# 内核驱动学习

**中断的实现：（**[**http://blog.csdn.net/w282529350/article/details/7359569**](http://blog.csdn.net/w282529350/article/details/7359569)**）**

中断分两种：

1）中断，又叫外部中断或异步中断，它的产生是由于外设向处理器发出中断请求。其中外部中断也有两种，这是由配置寄存器设定的：普通中断请求（IRQ）和快速中断请求（FIQ）。一般地，linux下很少使用快速中断请求。

2）异常，又叫内部中断或同步中断，它的产生是由于处理器执行指令出错。

在以下的内容我是要介绍由于外部设备产生的中断。

这里我还有两个名词要说清楚

1）中断请求线：在后面也叫中断号，每个中断都会通过一个唯一的数值来标识，而这个值就称做中断请求线

2）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。具体可以查看芯片手册。

二、什么是中断处理函数

在相应一个中断是，内核会执行该信号对应的一个函数，该函数就叫做该中断对应的中断处理函数。一般来说，中断的优先级是最高的，一但接收到中断，内核就会调用对应的中断处理函数。

中断处理函数运行在中断上下文中。中断上下文与内核上下文有一点区别：

内核上下文是指应用层调用系统调用陷入内核执行，内核代表陷入的进程执行操作。函数中可以通过current查看当前进程（即应用层的进程）的信息，并且可以睡眠。

中断上下文中，不能通过current查看调用它的应用层进程的信息，同时，处于中断上下文时，不能睡眠。

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

一、从硬件角度看中断

中断的产生到处理器获得中断这段过程中，还要通过中断处理器来筛选信号。

先温习一下S3C2440芯片手册的知识：中断是如何产生的，中断处理器本身如何处理中断。先看一下一幅经典的图，这是介绍中断控制器的工作流程：

从硬件上的分类，有两种不同的中断类型：

1）自己占有SORCPND寄存器的一位（without sub-register）。

2）几个中断共同享用SRCPND寄存器的一位（with sub-register）。

其实两种都差不多，只是多了两步的检测。我以自己占用一位的中断来举例，如EINT1，在我的开发板，EINT1上接了一个按键。

1）当我按下按键产生电平变化，传到S3C2440的中断控制器上（即将要进入上面图的流程图）。

2）首先，信号要经过寄存器SRCPND，SRCPND是用来配置当前的处理器要接收什么中断，如果该寄存器配置成接收EINT1中断（对应位置一），则允许继续下一步。

3）然后，信号经过寄存器MASK，这是用来设置当前系统需要屏蔽的中断。注意，这里的屏蔽跟上一个寄存器的不接收中断是不一样的。这里的屏蔽是指，中断是接受了，但是由于某种原因，先暂时不屏蔽产生的中断。

4）通过INTPND寄存器，查看当前是否有相同的中断已经被请求（如果是，INTPND对应位置一）。

5）如果没有相同的中断在请求，中断处理器才会把这个信号传给处理器，这时处理器才会知道有EINT0的中断真正来了，要对信号进行处理了。

注：如果设定了EINT0是快速中断模式（FIQ），中断通过SRCPND寄存器后就会通过MODE寄存器的判断，确定是FIQ后，中断控制器优先将该中断传给CPU处理。

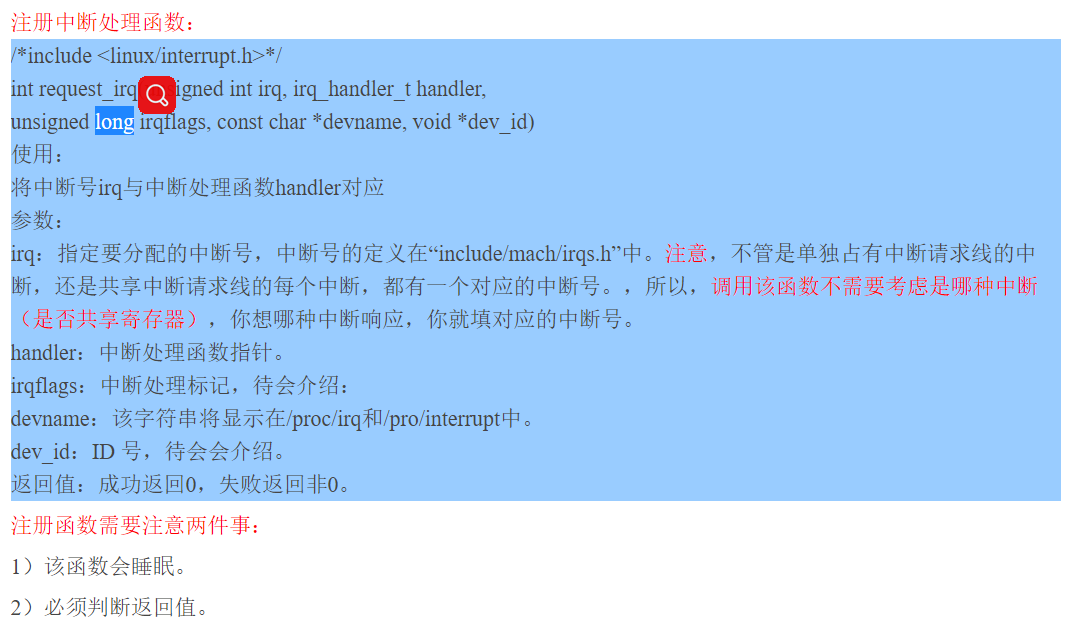
6）对应传来的中断类型（IRQ或FIQ），通过CPSR寄存器切换到对应的工作模式（ARM有七种工作模式）。

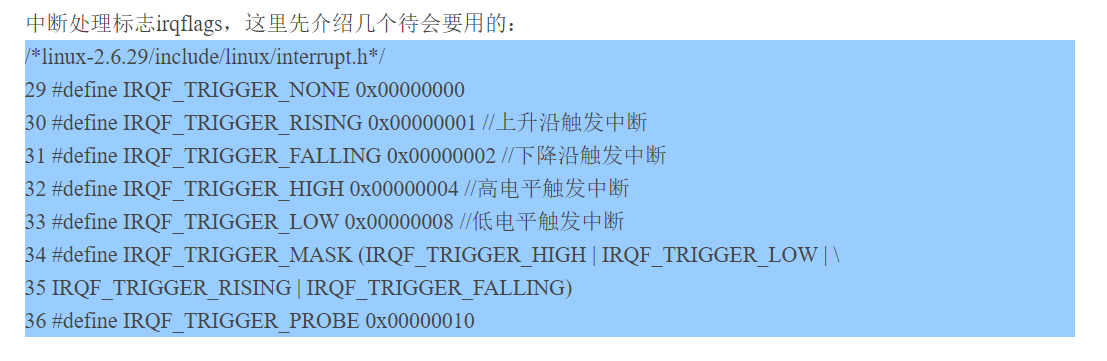
7）切换工作模式后，进入指定的中断处理入口执行中断处理函数。

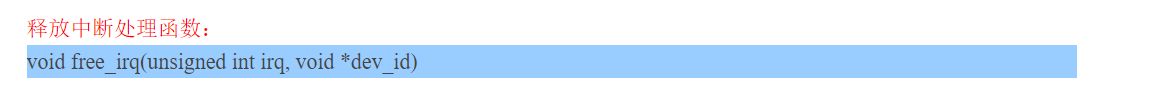
注意：第6、7步在linux下的实现相对复杂，不像在裸板程序，只需要切换一下工作模式，执行相应的函数就可以了。迟点会介绍linux如何实现。

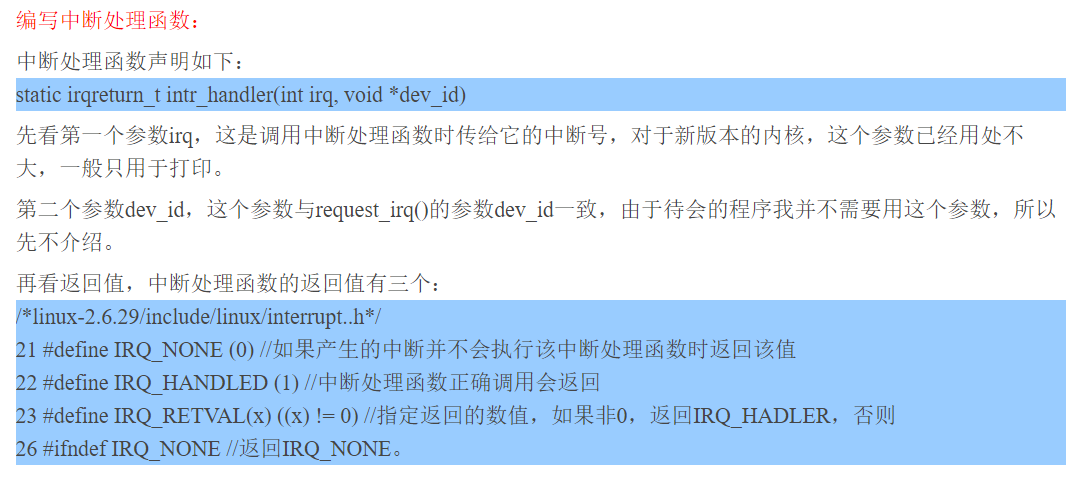
来个流程图比较只在，同时来个类比，处理器是老板，中断处理器是小秘：

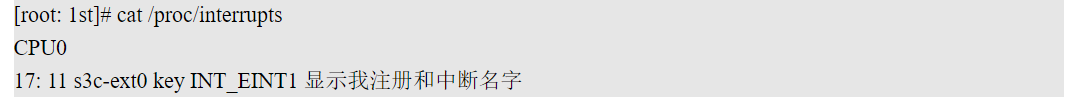
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx











第一列：中断号。

第二列：“11”对应处理器响应中断的次数。

第三列：“s3c-ext0”处理这个中断的中断控制器。

第四列：自定义的中断名字

**输入子系统：（learnaddr：http://blog.csdn.net/ielife/article/details/7798952）**

linux输入子系统（linux input subsystem）从上到下由三层实现，分别为：

1. 输入子系统事件处理层（EventHandler）。

则是用户编程的接口（设备节点），并处理驱动层提交的数据处理。

2.输入子系统核心层（InputCore）。

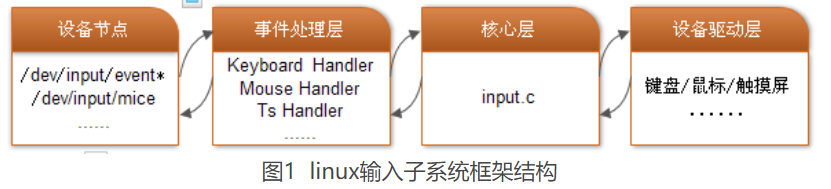
为设备驱动层提供了规范和接口。设备驱动层只要关心如何驱动硬件

并获得硬件数据（例如按下的按键数据），然后调用核心层提供的接口，核心层会自动把数据提交给事件处理层。

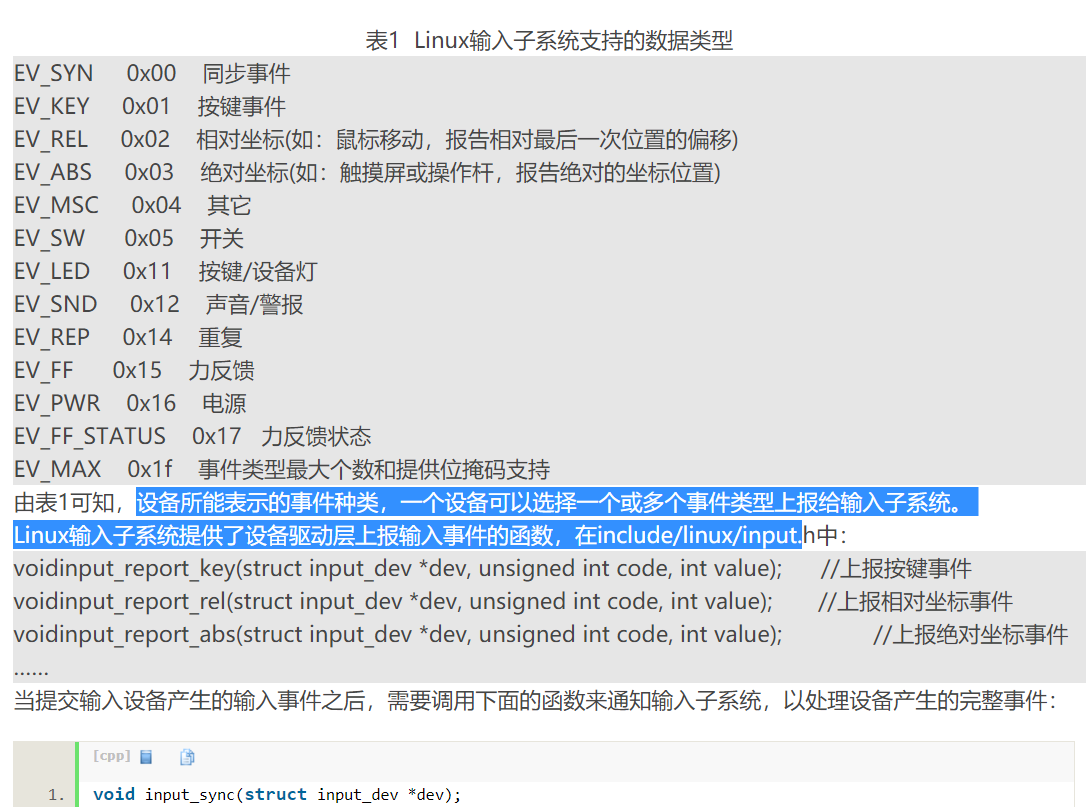
3.输入子系统设备驱动层。

主要实现对硬件设备的读写访问，中断设置，并把硬件产生的事件转换为

核心层定义的规范提交给事件处理层。



/dev/input目录下显示的是已经注册在内核中的设备编程接口



struct input\_dev {

const char \*name;

const char \*phys;

const char \*uniq;

struct input\_id id;

unsigned long propbit[BITS\_TO\_LONGS(INPUT\_PROP\_CNT)];

unsigned long evbit[BITS\_TO\_LONGS(EV\_CNT)];

unsigned long keybit[BITS\_TO\_LONGS(KEY\_CNT)];

unsigned long relbit[BITS\_TO\_LONGS(REL\_CNT)];

unsigned long absbit[BITS\_TO\_LONGS(ABS\_CNT)];

unsigned long mscbit[BITS\_TO\_LONGS(MSC\_CNT)];

unsigned long ledbit[BITS\_TO\_LONGS(LED\_CNT)];

unsigned long sndbit[BITS\_TO\_LONGS(SND\_CNT)];

unsigned long ffbit[BITS\_TO\_LONGS(FF\_CNT)];

unsigned long swbit[BITS\_TO\_LONGS(SW\_CNT)];

unsigned int hint\_events\_per\_packet;

unsigned int keycodemax;

unsigned int keycodesize;

void \*keycode;

int (\*setkeycode)(struct input\_dev \*dev,

const struct input\_keymap\_entry \*ke,

unsigned int \*old\_keycode);

int (\*getkeycode)(struct input\_dev \*dev,

struct input\_keymap\_entry \*ke);

struct ff\_device \*ff;

unsigned int repeat\_key;

struct timer\_list timer;

int rep[REP\_CNT];

struct input\_mt\_slot \*mt;

int mtsize;

int slot;

int trkid;

struct input\_absinfo \*absinfo;

unsigned long key[BITS\_TO\_LONGS(KEY\_CNT)];

unsigned long led[BITS\_TO\_LONGS(LED\_CNT)];

unsigned long snd[BITS\_TO\_LONGS(SND\_CNT)];

unsigned long sw[BITS\_TO\_LONGS(SW\_CNT)];

int (\*open)(struct input\_dev \*dev);

void (\*close)(struct input\_dev \*dev);

int (\*flush)(struct input\_dev \*dev, struct file \*file);

int (\*event)(struct input\_dev \*dev, unsigned int type, unsigned int code, int value);

struct input\_handle \_\_rcu \*grab;

spinlock\_t event\_lock;

struct mutex mutex;

unsigned int users;

bool going\_away;

bool sync;

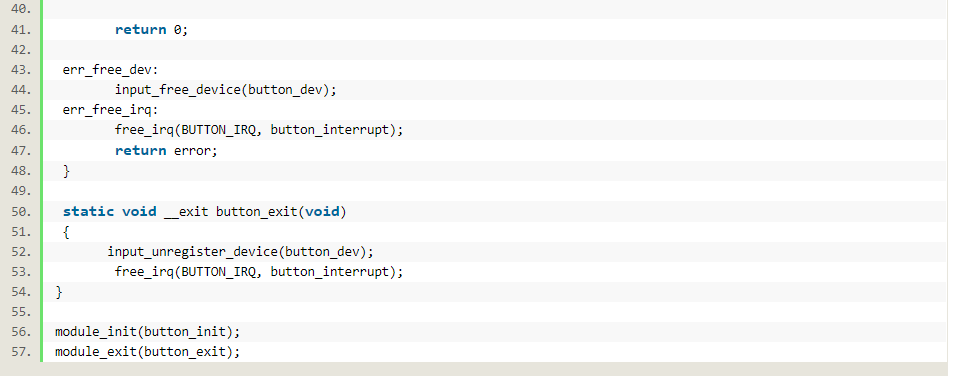
struct device dev;

struct list\_head h\_list;

struct list\_head node;

};





编写基于输入子系统的设备驱动程序需要包含<linux/input.h>，因为它包含了输入子系统的接口和所有的宏定义，这些内容在编写输入设备驱动程序时需要用到。

Button\_init函数说明：

当模块加载（insmod）或内核引导过程中，button\_init函数被调用。首先做的工作是能过获取正确的控制硬件设备的硬件资源（例如内存、IO内存、中断和DMA），在代码BUTTON\_IRQ作为BUTTON设备的中断资源、通过request\_irq()函数被申请注册。当有按键按下/释放时，调用button\_interrupt(),中断处理函数获取按键值BUTTON\_PORT(BUTTON设备的I/O资源)。

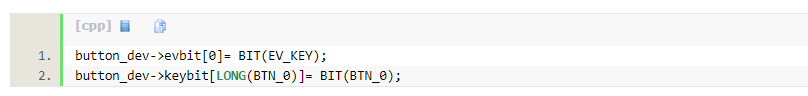
那么输入子系统怎么能够知道这个设备为输入设备呢？通过第8行为设备定义一个用于描述一个输入设备对象。

  
定义了button\_dev之后，如何通知输入子系统有新的输入设备了呢？或者说如何把一个新的输入设备加入到输入子系统中呢？可以通过输入子系统核心层input.c中提供的函数分配一个输入设备，在代码的第25行。

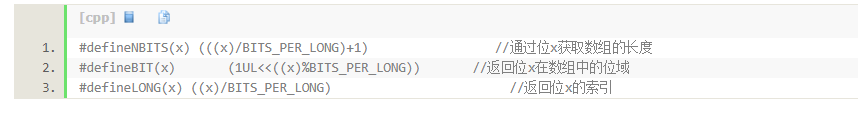
struct input\_dev \*input\_allocate\_device(void);

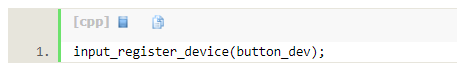


有了输入设备的描述，当事件产生时，输入子系统怎么能够知道设备产生的事件类型呢？通过32和33行的代码。



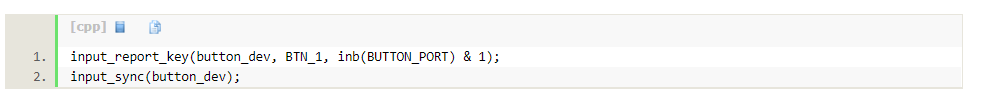
其中evbit和keybit成员分别代表设备产生的事件类型和上报的按键值。其中输入子系统的一些位操作NBITS、BIT、LONG经常被用到：



以上的工作做完之后，即可注册为输入设备了，代码的35行。

这个函数把button\_dev输入设备挂入输入设备链表中，并且通知事件处理层调用connect函数完成设备和事件处理的绑定，当用户打开设备时，便能够调用到相应的事件处理接口获得硬件上报的数据了。input\_register\_device（）函数是会睡眠的函数，因此不能够在中断上下文和持有自旋锁的代码中调用。

当我们把上面的工作做完之后，设备驱动中唯一值得关注的就是button\_interrupt（）中断处理函数了。当按键动作发生，button\_interrupt（）函数被调用，完成事件的上报由其中的两条语句完成。



其中input\_report\_key上报了这是一个按键事件，且它的值为inb(BUTTON\_PORT) & 1，由于案例代码只产生一个按键的值，因此input\_sync（）在这里不起关键作用。但如果是一个触摸屏，即有x坐标和y坐标，则需要通过input\_sync（）函数把x和y坐标完整地传递给输入子系统。

Device\_create（）详解：

在驱动程序中初始化入口函数，向内核注册一个设备后，往往要注册一个类

#define MYBUTTONNAME "iButtonDev"

static int major = 248,minor = 0;

static dev\_t devno;

static struct class \*cls;

static struct device \*test\_device;

devno = MKDEV(major,minor);

ret = register\_chrdev(major,MYBUTTONNAME,&iButtonOpr); //注册设备 ^M

cls = class\_create(THIS\_MODULE, "myclass"); //创建设备类

if(IS\_ERR(cls)) {

unregister\_chrdev(major,MYBUTTONNAME);

return -EBUSY;

}

test\_device = device\_create(cls,NULL,devno,NULL,MYBUTTONNAME);//创建设备节点 mknod /dev/hello

if(IS\_ERR(test\_device)){

class\_destroy(cls);

unregister\_chrdev(major,MYBUTTONNAME);

return -EBUSY;

}

**字符设备驱动程序之异步通知：**

驱动 应用

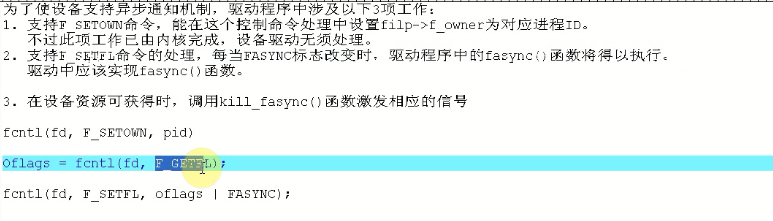
提醒 异步通知 signal

1. 注册一个信号处理函数
2. 谁发
3. 发给谁
4. 怎么发

目标：按下按键时

驱动通知应用程序

1. 应用程序注册信号处理函数
2. 谁发 ->驱动
3. 发给谁 -> app =>驱动PID
4. 怎么发：kill\_fasync



**字符设备驱动程序之同步互斥组赛：**

目的：同一时刻只允许一个进程访问该变量

Int canopen = 1;

DriverOpen:

If(--canopen != 0){

有Bug，多任务不成立。

Canopen++;

Return -EBUSY;

}

DriverClose

Canopen++;

**原子操作：**

原子操作指的时在执行的过程中不会被别的代码路径所中断的操作。

常用的原子操作函数举例：

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\_test(atomic\_t \*v); //自减操作后测试其是否为0，为0返回

True,否则返回false

**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。当获取不到信号粮食，进程进入休眠状态。

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

获取信号量：

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);

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，知道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，知道等待的条件被满足。

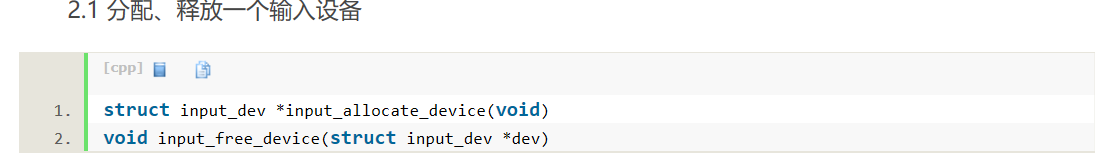
非阻塞：

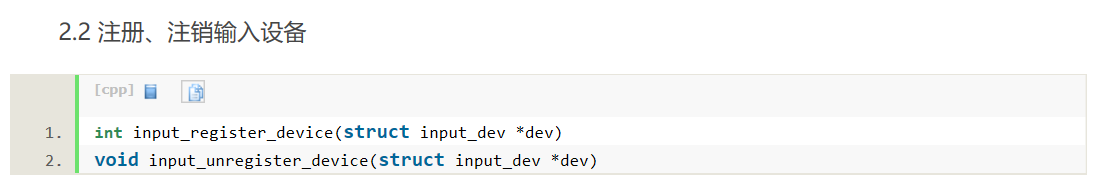
进程在不能进行设备操作时并不挂起，它或者放弃，或不停的查询，直至可以进行操作为止。

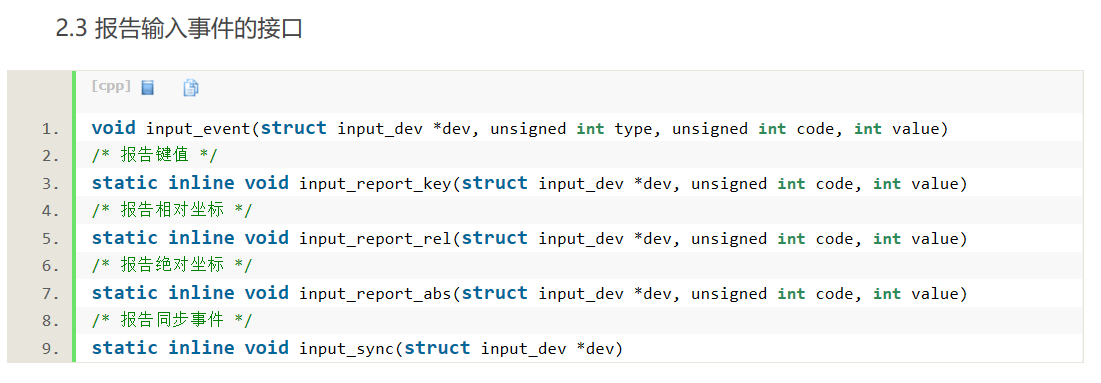
Fd = open(“…”,O\_RDWR | O\_NONBLOCK);

**输入子系统：**

核心函数在内核的位置：drivers\input\input.c 和 include\linux\input.h







**3、输入子系统驱动的一般编写步骤**  
           a、分配一个input\_dev结构体  
           b、设置input\_dev结构体  ：  支持哪类事件， 支持该类事件中的那些事件  
           c、注册input\_dev结构体  
           d、硬件相关的操作  ： 中断申请，定时器的设置等

static struct input\_dev \*button\_dev; /\*输入设备结构体\*/

button\_dev = input\_allocate\_device(); /\*分配一个设备结构体\*/

button\_dev->evbit[0] = BIT(EV\_KEY); /\*能产生哪类事件

\_\_set\_bit(KEY\_2, button\_dev->keybit); /\*能产生哪些事件

\_\_set\_bit(KEY\_1, button\_dev->keybit);

ret = input\_register\_device(button\_dev); /\*注册一个输入设备\*/

/\*KeyTmp->Event KEY\_1 KEY\_2 事件产生\*/

input\_event(button\_dev, EV\_KEY, KeyTmp->Event,dome); /\*上报事件

input\_sync(button\_dev); /\*通知接收者,一个报告发送完毕\*/