专题12-按键驱动程序设计

一、混杂驱动设备模型

1.1、混杂设备描述

在Linux系统中，存在一类字符设备，它们拥有 相同的主设备号（10），但次设备号不同，我 们称这类设备为混杂设备(miscdevice)。所有的 混杂设备形成一个链表，对设备访问时内核根据 次设备号查找到相应的混杂设备。

Linux中使用struct miscdevice来描述一个混杂设备。

```
struct miscdevice {
  int minor; /* 次设备号*/
  const char *name; /* 设备名*/
  const struct file_operations *fops; /*文件操作*/
  struct list_head list;
  struct device *parent;
  struct device *this_device;
};
```

1.2、混杂设备注册

Linux中使用misc_register函数来注册一个混杂设 备驱动。

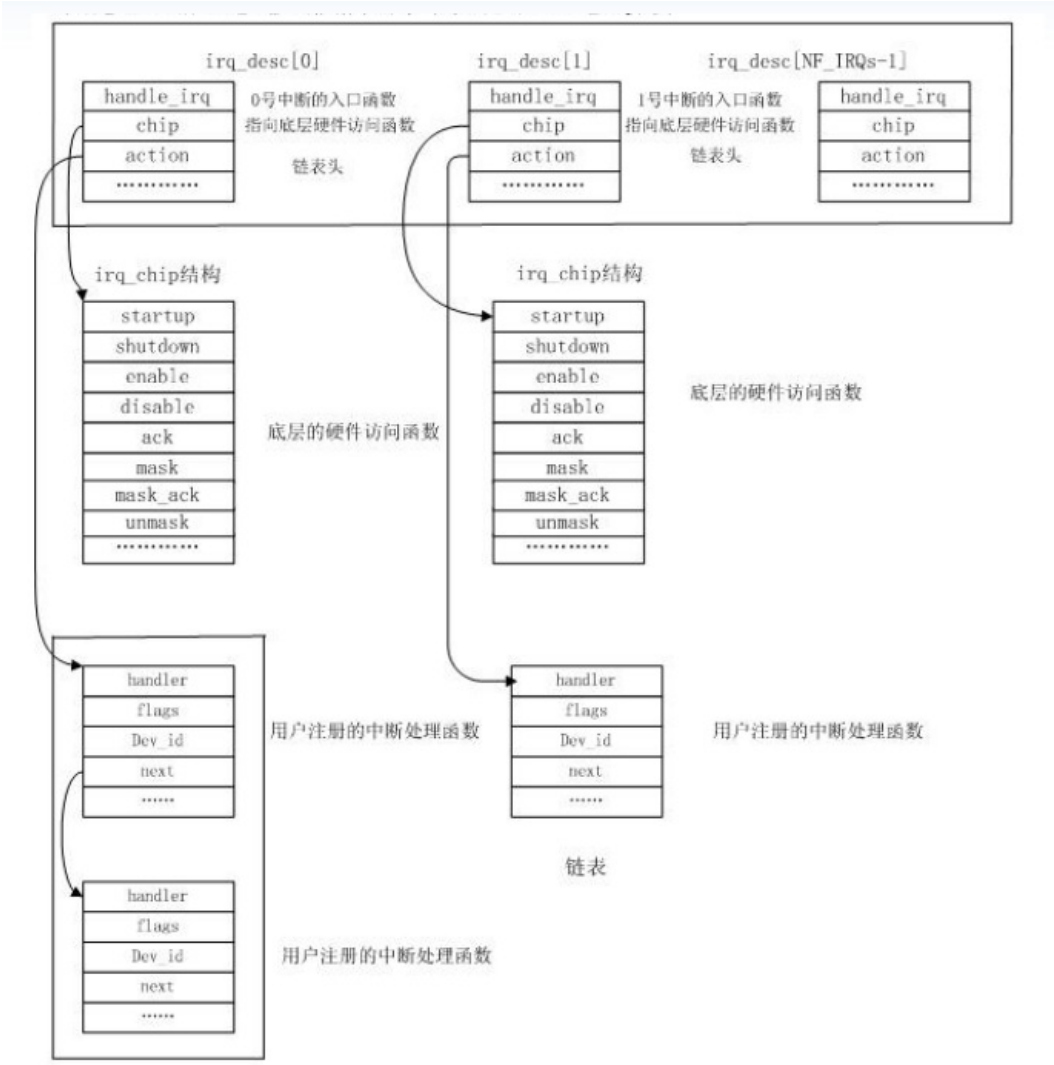
```
int misc_register(struct miscdevice * misc)
```

1.3、范例驱动分析

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

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


struct file_operations key_fops = {
 .open = key_open,
}


struct miscdevice key_miscdev = {
 .minor = 200,
 .name = "key",
 .fops = &key_fops,
}

static int key_init(void)
{
 misc_register(&key_miscdev);
}

static void key_exit(void)
{
 misc_deregister(&key_miscdev);
}

module_init(key_init);
module_exit(key_exit);
```

二、Linux中断处理

2.1、Linux中断处理流程分析



2.2、Linux中断处理程序设计

2.2.1、注册中断

request_irq函数用于注册中断。

int request_irq(unsigned int irq, void (*handler)(int, void*, struct pt_regs *), unsigned long flags, const char *devname, void *dev_id)

返回0表示成功，或者返回一个错误码

unsigned int irq

中断号。

void (*handler)(int,void *)

中断处理函数。

unsigned long flags

与中断管理有关的各种选项。

const char * devname

设备名

void *dev_id

共享中断时使用。

在flags参数中，可以选择一些与中断管理有关 的选项如:

• IRQF_DISABLED（SA_INTERRUPT）

如果设置该位，表示是一个“快速”中断处理程序； 如果没有设置这位，那么是一个“慢速”中断处理程序。

• IRQF_SHARED（SA_SHIRQ）

该位表明该中断号是多个设备共享的。

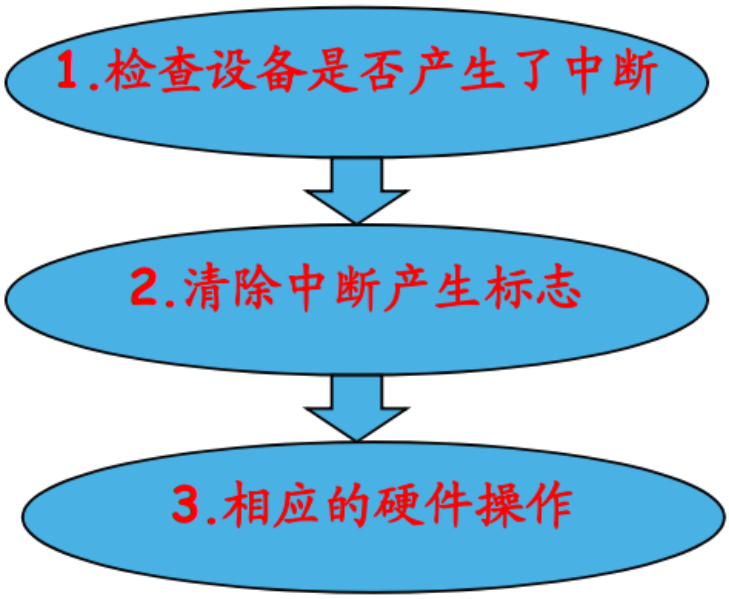
快/慢速中断的主要区别在于：快速中断保证中 断处理的原子性(不被打断)，而慢速中断则不保 证。换句话说，也就是“开启中断”标志位(处理 器IF)在运行快速中断处理程序时是关闭的，因 此在服务该中断时,不会被其他类型的中断打断; 而调用慢速中断处理时，其它类型的中断仍可以 得到服务。

2.2.2、中断处理

中断处理程序的特别之处是在中断上下文中运行的，它的行为受到某些限制:

1.不能使用可能引起阻塞的函数

2.不能使用可能引起调度的函数



2.2.3、注销处理

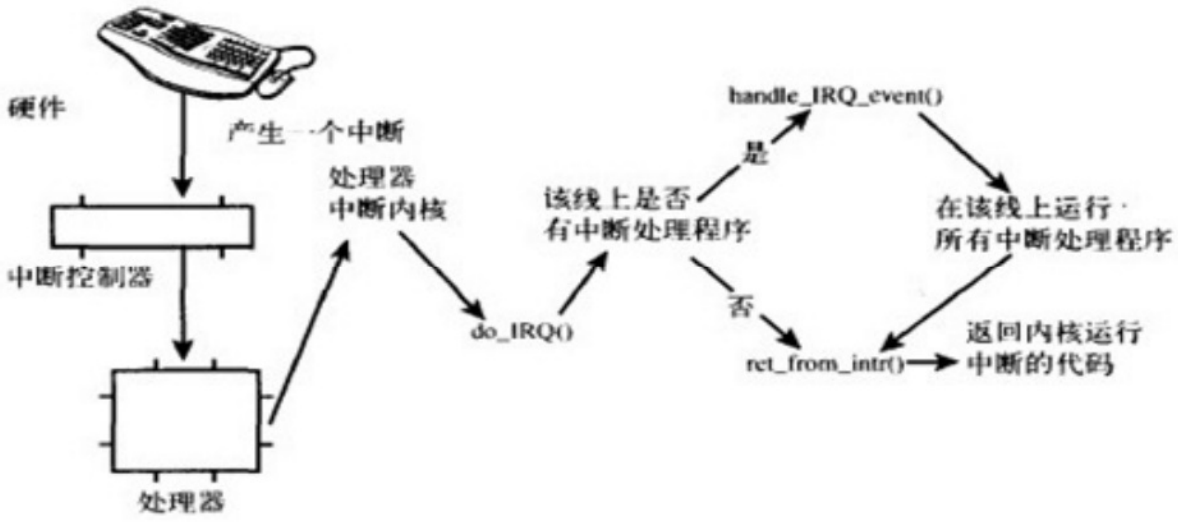
当设备不再需要使用中断时(通常在驱 动卸载时), 应当把它们注销, 使用函数:

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

三、按键驱动硬件操作实现

四、中断分层处理

4.1、中断嵌套



4.2、中断分层方式

上半部：当中断发生时，它进行相应地硬件读 写，并“登记”该中断。通常由中断处理程序充当上半部。

下半部：在系统空闲的时候对上半部“登记” 的中断进行后续处理。

4.2.1、软中断

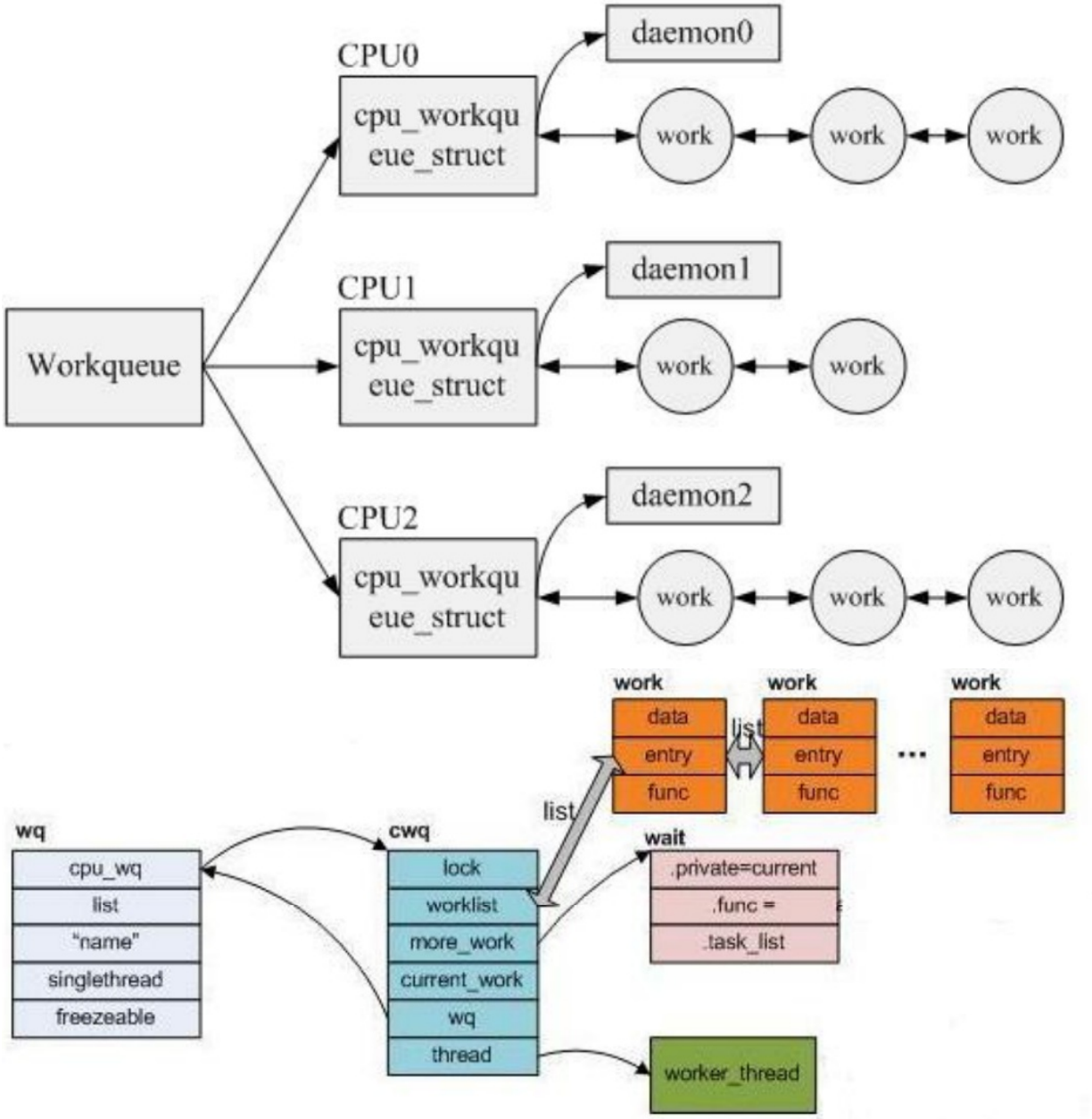
4.2.2、tasklet

4.2.3、工作队列

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/workqueue.h>
#include <linux/slab.h>

struct workqueue_struct * my_wq;
struct work_struct * work1;
struct work_struct * work2;
```

```
void work1_func(struct work_struct * work)
{
    printk(KERN_WARNING"this is work1->\n");
}

void work2_func(struct work_struct * work)
{
    printk(KERN_WARNING"this is work2->\n");
}

static int queue_init(void)
{
    /*creat workqueue*/
    my_wq = create_workqueue("my_wq");

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    /*queue work*/
    queue_work(my_wq, work1);

    /*init work*/
    work2 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work2, work2_func);

    /*queue work*/
    queue_work(my_wq, work2);

    return 0;
}

static void queue_exit(void)
{

}


module_init(queue_init);
module_exit(queue_exit);

MODULE_LICENSE("GPL");
```

工作队列是一种将任务推后执行的形式，他 把推后的任务交由一个内核线程去执行。这 样下半部会在进程上下文执行，它允许重新 调度甚至睡眠。 每个被推后的任务叫做“工 作” ，由这些工作组成的队列称为工作队列。



Linux内核使用struct work_struct来描述一个工作队列：

```
struct workqueue_struct {
    struct cpu_workqueue_struct *cpu_wq;
    struct list_head list;
    const char *name; /*workqueue name*/
    int singlethread;
    int freezeable; /* Freeze threads during suspend */
    int rt;
};
```

Linux内核使用struct work_struct来描述一个工作项：

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;
};
typedef void (*work_func_t)(struct work_struct *work);
```

step1. 创建工作队列

```
create_workqueue
```

step2. 创建工作

```
INIT_WORK
```

step3. 提交工作

```
queue_work
```

4.3、使用工作队列实现分层

在大多数情况下, 驱动并不需要自己建立工作 队列，只需定义工作, 然后将工作提交到内核已 经定义好的工作队列keventd_wq中。

1. 提交工作到默认队列

schedule_work

key.c:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <linux/io.h>
#include <linux/workqueue.h>
#include <linux/slab.h>

#define GPNCON    0x7f008830

struct work_struct * work1;

void work1_func(struct work_struct * work)
{
    printk(KERN_WARNING"key down!\n");
}

static irqreturn_t key_int(int irq, void *dev_id)
{
    /*Check if a key interrupt has occurred */

    /*Clear key interrupts that have occurred(If it is a CPU internal interrupt (non-peripheral), the system will help clear) */

    /*Submit the bottom half */
        /*queue work*/
    schedule_work(work1);

    return 0;
}

void key_hw_init(void)
{
    unsigned int data;
    unsigned int * gpio_config;

    gpio_config = ioremap(GPNCON, 4);
    data = readl(gpio_config);
    data &= ~0b11;                  //set key1
    data |= 0b10;
    writel(data, gpio_config);
}

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


struct file_operations key_fops = {
    .open = key_open,
};


struct miscdevice key_miscdev = {
    .minor = 200,
    .name = "key",
    .fops = &key_fops,
};

static int key_init(void)
{
    printk(KERN_WARNING"key init\n");
```

```
    misc_register(&key_miscdev);

    request_irq(S3C_EINT(0), key_int, IRQF_TRIGGER_FALLING, "key", 0);

    key_hw_init();

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    return 0;
}

static void key_exit(void)
{
    printk(KERN_WARNING"key exit\n");

    free_irq(S3C_EINT(0), 0);

    misc_deregister(&key_miscdev);
}


module_init(key_init);
module_exit(key_exit);
MODULE_LICENSE("GPL");
```

Makefile:

```
obj-m := key.o

KDIR := /home/S5-driver/lesson7/linux-ok6410

all:
    make -C $(KDIR) M=$(PWD) modules CROSS_COMPILE=arm-linux- ARCH=arm

clean:
    rm -f *.order *.symvers *.mod.o *.o *.ko *.mod.c
```

五、按键定时器去抖

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <linux/io.h>
#include <linux/workqueue.h>
#include <linux/slab.h>
#include <linux/timer.h>

#define GPNCON    0x7f008830
#define GPNDAT    0x7f008834

struct work_struct * work1;

struct timer_list key_timer;

unsigned int * gpio_data;

void work1_func(struct work_struct * work)
{
    mod_timer(&key_timer, jiffies + HZ/10);
}

void key_timer_func(unsigned long data)
{
    unsigned int key_val;
```

```
    key_val = readl(gpio_data) & 0x01;

    if (key_val == 0) {
        printk(KERN_WARNING"key down!\n");
    }
}

static irqreturn_t key_int(int irq, void *dev_id)
{
    /*Check if a key interrupt has occurred */

    /*Clear key interrupts that have occurred(If it is a CPU internal interrupt (non-peripheral), the system will help clear) */

    /*Submit the bottom half */
        /*queue work*/
    schedule_work(work1);

    return 0;
}

void key_hw_init(void)
{
    unsigned int data;
    unsigned int * gpio_config;

    gpio_config = ioremap(GPNCON, 4);
    data = readl(gpio_config);
    data &= ~0b11;                  //set key1
    data |= 0b10;
    writel(data, gpio_config);

    gpio_data = ioremap(GPNDAT, 4);
}

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


struct file_operations key_fops = {
    .open = key_open,
};


struct miscdevice key_miscdev = {
    .minor = 200,
    .name = "key",
    .fops = &key_fops,
};

static int key_init(void)
{
    printk(KERN_WARNING"key init\n");

    misc_register(&key_miscdev);

    request_irq(S3C_EINT(0), key_int, IRQF_TRIGGER_FALLING, "key", 0);

    key_hw_init();

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    /*init timer */
```

```
    init_timer(&key_timer);
    key_timer.function = key_timer_func;

    /*register timer */
    add_timer(&key_timer);

    return 0;
}

static void key_exit(void)
{
    printk(KERN_WARNING"key exit\n");

    free_irq(S3C_EINT(0), 0);

    misc_deregister(&key_miscdev);
}


module_init(key_init);
module_exit(key_exit);
MODULE_LICENSE("GPL");
```

5.1、按键抖动

按键所用开关为机械弹性开关，当机械触点 断开、闭合时，由于机械触点的弹性作用， 开关不会马上稳定地接通或断开。因而在闭合及断开的瞬间总是伴随有一连串的抖动。

5.2、按键消抖的方法

按键去抖动的方法主要有二种，一种是硬件 电路去抖动;另一种就是软件延时去抖。而延 时又一般分为二种，一种是for循环等待，另 一种是定时器延时。在操作系统中，由于效 率方面的原因，一般不允许使用for循环来等 待，只能使用定制器。

5.3、内核定时器

Linux内核使用struct timer_list来描述一个定 时器：

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

5.4、定时器使用流程

5.4.1、定义定时器变量

5.4.2、初始化定时器

5.4.2.1、init_timer初始化

5.4.2.1、设置超时函数

5.4.3、add_timer注册定时器

5.4.4、mod_timer启动定时器

5.5、Linux定时器理解内容

在linux内核里，有一个叫jiffies的变量(定义在linux/jiffies)记录了自系统启动以来产生的节拍的总数。启动时，内核将该变量初始化为0，此后每次时钟中断处理程序都会增加该变量的值。因为一秒内时钟中断的次数等于HZ,所以jiffies一秒内增加的值也就为HZ.系统运行时间以秒为单位计算，就等于jiffes/HZ.它作为在计算机表示的变量，就总存在大小，当这个变量增加到超出它的表示上限时，就要回绕到0.这个回绕看起来很简单，但实际上还是给我们编程造成了很大的麻烦，比如边界条件判断时。幸好，内核提供了四个宏来帮助比较节拍计数，这些宏定义在linux/jiffies.h可以很好的处理节拍回绕的情况。

Linux是使用一个硬件定时器模拟成一个超级定时器，可以产生多个软中断。

六、驱动支持多按键优化

key.c:

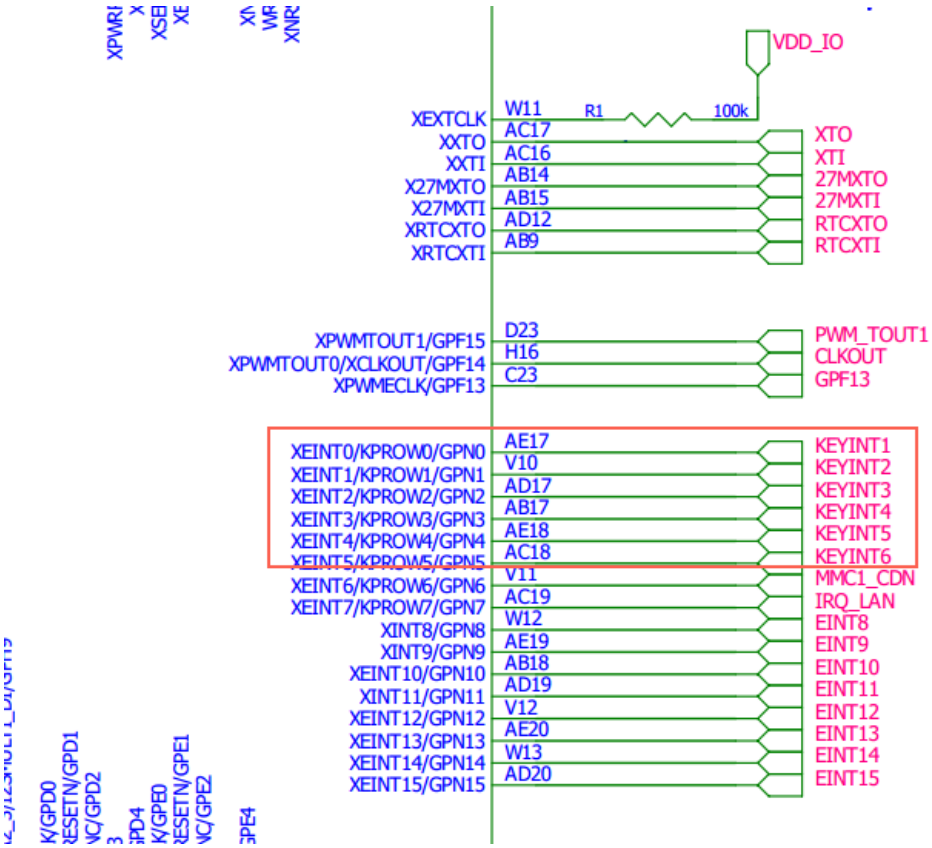
```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <linux/io.h>
#include <linux/workqueue.h>
#include <linux/slab.h>
#include <linux/timer.h>
#include <linux/uaccess.h>

#define GPNCON    0x7f008830
#define GPNDAT    0x7f008834

struct work_struct * work1;

struct timer_list key_timer;

unsigned int * gpio_data;
unsigned int key_num;

void work1_func(struct work_struct * work)
{
    mod_timer(&key_timer, jiffies + HZ/10);
}

void key_timer_func(unsigned long data)
{
    unsigned int key_val;

    key_val = readl(gpio_data) & 0b11;

    if (key_val == 0b10) {
        key_num = 1;
    }

    if (key_val == 0b01) {
        key_num = 2;
    }
}

static irqreturn_t key_int(int irq, void *dev_id)
{
    /*Check if a key interrupt has occurred */

    /*Clear key interrupts that have occurred(If it is a CPU internal interrupt (non-peripheral), the system will help clear) */

    /*Submit the bottom half */
        /*queue work*/
    schedule_work(work1);

    return 0;
}

void key_hw_init(void)
{
    unsigned int data;
    unsigned int * gpio_config;

    gpio_config = ioremap(GPNCON, 4);
    data = readl(gpio_config);
    data &= ~0b1111;                    //set key1 and key2
    data |= 0b1010;
    writel(data, gpio_config);

    gpio_data = ioremap(GPNDAT, 4);
}
```

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

ssize_t key_read (struct file * filp, char __user * buf, size_t size, loff_t * pos)
{
    copy_to_user(buf, &key_num, 4);

    return 4;
}

ssize_t key_write (struct file * filp, const char __user * buf, size_t size, loff_t * pos)
{
    return 0;
}

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

struct file_operations key_fops = {
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release = key_close,
};


struct miscdevice key_miscdev = {
    .minor = 200,
    .name = "key",
    .fops = &key_fops,
};

static int key_init(void)
{
    printk(KERN_WARNING"key init\n");

    misc_register(&key_miscdev);

    request_irq(S3C_EINT(0), key_int, IRQF_TRIGGER_FALLING, "key", 0);
    request_irq(S3C_EINT(1), key_int, IRQF_TRIGGER_FALLING, "key", 0);

    key_hw_init();

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    /*init timer */
    init_timer(&key_timer);
    key_timer.function = key_timer_func;

    /*register timer */
    add_timer(&key_timer);

    return 0;
}

static void key_exit(void)
{
    printk(KERN_WARNING"key exit\n");

    free_irq(S3C_EINT(0), 0);
```

```
    misc_deregister(&key_miscdev);
}


module_init(key_init);
module_exit(key_exit);
MODULE_LICENSE("GPL");
```

key_app.c:

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
 int fd;
 int key_num;

 /*open device*/
 fd = open("/dev/ok6410key", 0);

 if (fd < 0)
  printf("open device fail!\n");

 /*read device*/
 read(fd, &key_num, 4);
 printf("key is %d\n", key_num);

 /*close device*/
 close(fd);
}
```



七、阻塞型驱动设计

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/miscdevice.h>
#include <linux/interrupt.h>
#include <linux/fs.h>
#include <linux/io.h>
```

```c
#include <linux/workqueue.h>
#include <linux/slab.h>
#include <linux/timer.h>
#include <linux/uaccess.h>
#include <linux/wait.h>
#include <linux/sched.h>

#define GPNCON    0x7f008830
#define GPNDAT    0x7f008834

struct work_struct * work1;

struct timer_list key_timer;

unsigned int * gpio_data;
unsigned int key_num = 0;

wait_queue_head_t key_q;

void work1_func(struct work_struct * work)
{
    mod_timer(&key_timer, jiffies + HZ/10);
}

void key_timer_func(unsigned long data)
{
    unsigned int key_val;

    key_val = readl(gpio_data) & 0b11;

    if (key_val == 0b10) {
        key_num = 1;
    }

    if (key_val == 0b01) {
        key_num = 2;
    }

    wake_up(&key_q);
}

static irqreturn_t key_int(int irq, void *dev_id)
{
    /*Check if a key interrupt has occurred */

    /*Clear key interrupts that have occurred(If it is a CPU internal interrupt (non-peripheral), the system will help clear) */

    /*Submit the bottom half */
        /*queue work*/
    schedule_work(work1);

    return 0;
}

void key_hw_init(void)
{
    unsigned int data;
    unsigned int * gpio_config;

    gpio_config = ioremap(GPNCON, 4);
    data = readl(gpio_config);
    data &= ~0b1111;                    //set key1 and key2
    data |= 0b1010;
    writel(data, gpio_config);

    gpio_data = ioremap(GPNDAT, 4);
}
```

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

ssize_t key_read (struct file * filp, char __user * buf, size_t size, loff_t * pos)
{
    wait_event(key_q, key_num);

    copy_to_user(buf, &key_num, 4);

    key_num = 0;

    return 4;
}

ssize_t key_write (struct file * filp, const char __user * buf, size_t size, loff_t * pos)
{
    return 0;
}

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

struct file_operations key_fops = {
    .open = key_open,
    .read = key_read,
    .write = key_write,
    .release = key_close,
};


struct miscdevice key_miscdev = {
    .minor = 200,
    .name = "key",
    .fops = &key_fops,
};

static int __init ok6410_key_init(void)
{
    printk(KERN_WARNING"key init\n");

    misc_register(&key_miscdev);

    request_irq(S3C_EINT(0), key_int, IRQF_TRIGGER_FALLING, "key", 0);
    request_irq(S3C_EINT(1), key_int, IRQF_TRIGGER_FALLING, "key", 0);

    key_hw_init();

    /*init work*/
    work1 = kmalloc(sizeof(struct work_struct), GFP_KERNEL);

    INIT_WORK(work1, work1_func);

    /*init timer */
    init_timer(&key_timer);
    key_timer.function = key_timer_func;

    /*register timer */
    add_timer(&key_timer);

    /*init wait queue*/
    init_waitqueue_head(&key_q);

    return 0;
}
```

```
static void __exit ok6410_key_exit(void)
{
    printk(KERN_WARNING"key exit\n");

    free_irq(S3C_EINT(0), 0);

    misc_deregister(&key_miscdev);
}


module_init(ok6410_key_init);
module_exit(ok6410_key_exit);
MODULE_LICENSE("GPL");
```

7.1、阻塞必要性

当一个设备无法立刻满足用户的读写请 求时应当如何处理？ 例如：调用read 时，设备没有数据提供 但以后可能会 有；或者一个进程试图向设备写入数据， 但是设备暂时没有准备好接收数据。当上述情况发生的时候，驱动程序应当 (缺省地)阻塞进程，使它进入等待（睡眠）状态，直到请求可以得到满足。

7.2、内核等待队列0

在实现阻塞驱动的过程中，也 需要有一个“候车室”来安排 被阻塞的进程“休息” ，当唤 醒它们的条件成熟时，则可以 从“候车室”中将这些进程唤 醒。而这个“候车室”就是等 待队列。

1、定义等待队列

wait_queue_head_t my_queue

2、初始化等待队列

init_waitqueue_head(&my_queue)

3、定义+初始化等待队列

DECLARE_WAIT_QUEUE_HEAD(my_queue)

4、进入等待队列，睡眠

wait_event(queue,condition)

当condition(布尔表达式)为真时，立即返回；否则让进程 进入TASK_UNINTERRUPTIBLE模式的睡眠，并挂在queue参 数所指定的等待队列上。

wait_event_interruptible(queue,condition)

当condition(布尔表达式)为真时，立即返回；否则让 进程进入TASK_INTERRUPTIBLE的睡眠，并挂在queue参数 所指定的等待队列上。

int wait_event_killable(queue, condition)

当condition(一个布尔表达式)为真时，立即返回； 否则让进程进入TASK_KILLABLE的睡眠，并挂 在queue参数所指定的等待队列上。

5、从等待队列中唤醒进程

wake_up(wait_queue_t *q)

从等待队列q中唤醒状态为TASK_UNINTERRUPTIBLE， TASK_INTERRUPTIBLE，TASK_KILLABLE 的所有进程。

wake_up_interruptible(wait_queue_t *q)

从等待队列q中唤醒状态为TASK_INTERRUPTIBLE 的进程

7.3、阻塞驱动优化

驱动程序通常需要提供这样的能力：当应用程序进行read(),write()等系统调用时，若设备的资源不能获取，而用户又希望已阻塞的方式访问设备，驱动程序应在设备的xxx_read(),xxx_writ()等操作过程中将进程阻塞直到资源可以获取，以后，应用程序的read(),write()等调用才能返回，整个过程仍然进行了正确的设备访问，用户并没有感值到。