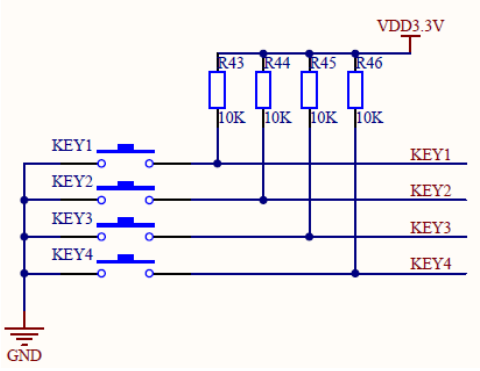
```c
/* 5，为系统提供 sys 目录下的设备相关信息，让应用程序 udev 自动创建设备节点。 */
static struct class *second_drv_class;
static struct class_device *second_drv_class_dev;

/*1.1.定义file_operations结构中的‘打开’操作。struct file 表示一个打开的文件。Struct inode 表示一个磁盘上的具体文件。*/
static int second_drv_open(struct inode *inode, struct file *file)
{
    return 0;
}

/*
 *  1.2.file 是要读的文件，buf是指文件要读到的地方，size是指要读文件多少，ppos是指这次对文件进行操作的起始位置。
 *  struct file 中的f_pos是最后一次文件操作以后的当前读写位置.
 */
ssize_t second_drv_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
    return 0;
}

/*1，定义一个一个file_operations 结构体，有'打开'和'读'操作。 */
static struct file_operations second_drv_fops = {
    .owner = THIS_MODULE,
    .open = second_drv_open,
    .read = second_drv_read,
};
```

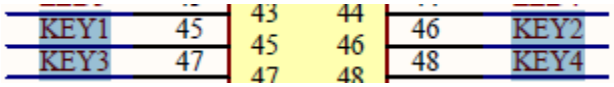
```c
int major;
/* 2，定义入口函数 */
static int second_drv_init(void)
{
    /* 2，1，注册 file_operations 结构，主设备号自动分配，设备名为 second_drv */
    major = register_chrdev (0, DEV_NAME, &second_drv_fops);

    /* 5.1，创建设备类和类一的设备 */
    second_drv_class = class_create (THIS_MODULE, DEV_NAME);
    /* 要自动生成设备节点，则要为 MKDEV(主设备号，次设备号) */
    second_drv_class_dev = class_device_create (second_drv_class, NULL, MKDEV(major, 10), NULL, "buttuons");
    return 0;
}

/* 3，定义出口函数 */
static void second_drv_exit(void)
{
    unregister_chrdev (major, DEV_NAME); //注销注册到内核中的设备。
    class_device_unregister (second_drv_class_dev);
    class_unregister (second_drv_class);
}

/* 4，修饰上面的 入口 和 出口 函数 */
module_init (second_drv_init);
module_exit (second_drv_exit);

MODULE_LICENSE("GPL"); //要上下空一行，不然有警告。
```

二，硬件操作：

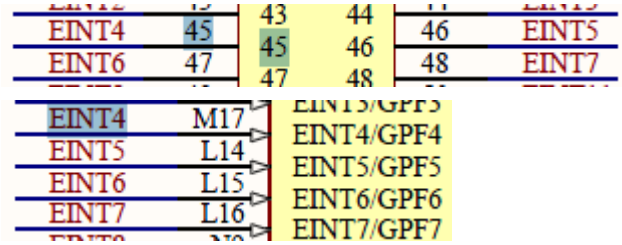
1，看原理图：查找引脚定义。

查主板：





查核心板：



所以可以知道：KEY1 - GPF4

KEY2 - GPF5

KEY3 - GPF6

KEY4 - GPF7

2，设置 4 个引脚为输入引脚：从上面的原理图可以看到，低电平要从开关接地端输入。要有电流流过，就是有压差，从上面的原理图上知道，KEY 一端接了高电平，当 KEY 没按下时，这个 KEY 接 2440 的引脚都是高电平。当 KEY 按键按下，就接通了地，这时这些引脚就返回 0 。

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configure the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undef.

要操作这两个寄存器，驱动中要 ioremap

```
/* 6.2，在入口函数中建立寄存器的地址映射 */
gpfcon = (volatile unsigned long *)ioremap(0x56000050, 4);
gpfdat = gpfcon + 1;   //指针加1，这里即上面的gpfcon地址加4. GPFDAT寄存器的物理地址：0x56000054。
```

GPFCON	Bit	Description	
GPF7	[15:14]	00 = Input	01 = Output
		10 = EINT[7]	11 = Reserved
GPF6	[13:12]	00 = Input	01 = Output
		10 = EINT[6]	11 = Reserved
GPF5	[11:10]	00 = Input	01 = Output
		10 = EINT[5]	11 = Reserved
GPF4	[9:8]	00 = Input	01 = Output
		10 = EINT[4]	11 = Reserved

00 为输入

```
/* 6.3，配置 GPF4~7 为输入引脚：相应的位清0. GPFCON寄存器每个引脚占2位，则可以让0x3（即二进制为 11）左移到相应的位上去，
 * 然后取反即清 0 。
 */
*gpfcon &= ~((0x3<<(4*2)) | (0x3<<(5*2)) | (0x3<<(6*2)) | (0x3<<(7*2)));
```

3，在 read 操作中，返回 4 个引脚的状态。然后将状态值从内核空间拷贝到用户空间。当驱动测试程序读设备节点时，调用到驱动中的“second_drv_read()”函数。

```c
ssize_t second_drv_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{ //参1，要读的文件；参2，将文件读到哪里去；参3，要读文件多少内容；参5，修改文件当前的读写位置，返回新位置。
    /* 6.4，在read函数中返回 4 个引脚的电平状态.GPFDAT 为 0-7 位。 */
    unsigned char key_vals[4];  //每组元对应一个引脚。
    int regval;

    if(size != sizeof(key_vals))      //若存放4个KEY的引脚数据的数组大小不等于‘要读的内容大小’，则返回错误。
        return -EINVAL;               //EINVAL是指‘参数值不正确’。

    regval = *gpfdat;    //取出gpfdat值给regval.
    /* regval & (1<<4) 是指让regval的bit4位与1(1<<4即1左移4位，将bit4置为1)相与,若regval的bit4位原来是'1'则与上1结果为1.
     * (regval & (1<<4)) ? 1 : 0;结果则为1.引脚状态的值是‘1’或‘0’。
     * 因为 GPFDAT 端口F的数据寄存器是 GPF[7:0]这8位。是每一个引脚就占1bit，不像GPFCON是每一引脚占2位。所以这里用 0x1<<(1*0)
     * 这样的计算方法。
     */
    key_vals[0] = (regval & (1<<4)) ? 1 : 0;
  //  printk("key_vals[0]:%d\n", key_vals[0]);
    key_vals[1] = (regval & (1<<5)) ? 1 : 0;
    key_vals[2] = (regval & (1<<6)) ? 1 : 0;
    key_vals[3] = (regval & (1<<7)) ? 1 : 0;
  //  printk("key_vals[3]:%d\n", key_vals[3]);

    /* 6.5，将端口F的数据寄存器bit4~7的值拷贝到‘用户空间’去。file_operations结构中read操作函数的参2即为用户空间buf地址。 */
    if(copy_to_user ((char __user *)buf, key_vals, sizeof(key_vals))) //将key_val数组中的引脚数据拷贝到用户空间buf缓充区。
        return -EFAULT;

    return sizeof(key_vals);     //返回读到的数据大小。
} ? end second_drv_read ?
```

4，测试程序：

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

#define DEV_NAME    "/dev/buttons"
int main(int argc, char **argv)
{
    int fd;
    unsigned char key_vals[4];
    int cnt = 0;

    fd = open(DEV_NAME, O_RDWR);
    if (fd < 0)
    {
        printf("can't open\n");
    }

    while (1)
    {
        read(fd, key_vals, sizeof(key_vals));
        if(!key_vals[0] || !key_vals[1] || !key_vals[2] || !key_vals[3])
        {
            printf("%04d key pressed: %d %d %d %d\n", cnt++, key_vals[0], key_vals[1], key_vals[2], key_vals[3]);
        }
    }
    close(fd);
    return 0;
} ? end main ?
```

```
HH:/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key # make
make -C /work/arm/system/linux-2.6.22.6 M=`pwd` modules
make[1]: 进入目录“/work/arm/system/linux-2.6.22.6”
  CC [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_2.o
include/asm/uaccess.h: In function `second_drv_read':
/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_2.c:86: warning: ignorin
g return value of `copy_to_user', declared with attribute warn_unused_result
  Building modules, stage 2.
  MODPOST 1 modules                    要判断 copy_to_user的返回值
  CC      /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_2.mod.o
  LD [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_2.ko
make[1]: 离开目录“/work/arm/system/linux-2.6.22.6”
HH:/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key #
```

```c
if(copy_to_user ((char __user *)buf, key_vals, sizeof(key_vals)))
    return -EFAULT; 判断copy_to_user的返回值。
```

判断 copy_from_user、copy_to_user 两个函数的返回值，正常返回 0，出错则返回没有复制成功的字节数。出错时返回-EFAULT。

加载驱动：

```
/drivers_2.6.22/2_key # insmod second_2.ko
/drivers_2.6.22/2_key # lsmod
second_2 2516 0 - Live 0xbf000000
/drivers_2.6.22/2_key # cat /proc/devices |grep second
252 second_drv
/drivers_2.6.22/2_key # ls /dev/buttuons -l
crw-rw----    1 0        0         252,  10 Jan  1 02:30 /dev/buttuons
/drivers_2.6.22/2_key # x
```

KEY2

```
0463 key pressed: 1 0 1 1
0464 key pressed: 1 0 1 1
```

KEY3

```
0297 key pressed: 1 1 0 1
0298 key pressed: 1 1 0 1
```

KEY4

```
0124 key pressed: 1 1 1 0
0125 key pressed: 1 1 1 0
```

抖动的很厉害。

查询的按键方式因为里面有一个死循环 while，会占用 CPU 资源：

```
Mem: 6644K used, 54544K free, 0K shrd, 0K buff, 2068K cached
CPU: 14.9% usr 84.8% sys  0.0% nic  0.0% idle  0.0% io  0.1% irq  0.0% sirq
Load average: 0.08 0.02 0.01 2/20 788
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
  787   773 0        R     1308  2.1   0 99.2 ./second_drv_test
  788   773 0        R     3060  4.9   0  0.4 top
```

所以查询的方式不可取。查询是连续去读取按键值，去比较值是否有变化。这种方法很耗资源。

LINUX 异常处理结构、中断处理结构：

一，Linux 异常处理体系结构 框架：

将 2440 作为单片机使用时：裸机程序时
1，按键按下时。
2，CPU 发生中断。
强制的跳到异常向量处执行（中断是异常的一种）。
3，“入口函数”是一条跳转指令。跳到某个函数：（作用）
①，保存被中断处的现场（各种寄存器的值）。
②，执行中断处理函数。
③，恢复被中断的现场。

LINUX 中处理中断的过程：

1，写程序时先设置异常入口：

```
@ 0x18：中断模式的向量地址
@这里很多都没有实现，只是从这里执行。中断后跳到0x18处开始执行。
@做的事情是跳转到（b）函数HandleIRQ处执行。这些异常的入口只有4字节，显然4字节不能处理那么多的事情。所以通常这4
@一条跳转指令。转到更为复杂的处理上去。
    b    HandleIRQ                    @发生中断时，是跳到 0x18 地址，这是“异常向量 ”的入口。跳转到“HandleIRQ ”
```

发生“中断”时，就跳到 0x18 地址处，跳转到“HandleIRQ”是执行下面的指令：

linux 中：

```
HandleIRQ:
    sub lr, lr, #4                  @ 计算返回地址。中断了别的程序，中断完后还要返回来继续执行被中断的程序。当前lr值减4（ARM规定）
    stmdb   sp!,    { r0-r12,lr }   @ 保存使用到的寄存器，保存现场被中断的那些寄存器值。
                                    @ 注意，此时的sp是中断模式的sp
                                    @ 初始值是上面设置的3072

    ldr lr, =int_return             @ 设置调用ISR即EINT_Handle函数后的返回地址
    ldr pc, =EINT_Handle            @ 调用中断服务函数，在interrupt.c中。调用中断程序。处理完后下面就恢复之前的被中断程序。
int_return:
    ldmia   sp!,    { r0-r12,pc }^  @ 中断返回，^表示将spsr的值复制到cpsr
```

中断处理完后，要返回去继续执行之前被中断的那个程序。
保存寄存器就是保存中断前那个程序的所用到的寄存器。
然后是处理中断，最后是恢复。

异常向量在哪里：

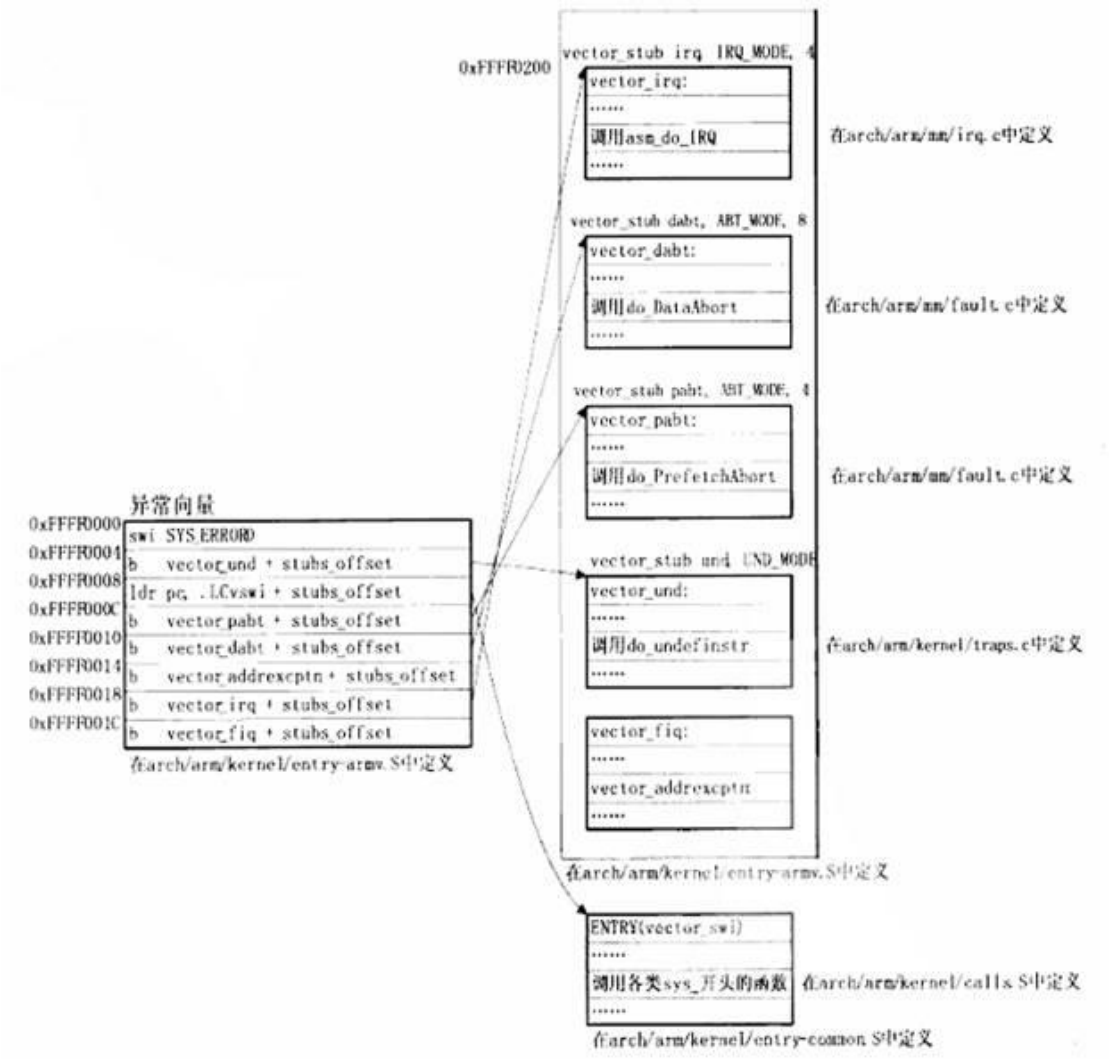


图 20.1　ARM 架构 Linux 内核的异常处理体系结构

表 20.1　　**ARM 架构 Linux 中常见的异常**

异 常 总 类	异 常 细 分
未定义指令异常	ARM 指令 break
	Thumb 指令 break
	ARM 指令 mrc
指令预取中止异常	取指时地址翻译错误（translation fault），系统中还没有为这个指令的地址建立映射关系
数据访问中止异常	访问数据时段地址翻译错误（section translation fault）
	访问数据时页地址翻译错误（page translation fault）
	地址对齐错误
	段权限错误（section permission fault）
	页权限错误（page permission fault）
	…
中断异常	GPIO 引脚中断、WDT 中断、定时器中断、USB 中断、UART 中断等
swi 异常	各类系统调用，sys_open、sys_read、sys_write 等

①，LINUX 的异常向量在哪里：

ARM 架构的 CPU 的异常向量基址可以是 0x0000 0000,也可以是 0xffff0000，LINUX 内核使用后者，只需要在某个寄存器里设置下，就可以将异常基址定位到这里来。这个地址并不代表实际的内存，是虚拟地址。当建立了虚拟地址与物理地址间的映射后，得将那些异常向量，即相当于把那些跳转指令（如:HandleSWI 等）复制拷贝到这个 0xffff0000 这个地址处去。（“那些跳转指令”是指 head.S 中那些跳转）。
这个过程是在 trap_init 这个函数里做。

```
void __init trap_init(void)
{
#if    defined(CONFIG_KGDB)
    return;
}
```

trap_init 函数将异常向量复制到 0xffff0000 处，部分代码如下：

```
unsigned long vectors = CONFIG_VECTORS_BASE;
00732:        memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
00733:        memcpy((void *)vectors + 0x200, __stubs_start, __stubs_end - __stubs_start);
00734:        memcpy((void *)vectors + 0x1000 - kuser_sz, __kuser_helper_start, kuser_sz);
```

如上：
将 __vectors_start, __vectors_end - __vectors_start 这段代码拷贝到 vectors 来。
vectors 是“CONFIG_VECTORS_BASE” 是个配置项（内核的配置选项）。

在 linux 源码顶层目录下：vim .config，搜索“CONFIG_VECTORS_BASE”。

```
# CONFIG_ARCH_HAS_ILOG2_U32 is not set
# CONFIG_ARCH_HAS_ILOG2_U64 is not set
CONFIG_GENERIC_HWEIGHT=y
CONFIG_GENERIC_CALIBRATE_DELAY=y
CONFIG_ZONE_DMA=y
CONFIG_VECTORS_BASE=0xffff0002
```

我的内核配置文件这个地址是“0xffff0002”和书上的不同。

```
---- __stubs_start Matches (6 in 2 files) ----
entry-armv.S (arch\arm\kernel): .globl  __stubs_start
entry-armv.S (arch\arm\kernel):__stubs_start:
```

__vectors_start 在 entry-armv.S 中定义，也是些跳转指令。

```
    .globl  __vectors_start
__vectors_start:
    swi SYS_ERROR0
    b   vector_und + stubs_offset
    ldr pc, .LCvswi + stubs_offset
    b   vector_pabt + stubs_offset
    b   vector_dabt + stubs_offset
    b   vector_addrexcptn + stubs_offset
    b   vector_irq + stubs_offset
    b   vector_fiq + stubs_offset

    .globl  __vectors_end
__vectors_end:
```

可见和单片中的一样（都是跳转指令）。

A:假设发生了 vector_und （undefined）异常未定义指令后，会跳转到 vector_und 加一个偏移地址 stubs_offset（b vector_und + stubs_offset）。这个 vector_und 地址标号源代码里没有，它是一个宏：

```
/*
 * Undef instr entry dispatcher
 * Enter in UND mode, spsr = SVC/USR CPSR, lr = SVC/USR PC
 */
    vector_stub und, UND_MODE

    .long   __und_usr               @  0 (USR_26 / USR_32)
    .long   __und_invalid               @  1 (FIQ_26 / FIQ_32)
    .long   __und_invalid               @  2 (IRQ_26 / IRQ_32)
    .long   __und_svc               @  3 (SVC_26 / SVC_32)
    .long   __und_invalid               @  4
    .long   __und_invalid               @  5
    .long   __und_invalid               @  6
    .long   __und_invalid               @  7
    .long   __und_invalid               @  8
    .long   __und_invalid               @  9
    .long   __und_invalid               @  a
    .long   __und_invalid               @  b
    .long   __und_invalid               @  c
    .long   __und_invalid               @  d
    .long   __und_invalid               @  e
    .long   __und_invalid               @  f

    .align  5
```

将这个宏展开：

```
00896:      .macro    vector_stub, name, mode, correction=0
00897:      .align    5
00898:
00899: vector_\name:
00900:      .if \correction
00901:      sub  lr, lr, #\correction
00902:      .endif
00903:
00904:      @
00905:      @ Save r0, lr_<exception> (parent PC) and spsr_<exception>
00906:      @ (parent CPSR)
00907:      @
00908:      stmia     sp, {r0, lr}            @ save r0, lr
00909:      mrs  lr, spsr
00910:      str   lr, [sp, #8]            @ save spsr
00911:
00912:      @
00913:      @ Prepare for SVC32 mode.  IRQs remain disabled.
00914:      @
00915:      mrs  r0, cpsr
00916:      eor  r0, r0, #(\mode ^ SVC_MODE)
00917:      msr  spsr_cxsf, r0
00918:
00919:      @
00920:      @ the branch table must immediately follow this code
00921:      @
00922:      and  lr, lr, #0x0f
00923:      mov r0, sp
00924:      ldr   lr, [pc, lr, lsl #2]
00925:      movs      pc, lr              @ branch to handler in SVC mode
00926:      .endm
```


```
00896:      .macro    vector_stub, name, mode, correction=0
00897:      .align    5
00898:
00899: vector_\name:
00900:      .if \correction
00901:      sub  lr, lr, #\correction
00902:      .endif
00903:
00904:      @
00905:      @ Save r0, lr_<exception> (parent PC) and spsr_<exception>
00906:      @ (parent CPSR)
00907:      @
00908:      stmia     sp, {r0, lr}            @ save r0, lr
00909:      mrs  lr, spsr
00910:      str   lr, [sp, #8]            @ save spsr
00911:
00912:      @
00913:      @ Prepare for SVC32 mode.  IRQs remain disabled.
00914:      @
00915:      mrs  r0, cpsr
00916:      eor  r0, r0, #(\mode ^ SVC_MODE)
00917:      msr  spsr_cxsf, r0
00918:
00919:      @
00920:      @ the branch table must immediately follow this code
00921:      @
00922:      and  lr, lr, #0x0f
00923:      mov r0, sp
00924:      ldr   lr, [pc, lr, lsl #2]
00925:      movs      pc, lr              @ branch to handler in SVC mode
00926:      .endm
```

vector_stub und, UND_MODE 这个宏展开替换下面的语句：

.macro vector_stub, name, mode, correction=0 （.macro 开始定义宏）

把宏展开，上面的 name 就是"und"。则下来替换后，“vector_\name”就成了“vector_und”
vector_\name:（变成 vector_und: ）定义了一个 vector_und 标号。做的事情如下。
.if \correction 因为上面“correction=0”，即这里是：.if 0.所以 if...endif 间的代码忽略。
sub lr, lr, #\correction
.endif（这三句因为"correction=0,忽略不要"）

@
@ Save r0, lr_<exception> (parent PC) and spsr_<exception>先保存。
@ (parent CPSR)
@
stmia sp, {r0, lr} @ save r0, lr
mrs lr, spsr
str lr, [sp, #8] @ save spsr

@
@ Prepare for SVC32 mode. IRQs remain disabled.转换到管理模式。
@
mrs r0, cpsr
eor r0, r0, #(\mode ^ SVC_MODE)
msr spsr_cxsf, r0

@
@ the branch table must immediately follow this code 这里是下一级跳转
@
and lr, lr, #0x0f
mov r0, sp
ldr lr, [pc, lr, lsl #2]
movs pc, lr @ branch to handler in SVC mode
.endm （.endm 结束宏定义）

这里是下一级的跳转，跳转表如下：

```
/*
 * Undef instr entry dispatcher
 * Enter in UND mode, spsr = SVC/USR CPSR, lr = SVC/USR PC
 */
    vector_stub und, UND_MODE

    .long   __und_usr              @  0 (USR_26 / USR_32)
    .long   __und_invalid          @  1 (FIQ_26 / FIQ_32)
    .long   __und_invalid          @  2 (IRQ_26 / IRQ_32)
    .long   __und_svc              @  3 (SVC_26 / SVC_32)
    .long   __und_invalid          @  4
    .long   __und_invalid          @  5
    .long   __und_invalid          @  6
    .long   __und_invalid          @  7   跳转表
    .long   __und_invalid          @  8
    .long   __und_invalid          @  9
    .long   __und_invalid          @  a
    .long   __und_invalid          @  b
    .long   __und_invalid          @  c
    .long   __und_invalid          @  d
    .long   __und_invalid          @  e
    .long   __und_invalid          @  f

    .align  5
```

①，name 是 und

②，correction=0 默认是 0，则：

```
.if \correction
sub     lr, lr, #\correction
.endif
```

则这三句不用管。

③，下一级跳转表：

```
vector_stub   und, UND_MODE

.long     __und_usr             @ 0 (USR_26 / USR_32)
.long     __und_invalid         @ 1 (FIQ_26 / FIQ_32)
.long     __und_invalid         @ 2 (IRQ_26 / IRQ_32)
.long     __und_svc             @ 3 (SVC_26 / SVC_32)
.long     __und_invalid         @ 4
.long     __und_invalid         @ 5
.long     __und_invalid         @ 6
.long     __und_invalid         @ 7
.long     __und_invalid         @ 8
.long     __und_invalid         @ 9
.long     __und_invalid         @ a
.long     __und_invalid         @ b
.long     __und_invalid         @ c
.long     __und_invalid         @ d
.long     __und_invalid         @ e
.long     __und_invalid         @ f

.align    5
```

后面 __und_usr 等标号中，去保存那些寄存器，作处理。处理完后，再恢复那些寄存器，即恢复那些被中断的程序。

B:下面是一个实例：在 entry-armv.S 源码中的第 1069 行：

```
    .globl  __vectors_start
__vectors_start:
    swi SYS_ERROR0
    b   vector_und + stubs_offset        //vector_und是undefined未定义异常指令。它是一个宏，由上面的"vector_stub   und, UND_MODE
    ldr pc, .LCvswi + stubs_offset
    b   vector_pabt + stubs_offset
    b   vector_dabt + stubs_offset
    b   vector_addrexcptn + stubs_offset
    b   vector_irq + stubs_offset        //发生中断便跳转到这里。这个地址标号“vector_irq”在代码中也没有。也是一个宏来定义。
    b   vector_fiq + stubs_offset

    .globl  __vectors_end
__vectors_end:
```

发生中断便跳转到这里。这个地址标号“vector_irq”在代码中也没有。也是一个宏来定义的：

```
00933:      vector_stub   irq, IRQ_MODE, 4
00934:
00935:      .long     __irq_usr          @ 0 (USR_26 / USR_32)
00936:      .long     __irq_invalid      @ 1 (FIQ_26 / FIQ_32)
00937:      .long     __irq_invalid      @ 2 (IRQ_26 / IRQ_32)
00938:      .long     __irq_svc          @ 3 (SVC_26 / SVC_32)
00939:      .long     __irq_invalid      @ 4
00940:      .long     __irq_invalid      @ 5
00941:      .long     __irq_invalid      @ 6
00942:      .long     __irq_invalid      @ 7
00943:      .long     __irq_invalid      @ 8
00944:      .long     __irq_invalid      @ 9
00945:      .long     __irq_invalid      @ a
00946:      .long     __irq_invalid      @ b
00947:      .long     __irq_invalid      @ c
00948:      .long     __irq_invalid      @ d
00949:      .long     __irq_invalid      @ e
00950:      .long     __irq_invalid      @ f
```


也是和上面的“vector_stub und, UND_MODE”一样，是用下面的宏展开：

```
00896:      .macro    vector_stub, name, mode, correction=0
00897:      .align    5
00898:
00899: vector_\name:
00900:      .if \correction
00901:      sub  lr, lr, #\correction
00902:      .endif
00903:
00904:      @
00905:      @ Save r0, lr_<exception> (parent PC) and spsr_<exception>
00906:      @ (parent CPSR)
00907:      @
00908:      stmia     sp, {r0, lr}          @ save r0, lr
00909:      mrs  lr, spsr
00910:      str  lr, [sp, #8]          @ save spsr
00911:
00912:      @
00913:      @ Prepare for SVC32 mode.  IRQs remain disabled.
00914:      @
00915:      mrs  r0, cpsr
00916:      eor  r0, r0, #(\mode ^ SVC_MODE)
00917:      msr  spsr_cxsf, r0
00918:
00919:      @
00920:      @ the branch table must immediately follow this code
00921:      @
00922:      and  lr, lr, #0x0f
00923:      mov r0, sp
00924:      ldr  lr, [pc, lr, lsl #2]
00925:      movs      pc, lr          @ branch to handler in SVC mode
00926:      .endm
```

vector_stub irq, IRQ_MODE, 4 这里是一个宏，将这个宏和宏参数替换到下面的宏定义中去：

“name”为“irq”.
“correction”为“4”。

```
.macro  vector_stub, name, mode, correction=0
.align  5
```

```
vector_\name:                    宏替换后：vector_irq:
.if \correction                  宏替换后：.if 4
sub  lr, lr, #\correction        宏替换后：sub lr, lr, #4
.endif
```

这里宏替换后就变成：

```
vector_irq:
.if 4
sub lr, lr, #4
.endif
```

```
vector_\name:  vector_irq:
    .if \correction
    sub  lr, lr, #\correction
    .endif
```

sub lr,lr,#4
计算这个返回地址

"sub lr, lr, #4"这是计算返回地址，比较裸机程序：
“F:\embedded\嵌入式 LINUX 开发完全手册\嵌入式 LINUX 开发完全手册光盘\视频\ask100_example\int”下的：head.S

```
HandleIRQ:
    sub lr, lr, #4                  @ 计算返回地址。中断了别的程序，中断完后还要返回来继续执行被中断的程序。当前lr值减4（A
    stmdb   sp!,    { r0-r12,lr }   @ 保存使用到的寄存器，保存现场被中断的那些寄存器值。
                                    @ 注意，此时的sp是中断模式的sp
                                    @ 初始值是上面设置的3072

    ldr lr, =int_return             @ 设置调用ISR即EINT_Handle函数后的返回地址
    ldr pc, =EINT_Handle            @ 调用中断服务函数，在interrupt.c中。调用中断程序。处理完后下面就恢复之前的被中断程序。
int_return:
    ldmia   sp!,    { r0-r12,pc }^  @ 中断返回，^表示将spsr的值复制到cpsr
```

```
@
@ Save r0, lr_<exception> (parent PC) and spsr_<exception>      转换到“管理模式”。
@ (parent CPSR)
@
stmia       sp, {r0, lr}            @ save r0, lr
mrs         lr, spsr
str         lr, [sp, #8]            @ save spsr

@
@ Prepare for SVC32 mode.    IRQs remain disabled.
@
```

```
mrs     r0, cpsr
eor     r0, r0, #(\mode ^ SVC_MODE)
msr     spsr_cxsf, r0

@
@ the branch table must immediately follow this code      这里是下一级跳转
@
and     lr, lr, #0x0f
mov     r0, sp
ldr     lr, [pc, lr, lsl #2]
movs    pc, lr                  @ branch to handler in SVC mode
.endm
```

下一级跳转的跳转表：

```
vector_stub irq, IRQ_MODE, 4

.long   __irq_usr               @  0  (USR_26 / USR_32)
.long   __irq_invalid           @  1  (FIQ_26 / FIQ_32)
.long   __irq_invalid           @  2  (IRQ_26 / IRQ_32)
.long   __irq_svc               @  3  (SVC_26 / SVC_32)
.long   __irq_invalid           @  4
.long   __irq_invalid           @  5
.long   __irq_invalid           @  6
.long   __irq_invalid           @  7
.long   __irq_invalid           @  8   下一级跳转表
.long   __irq_invalid           @  9
.long   __irq_invalid           @  a
.long   __irq_invalid           @  b
.long   __irq_invalid           @  c
.long   __irq_invalid           @  d
.long   __irq_invalid           @  e
.long   __irq_invalid           @  f
```

上面的跳转表，从名字上猜测：

__irq_usr：应该是用户态发生中断时，就跳转到“__irq_usr”标号中去。

__irq_svc：在管理模式里发生中断时，就跳转到“__irq_svc”这个标号中去。

查看标号“__irq_usr”：

```
00405: __irq_usr:
00406:      usr_entry
```

这个宏 usr_entry 应该是保存那些寄存器。搜索“usr_entry”的宏定义如下：

```
entry-armv.S (arch\arm\kernel): .macro svc_entry
```

```
    .macro  svc_entry
    sub sp, sp, #S_FRAME_SIZE
 SPFIX( tst sp, #4       )
 SPFIX( bicne    sp, sp, #4   )
    stmib   sp, {r1 - r12}

    ldmia   r0, {r1 - r3}
    add r5, sp, #S_SP       @ here for interlock avoidance
    mov r4, #-1             @  ""  ""       ""        ""
    add r0, sp, #S_FRAME_SIZE   @  ""  ""       ""        ""
 SPFIX( addne    r0, r0, #4   )
    str r1, [sp]            @ save the "real" r0 copied
                    @ from the exception stack

    mov r1, lr

    @
    @ We are now ready to fill in the remaining blanks on the stack:
    @
    @  r0 - sp_svc
    @  r1 - lr_svc
    @  r2 - lr_<exception>, already fixed up for correct return/restart
    @  r3 - spsr_<exception>
    @  r4 - orig_r0 (see pt_regs definition in ptrace.h)
    @
    stmia   r5, {r0 - r4}
    .endm
```

在这个栈（sp）中将寄存器保存进去。

接着再看：__irq_svc 标号的实现，会有个：

```
00418:      irq_handler
```

"irq_handler"也是一个宏：

```
/*
 * Interrupt handling.  Preserves r7, r8, r9
 */
    .macro  irq_handler
    get_irqnr_preamble r5, lr
1:  get_irqnr_and_base r0, r6, r5, lr
    movne   r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @
    adrne   lr, 1b
    bne asm_do_IRQ

#ifdef CONFIG_SMP
    /*
     * XXX
     * this macro assumes that irqstat (r6) and base (r5) are
     * preserved from get_irqnr_and_base above
     */
    test_for_ipi r0, r6, r5, lr
    movne   r0, sp
    adrne   lr, 1b
    bne do_IPI

#ifdef CONFIG_LOCAL_TIMERS
    test_for_ltirq r0, r6, r5, lr
    movne   r0, sp
    adrne   lr, 1b
    bne do_local_timer
#endif
#endif
    .endm
```

从上面的宏定义可以看到，最终会调用一个 asm_do_IRQ .就是处理函数，比较复杂的代码

就用 C 语言实现。

最后是调用：

```
00111: asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
00112: {
00113:      struct pt_regs *old_regs = set_irq_regs(regs);
00114:      struct irq_desc *desc = irq_desc + irq;
00115:
00116:      /*
00117:       * Some hardware gives randomly wrong interrupts.  Rather
00118:       * than crashing, do something sensible.
00119:       */
00120:      if (irq >= NR_IRQS)
00121:           desc = &bad_irq_desc;
00122:
00123:      irq_enter();
00124:
00125:      desc_handle_irq(irq, desc);
00126:
00127:      /* AT91 specific workaround */
00128:      irq_finish(irq);
00129:
00130:      irq_exit();
00131:      set_irq_regs(old_regs);
00132: } ? end asm_do_IRQ ?
```

总结：

linux 内核中处理异常的流程，最后调用到“asm_do_IRQ（）”

①，异常向量：

```
00712: void __init trap_init(void)
00713: {
00714: #if  defined(CONFIG_KGDB)
00715:       return;
00716: }
```


首先“trap_init”构造了“异常向量”。

②，异常向量是什么？

```
00732:       memcpy((void *)vectors, __vectors_start, __vectors_end - __vectors_start);
```

异常向量就是将这段代码“__vectors_start”拷贝到 0xffff0000“vectors”处：

```
01062: __vectors_start:
01063:       swi  SYS_ERROR0
01064:       b    vector_und + stubs_offset
01065:       ldr  pc, .LCvswi + stubs_offset
01066:       b    vector_pabt + stubs_offset
01067:       b    vector_dabt + stubs_offset
01068:       b    vector_addrexcptn + stubs_offset
01069:       b    vector_irq + stubs_offset
01070:       b    vector_fiq + stubs_offset
01071:
```

异常向量就在这里。这“异常向量”也是某些跳转。如：“b　　vector_irq + stubs_offset”因为向量已重新定位了，所以得加上“stubs_offset”偏移地址。“vector_irq”是链接地址，要加上一个偏移地址才能正确的跳转。

③，vector_irq 做的事：

它是由一个宏定义的。做的事和单片机程序一样。

a，计算返回地址：sub lr, lr, #4

b, 保存寄存器值：
c，调用处理函数（如：__irq_usr 若用户态发生中断，就跳转到这个标号处。）
d, 处理函数又去调用“宏”：如"__irq_usr"标号处理是“usb_entry”宏，此宏先保存环境变量诸多寄存器。然后就调用宏“irq_handler”。此宏的定义会调用函数“asm_do_IRQ”。
如：　　__irq_usr:
usr_entry（这个宏也是保存些寄存器）
　　irq_handler（从__irq_usr 后调用这个函数，它也是一个宏。）
　　asm_do_IRQ（irq_hander 这个宏是做 asm_do_IRQ 函数）
e:恢复（调用完 asm_do_IRQ 函数后）

二，LINUX 的中断框架：内核中断框架

了解“asm_do_IRQ”，理解中断处理的过程。

A,单片机下的中断处理：
1，分辨是哪个中断。
2，调用处理函数（哪个中断就调用哪个处理函数）。
2， 清中断。

```
#include "s3c24xx.h"
void EINT_Handle() {
    unsigned long oft = INTOFFSET;//先读下此寄存器，看当前是在处理哪个中断。
    unsigned long val;
    switch( oft )
    {// S2被按下
        case 0:
        {
            GPFDAT |= (0x7<<4);    // 所有LED熄灭
            GPFDAT &= ~(1<<4);      // LED1点亮
            break;
        }
        // S3被按下
        case 2:
        {
            GPFDAT |= (0x7<<4);    // 所有LED熄灭
            GPFDAT &= ~(1<<5);      // LED2点亮
            break;
        }
        // K4被按下
        case 5:
        {
            GPFDAT |= (0x7<<4);    // 所有LED熄灭
            GPFDAT &= ~(1<<6);      // LED4点亮
            break;
        }
        default:
            break;
    }
    //清中断
    if( oft == 5 )
        EINTPEND = (1<<11);    // EINT8_23合用IRQ5
    SRCPND = 1<<oft;
    INTPND = 1<<oft;
}
```

1，上面是先读寄存器“INTOFFSET”，看是哪个中断。

2，中间是中断处理。
3，最后是清中断。
内核的也差不多。

B,内核的中断处理：
以上单片机的 3 个过程，都是在 asm_do_IRQ 中实现的。最终在“handle_irq”中实现的。

```
00111: asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
00112: {
00113:      struct pt_regs *old_regs = set_irq_regs(regs);
00114:      struct irq_desc *desc = irq_desc + irq;
00115:
00116:      /*
00117:       * Some hardware gives randomly wrong interrupts.  Rather
00118:       * than crashing, do something sensible.
00119:       */
00120:      if (irq >= NR_IRQS)
00121:           desc = &bad_irq_desc;
00122:
00123:      irq_enter();
00124:
00125:      desc_handle_irq(irq, desc);
00126:
00127:      /* AT91 specific workaround */
00128:      irq_finish(irq);
00129:
00130:      irq_exit();
00131:      set_irq_regs(old_regs);
00132: } ? end asm_do_IRQ ?
```

1，首先是根据传进来的 IRQ 中断号（参 1），
struct irq_desc *desc = irq_desc + irq;
irq_desc 是一个数组。在“Handle.c”中定义：

```
00051: struct irq_desc irq_desc[NR_IRQS] __cacheline_aligned_in_smp = {
00052:      [0 ... NR_IRQS-1] = {
00053:           .status = IRQ_DISABLED,
00054:           .chip = &no_irq_chip,
00055:           .handle_irq = handle_bad_irq,
00056:           .depth = 1,
00057:           .lock = __SPIN_LOCK_UNLOCKED(irq_desc->lock),
00058: #ifdef CONFIG_SMP
00059:           .affinity = CPU_MASK_ALL
00060: #endif
00061:      }
00062: };
```

从名字上看可知这是一个“中断描述”数组。以中断号“NR_IRQS”为下标。

struct irq_desc *desc = irq_desc + irq;
在这个数组下标处取到一个“数组项”。
irq_enter();不用管。

然后进入处理：
desc_handle_irq(irq, desc);

```
static inline void desc_handle_irq(unsigned int irq, struct irq_desc *desc)
{
    desc->handle_irq(irq, desc);
}
```

asmlinkage void __exception asm_do_IRQ(unsigned int irq, struct pt_regs *regs)
-->struct irq_desc *desc = irq_desc + irq;
    desc_handle_irq(irq, desc);

static inline void desc_handle_irq(unsigned int irq, struct irq_desc *desc)
--> desc->handle_irq(irq, desc);

desc->handle_irq(irq, desc);
即为：
(irq_desc + irq)->handle_irq(irq, desc);

desc 是全局数组“irq_desc（中断号为下标） ”
裸机程序中的 3 过程是在这个“handle_irq”中实现的。

2，查找“handle_irq”时，发现它有在“/kernel/irq/Chip.c”中的“__set_irq_handler（）”函数中有被用到。

```
---- handle_irq Matches (19 in 9 files) ----
Chip.c (kernel\irq):    desc->handle_irq = handle_bad_irq;
Chip.c (kernel\irq):    desc->handle_irq = handle_bad_irq;
Chip.c (kernel\irq):    desc->handle_irq = handle;
Handle.c (kernel\irq):      .handle_irq = handle_bad_irq,
Internals.h (kernel\irq):   printk("->handle_irq():  %p, ", desc->handle_irq);
Internals.h (kernel\irq):   print_symbol("%s\n", (unsigned long)desc->handle_irq);
Irq.c (arch\arm\kernel):    .handle_irq = handle_bad_irq,
Irq.h (include\asm-arm\mach):   desc->handle_irq(irq, desc);
Irq.h (include\linux): * @handle_irq:       highlevel irq-events handler [if NULL, __do_IRQ()]
Irq.h (include\linux):  irq_flow_handler_t  handle_irq;
Irq.h (include\linux): * callable via desc->chip->handle_irq()
Irq.h (include\linux): * irqchip-style controller then we call the ->handle_irq() handler,
Irq.h (include\linux):  desc->handle_irq(irq, desc);
Irq.h (include\linux):  if (likely(desc->handle_irq))
Irq.h (include\linux):      desc->handle_irq(irq, desc);
Manage.c (kernel\irq):  if (desc->handle_irq == &handle_bad_irq)
Manage.c (kernel\irq):      desc->handle_irq = NULL;
NCR53C9x.c (drivers\scsi):  ESPIRQ(("handle_irq: [sreg<%02x> sstep<%02x> ireg<%02x>]\n",
Resend.c (kernel\irq):      desc->handle_irq(irq, desc);
```

bad_irq应该不是我们想要的。

从上面看到调用过 handle_irq的地方，要么是在头文件，要么是打印。最终有一个地方 chip.c 中有

在“chip.c”的如下函数中有调用过“handle_irq”：
Void __set_irq_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
  const char *name)
-->desc->handle_irq = handle;

接着查“__set_irq_handler（）”这个函数有谁调用过。在 Irq.h 中：

```
---- __set_irq_handler Matches (6 in 2 files) ----
Chip.c (kernel\irq):__set_irq_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
Chip.c (kernel\irq):    __set_irq_handler(irq, handle, 0, NULL);
Chip.c (kernel\irq):    __set_irq_handler(irq, handle, 0, name);
Irq.h (include\linux):  set_irq_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
Irq.h (include\linux):  __set_irq_handler(irq, handle, 0, NULL);
Irq.h (include\linux):  __set_irq_handler(irq, handle, 1, NULL);
```

看这一个调用过“__set_irq_hander()”的地方。

```
/*
 * Set a highlevel flow handler for a given IRQ:
 */
static inline void
set_irq_handler(unsigned int irq, irq_flow_handler_t handle)
{
	__set_irq_handler(irq, handle, 0, NULL);
}
```

再搜索“set_irq_handler”时在“arch/arm/plat-s3c24xx/Irq.c”中找到一个函数：

```
00660: void __init s3c24xx_init_irq(void)
```

这个函数中有：

```
00753:	for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
00754:		irqdbf("registering irq %d (ext int)\n", irqno);
00755:		set_irq_chip(irqno, &s3c_irq_eint0t4);
00756:		set_irq_handler(irqno, handle_edge_irq);
00757:		set_irq_flags(irqno, IRQF_VALID);
00758:	}
```

下面是在"Chip.c"中

```
00537: void
00538: __set_irq_handler(unsigned int irq, irq_flow_handler_t handle, int is_chained,
00539:		const char *name)
```

```
struct irq_desc *desc;
```



```
00550:	desc = irq_desc + irq;
```

```
00576:	desc->handle_irq = handle;
00577:	desc->name = name;
```

是以形参“irq”这索引，在“irq_desc”中断描述数组中找到一项，将这一项给“desc”.
将通过“形参 irq”索引找到的那一项“desc”的“handle_irq”指向传进函数“__set_irq_handler”的形参“irq_flow_handler_t handle”。
这样显然在“void __init s3c24xx_init_irq(void)”函数中就构造了这个数组的很多项。

结构体“desc”的原型的成员：

```
struct irq_desc {
    irq_flow_handler_t  handle_irq;
    struct irq_chip     *chip;
    struct msi_desc     *msi_desc;
    void                *handler_data;
    void                *chip_data;
    struct irqaction    *action;    /* IRQ action list */
    unsigned int        status;     /* IRQ status */

    unsigned int        depth;      /* nested irq disables */
    unsigned int        wake_depth; /* nested wake enables */
    unsigned int        irq_count;  /* For detecting broken IRQs */
    unsigned int        irqs_unhandled;
    spinlock_t      lock;
#ifdef CONFIG_SMP
    cpumask_t       affinity;
    unsigned int        cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
    cpumask_t       pending_mask;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry   *dir;
#endif
    const char      *name;
} ? end irq_desc ?  ____cacheline_internodealigned_in_smp;
```

在这个“irq_desc”结构中有“handle_irq”等于“某个函数”；如下举例，看此结构成员 handle_irq 等于什么函数，"*chip" 等于什么函数。

举例：IRQ_EINT0 到 IRQ_EINT3：
在 Irq.c 中/*external interrupts*/外部中断中。
IRQ_EINT0 到 IRQ_EINT3 （外部中断 0 到外部中断 3）。设置了上面结构中的“handle_irq”和“*chip”

```
00753:      for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
00754:          irqdbf("registering irq %d (ext int)\n", irqno);
00755:          set_irq_chip(irqno, &s3c_irq_eint0t4);
00756:          set_irq_handler(irqno, handle_edge_irq);
00757:          set_irq_flags(irqno, IRQF_VALID);
00758:      }
```

1，成员“handle_irq”：
如上外部中断 0“IRQ_EINT0”（从名字猜测这是中断号），在\arch-s3c2410\Irqs.h 中有定义。

```
: /* main cpu interrupts */
: #define IRQ_EINT0     S3C2410_IRQ(0)    /* 16 */
: #define IRQ_EINT1     S3C2410_IRQ(1)
: #define IRQ_EINT2     S3C2410_IRQ(2)
: #define IRQ_EINT3     S3C2410_IRQ(3)
: #define IRQ_EINT4t7   S3C2410_IRQ(4)    /* 20 */
: #define IRQ_EINT8t23  S3C2410_IRQ(5)
```

那么“16”这个数组项（irq_desc[16]）的“handle_irq”就等于这个函数“handle_edge_irq”边缘触发。
（set_irq_handler(irqno, handle_edge_irq);）
从“handle_edge_irq”名字中“edge”可猜测到是处理那些边缘触发的中断。

2，成员“*chip”：
对于上面的“set_irq_chip(irqno, &s3c_irqext_chip);”也是找到相应的数组项：

```
00093: int set_irq_chip(unsigned int irq, struct irq_chip *chip)

00107:         desc = irq_desc + irq;
```

所以这里“chip”等于“s3c_irq_eint0t4”是 s3c 外部中断 0－4.

举例 2：外部中断 IRQ_EINT4;到外部中断 IRQ_EINT23

```
for (irqno = IRQ_EINT4; irqno <= IRQ_EINT23; irqno++) {
    irqdbf("registering irq %d (extended s3c irq)\n", irqno);
    set_irq_chip(irqno, &s3c_irqext_chip);
    set_irq_handler(irqno, handle_edge_irq);
    set_irq_flags(irqno, IRQF_VALID);
}
```

Irq_desc[外部中断 4 到 外部中断 23]

Handle_irq = handle_edge_irq;
Chip = s3c_irqext_chip;

总结：
Irq_desc[外部中断 0 ~ 3]：
{
Handle_irq = handle_edge_irq;
Chip = s3c_irq_eint0t4;
}

Irq_desc[外部中断 4 到 外部中断 23]：
{
Handle_irq = handle_edge_irq;
Chip = s3c_irqext_chip;
}
以上就是初始化，在“void __init s3c24xx_init_irq(void)”中实现的。

中断处理 C 函数入口 “asm_do_IRQ”会调用到事先初始的“handle_irq”函数。如外部中断调用
“handle_edge_irq”.是处理边缘触发的中断。

void __init s3c24xx_init_irq(void)
-->for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
set_irq_chip(irqno, &s3c_irq_eint0t4);
set_irq_handler(irqno, handle_edge_irq);
}

for (irqno = IRQ_EINT4; irqno <= IRQ_EINT23; irqno++) {
set_irq_chip(irqno, &s3c_irqext_chip);
set_irq_handler(irqno, handle_edge_irq);
}

在函数“/kernel/irq/Chip.c”中，可以看到“handle_edge_irq”做的事情。
void fastcall handle_edge_irq(unsigned int irq, struct irq_desc *desc)
-->kstat_cpu(cpu).irqs[irq]++; /* 发生了多少次中断的计数 */
-->desc->chip->ack(irq); /* Start handling the irq 开始处理中断*/
  -->if (unlikely(!action)) { /*这里是出错处理。 action 是链表。若此链表为空 */
desc->chip->mask(irq); /* 若链表为空则屏蔽中断 */
goto out_unlock;
}
-->action_ret = handle_IRQ_event(irq, action); /* 这是真正的中断处理过程 */

action_ret = handle_IRQ_event(irq, action);
其中"irq"是“handle_edge_irq”参 1；“action”是“struct irqaction *action = desc->action;”参 2 结构中的成员。
handle_edge_irq(unsigned int irq, struct irq_desc *desc)

总结：
handle_edge_irq
-->desc->chip->ack(irq). 清中断
-->handle_IRQ_event(irq, action). 处理中断

“handle_edge_irq”的处理过程：处理边缘触发的中断。
①，desc->chip->ack(irq)：清中断
假设是如下外部中断，irq.c 中 “s3c24xx_init_irq()”初始化中断中：

```
for (irqno = IRQ_EINT4; irqno <= IRQ_EINT23; irqno++) {
    irqdbf("registering irq %d (extended s3c irq)\n", irqno);
    set_irq_chip(irqno, &s3c_irqext_chip);
    set_irq_handler(irqno, handle_edge_irq);
    set_irq_flags(irqno, IRQF_VALID);
}
```

在“chip：s3c_irqext_chip”定义如下：desc->chip 就是“s3c_irqext_chip”：
S3c_irqext_chip 名字上猜测是 irq(中断)ext(外部)。

```
static struct irq_chip s3c_irqext_chip = {
    .name       = "s3c-ext",
    .mask       = s3c_irqext_mask,
    .unmask     = s3c_irqext_unmask,
    .ack        = s3c_irqext_ack,
    .set_type   = s3c_irqext_type,
    .set_wake   = s3c_irqext_wake
};
```

那么：desc->chip->ack：这个成员"ack"即：
.ack = s3c_irqext_ack,

static void s3c_irqext_ack(unsigned int irqno)
-->mask = __raw_readl(S3C24XX_EINTMASK); /* 读出外部中断屏蔽寄存器 */
-->__raw_writel(bit, S3C24XX_EINTPEND); /* 外部中断等待寄存器‘pending’。这里是清0 */
从这个函数的具体实现，可知它应该是“清中断”。

②，如果链表“action”是空的，就将它屏蔽。（出错处理）

```
if (unlikely(!action)) {
    desc->chip->mask(irq);
    goto out_unlock;
}
```

③，handle_IRQ_event 这是真正的正理过程：处理中断。

```
irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action)
{
    irqreturn_t ret, retval = IRQ_NONE;
    unsigned int status = 0;

    handle_dynamic_tick(action);

    if (!(action->flags & IRQF_DISABLED))
        local_irq_enable_in_hardirq();

    do {
        ret = action->handler(irq, action->dev_id);
        if (ret == IRQ_HANDLED)
            status |= action->flags;
        retval |= ret;
        action = action->next;
    } while (action);

    if (status & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness(irq);
    local_irq_disable();

    return retval;
} ? end handle_IRQ_event ?
```

显然action是一个链表

取出链表中的各成员，给handler。

显然action是链表

取出链表 action 中的成员，执行“action->handler（）”。

总结：“中断框架”（按下按键）。

1，CPU 自动进入“异常模式”。调用“异常处理函数”。

linux-2.6.22.6\arch\arm\kernel\entry-armv.S 中

2，在“异常处理函数”中如跳到“b vector_irq + stubs_offset”。

```
    .globl  __vectors_start
__vectors_start:
    swi SYS_ERROR0
    b   vector_und + stubs_offset       //vector_und是undefined未定义异常指令。它是一个宏，由上面的"vector_stub   und, UND_MODE"定义。
    ldr pc, .LCvswi + stubs_offset
    b   vector_pabt + stubs_offset
    b   vector_dabt + stubs_offset
    b   vector_addrexcptn + stubs_offset
    b   vector_irq + stubs_offset       //发生中断便跳转到这里。这个地址标号“vector_irq”在代码中也没有。也是一个宏来定义.
    b   vector_fiq + stubs_offset

    .globl  __vectors_end
__vectors_end:
```

“vector_irq”是一个宏定义的。找到这个“宏”，假设它调用“__irq_usr”。

vector_stub irq, IRQ_MODE, 4

此宏展开：

Vector_irq:

Sub lr, lr, #4

@

@ Save r0, lr_<exception> (parent PC) and spsr_<exception>

@ (parent CPSR)

@

stmia sp, {r0, lr} @ save r0, lr

mrs lr, spsr

str lr, [sp, #8] @ save spsr

@

@ Prepare for SVC32 mode. IRQs remain disabled.

@

mrs r0, cpsr

eor r0, r0, #(\mode ^ SVC_MODE)

msr spsr_cxsf, r0

@

@ the branch table must immediately follow this code

@ 这里调用某个列表。如下图：

and lr, lr, #0x0f

mov r0, sp

ldr lr, [pc, lr, lsl #2]

movs pc, lr @ branch to handler in SVC mode

```
vector_stub irq, IRQ_MODE, 4

.long   __irq_usr                       @  0  (USR_26 / USR_32)
.long   __irq_invalid                   @  1  (FIQ_26 / FIQ_32)
.long   __irq_invalid                   @  2  (IRQ_26 / IRQ_32)
.long   __irq_svc                       @  3  (SVC_26 / SVC_32)
.long   __irq_invalid                   @  4
.long   __irq_invalid                   @  5
.long   __irq_invalid                   @  6
.long   __irq_invalid                   @  7
.long   __irq_invalid                   @  8
.long   __irq_invalid                   @  9
.long   __irq_invalid                   @  a
.long   __irq_invalid                   @  b
.long   __irq_invalid                   @  c
.long   __irq_invalid                   @  d
.long   __irq_invalid                   @  e
.long   __irq_invalid                   @  f
```

假若是调用的列表中的“__irq_usr”。

3，调用到列表中的“__irq_usr”后，可以具体分析这个“__irq_usr”中处理的情况：

```
__irq_usr:
    usr_entry

#ifdef CONFIG_TRACE_IRQFLAGS
    bl  trace_hardirqs_off
#endif
    get_thread_info tsk
#ifdef CONFIG_PREEMPT
    ldr r8, [tsk, #TI_PREEMPT]      @ get preempt count
    add r7, r8, #1          @ increment it
    str r7, [tsk, #TI_PREEMPT]
#endif

    irq_handler
#ifdef CONFIG_PREEMPT
    ldr r0, [tsk, #TI_PREEMPT]
    str r8, [tsk, #TI_PREEMPT]
    teq r0, r7
    strne   r0, [r0, -r0]
#endif
#ifdef CONFIG_TRACE_IRQFLAGS
    bl  trace_hardirqs_on
#endif

    mov why, #0
    b   ret_to_user

    .ltorg

    .align  5
```

__irq__usr 是用户态中断。

“usr_entry”这是它的入口，其中保存寄存器（中断现场）。

再调用“irq_handler”，这也是一个宏。

```
    .macro  irq_handler                    irq_handler 是一个宏
    get_irqnr_preamble r5, lr
1:  get_irqnr_and_base r0, r6, r5, lr
    movne   r1, sp
    @
    @ routine called with r0 = irq number, r1 = struct pt_regs *
    @                       最终调用"asm_do_IRQ"函数。
    adrne   lr, 1b
    bne asm_do_IRQ
```

最终会调用到“asm_do_IRQ”这个 C 函数。

4，“asm_do_IRQ”调用“irq_desc[IRQ 中断下标]以中断为下标取出里面的一项“handle_irq””

```
desc_handle_irq(irq, desc);
```

```
static inline void desc_handle_irq(unsigned int irq, struct irq_desc *desc)
{
    desc->handle_irq(irq, desc);
}
```

desc->handle_irq(irq, desc);指向谁：

举例说明：在：linux-2.6.22.6\arch\arm\plat-s3c24xx\Irq.c 中，s3c24xx_init_irq(void)初始化中断时，若是外部中断 0-3，或外部中断 4-23 时，，指向“handle_edge_irq 函数”

```
for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
    irqdbf("registering irq %d (ext int)\n", irqno);
    set_irq_chip(irqno, &s3c_irq_eint0t4);
    set_irq_handler(irqno, handle_edge_irq);
    set_irq_flags(irqno, IRQF_VALID);
}

for (irqno = IRQ_EINT4; irqno <= IRQ_EINT23; irqno++) {
    irqdbf("registering irq %d (extended s3c irq)\n", irqno);
    set_irq_chip(irqno, &s3c_irqext_chip);
    set_irq_handler(irqno, handle_edge_irq);
    set_irq_flags(irqno, IRQF_VALID);
}
```

，指向“handle_edge_irq 函数”

5，“handle_edge_irq”做的事：
①，desc->chip->ack()　　清中断
②，handle_IRQ_event()　处理中断
③，取出“action”链表中的成员，执行“action->handler”。

6，核心在“irq_desc” 结构数组：
分析“irq_desc[]”结构数组：里面有“action 链表”，有“chip”等。

```
struct irq_desc {
    irq_flow_handler_t  handle_irq;      /* 函数指针，如指向handle_edge_irq() */
    struct irq_chip     *chip;           /* 芯片相关底层的处理函数。如指向 s3c_irqext_chip */
    struct msi_desc     *msi_desc;
    void                *handler_data;
    void                *chip_data;
    struct irqaction    *action;     /* IRQ action list 中断的动作链表 */
    unsigned int        status;      /* IRQ status */

    unsigned int        depth;       /* nested irq disables */
    unsigned int        wake_depth;  /* nested wake enables */
    unsigned int        irq_count;   /* For detecting broken IRQs */
    unsigned int        irqs_unhandled;
    spinlock_t      lock;
#ifdef CONFIG_SMP
    cpumask_t       affinity;
    unsigned int        cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
    cpumask_t       pending_mask;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry   *dir;
#endif
    const char      *name;
} ? end irq_desc ?  ____cacheline_internodealigned_in_smp;
```

handle_irq:是函数指针。在上面的例子中这个函数指针指向了“handle_edge_irq”。
首先是清中断，然后把链表“action”中的成员取出，一一执行里面的“action->handler”
*chip ：是芯片相关底层的一些处理函数。在例子中是指向"&s3c_irqext_chip”
　这个结构体“irq_chip”作些底层的操作：
起动、关闭、使用、禁止、ack 响应中断（响应中断即清中断而已）、

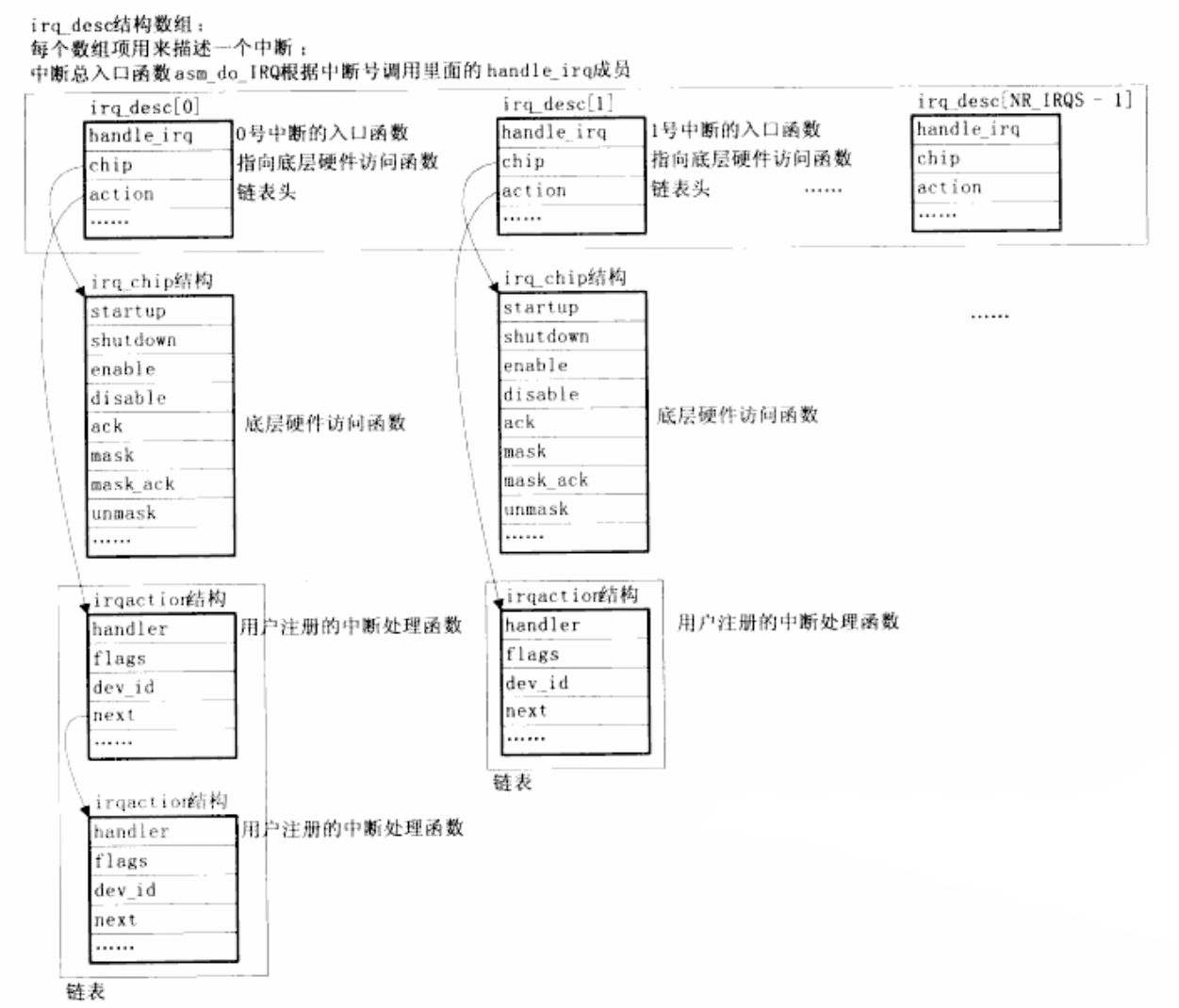
```
struct irq_chip {
    const char  *name;
    unsigned int    (*startup)(unsigned int irq);    /* 启动中断 */
    void        (*shutdown)(unsigned int irq);       /* 关闭中断 */
    void        (*enable)(unsigned int irq);         /* 使能中断 */
    void        (*disable)(unsigned int irq);        /* 禁止中断 */

    void        (*ack)(unsigned int irq);            /* 响应中断：通过分析应该是清中断 */
    void        (*mask)(unsigned int irq);
    void        (*mask_ack)(unsigned int irq);
    void        (*unmask)(unsigned int irq);
    void        (*eoi)(unsigned int irq);

    void        (*end)(unsigned int irq);
    void        (*set_affinity)(unsigned int irq, cpumask_t dest);
    int     (*retrigger)(unsigned int irq);
    int     (*set_type)(unsigned int irq, unsigned int flow_type);
    int     (*set_wake)(unsigned int irq, unsigned int on);

    /* Currently used only by UML, might disappear one day.*/
#ifdef CONFIG_IRQ_RELEASE_METHOD
    void        (*release)(unsigned int irq, void *dev_id);
#endif
    /*
     * For compatibility, ->typename is copied into ->name.
     * Will disappear.
     */
    const char  *typename;
} ? end irq_chip ? ;
```

*action:中断的动作列表。
里面有：handler（处理函数），还有“flags”等。

```
struct irqaction {
    irq_handler_t handler;      /* 处理函数 */
    unsigned long flags;        /* 标志 */
    cpumask_t mask;             /* 掩码 */
    const char *name;
    void *dev_id;
    struct irqaction *next;
    int irq;
    struct proc_dir_entry *dir;
};
```

以上 1—6 都是系统做好的，这便是系统的“中断处理框架”。
但是我们写中断处理时，是想执行自己的中断代码。那么我们的代码就应该在“action->handler”这里。
我们要用“request_irq()”告诉内核我们的处理函数是什么。

LINUX 内核中断框架：



首先内核中有个数组“irq_desc[]”IRQ 的描述结构数组，这个结构数组是以“中断号”为下标。

结构中有：

handle_irq:中断入口函数。这个入口函数中，是做清中断，再调用 action 链表中的各种处理函数。

chip:　　指向底层硬件的访问函数，它做屏蔽中断、使能中断，响应中断等。

action：链表头，它指向“irqaction 结构”，这个结构中存放“handler 用户注册的中断处理函数”等。

我们想使用自己的中断处理函数时，就是在内核的这个中断框架里在其中的“action 链表”中添加进我们的“中断函数”。

三，分析“request_irq()”：

“request_irq()”在 kernel\irq\Manage.c 中。

```
0500: int request_irq(unsigned int irq, irq_handler_t handler,
0501:                 unsigned long irqflags, const char *devname, void *dev_id)
0502: {
0503:         struct irqaction *action;
0504:         int retval;
```

在发生对应于第 1 个参数 irq 的中断时，则调用第 2 个参数 handler 指定的中断服务函数(也就是把 handler() 中断服务函数注册到内核中)。

irq:中断号。
handler：处理函数。
irqflags:上升沿触发，下降沿触发，边沿触发等。指定了快速中断或中断共享等中断处理属性.
*devname：中断名字。通常是设备驱动程序的名称。改值用在 /proc/interrupt 系统 (虚拟) 文件上，或内核发生中断错误时使用。
第 5 个参数 dev_id 可作为共享中断时的中断区别参数，也可以用来指定中断服务函数需要参考的数据地址。

返回值：
函数运行正常时返回 0 ，否则返回对应错误的负值。

```
int request_irq(unsigned int irq, irq_handler_t handler,
unsigned long irqflags, const char *devname, void *dev_id)
--> action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC); /* 分配irqaction结构 */
--> action->handler = handler;   /* 记录形参传进来的handler */
action->flags = irqflags;     /* 记录形参传进来的irqflags */
cpus_clear(action->mask);
action->name = devname;       /* 记录形参传进来的中断名字 */
action->next = NULL;
action->dev_id = dev_id;
-->retval = setup_irq(irq, action);      /* 设置中断 */
```

```
00527:         action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
00528:         if (! action)
00529:                 return - ENOMEM;
00530:
00531:         action- >handler = handler;
00532:         action- >flags = irqflags;
00533:         cpus_clear(action- >mask);
00534:         action- >name = devname;
00535:         action- >next = NULL;
00536:         action- >dev_id = dev_id;
```

① ，分配一个“irqaction”结构。

```
    action->handler = handler;// 将传进来的中断处理函数指针 handler 传给
action->handler
    action->flags = irq_flags;//将传进来的中断标志字传给 action->flags
    cpus_clear(action->mask);//将中断操作行为描述结构的 CPU 掩码字中全部有效位码清 0，
也就是将系统中所有 CPU 对应于 action->mask 中的位码全部清 0
    action->name = devname;//将传进来的设备名字传给 action->name
    action->next = NULL;//将 next 指针设置为空
    action->dev_id = dev_id;//将传进来的 dev_id 指针传给 action->dev_id
```

这个申请的结构将传来的 handler 处理函数等都记录下来。这个结构的成员都指向函数“request_irq()”传进来的参数。

②，最后调用“setup_irq()”函数。设置中断

```
00559:        retval = setup_irq(irq, action);
00560:        if (retval)
00561:              kfree(action);
00562:
```

参 1：要安装中断处理程序的中断号。

参 2：irq 号中断的中断操作行为描述结构指针。

这里是上面 kmalloc 分配的 action 结构。

```
构造了action结构，再放入action结构链表。再将对应的引脚设置成中断引脚。最后使能中断。
int setup_irq(unsigned int irq, struct irqaction *new)
-->struct irq_desc *desc = irq_desc + irq;
    p = &desc->action;
    以中断号为下标，找到“irq_desc[]”数组中的一项。
-->if (!((old->flags & new->flags) & IRQF_SHARED) ||    /* 是否是共享中断。若action链表头里挂接了多个
项就表明此中断为共享中断 */
     ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
old_name = old->name;
goto mismatch;
}
-->/* add new interrupt at end of irq queue */
do {
p = &old->next;
old = *p;
} while (old);
-->if (!shared) {   /* 若是共享中断则表示之前有内容挂接上去 */
irq_chip_set_defaults(desc->chip); /* 开始chip结构 */
-->if (!chip->enable) /* 若chip中没有enable函数时，就使用下面默认的函数 */
chip->enable = default_enable;
if (!chip->disable)
chip->disable = default_disable;
if (!chip->startup)
chip->startup = default_startup;
if (!chip->shutdown)
```

```
chip->shutdown = chip->disable;
if (!chip->name)
chip->name = chip->typename;
if (!chip->end)
chip->end = dummy_irq_chip.end;
-->desc->chip->set_type(irq,new->flags & IRQF_TRIGGER_MASK); /* 将那些引脚设置成中断引脚 */
-->desc->chip->startup(irq);
    或者：desc->chip->enable(irq);
```

```
    desc = irq_desc + irq;//找到 irq 号中断对应的 struct irqdesc 结构 irq_desc[irq]
    spin_lock_irqsave(&irq_controller_lock, flags);//禁止 IRQ 中断，将系统当前程序状态寄存器 CPSR 保存到 flags 中，获取中断操作控制锁
    p = &desc->action;//将 p 指向 irq_desc[irq].action
    if ((old = *p) != NULL) { //如果 irq_desc[irq].action 指针非空，也就是说 irq 号已经安装了中断处理程序
        if (!(old->flags & new->flags & SA_SHIRQ)) { //如果已经安装的中断处理程序不支持共享 irq 号中断
            spin_unlock_irqrestore(&irq_controller_lock, flags);//释放中断操作控制锁，将 flags 的值恢复到 CPSR 中，使能系统 IRQ 中断
            return -EBUSY;//返回忙错误
        }
```

p = &desc->action;
desc 是“irq_desc[]”里面的数组项。这里即是找到了这个数组项。
action 是链表头，下面是判断这个链表开始有没有结构。若链表中原来一个中断结构。则要判断是否“共享中断”。
若一个“action”链表头中挂接有多个项，就表明这个中断是“共享中断”。
“共享中断”：表示中断来源有很多种，但它们共享同一个引脚。

if (!((old->flags & new->flags) & IRQF_SHARED) || /* 是否是共享中断。若 action 链表头里挂接了多个项就表明此中断为共享中断 */
    ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
old_name = old->name;
goto mismatch;
}

如果之前那个老的 action 结构不是作“共享中断”时，我们就不能将申请的新结构放进这个老的“action”链表中去。就不能再次链接新的结构进去了。就表示出“goto mismatch;”出错。

```
    //如果已经安装的中断处理程序支持共享 irq 号中断
        do {
            p = &old->next; //将 p 指向已经安转的中断操作行为描述结构的下一个中断操作行为描述结构指针
            old = *p; //将 old 指向 p 所指中断操作行为描述结构
```

```
        } while (old); //直到最后一个安转到 irq 号中断的中断操作行为描述结构，因为可能已经安装了多个中断处理程序到 irq 号中断上
        shared = 1; //将共享标志设置为 1
    }
```

这是将新的结构放进去。

接着“chip->set_type”将对应的引脚设置为“中断引脚”（外部中断，中断触发方式）。
然后“desc->chip->startup(irq);”和“desc->chip->enable(irq);”。这是最后“使能中断”

request_irq 是注册中断服务程序。
free_irq（irq,dev_id）来卸载这个中断服务程序。
参 1：要卸载中断处理程序的中断号。定位“action”链表。
参 2：使用 irq 号中断的设备的 ID 号或者设备驱动灵气指针。在“action”链表中找到要卸载的表项。
同一个中断的不同中断处理函数必须使用不同的 dev_id 来区分。

构造了 action 结构，再放入 action 结构链表。再将对应的引脚设置成中断引脚。最后使能中断。
原来假设事例中的 chip 结构：

```
for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
irqdbf("registering irq %d (ext int)\n", irqno);
set_irq_chip(irqno, &s3c_irq_eint0t4);
set_irq_handler(irqno, handle_edge_irq);
set_irq_flags(irqno, IRQF_VALID);
}
```

```
static struct irq_chip s3c_irq_eint0t4 = {
    .name       = "s3c-ext0",
    .ack        = s3c_irq_ack,
    .mask       = s3c_irq_mask,
    .unmask     = s3c_irq_unmask,
    .set_wake   = s3c_irq_wake,
    .set_type   = s3c_irqext_type,
};
```

Int s3c_irqext_type(unsigned int irq, unsigned int type)

```
if ((irq >= IRQ_EINT0) && (irq <= IRQ_EINT3))
{   /* 相关的中断类型，就配置相关的引脚为中断引脚。 */
    gpcon_reg = S3C2410_GPFCON;
    extint_reg = S3C24XX_EXTINT0;
    gpcon_offset = (irq - IRQ_EINT0) * 2;
    extint_offset = (irq - IRQ_EINT0) * 4;
}
```

若为外部中断 0-3，则将上面的相关引脚配置成中断引脚。

```
/* Set the external interrupt to pointed trigger type */
switch (type)    /* 根据request_irq()参3的irqflags标志来判断中断触发方式。 */
{
    case IRQT_NOEDGE:
        printk(KERN_WARNING "No edge setting!\n");
        break;

    case IRQT_RISING:    /* 上升沿触发 */
        newvalue = S3C2410_EXTINT_RISEEDGE;
        break;

    case IRQT_FALLING:   /* 下降沿触发 */
        newvalue = S3C2410_EXTINT_FALLEDGE;
        break;

    case IRQT_BOTHEDGE: /* 双边沿触发 */
        newvalue = S3C2410_EXTINT_BOTHEDGE;
        break;

    case IRQT_LOW:       /* 低电平触发 */
        newvalue = S3C2410_EXTINT_LOWLEV;
        break;

    case IRQT_HIGH:      /* 高电平触发 */
        newvalue = S3C2410_EXTINT_HILEV;
        break;

    default:
        printk(KERN_ERR "No such irq type %d", type);
        return -1;
} ? end switch type ?
```

卸载中断处理函数：

```
void free_irq(unsigned int irq, void *dev_id)
-->desc = irq_desc + irq;   /* 找到数组项 */
-->if (action->dev_id != dev_id)    /* 判断dev_id是否为free_irq()传来的参2 */
continue;
-->if (!desc->action) {      /* 若链表空了，就禁止或屏蔽这个chip */
desc->status |= IRQ_DISABLED;
if (desc->chip->shutdown)
desc->chip->shutdown(irq);
else
desc->chip->disable(irq);
}
```

总结：

①，要注册用户中断处理函数时，用：
request_irq(中断号,处理函数,硬件触发试,中断名字,dev_id).
{
分配一个 irqaction 结构。结构指向 request_irq 中的参数。
将上面的 irqaction 结构放到 irq_desc[irq]数组项中的 action 链表中。
设置引脚成为中断引脚。
使能中断。
}
②，卸载中断处理函数。

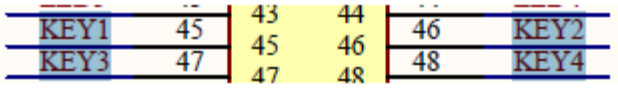
```
void free_irq(unsigned int irq, void *dev_id)
```

{

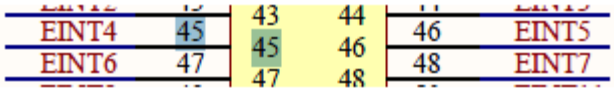
将其从 action 链表中删除结构。
禁止中断（在 action 链表中没有成员结构 irqacton 时了，要是共享中断中删除一个结构还有其他结构时，中断也不会禁止）。
}

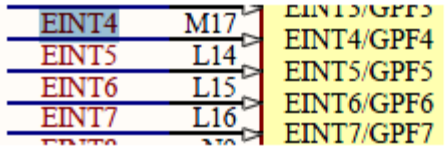
四，按键中断处理实例：

要配置成‘中断引脚’，request_irq()中会自动将相关引脚配置成‘中断引脚’。再根据它的参3flags 决定是什么触发方式。
要配置成中断引脚: request_irq 函数会将引脚自动配置成中断引脚
由其中的参数"flages"要决定是上升沿等等方式触发中断.
其中有几个参数，我们不知道 IRQ 号是什么 。找到按键的原理图：



查核心板：





从原理图上知道是“外部中断 4、5、6、7”.

```
request_irq(IRQ_EINT0, buttons_irq, IRQT_BOTHEDGE, "S2", 1);
```

1，有一个初始化中断的函数例子“s3c24xx_init_irq”里面设置“irq_desc 数组”里面用到了很多“IRQ_EINT3”等这样的中断号。找其中一个可见其在头文件“Irqs.h”有定义：

```
/* main cpu interrupts */
#define IRQ_EINT0       S3C2410_IRQ(0)      /* 16 */
#define IRQ_EINT1       S3C2410_IRQ(1)
#define IRQ_EINT2       S3C2410_IRQ(2)
#define IRQ_EINT3       S3C2410_IRQ(3)
#define IRQ_EINT4t7     S3C2410_IRQ(4)      /* 20 */
#define IRQ_EINT8t23    S3C2410_IRQ(5)
#define IRQ_RESERVED6   S3C2410_IRQ(6)      /* for s3c2410 */
#define IRQ_CAM         S3C2410_IRQ(6)      /* for s3c2440, s3c2443 */
#define IRQ_BATT_FLT    S3C2410_IRQ(7)
```

这些便是它的中断号。

2，参 2 是一个处理函数，这里还没有写，但先将名字写出来“buttons_irq”。

3，参 3 是“irqflags”。查看有哪些宏可以来用。 可以看实例：linux-2.6.22.6\arch\arm\plat-

s3c24xx\Irq.c 看“request_irq”这个函数。

```
for (irqno = IRQ_EINT0; irqno <= IRQ_EINT3; irqno++) {
irqdbf("registering irq %d (ext int)\n", irqno);
set_irq_chip(irqno, &s3c_irq_eint0t4);
set_irq_handler(irqno, handle_edge_irq);
set_irq_flags(irqno, IRQF_VALID);
}
static struct irq_chip s3c_irq_eint0t4 = {
.name                   = "s3c-ext0",
.ack                    = s3c_irq_ack,
.mask                   = s3c_irq_mask,
.unmask                 = s3c_irq_unmask,
.set_wake               = s3c_irq_wake,
.set_type               = s3c_irqext_type,
};
```

里面会根据“set_type”函数设置：
那些引脚，比如“s3c_irqext_chip”这个结构中成员“.set_type = s3c_irqext_type”

```
int
s3c_irqext_type(unsigned int irq, unsigned int type)
```

里面有个“type”。有比较多，如下想用一个“双边沿触发”上升沿和下降沿都可以触发中断：

```
case IRQT_BOTHEDGE: /* 双边沿触发 */
    newvalue = S3C2410_EXTINT_BOTHEDGE;
    break;
```

所以在“request_irq” ，函数的第 3 个参数写成“IRQT_BOTHEDGE”.它在“include\asm-arm\Irq.h”中定义：

```
/*
 * Migration helpers
 */
#define __IRQT_FALEDGE   IRQ_TYPE_EDGE_FALLING
#define __IRQT_RISEDGE   IRQ_TYPE_EDGE_RISING
#define __IRQT_LOWLVL    IRQ_TYPE_LEVEL_LOW
#define __IRQT_HIGHLVL   IRQ_TYPE_LEVEL_HIGH

#define IRQT_NOEDGE (0)
#define IRQT_RISING (__IRQT_RISEDGE)
#define IRQT_FALLING    (__IRQT_FALEDGE)
#define IRQT_BOTHEDGE   (__IRQT_RISEDGE|__IRQT_FALEDGE)
#define IRQT_LOW    (__IRQT_LOWLVL)
#define IRQT_HIGH   (__IRQT_HIGHLVL)
#define IRQT_PROBE  IRQ_TYPE_PROBE
```

4，参 4，取一个名字。如“原理图”有按键 key4，所以这个名字用“S4”.

5,参 5，是 dev_id，它是“free_irq”函数用来从“action”链表中删除“irqaction”结构。使用参 1 中断号“irq”定位“action”链表，再使用“dev_id”在“action”链表中找到卸载的表

项。同一个中断的不同中断处理函数必须使用不同的“dev_id”来区分，这要求注册共享中断时参数“dev_id”必须惟一。
这里参 5 先写成“1”。

6，对于其他按键的中断也和上面一样设置方法。

```
/*1.1.定义file_operations结构中的‘打开’操作。struct file 表示一个打开的文件。Struct inode 表示一个磁盘上的具体文件。*/
static int second_drv_open(struct inode *inode, struct file *file)
{
    /* 6.3，以中断处理的方式控制招安时，要配置成‘中断引脚’，request_irq()中会自动将相关引脚配置成‘中断引脚’。
     * 再根据它的参3flags决定是什么触发方式。
     * request_irq参数：
     *    irq:可以从按键的原理图上知道是‘EINT4-7’，再从‘s3c24xx_init_irq()’这个初始化中断的函数中知道中断号的定义，如：
     *        #define IRQ_EINT4t7     S3C2410_IRQ(4)  这是20.
     *    handler : buttons_irq 先定义一个处理函数，一会再去实现。
     *    irqflags : 可以从实例中找到相关宏.如int s3c_irqext_type()中有很多中断触发方式的宏，这里我们取‘IRQT_OTHEDGE’
     *    devname: 取个中断名字，KEY1 就取 KEY1.
     *    dev_id: 它是“free_irq”函数用来从“action”链表中删除“irqaction”结构。使用参1中断号“irq”定位“action”链表，
     *          再使用“dev_id”在“action”链表中找到卸载的表项。同一个中断的不同中断处理函数必须使用不同的“dev_id”来
     *          区分，这要求注册共享中断时参数“dev_id”必须惟一。 这里先写成“1”。
     */
    request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY1", 1);
    request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY2", 1);
    request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY3", 1);
    request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY4", 1);

    return 0;
} ? end second_drv_open ?
```

这是“open 函数”中“request_irq 函数”。在“关闭”这个设备时，去释放这些“request_irq 函数”。
①，在“file_operations”结构中加一个函数

```
00054: static struct file_operations third_drv_fops = {
00055:     .owner   = THIS_MODULE,
00056:     .open    = third_drv_open,
00057:     .read    = third_drv_read,
00058:     .release = third_drv_close,
00059: }
```

②，

```
/* 定义关闭设备时，释放去中断 */
int second_drv_release(struct inode *inode, struct file *file)
{
    /* 释放中断 */
    free_irq (IRQ_EINT4t7, 1);
    return 0;
}
```

7，定义：“buttons_irq”函数。从内核中搜索“request_irq()”初调用的地址，找一个相应的例子来仿写：

```
if (*irqp == probe_irq_off(cookie) /* It's a good IRQ line! */
    && ((retval = request_irq(dev->irq = *irqp,
    ei_interrupt, 0, dev->name, dev)) == 0))
    break;
```

```
irqreturn_t ei_interrupt(int irq, void *dev_id)
{
    return __ei_interrupt(irq, dev_id);
}
```

“buttons_irq”被调用时，会从“request_irq”传入“dev_id”。
我们只是在里面打印下停下。

```
/* 定义中断处理函数 */
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    printk("irq = %d\n", irq);

    return IRQ_HANDLED;
}
```

以上驱动写好了。可以编译下到开发板检验。
<<second_irq.c>>
上面这个代码有问题。下面这个可以：
<<second_irq_没有中断共享.c>>

```
HH:/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key # make
make -C /work/arm/system/linux-2.6.22.6 M=`pwd` modules
make[1]: 进入目录“/work/arm/system/linux-2.6.22.6”
  CC [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.o
/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.c: In function `second_drv_open':
/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.c:71: warning: implicit declaration of functi
on `request_irq'
/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.c: In function `second_drv_release':
/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.c:116: warning: implicit declaration of funct
ion `free_irq'
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.mod.o
  LD [M]  /work/arm/nfs_romfs_2440/drivers_2.6.22/2_key/second_irq.ko
make[1]: 离开目录“/work/arm/system/linux-2.6.22.6”
HH:/work/arm/nfs_romfs_2440/drivers_2.6.22/2_key #
```

这里我遇到一个问题，因为开发板的 4 个按键，对应的外部中断是 “IRQ_EINT4t7”，都是占用中断号 “20”。
在 2440 芯片中，有些中断是需要共享一个中断寄存器中的一位，如 EINT4——EINT7，它们是共享寄存器 SRCPEND 的第 4 位。

request_irq (IRQ_EINT4, buttons_irq, IRQT_BOTHEDGE, "KEY1", (int *)1);

这里加上：

```
static inline void
s3c_irq_ack(unsigned int irqno)
{
    unsigned long bitval = 1UL << (irqno - IRQ_EINT0);

    __raw_writel(bitval, S3C2410_SRCPND);
    __raw_writel(bitval, S3C2410_INTPND);
}
```

```
s3c_irq_ack(IRQ_EINT4t7);
request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY1", (int *)1);
request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY2", (int *)1);
request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY3", (int *)1);
request_irq (IRQ_EINT4t7, buttons_irq, IRQT_BOTHEDGE, "KEY4", (int *)1);
```

加上对共享中断的处理后，会有“cat /proc/interrupts”：但是有乱码。并且按下按键时，不会触发中断服务程序显示中断号。

后面没有使用共享中断号，而是如下：

```
request_irq (IRQ_EINT4, buttons_irq, IRQT_BOTHEDGE, "KEY4", 1);
request_irq (IRQ_EINT5, buttons_irq, IRQT_BOTHEDGE, "KEY4", 1);
request_irq (IRQ_EINT6, buttons_irq, IRQT_BOTHEDGE, "KEY4", 1);
request_irq (IRQ_EINT7, buttons_irq, IRQT_BOTHEDGE, "KEY4", 1);
```

即可。

验证中断：

```
/drivers_2.6.22/2_key # insmod second_irq.ko
second_drv_init
/drivers_2.6.22/2_key # lsmod /dev/buttons -l
second_irq 2900 0 - Live 0xbf000000
/drivers_2.6.22/2_key # cat /proc/devices |grep secon
252 second_irq_drv
/drivers_2.6.22/2_key #
```


在驱动程序中只要调用“third_drv_open”打开设备，就会去调用“request_irq”
在系统中查看当前“中断”。

```
/drivers_2.6.22/2_key # cat /proc/interrupts
           CPU0
 30:    5657044          s3c  S3C2410 Timer Tick
 32:          0          s3c  s3c2410-lcd
 33:          0          s3c  s3c-mci
 34:          0          s3c  I2SSDI
 35:          0          s3c  I2SSDO
 37:         13          s3c  s3c-mci
 42:         88          s3c  ohci_hcd:usb1
 43:          0          s3c  s3c2440-i2c
 60:          0      s3c-ext  s3c-mci
 62:      94931      s3c-ext  eth0
 70:        539   s3c-uart0  s3c2440-uart
 71:     305362   s3c-uart0  s3c2440-uart
 79:          0      s3c-adc  s3c2410_action
 80:          0      s3c-adc  s3c2410_action
 83:          0            -  s3c2410-wdt
Err:          0
```

加载驱动后：打开设备。

打开设备“/dev/buttons”定位到“5”去。
关闭这个设备：exec 5<&-

```
/drivers_2.6.22/2_key # ps
PID   USER     TIME   COMMAND
    1 0        0:01 {linuxrc} init
    2 0        0:00 [kthreadd]
    3 0        0:00 [ksoftirqd/0]
    4 0        0:00 [watchdog/0]
    5 0        0:00 [events/0]
    6 0        0:00 [khelper]
   55 0        0:00 [kblockd/0]
   56 0        0:00 [ksuspend_usbd]
   59 0        0:00 [khubd]
   61 0        0:00 [kseriod]
   73 0        0:00 [pdflush]
   74 0        0:00 [pdflush]
   75 0        0:00 [kswapd0]
   76 0        0:00 [aio/0]
  711 0        0:00 [mtdblockd]
  750 0        0:00 [kmmcd]
  767 0        0:00 [rpciod/0]
  773 0        0:00 -/bin/sh
  783 0        0:00 ps
/drivers_2.6.22/2_key # ls /proc/773/fd -l
total 0
lrwx------    1 0        0               64 Jan  1 00:01 0 -> /dev/console
lrwx------    1 0        0               64 Jan  1 00:01 1 -> /dev/console
lrwx------    1 0        0               64 Jan  1 00:01 10 -> /dev/tty
lrwx------    1 0        0               64 Jan  1 00:01 2 -> /dev/console
lr-x------    1 0        0               64 Jan  1 00:01 5 -> /dev/buttons
```

这时可以看到中断信息：

```
/drivers_2.6.22/2_key # cat /proc/interrupts
           CPU0
 30:      19234          s3c  S3C2410 Timer Tick
 32:          0          s3c  s3c2410-lcd
 33:          0          s3c  s3c-mci
 34:          0          s3c  I2SSDI
 35:          0          s3c  I2SSDO
 37:         13          s3c  s3c-mci
 42:         88          s3c  ohci_hcd:usb1
 43:          0          s3c  s3c2440-i2c
 48:          0      s3c-ext  KEY1
 49:          0      s3c-ext  KEY2
 50:          0      s3c-ext  KEY3
 51:          0      s3c-ext  KEY4
 60:          0      s3c-ext  s3c-mci
 62:       1549      s3c-ext  eth0
 70:        218   s3c-uart0  s3c2440-uart
 71:        304   s3c-uart0  s3c2440-uart
 79:          0      s3c-adc  s3c2410_action
 80:          0      s3c-adc  s3c2410_action
 83:          0            -  s3c2410-wdt
Err:          0
```

这时按下 4 个按键，会由中断处理程序打印中断号出来。

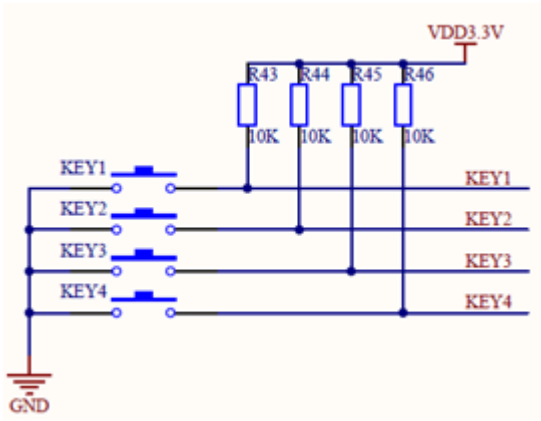
```
/drivers_2.6.22/2_key # irq = 49
irq = 49
```

因为这个驱动程序所做的事，就是打印一句话。

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    printk("irq = %d",irq);
    return IRQ_HANDLED;
}
```

驱动成功后，可以“exec 5</dev/buttons”后，可以按单板上的按键，会有相关的 irq 中断号显示出来。按下松开都有中断“双边沿触发”。

接着完善中断处理函数，确定按键值，从原理图上看到，是按键按下后，管脚成低电平，松开后是高电平：



在“中断服务”程序中判断是哪个按键被按下了。根据“irq”号就知道是哪个按键被按下了。是按下还是松开也是确定。
原理图上，当发生了某个中断后，这个引脚是按下还是松开，要去读“管脚”，若按下则这个
“管脚”是低电平，松开时是高电平。
可以用“switch”来判断，但比较麻烦。用结构体。

内核中有一个系统函数，这个系统函数可以去读出那些引脚值。在第 2 个驱动程序中，是读那些寄存器，
再移位再判断，对这些步骤有一个系统函数可以完成。

①，定义一个结构体，和一个结构数组。

```
struct pin_desc {
	unsigned int pin;
	unsigned int key_val;//按 键 值
};

/* 键 值 :按 下 时 ， 值 0x01,0x02,0x03,0x04 */
/* 键 值 :松 开 时 ， 值 0x81,0x82,0x83,0x84 */

struct pin_desc pins_desc[4] = {
		{ s3c2410_GPF0, 0x01},
		{ s3c2410_GPF2, 0x02},
		{ s3c2410_GPG3, 0x03},
		{ s3c2410_GPG11, 0x04},
};
```

给原理图上的按键赋几个值。

```
static int third_drv_open(struct inode *inode, struct file *file)
{
	request_irq(IRQ_EINT0,  buttons_irq, IRQT_BOTHEDGE, "S2", &pins_desc[0]);
	request_irq(IRQ_EINT2,  buttons_irq, IRQT_BOTHEDGE, "S3", &pins_desc[1]);
	request_irq(IRQ_EINT11, buttons_irq, IRQT_BOTHEDGE, "S4", &pins_desc[2]);
	request_irq(IRQ_EINT19, buttons_irq, IRQT_BOTHEDGE, "S5", &pins_desc[3]);

	return 0;
}


int third_drv_close(struct inode *inode, struct file *file)
{
	/*这 里 释 放 中 断 */
	free_irq(IRQ_EINT0, &pins_desc[0]);
	free_irq(IRQ_EINT2, &pins_desc[1]);
	free_irq(IRQ_EINT11,&pins_desc[2]);
	free_irq(IRQ_EINT19, &pins_desc[3]);

	return 0;
}
```

当发生“IRQ_EINT0”这个中断时，就会调用“buttons_irq”这个中断处理函数。这个中断处理函数有
两个参数：

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
```

参 1 中断号 = “IRQ_EINT0”
参 2 是 dev_id = “&pins_desc[0]”

中断处理函数“buttons_irq”如何用这些东西：
定义一个指针“sturct pin_desc * pindesc = (sturct pin_desc *)dev_id ”

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
	struct pin_desc * pindesc = (struct pin_desc *)dev_id;
	unsigned pinval;

	pinval = s3c2410_gpio_getpin(pindesc->pin);
	return IRQ_HANDLED;
}
```

接下来：如果这个引脚值为 1，表明按键是松开的。若这个引脚的值为 0，表明按键是按下的。
将键值保存在一个全局数组中。

```
static unsigned char key_val;
```



```
if (pinval)
	{
		key_val = 0x80 | pindesc->key_val;
	}
else
	{
		key_val = pindesc->key_val;
	}
```

以上在“中断服务程序 buttons_irq” 中已经可以确定键值了。接着要将键值返回给应用程序。

```
#define wait_event_interruptible(wq, condition)                 \
({                                                              \
	int __ret = 0;                                          \
	if (!(condition))                                       \
		__wait_event_interruptible(wq, condition, __ret);   \
	__ret;                                                  \
})
```

这个函数是先判断“condition”，如果它等于 0，就会调用函数“__wait_event_interruptible(wq, condition, __ret);”让这个应用程序“休眠”。
若休眠被唤醒，则会继续往下跑。

要定义两个全局变量：

```
/* 中断事件标志，中断服务程序将它置1，second_drv_read将它清0 */
static volatile int ev_press = 0;
```

```
static DECLARE_WAIT_QUEUE_HEAD (button_waitq);
```

这个宏在“wait.h”中定义：

```
#define DECLARE_WAIT_QUEUE_HEAD(name) \
    wait_queue_head_t name = __WAIT_QUEUE_HEAD_INITIALIZER(name)
```

定义一个 wait_queue_head_t 等待队列头这样一个结构体。这个结构体用后面的宏初始化。

休眠后若有动作发生，就将“键值”拷贝回去。
有休眠谁来唤醒：

```
ev_press = 1;                  /* 表示中断发生 */
wake_up_interruptible (&button_waitq);  /* 唤醒休眠的进程 */
```

休眠时是将进程挂到队列里面：唤醒是将挂在这个队列中的进程唤醒。

```
/* 如 果 没 有 按 键 动 作 ,休 眠 */
wait_event_interruptible(button_waitq, ev_press);
```

字符 设备驱动-POLL 机制：

poll 机制分析

所有的系统调用，基于都可以在它的名字前加上“sys_”前缀，这就是它在内核中对应的函数。比如系统调用 open、read、write、poll，与之对应的内核函数为：sys_open、sys_read、sys_write、sys_poll。

一、内核框架：

对于系统调用 poll 或 select，它们对应的内核函数都是 sys_poll。分析 sys_poll，即可理解 poll 机制。

sys_poll 函数位于 fs/select.c 文件中，代码如下：

```
asmlinkage long sys_poll(struct pollfd __user *ufds, unsigned int nfds,
            long timeout_msecs)
{
    s64 timeout_jiffies;

    if (timeout_msecs > 0) {
#if HZ > 1000
        /* We can only overflow if HZ > 1000 */
        if (timeout_msecs / 1000 > (s64)0x7fffffffffffffffULL / (s64)HZ)
            timeout_jiffies = -1;
        else
#endif
            timeout_jiffies = msecs_to_jiffies(timeout_msecs);
    } else {
        /* Infinite (< 0) or no (0) timeout */
        timeout_jiffies = timeout_msecs;
    }

    return do_sys_poll(ufds, nfds, &timeout_jiffies);
} ? end sys_poll ?
```

其中会去调用 do_sys_poll

它对超时参数稍作处理后，直接调用 do_sys_poll。

do_sys_poll 函数也位于位于 fs/select.c 文件中，我们忽略其他代码：

```
int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds, s64 *timeout)
{
……
poll_initwait(&table);
……
fdcount = do_poll(nfds, head, &table, timeout);
……
```

}

poll_initwait 函数非常简单，它初始化一个 poll_wqueues 变量 table：
poll_initwait > init_poll_funcptr(&pwq->pt, __pollwait); > pt->qproc = qproc;
即 table->pt->qproc = __pollwait，__pollwait 将在驱动的 poll 函数里用到。
table 中的 qproc 函数指针指向“__pollwait”。

```
void poll_initwait(struct poll_wqueues *pwq)
{
     init_poll_funcptr(&pwq->pt, __pollwait);
     pwq->error = 0;
     pwq->table = NULL;
     pwq->inline_index = 0;
}
```

```
static void __pollwait(struct file *filp, wait_queue_head_t *wait_address,
                poll_table *p)
{
    struct poll_table_entry *entry = poll_get_entry(p);
    if (!entry)
        return;
    get_file(filp);
    entry->filp = filp;
    entry->wait_address = wait_address;
    init_waitqueue_entry(&entry->wait, current);
    add_wait_queue(wait_address, &entry->wait);
}
```

```
static inline void init_poll_funcptr(poll_table *pt, poll_queue_proc qproc)
{
    pt->qproc = qproc;
}
```

do_poll 函数位于 fs/select.c 文件中，代码如下：

```
static int do_poll(unsigned int nfds,    struct poll_list *list,
    struct poll_wqueues *wait, s64 *timeout)
{
01 ……
02       for (;;) { //有一个死循环.
03 ……
04                          if (do_pollfd(pfd, pt)) {
05                              count++;
06                              pt = NULL;
07                          }
08 ……
09           if (count || !*timeout || signal_pending(current))
10                 break; //超时跳出。count不为0时，就跳出返回到应用程序中去了。
11           count = wait->error;
12           if (count)
```

```
13                  break;
14
15          if (*timeout < 0) {
16                  /* Wait indefinitely */
17                  __timeout = MAX_SCHEDULE_TIMEOUT;
18          } else if (unlikely(*timeout >= (s64)MAX_SCHEDULE_TIMEOUT-1)) {
19                  /*
20                   * Wait for longer than MAX_SCHEDULE_TIMEOUT. Do it in
21                   * a loop
22                   */
23                  __timeout = MAX_SCHEDULE_TIMEOUT - 1;
24                  *timeout -= __timeout;
25          } else {
26                  __timeout = *timeout;
27                  *timeout = 0;
28          }
29
30          __timeout = schedule_timeout(__timeout); //休眠。
31          if (*timeout >= 0)
32                  *timeout += __timeout;
33      }
34      __set_current_state(TASK_RUNNING);
35      return count;
36 }
```


分析其中的代码，可以发现，它的作用如下：
从 02 行可以知道，这是个循环，它退出的条件为：
09 行的 3 个条件之一(count 非 0，超时、有信号等待处理)
count 非 0 表示 04 行的 do_pollfd 至少有一个成功。
11、12 行：发生错误
重点在 do_pollfd 函数，后面再分析
第 30 行，让本进程休眠一段时间，注意：应用程序执行 poll 调用后，如果①②的条件不满足，进程就会进入休眠。那么，谁唤醒呢？除了休眠到指定时间被系统唤醒外，还可以被驱动程序唤醒——记住这点，这就是为什么驱动的 poll 里要调用 poll_wait 的原因，后面分析。

```
app: poll 应用程序调用poll时，内核就调用了sys_poll。
kernel:  sys_poll
            do_sys_poll(...., timeout_jiffies)
                poll_initwait(&table); 初始化
                    init_poll_funcptr(&pwq->pt, __pollwait); > table->qproc = __pollwait
                do_poll(nfds, head, &table, timeout)
                    for (;;)  死循环
                    {
                        if (do_pollfd(pfd, pt)) {  > mask = file->f_op->poll(file, pwait);  return mask;
                            count++;  // 如果驱动的poll返回非0值，那么count++
                            pt = NULL;
                        }

                        // break的条件：count非0，超时，有信号在等待处理
                        if (count || !*timeout || signal_pending(current))
                            break;

                          若上面的条件都不满足时执行下面的“休眠”，休眠时间为__timeout。

                        // 休眠
                        __timeout = schedule_timeout(   timeout);

                        当休眠到“__timeout”时间后，__timeout变为0，就又开始for(;;)起始处执行，而这时“break条件”中“超时” “!*timeout”为真，则执行了“break”。跳出了do_poll().
                    }
```

do_pollfd 函数位于 fs/select.c 文件中，代码如下：

```
static inline unsigned int do_pollfd(struct pollfd *pollfd, poll_table *pwait)
{
......
if (file->f_op && file->f_op->poll)
mask = file->f_op->poll(file, pwait);
......
}
```

```
const struct file_operations    *f_op;
```

可见，do_pollfd 函数就是调用我们的驱动程序里注册的 poll 函数。这里有个“f_op”这就是一个 file_operation。

```
static struct file_operations sencod_drv_fops = {
    .owner   =  THIS_MODULE,    /* 这是一个宏，推向编译模块时自动创建的__this_module变量 */
    .open    =  forth_drv_open,
    .read    =  forth_drv_read,
    .release =  forth_drv_close,
    .poll    =  forth_drv_poll,   驱动程序中的 poll 。
};
```

```
static unsigned forth_drv_poll(struct file *file, poll_table *wait)
{
    unsigned int mask = 0;
    poll_wait(file, &button_waitq, wait); // 不会立即休眠

    if (ev_press)          驱动程序中的poll会调用 poll_wait () 。
        mask |= POLLIN | POLLRDNORM;

    return mask;
}
```

二、驱动程序：

驱动程序里与 poll 相关的地方有两处：一是构造 file_operation 结构时，要定义自己的 poll 函数。二是通过 poll_wait 来调用上面说到的__pollwait 函数，pollwait 的代码如下：

```
static inline void poll_wait(struct file * filp, wait_queue_head_t * wait_address, poll_tabl
{
    if (p && wait_address)
        p->qproc(filp, wait_address, p);
}
```

p->qproc 就是“__pollwait 函数”：
在“do_sys_poll”函数中定义一个“poll_wqueues”结构的变量“table”，且在“do_sys_poll”中一个函数“poll_initwait(&table);”使用了这个变量。而"void poll_initwait(struct poll_wqueues *pwq)"中调用 了一个函数“init_poll_funcptr(&pwq->pt, __pollwait);”则，最后要看如下操作：“init_poll_funcptr(&table->pt, __pollwait);”，接着再看“init_poll_funcptr()”函数。

```
static inline void init_poll_funcptr(poll_table *pt, poll_queue_proc qproc)
{
pt->qproc = qproc;
}
```


p->qproc 就是__pollwait 函数，从它的代码可知，它只是把当前进程挂入我们驱动程序里定义的一个队列里而已。它的代码如下：

```
static void __pollwait(struct file *filp, wait_queue_head_t *wait_address,
poll_table *p)
{
struct poll_table_entry *entry = poll_get_entry(p);
if (!entry)
return;
get_file(filp);
entry->filp = filp;
entry->wait_address = wait_address;
init_waitqueue_entry(&entry->wait, current);
add_wait_queue(wait_address, &entry->wait);
}
```

执行到驱动程序的 poll_wait 函数时，进程并没有休眠，我们的驱动程序里实现的 poll 函数是不会引起休眠的。让进程进入休眠，是前面分析的 do_sys_poll 函数的 30 行“__timeout = schedule_timeout(__timeout)”。
poll_wait 只是把本进程挂入某个队列，应用程序调用 poll > sys_poll > do_sys_poll > poll_initwait，do_poll > do_pollfd > 我们自己写的 poll 函数后，再调用 schedule_timeout 进入休眠。如果我们的驱动程序发现情况就绪，可以把这个队列上挂着的进程唤醒。可见，poll_wait 的作用，只是为了让驱动程序能找到要唤醒的进程。即使不用 poll_wait，我们的程序也有机会被唤醒：chedule_timeout(__timeout)，只是要休眠__time_out 这段时间。

```c
static unsigned forth_drv_poll(struct file *file, poll_table *wait)
{
    unsigned int mask = 0;
    poll_wait(file, &button_waitq, wait); // 不会立即休眠,这只是让进程挂到队列里面去。
    // 休眠是在 "do_poll()" 中的 "schedule_timeout()"
    if (ev_press) // 若当前有数据可以返回到应用程序时，就返回 mask |=POLLIN | POLLRDNORM.否则mask=0。
    //若mask=0,则 do_poll()中的 count++就不会执行，则判断if(count || !*timeout || signal_pending(current)) ，此
    //时conut为0，则不会执行if下的break跳出，而是执行if后面的"休眠"schedule_timeout().
        mask |= POLLIN | POLLRDNORM;

    return mask;
}
```

```c
app: poll
kernel:  sys_poll
            do_sys_poll(...., timeout_jiffies)
                poll_initwait(&table);
                    init_poll_funcptr(&pwq->pt, __pollwait); > table->qproc = __pollwait
                do_poll(nfds, head, &table, timeout)
                   for (;;)
                   {
                       if (do_pollfd(pfd, pt)) {  > mask = file->f_op->poll(file, pwait);  return mask;
                                                  // 驱动的poll:
                                                    pollwait(filp, &button_waitq, p);
                                                  把当前进程挂到button_waitq队列里去
                           在驱动中“forth_drv_poll()”中若mask返回0时，就不会count++.
                           count++;  // 如果驱动的poll返回非0值，那么count++
                           pt = NULL;
                       }
                       上面驱动中的poll"forth_drv_poll()"的返回的mask=0时，count为0.
                       // break的条件：count非0，超时，有信号在等待处理
                       if (count || !*timeout || signal_pending(current))
                           break;                                          这时if不会执行“break”，接着直接执行下现的“schedule_timeout”休眠。


                       // 休眠__timeout 5s
                       __timeout = schedule_timeout(__timeout);

                   }
```

现在来总结一下 poll 机制：

1. poll > sys_poll > do_sys_poll > poll_initwait，poll_initwait 函数注册一下回调函数__pollwait，它就是我们的驱动程序执行 poll_wait 时，真正被调用的函数。

2. 接下来执行 file->f_op->poll，即我们驱动程序里自己实现的 poll 函数
   它会调用 poll_wait 把自己挂入某个队列，这个队列也是我们的驱动自己定义的；
   它还判断一下设备是否就绪。

3. 如果设备未就绪，do_sys_poll 里会让进程休眠一定时间

4. 进程被唤醒的条件有 2：一是上面说的“一定时间”到了，二是被驱动程序唤醒。驱动程序发现条件就绪时，就把“某个队列”上挂着的进程唤醒，这个队列，就是前面通过 poll_wait 把本进程挂过去的队列。

5. 如果驱动程序没有去唤醒进程，那么 chedule_timeout(__timeou)超时后，会重复 2、3 动作，直到应用程序的 poll 调用传入的时间到达。

字符设备驱动 异步通知：

为了使设备支持异步通知机制，驱动程序中涉及以下 3 项工作：
1. 支持 F_SETOWN 命令，能在这个控制命令处理中设置 filp->f_owner 为对应进程 ID。
   不过此项工作已由内核完成，设备驱动无须处理。
2. 支持 F_SETFL 命令的处理，每当 FASYNC 标志改变时，驱动程序中的 fasync()函数将得以执行。
   驱动中应该实现 fasync()函数。

3. 在设备资源可获得时，调用 kill_fasync()函数激发相应的信号

应用程序：

fcntl(fd, F_SETOWN, getpid()); // 告诉内核，发给谁
应用程序会调用“fcntl()”这个函数，把进程的 PID 号告诉给驱动程序。
应用程序还要通过“F_GETFL”读出“flags”，在 flags 上置上“FASYNC”位。
Oflags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, Oflags | FASYNC); // 改变 fasync 标记，最终会调用到驱动的 faync > fasync_helper：初始化/释放 fasync_struct

一，应用程序主动的去查询或 read。

1.查询方式：很占资源。
2.中断机制：虽然有休眠，但在没有按键按下时 read（）会一直等待，永远不会返回。
3.poll 机制：指定超时时间。
以上都是“应用程序”主动去读或查询。

二，异步通知：

有按键按下了，驱动程序来提醒（触发）“应用程序”去读键值。用“signal”
进程之间发信号：
kill -9 pid
kill 是信号发送者，
pid 具体进程是信号接收者。
信号值是“9”

“信号”与“中断”差不多。注册中断处理函数时是用“request_irq(中断号，处理函数)”。
信号也是有一个“信号”和“处理函数”。

参数是“信号的值”，和要挂接的“信号处理函数”。

```c
#include <stdio.h>
#include <signal.h> //信号处理需要这个头文件

void my_signal_fun(int signum)
{
    static int cnt = 0;
    printf("signal = %d, %d times\n", signum, ++cnt);
}

int main(int argc, char **argv)
{
    //做什么事情，设置一个“信号处理函数”。
    signal(SIGUSR1, my_signal_fun);

    while (1)
    {
        sleep(1000);//这个主函数实现的是“休眠”。
    }
    return 0;
}
```

在后台执行：

```
root@book-desktop:/home/book/testCode# ./signal &
[1] 2862
```

```
root      2862  0.0  0.0   1524   280 pts/2    S    11:19   0:00 ./signal
```

可见一直在“休眠”状态。

这时发送一个信号给它。用应用程序“kill”给它发一个信号。发“USR1”信号给进程。首先要知道进程号是多少。上面是“2862”

```
root@book-desktop:/home/book/testCode# kill -USR1 2862
root@book-desktop:/home/book/testCode# signal = 10, 1 times
^C
root@book-desktop:/home/book/testCode# kill -USR1 2862
root@book-desktop:/home/book/testCode# signal = 10, 2 times
^C
root@book-desktop:/home/book/testCode# kill -USR1 2862
root@book-desktop:/home/book/testCode# signal = 10, 3 times
^C
root@book-desktop:/home/book/testCode# kill -10 2862
root@book-desktop:/home/book/testCode# signal = 10, 4 times
```

程序"signal"一收到这个“Kill”信号时，就会调用里面的“信号处理函数”“my_signal_fun”。
－USR1 与 －10 是一样的。

kill -9 pid：9 这个信号处理函数就是让这个进程退出来。

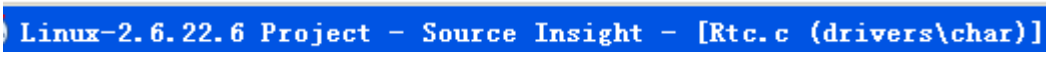
1，先注册“信号处理函数”。
2，发送信号。

①，谁来发信号。
②，发给谁。
③，怎么发。

三，异步通知功能的驱动函数的应用程序：

目标：按下按键时，驱动程序通知应用程序。（以前是应用程序主动读取按键值）
1，应用程序中要注册“信号处理函数”。因为通知它做什么事情，这要一个函数来做。
2，谁发：是驱动程序发送信号。
3，发给谁：信号发送给应用程序。应用程序要告诉驱动程序它自己的 PID，
4，如何发：驱动程序中调用某个函数（kill_fasync()）

找一个字符驱动查看 kill_fasync 函数的用法。接上面 4 个步骤查看用法，下面找到一个：



1，首先看这个字符设备中定义的一个“fasync_struct”结构的变量：rtc_async_queue。
2，看这个变量的定义，初始化和使用，整个 rtc.c 中有三处出现“rtc_async_queue”：
定义：

```
static struct fasync_struct *rtc_async_queue;
```

使用：

```
kill_fasync (&rtc_async_queue, SIGIO, POLL_IN);
```

初始化：

```
static int rtc_fasync (int fd, struct file *filp, int on)
{
    return fasync_helper (fd, filp, on, &rtc_async_queue);
}
```

3，要清楚使用方法就要清楚“rtc_fasync（）”在哪里定义：

```
static const struct file_operations rtc_fops = {
    .owner      = THIS_MODULE,
    .llseek     = no_llseek,
    .read       = rtc_read,
#ifdef RTC_IRQ
    .poll       = rtc_poll,
#endif
    .ioctl      = rtc_ioctl,
    .open       = rtc_open,
    .release    = rtc_release,
    .fasync     = rtc_fasync,
};
```

在这个“rtc.c”中可以找到这个函数的定义，它是一个“file_operatons”结构中的成员。

以上弄清楚“kill_fasync”函数的用法之后，可以依照：
在“中断服务程序”中发：即当有按键按下时就发送一个信号给应用程序。
首先声明一个“fasync_struct”结构变量。

```
static struct fasync_struct *button_async;
```

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    struct pin_desc * pindesc = (struct pin_desc *)dev_id;
    unsigned int pinval;

    pinval = s3c2410_gpio_getpin(pindesc->pin);

    if (pinval)
    {
        /* 松 开 */
        key_val = 0x80 | pindesc->key_val;
    }
    else
    {
        /* 按 下 */
        key_val = pindesc->key_val;
    }

  ev_press = 1;                  /* 表 示 中 断 发 生 了 */
  wake_up_interruptible(&button_waitq);  /* 唤 醒 休 眠 的 进 程 */

    kill_fasync (&button_async, SIGIO, POLL_IN);

    return IRQ_RETVAL(IRQ_HANDLED);
} ? end buttons_irq ?
```

有按键按下了就发一个信号给应用程序。
其中要定义一个结构，取名为“button_async”.
以上便可以发送信号出去了。

发给谁肯定是这个结构中定义的。这个结构“button_async”要做些初始化。
有如下例子，这里的结构体是“rtc_async_queue”，看看这个“fasync_helper”函数是否在初始化这个结构体。

```
static int rtc_fasync (int fd, struct file *filp, int on)
{
    return fasync_helper (fd, filp, on, &rtc_async_queue);
}
```

查询到函数“rtc_fasync”的定义：

```
static const struct file_operations rtc_fops = {
    .owner      = THIS_MODULE,
    .llseek     = no_llseek,
    .read       = rtc_read,
#ifdef RTC_IRQ
    .poll       = rtc_poll,
#endif
    .ioctl      = rtc_ioctl,
    .open       = rtc_open,
    .release    = rtc_release,
    .fasync     = rtc_fasync,
};
```

所以可以依照这个例子做。在“file_operations 结构中加一个成员。
原型是：

```
int (*fasync)(int, struct file *, int);
```

```
static struct file_operations sencod_drv_fops = {
    .owner  = THIS_MODULE,  /*这 是 一 个 宏
    .open   = fifth_drv_open,
    .read       =   fifth_drv_read,
    .release = fifth_drv_close,
    .poll   = fifth_drv_poll,
    .fasync     = fifth_drv_fasync,
};
```

然后加上这个初始化“button_async”结构的函数“fasync_help”：

```
static int fifth_drv_fasync (int fd, struct file *filp, int on)
{
    printk("driver: fifth_drv_fasync\n");
    return fasync_helper (fd, filp, on, &button_async);
}
```

这个函数“fifth_drv_fasync”什么情况下调用：

信号是驱动程序发的，在中断服务程序“buttons_irq”通过“kill_fasync (&button_async, SIGIO, POLL_IN);”来发。发给谁是包含在“button_async”这个结构体中。这个结构体在“fifth_drv_fasync”中由“fasync_helper”来初始化。这个函数“fifth_drv_fasync”什么时候被调用，是应用程序调用“.fasync”

```
static struct file_operations sencod_drv_fops = {
    .owner  = THIS_MODULE,  /*这 是 一 个
    .open   = fifth_drv_open,
    .read       =   fifth_drv_read,
    .release = fifth_drv_close,
    .poll   = fifth_drv_poll,
    .fasync     = fifth_drv_fasync,
};
```

这么一个“.fasync”系统调用时，就会调用到：fifth_drv_fasync

```
static int fifth_drv_fasync (int fd, struct file *filp, int on)
{
	printk("driver: fifth_drv_fasync\n");
	return fasync_helper (fd, filp, on, &button_async);
}
```

所以显然是应用程序要调用“.fasync”来设置那个“发给谁”。

```
为了使设备支持异步通知机制，驱动程序中涉及以下3项工作：
1. 支持F_SETOWN命令，能在这个控制命令处理中设置filp->f_owner为对应进程ID。
   不过此项工作已由内核完成，设备驱动无须处理。
2. 支持F_SETFL命令的处理，每当FASYNC标志改变时，驱动程序中的fasync()函数将得以执行。
   驱动中应该实现fasync()函数。

3. 在设备资源可获得时，调用kill_fasync()函数激发相应的信号


应用程序：
fcntl(fd, F_SETOWN, getpid());  // 告诉内核，发给谁

Oflags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, Oflags | FASYNC);  // 改变fasync标记，最终会调用到驱动的faync > fasync_helper：初始化/释放fasync_struct
```

为了使设备支持异步通知机制，驱动程序中涉及以下 3 项工作：

1. 支持 F_SETOWN 命令，能在这个控制命令处理中设置 filp->f_owner 为对应进程 ID。
   不过此项工作已由内核完成，设备驱动无须处理。
2. 支持 F_SETFL 命令的处理，每当 FASYNC 标志改变时，驱动程序中的 fasync()函数将得以执行。
   驱动中应该实现 fasync()函数。
3. 在设备资源可获得时，调用 kill_fasync()函数激发相应的信号

应用程序：

fcntl(fd, F_SETOWN, getpid()); // 告诉内核，发给谁
应用程序会调用“fcntl()”这个函数，把进程的 PID 号告诉给驱动程序。
应用程序还要通过“F_GETFL”读出“flags”，在 flags 上置上“FASYNC”位。
Oflags = fcntl(fd, F_GETFL);
fcntl(fd, F_SETFL, Oflags | FASYNC); // 改变 fasync 标记，最终会调用到驱动的 faync > fasync_helper：初始化/释放 fasync_struct
如上面“fcntl(fd, F_SETFL, Oflags | FASYNC);”接口被调用时，下在函数中的“fasync_helper（）”就
会被调用：

```
static int fifth_drv_fasync (int fd, struct file *filp, int on)
{
    printk("driver: fifth_drv_fasync\n");
    return fasync_helper (fd, filp, on, &button_async);
}
```

应用程序会调用“F_SETOWN”即“fcntl(fd, F_SETOWN, pid)”这样一个函数，来进程的 pid 告诉驱动程序。

然后应用程序还要通过“F_GETFL”读出标志位，即 flags，先读出来再在这个 flags 上面修改，加上一个“FASYNC”。异步通知的 flags.每当 FAYNC 标志“flags”改变时，驱动程序里面的

```
.fasync     = fifth_drv_fasync,
```

这个函数会被调用。
这个函数也只是用一个“fasync_helper”，这是内核帮做的一个函数。是辅助作用。将结构体变量
“button_async”初始化，我们定义了但没去初始化它，这个内核辅助函数就去初始化它。
这个结构体“button_async”初始化后，中断服务程序“buttons_irq”就能调用：

```
kill_fasync (&button_async, SIGIO, POLL_IN);
```

这个函数了。

当应用程序调用这个接口“fcntl(fd, SETFL, oflags | FASYNC)"时，函数“fasync_helper”就会被调用。
驱动程序对“F_SETOWN”的处理是由内核完成的。

```
int fd;
void my_signal_fun(int signum)
{
	unsigned char key_val;
	read(fd, &key_val, 1); //读 了 按 键 值 .
	printf("key_val: 0x%x\n", key_val);
}

int main(int argc, char **argv)
{
	unsigned char key_val;
	int ret;
	int Oflags;

	/*内 核 中 发 出 信 号 是 发 出 SIGIO,表 示 IO口 有 数 据 让 你 读 和 写 */
	signal(SIGIO, my_signal_fun);

	fd = open("/dev/buttons", O_RDWR);
	if (fd < 0)
	{
		printf("can't open! \n");
	}

	fcntl(fd, F_SETOWN, getpid());
	Oflags = fcntl(fd, F_GETFL);
	fcntl(fd, F_SETFL, Oflags | FASYNC);

	/*主 函 数 中 只 用 sleep， 所 有 工 作 在 "信 号 处 理 函 数 "中 实 现 */
	/*当 然 主 函 数 中 可 以 加 上 其 他 工 作 需 要 的 函 数 */
	while (1)
	{
		sleep(1000);
	}

	return 0;
} ? end main ?
```

这是驱动测试程序。它不会主动去读按键值“read(fd, &key_val, 1); "。这个“my_signal_fun”信号函数是在驱动服务程序“buttons_irq”里，发现有按键按下时，给应用程序发“kill_fasync (&button_async, SIGIO, POLL_IN);"信号，这个信号会触发应用程序来调用它的信号处理函数“my_signal_fun”。

编译运行测试：

```
# insmod ./fifth_drv.ko
# lsmod
Module                  Size  Used by    Not tainted
fifth_drv               3976  0
# ./fifthdrvtest &
#
#
# ps
  PID  Uid        VSZ Stat Command
    1 0          3092 S    init
    2 0               SW<  [kthreadd]
    3 0               SWN  [ksoftirqd/0]
    4 0               SW<  [watchdog/0]
    5 0               SW<  [events/0]
    6 0               SW<  [khelper]
   55 0               SW<  [kblockd/0]
   56 0               SW<  [ksuspend_usbd]
   59 0               SW<  [khubd]
   61 0               SW<  [kseriod]
   73 0               SW   [pdflush]
   74 0               SW   [pdflush]
   75 0               SW<  [kswapd0]
   76 0               SW<  [aio/0]
  711 0               SW<  [mtdblockd]
  746 0               SW<  [kmmcd]
  763 0               SW<  [rpciod/0]
  771 0          3096 S    -sh
  777 0          1308 S    ./fifthdrvtest
  778 0          3096 R    ps
#
```

得告诉驱动程序，我们应用程序的进程 PID 号，

7，字符设备驱动 同步互斥阻塞

1. 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。
常用原子操作函数举例：

```
atomic_t v = ATOMIC_INIT(0);        //定义原子变量 v 并初始化为 0
atomic_read(atomic_t *v);           //返回原子变量的值
void atomic_inc(atomic_t *v);       //原子变量增加 1
void atomic_dec(atomic_t *v);       //原子变量减少 1
int atomic_dec_and_test(atomic_t *v); //自减操作后测试其是否为 0，为 0 则返回 true，否则返回 false。
```

2. 信号量

信号量（semaphore）是用于保护临界区的一种常用方法，只有得到信号量的进程才能执行临界区代码。
当获取不到信号量时，进程进入休眠等待状态。

定义信号量

```
struct semaphore sem;
```

初始化信号量

```
void sema_init (struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);//初始化为 0

static DECLARE_MUTEX(button_lock);        //定义互斥锁
```

获得信号量

```
void down(struct semaphore * sem);
int down_interruptible(struct semaphore * sem);
int down_trylock(struct semaphore * sem);
```

释放信号量

```
void up(struct semaphore * sem);
```

3. 阻塞

阻塞操作
是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。

非阻塞操作
进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

fd = open("...", O_RDWR | O_NONBLOCK);

目标：同一时刻只能有一个应用程序打开驱动程序/dev/buttons。

1. 原子操作

原子操作指的是在执行过程中不会被别的代码路径所中断的操作。
常用原子操作函数举例：
atomic_t v = ATOMIC_INIT(0);　　//定义原子变量 v 并初始化为 0
atomic_read(atomic_t *v);　　//返回原子变量的值
void atomic_inc(atomic_t *v);　　//原子变量增加 1
void atomic_dec(atomic_t *v);　　//原子变量减少 1
int atomic_dec_and_test(atomic_t *v); //自减操作后测试其是否为 0，为 0 则返回 true，否则返回 false。

最简单的方法是在“open”中做一个标记。

```
struct pin_desc pins_desc[4] = {
    {S3C2410_GPF0, 0x01},
    {S3C2410_GPF2, 0x02},
    {S3C2410_GPG3, 0x03},
    {S3C2410_GPG11, 0x04},
};

static int canopen = 1;
```

```
static int sixth_drv_open(struct inode *inode, struct file *file)
{
    if (--canopen != 0)
    {
        canopen++;
        return -EBUSY;
    }

    /* 配 置 GPF0,2为 输 入 引 脚 */
    /* 配 置 GPG3,11为 输 入 引 脚 */
    request_irq(IRQ_EINT0,  buttons_irq, IRQT_BOTHEDGE, "S2", &pins_desc[0]);
    request_irq(IRQ_EINT2,  buttons_irq, IRQT_BOTHEDGE, "S3", &pins_desc[1]);
    request_irq(IRQ_EINT11, buttons_irq, IRQT_BOTHEDGE, "S4", &pins_desc[2]);
    request_irq(IRQ_EINT19, buttons_irq, IRQT_BOTHEDGE, "S5", &pins_desc[3]);

    return 0;
}
```

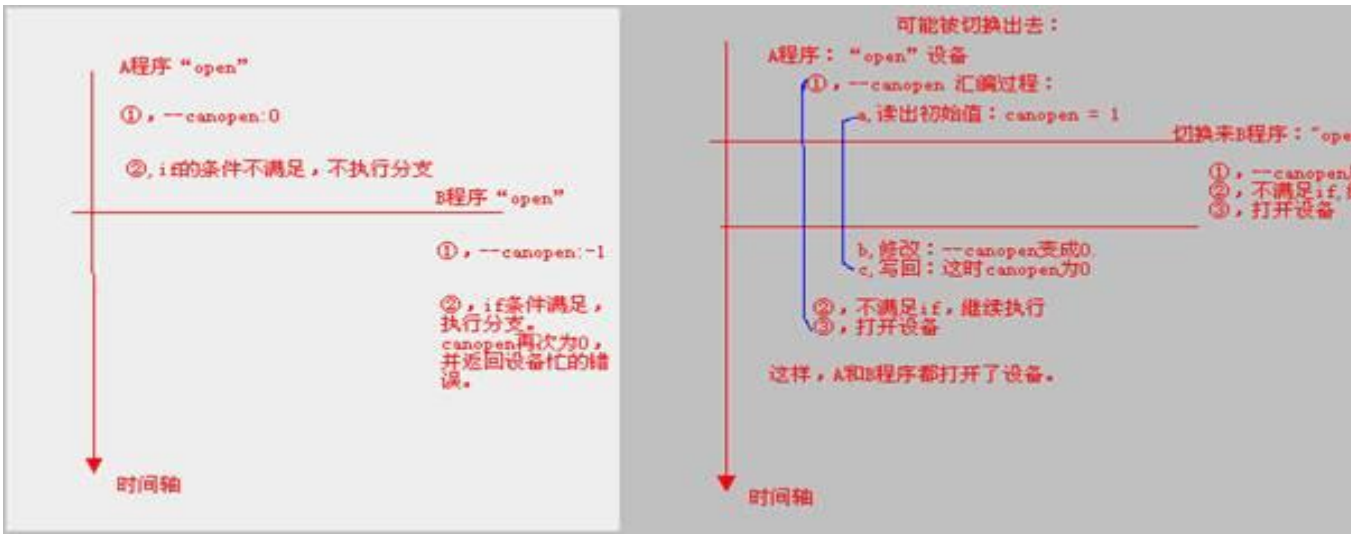
一上来让“canopen”自减，自减应该等于 0，如果等于 0 时，表明还没有打开过它。这样就可以往下走。
如果有人打开过了，则“canopen”为 0 了，“--canopen”时，为“－1”了，则这时，在“if”中会进入到里面
执行，就返回了“-EBUSY”.

关闭这个设备时，在关闭函数中加 1 即可。

```
int sixth_drv_close(struct inode *inode, struct file *file)
{
    canopen++;
    free_irq(IRQ_EINT0, &pins_desc[0]);
    free_irq(IRQ_EINT2, &pins_desc[1]);
    free_irq(IRQ_EINT11, &pins_desc[2]);
    free_irq(IRQ_EINT19, &pins_desc[3]);
    return 0;
}
```

A 程序来“open”时：
①，canopen 自减 1：--canopen。原来 canopen=1,则这时 canoepen=0
②，在 open 中判断"canopen"是否等于 0，刚开始是等于 0 的，则继续往下跑，不会执行 If 的分支。

这时又来了个 B 程序来“open”：
①，canopen 自减 1：--canopen。原来 canopen=0,则这时 canoepen=－1
②，在 open 中判断"canopen"是否等于 0，最开始是等于 0 的，但经过 A 程序 open 后，再次自减后为－1，则执行 If 的分支，
canopen++又变成 0 去，返回设备忙的错误“EBUSY”



感觉这个方法正确，但里面实质上有漏洞，如上右图。
LINUX 是多任务系统，A 程序执行过程中，有可能被切换出去换成 B 程序执行这种情况。
“--canopen”在汇编里看，是被分成了 3 步：“读出－－修改－－写回”。这个过程不是“原子操作”。中间有被切换出去的可能。

修改代码将“canopen”定义成“原子变量”。初始值“1”。

```
static atomic_t canopen = ATOMIC_INIT(1);
```

在“open”操作中：

```
if (!atomic_dec_and_test(&canopen))
```

“canopen”一开始是“1”，自减后判断它是否为“0”。是“0”则为“true”，这里前面加个“!”非，所以“if”中的分支不执行。
假设“canopen”已经被调用过一次变成“0”了，则“atomic_dec_and_test(&canopen)”中自减操作后变成“－1”了，则为“false”。
加上“!”非，则“if(true)”会执行 if 分支。在分支中将“canopen”自加一次。

```
static int sixth_drv_open(struct inode *inode, struct file *file)
{
    if (!atomic_dec_and_test(&canopen))
    {
        atomic_inc(&canopen);
        return -EBUSY;
    }
```

在“close”设备中：

```
int sixth_drv_close(struct inode *inode, struct file *file)
{
    atomic_inc(&canopen);
```

这样就不会发生“A 程序执行时被 B 程序切换出去”的情况。汇编过程中的“读出－修改－写回”在“int atomic_dec_and_test(atomic_t *v)”一次性完成。中间不会被打断。这就是“原子变量”。

2. 信号量

信号量（semaphore）是用于保护临界区的一种常用方法，只有得到信号量的进程才能执行临界区代码。
当获取不到信号量时，进程进入休眠等待状态。

定义信号量
struct semaphore sem;
初始化信号量
void sema_init (struct semaphore *sem, int val);
void init_MUTEX(struct semaphore *sem);//初始化为 0

static DECLARE_MUTEX(button_lock);　　//定义互斥锁

获得信号量
void down(struct semaphore * sem);//获取信号量
int down_interruptible(struct semaphore * sem); //这是获取不到信号量就去休眠。休眠状态可被打断。
int down_trylock(struct semaphore * sem);
释放信号量
void up(struct semaphore * sem);

就是在操作之前申请一个“信号量”。申请不到，则要么等待，要么休眠；如果申请到了“信号量”则继续往下执行代码。操作完毕，要释放这个信号量。这时若有其他程序在等待信号量，就去唤醒那个应用程序。
①，先定义“信号量”：这里用“DECLARE_MUTEX”宏，这个宏定义的初始化一起做好了。

```
#define __SEMAPHORE_INIT(name, cnt)                     \
{                                                       \
    .count      = ATOMIC_INIT(cnt),                     \
    .wait       = __WAIT_QUEUE_HEAD_INITIALIZER((name).wait), \
}

#define __DECLARE_SEMAPHORE_GENERIC(name,count)  \
    struct semaphore name = __SEMAPHORE_INIT(name,count)

#define DECLARE_MUTEX(name)         __DECLARE_SEMAPHORE_GENERIC(name,1)
```

```
static DECLARE_MUTEX(button_lock);   //定 义 互 斥 锁
```

②，在“open”中获取“信号量”。

```
/*获 取 信 号 量 */
down(&button_lock);
```

如果是第一次执行“open”时，这里便会获取到“信号量”。若是另一个程序又来调用“open”设备，这时
down(&button_lock);是无法获取信号量的。

第一个程序执行完成后，这个信号量就要释放出来给其他程序“open”这个设备时用。释放是在“close”、
函数中：

```
up(&button_lock);
```

这时编译执行测试程序：

```
# ./sixthdrvtest &
#
# ps
  PID  Uid        VSZ Stat Command
    1 0         3092 S    init
    2 0              SW<  [kthreadd]
    3 0              SWN  [ksoftirqd/0]
    4 0              SW<  [watchdog/0]
    5 0              SW<  [events/0]
    6 0              SW<  [khelper]
   55 0              SW<  [kblockd/0]
   56 0              SW<  [ksuspend_usbd]
   59 0              SW<  [khubd]
   61 0              SW<  [kseriod]
   73 0              SW   [pdflush]
   74 0              SW   [pdflush]
   75 0              SW<  [kswapd0]
   76 0              SW<  [aio/0]
  711 0              SW<  [mtdblockd]
  746 0              SW<  [kmmcd]
  763 0              SW<  [rpciod/0]
  771 0         3096 S    -sh
  846 0         1308 S    ./sixthdrvtest
  848 0         1308 D    ./sixthdrvtest
  849 0         3096 R    ps
```

加载驱动程序后，这是第2次执行“应用程序”测试驱动。

这里可见有两个进程。

可见可以同时运行两个测试程序。不像“原子变量”中：

```
fd = open("/dev/buttons", O_RDWR);
if (fd < 0)
{
    printf("can't open!\n");
    return -1;
}
```


一次打开一次，第二次直接打印信息了。
而“信号量”申请操作中，第二个程序的进程处于“D”状态，是种“僵死”状态或称为一种不可中断的“睡眠”状态。
这是因为第 2 次在“open”函数中在“获取信号量”时休眠。

```
/*获 取 信 号 量 */
down(&button_lock);
```

这个休眠只有在第一个应用程序“close”：

```
up(&button_lock);
```

释放“信号量”后，才会唤醒：

```
/*获 取 信 号 量 */
down(&button_lock);
```

这时杀掉一个进程：发现程序继续进行。

```
  846 0          1308 S    ./sixthdrvtest
  848 0          1308 D    ./sixthdrvtest
  850 0          3096 R    ps
# kill -9 846
# driver: sixth_drv_fasync

[1] - Killed                         ./sixthdrvtest
# ps
  PID  Uid        VSZ Stat Command
    1 0          3092 S    init
    2 0               SW<  [kthreadd]
    3 0               SWN  [ksoftirqd/0]
    4 0               SW<  [watchdog/0]
    5 0               SW<  [events/0]
    6 0               SW<  [khelper]
   55 0               SW<  [kblockd/0]
   56 0               SW<  [ksuspend_usbd]
   59 0               SW<  [khubd]
   61 0               SW<  [kseriod]
   73 0               SW   [pdflush]
   74 0               SW   [pdflush]
   75 0               SW<  [kswapd0]
   76 0               SW<  [aio/0]
  711 0               SW<  [mtdblockd]
  746 0               SW<  [kmmcd]
  763 0               SW<  [rpciod/0]
  771 0          3096 S    -sh
  848 0          1308 S    ./sixthdrvtest
  850 0          3096 R    ps
#
```

这时 PID848 处于正常的运行状态了。

3. 阻塞

阻塞操作
是指在执行设备操作时若不能获得资源则挂起进程，直到满足可操作的条件后再进行操作。被挂起的进程进入休眠状态，被从调度器的运行队列移走，直到等待的条件被满足。

非阻塞操作
进程在不能进行设备操作时并不挂起，它或者放弃，或者不停地查询，直至可以进行操作为止。

fd = open("...", O_RDWR | O_NONBLOCK);

如想读一个按键值，若当前没有按键按下，就一直等待有按下才返回，这是“阻塞操作”；“非阻塞操作”是指若想读一个按键值，若没有按键值可读，则立刻返回一个错误。
分辨他们，是在 open 设备时，加入一个参数。若传入“ O_NONBLOCK”就是非阻塞操作，若不传入“ O_NONBLOCK”标记时，
默认为“阻塞操作”。对于“阻塞、非阻塞”，驱动程序要对这个“ O_NONBLOCK”进行处理。

```
struct file {
    struct file *next;
    struct file *parent;
    char *name;
    int lineno;
    int flags;
};
```

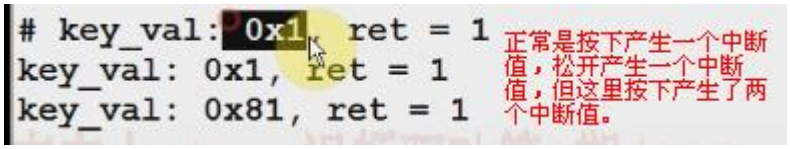
```c
static int sixth_drv_open(struct inode *inode, struct file *file)
{
	if (file->f_flags & O_NONBLOCK)
	{
		if (down_trylock(&button_lock))
			return -EBUSY;
	}
	else
	{
		/* 获取信号量 */
		down(&button_lock);
	}
```

这个“O_NONBLOCK”标记就是在上面“file”结构中获取。这个结构是内核提供的。如果“file->f_flags”是“ O_NONBLOCK”时，
是“非阻塞”操作，否则就是“阻塞”操作。
如果它无法获取这个“信号量”，这个“down(&button_lock);”会陷入休眠。如果“open”打不开，立刻返回一个错误，
就用了“down_trylock(&button_lock)”这个函数。表示如果无法获取到“信号量”就返回“return -EBUSY”.

“read”中也要做这样的事：

```c
ssize_t sixth_drv_read(struct file *file, char __user *buf, size_t size, loff_t *ppos)
{
	if (size != 1)
		return -EINVAL;

	if (file->f_flags & O_NONBLOCK)
	{
		if (!ev_press) /*如果是非阻塞，则立马判断有没有按键ev_press发生 */
			return -EAGAIN; /*如果没有按键发生则返回-EAGAIN再次来执行 */
	}
	else	/*否则若是阻塞方式，则要调用下来的代码判断下若没按键则休眠 */
	{
		/*如果没有按键动作,休眠 */
		wait_event_interruptible(button_waitq, ev_press);
	}

	/*如果有按键动作,返回键值 */
	copy_to_user(buf, &key_val, 1);
	ev_press = 0;

	return 1;
} ? end sixth_drv_read ?
```

8.按键消抖



这里产生了“抖动”，按键是机械开关，按下松开时里面的金属弹片可能抖动了好几次。这种抖动产生了多次“脉
冲”导致多次中断。
2410 核心

一，定时器：引入这个概念将“抖动”去掉。

定时器有两个概念：
1，超时时间：
2，时间到了之后的“处理函数”。
可以在中断处理中，如定时 10ms 后处理确定按键值上报。
产生中断
在中断中加定时器，当遇到 A 中断时加一个 10ms 的定时器，过了 10ms 后就去执行“处理函数”（确定按键值上
报）。因为机械的抖动会非常快，没等到 10ms 后的处理，这时因为抖动又来了一个中断 B，这时中断 B 把之前的那个定时器修改了。所以 A 中断的定时器就取消了。最后又来了一个中断 C，同样会修改掉 B 中断的定时器。上图中是假设抖动时产生了 3 个中断，所以对于同一个“定时器”，最终中断 C 的定时器没有被修改，所以 10ms 后由中断 C 的处理函数上报了按键值。最后这个 10ms 是从抖动 C 处开始。
这样 3 个抖动的中断只会导致最后处理一个“上报按键值”（定时器过后的处理函数只会执行一次）。以上便是用定
时器消除抖动的原理。
因为是修改同一个定时器，所以前面的定时又取消，相当于把“闹钟”时间往后调整时，最终只要响一次闹铃。

1，在内核中比较 add_timer(&timer)的用法：

```
static struct timer_list timer;
```

定义一个 timer_list 结构体变量“timer”。

```
init_timer(&timer);初始化时间
timer.data     = (unsigned long) SCpnt;
timer.expires  = jiffies + 100*HZ;   /* 10s */超时时间
timer.function = (void (*)(unsigned long)) timer_expired;
                                                处理函数
```

2，在自己的驱动程序中可以仿照着上面“定义-->初始化-->使用”

定时器的定义和触发时间：jiffies 相关概念。
在入口函数中使用：static int sixth_drv_init(void)：
初始化定时器：buttons_timer_function()。
设备定时器处理函数：buttons_timer.function = buttons_timer_function；
将定时器加到内核：add_timer(&buttons_timer);
当按下按键后--->到中断处理函数“buttons_irq()”中去。
中断处理函数以前是确实热键值，唤醒应用程序或者发信号等等操作。现在并没先做这些事情，而是为了防止按键抖动。加了定时器，让唤醒 APP 或发信号让定时器到达时间时的“定时器处理函数”中完成。而这里“中断处理函数”中先是修改定时器的超时时间为 10ms：Mod_timer(&buttons_timer, jiffies+HZ/100);
这里产生中断到到这个中断处理函数时，会来一个抖动就把时间基于当前时间值推后 10ms。（看上图），这样就把多个中断合并成了一个定时器处理。
接着“中断处理函数 buttons_irq()”就不再操作，返回：return IRQ_RETVAL(IRQ_HANDLED);

```
static struct timer_list buttons_timer; //定时器去抖:1,定义一个time_list结构的变量.

static int sixth_drv_init(void)
{
    /*定时器去抖:2,接着在入口函数中初始化buttons_times*/
    init_timer(&buttons_timer);
    /* 以下使用方法源于内核。
       timer.data = (unsigned long)SCpnt; 这个data是给下面的处理函数使用的。这时也先用不着。
       timer.expires = jiffies + 100*HZ; 10s 在中断里使用这个定时器故这里不管这句定时。
       timer.function = (void(*)(unsigned long))timer_expired (unsigned long p);
    */
    init_timer (&buttons_timer); //初始化buttons_timers结构。
    buttons_timer.function = buttons_timer_function; //定义处理函数.要自己写buttons_timer_function。
```

上面最后的处理函数我们自己定义的是“buttons_timer_function”，可以查看 timer_list 结构中 function 的定义原型：

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;

    void (*function)(unsigned long);
    unsigned long data;

    struct tvec_t_base_s *base;
#ifdef CONFIG_TIMER_STATS
    void *start_site;
    char start_comm[16];
    int start_pid;
#endif
};
```

根据定时器处理函数原型写一个定时器处理函数：

```
void buttons_timer_function(unsigned long data)
{//定时器去抖:定义定时器处理函数.

}
```

定时器有两要素：超时时间 和 定时器处理函数。上面时间还没有写，处理函数框架写出来了。

用“add_timer()”去使用这个时间，它是把“定时器”告诉内核，当定时器中的超时时间到了后，定时器处理函数“buttons_timer_function()”就会被调用。
在中断处理程序中启动修改定时器的超时时间，下面是以前的中断程序：

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    struct pin_desc * pindesc = (struct pin_desc *)dev_id;
    unsigned int pinval;

    pinval = s3c2410_gpio_getpin(pindesc->pin);

    if (pinval)
    {
        /* 松开 */
        key_val = 0x80 | pindesc->key_val;
    }
    else
    {
        /* 按下 */
        key_val = pindesc->key_val;
    }

    ev_press = 1;                   /* 表示中断发生了 */
    wake_up_interruptible(&button_waitq);   /* 唤醒休眠的进程 */

    kill_fasync (&button_async, SIGIO, POLL_IN);

    return IRQ_RETVAL(IRQ_HANDLED);
} ? end buttons_irq ?
```

用了定时器后，这部分代码交由定时器处理函数。这里只需要修改定时时间。

这个中断程序中只需要修改超时时间。Mod_timer();超时是指“闹钟”什么时间闹铃。基于“jiffles”这个值，这是一个全局变量，系统每隔 10ms,这个值就会产生一个系统时钟中断。

```
# cat /proc/interrupts
           CPU0
30:        3526          s3c  S3C2410 Timer Tick
32:           0          s3c  s3c2410-lcd
```

这是系统时钟中断，不断cat会有变化。

每隔 10ms 就有一个系统时钟中断。在系统时钟中断中这个“jiffies”值就会累加。

这里的“定时器”超时时间就是基于这个“jiffies”值。这里先定一个值，看“timer_list”结构中的定义：

```
struct timer_list {
    struct list_head entry;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
```

超时时间

```
# define HZ 100
```

```
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    /* 10ms后启动定时器 HZ是1秒，除以100就是10ms */
    add_timer(&buttons_timer,jiffies+HZ/100);
    return IRQ_RETVAL(IRQ_HANDLED);
}
```

超时时间可以设置成“当前值”加上某个值。如 1 秒就是 HZ，从定义中可以看到 HZ 是 100. 这里的意思是 1 秒钟里这个当前的 jiffies 值会增加 100.
add_timer(&buttons_timer,jiffies+HZ);是指当前 jiffies 时间过了 100 个系统时钟中断（系统滴答）后，这个定时器的超时时间“buttons_timer”就到达了。HZ 是 1 秒，那么定时器在 10ms 时启动的话，就是 HZ/100 即 10ms。
假设现在，jiffies 的值为“50”，HZ 为“100”，那么这个定时器 buttons_timer 的超时时间为：Buttons_times.expires = 50+100/100(jieeies+HZ/100) = 51.
系统是每隔 10ms 产生一个系统时钟中断，系统时钟中断中这个 jiffies 的值会累加。这里假设 jiffies 为 50，则下一个系统时钟时，jiffies 就变成 51 了，51 一到，在这个系统时钟中断处理函数里面会从这个定时器链表里面把这里的定时器找出来，看看哪个定时器的时间已经到了（buttons_timer 这个定时器就在这个链表中）。
若这里的 jiffies 已经大于等于这个“buttons_timer”的定时器超时时间“expires”时（jiffies>=buttons_timer.expirs）,就去调用与这个定时器相关的定时器处理函数。这里要是 buttons_timer.expirs 已经超时，就会调用上面自己定义的“buttons_timer_function（）”这个定时器处理函数。

当定时器超时时，定时器处理函数的工作：
首先“dev_id”要记录下来：
要定义一个结构体，static struct pid_desc *irq_pd;发生中断时的引脚描述。

```
static struct pid_desc *irq_pd; //发生中断时的引脚描述.
```

将它记录下来，在“buttons_irq()”中断处理函数中记录下来。

```
static struct pid_desc *irq_pd; //发生中断时的引脚描述.
//static atomic_t canopen = ATOMIC_INIT(1);     //定义原子变量并初始化为1

static DECLARE_MUTEX(button_lock);      //定义互斥锁

/*
 * 确定按键值
 */
static irqreturn_t buttons_irq(int irq, void *dev_id)
{
    /* 10ms后启动定时器 HZ是1秒，除以100就是10ms */
    irq_pd = (struct pin_desc *)dev_id;
    add_timer(&buttons_timer,jiffies+HZ/100);
    return IRQ_RETVAL(IRQ_HANDLED);
}
```

接着这个“irq_pd”就可以在“定时器”处理函数“buttons_timer_function()”中使用了：

```
void buttons_timer_function(unsigned long data)
{//定时器去抖:定义定时器处理函数.
    struct pin_desc * pindesc = irq_pd;
    unsigned int pinval;
```

这里要判断下，因为在入口函数中没有设置超时时间，这样超时时间就为 0，一旦 add_timer() 把这个定时器放到内核中去，那么在这个系统时钟中断里面，就会 jiffies >= 0 .这样就立即调用了定时器处理函数
buttons_timer_function()。但这个时间并没有按键中断产生。所以上面的“pindesc”要判断下。

```
void buttons_timer_function(unsigned long data)
{//定时器去抖:定义定时器处理函数.
    struct pin_desc * pindesc = irq_pd;
    unsigned int pinval;
    if(!pindesc) //若pindesc是空时直接return并不处理。
        return;
    //pindesc不为空时就接着下面的处理。
    pinval = s3c2410_gpio_getpin(pindesc->pin);//确定按键值。
```