**Linux 下input子系统**

Input子系统分为三层，从下至上分别是设备驱动层，输入核心层以及事件处理层，即inputdevice Driver->inputCore->Eventhandler->userspace.

输入设备主要的工作过程是动作产生（按键，触屏。。。）-->产生中断（数值，坐标。。。。。）-->将数值传递给应用程序。

一个大致的工作流程就是，input device向上层报告-->input core接收报告，并根据在注册inputdevice 时建立好的链接选择哪一类handler来处理事件-->通过handler将数据存放在相应的dev（evdev,mousedev。。。）实例的缓存区中，等待应用程序来读取。这三层的输入核心层和事件处理层都是内核已经完成的，因此需要我们完成的只有设备驱动层。

这里的[**C语言**](http://lib.csdn.net/base/c)逻辑比较简单，就是读取其设备文件，read的返回值是一个

struct input\_event {

      struct timeval time;

      \_\_u16 type;

      \_\_u16 code;

      \_\_s32 value;

};

其中struct timeval time是系统开机时间，其定义为：

struct timeval {

      \_\_kernel\_time\_t          tv\_sec;          /\* seconds \*/

      \_\_kernel\_suseconds\_t       tv\_usec; /\*microseconds \*/

};

可以看出来，这里是按键按下时候的系统时间（第一个参数单位是秒，第二个参数单位是微秒）后面的参数为事件驱动类型（对应的是ev\_key）后面两个是按键的编码和按键的值（这里的按键值是松开是0或者按键按键其值为1）。

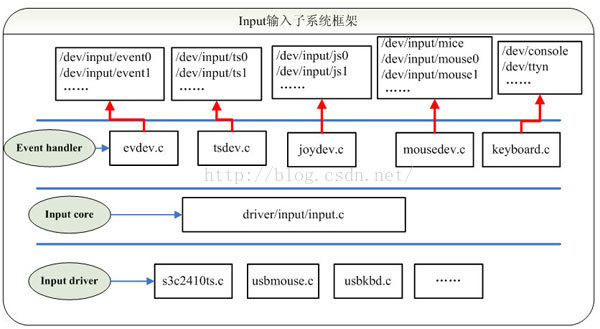
3、  对按键驱动分析

分析按键驱动之前，我觉得有必要先学习一下有关输入系统的介绍：

Input子系统分为三层，从下至上分别是设备驱动层，输入核心层以及事件处理层，即inputdevice Driver -> InputCore -> Eventhandler -> userspace。输入设备主要的工作过程都是动作产生(按键，触屏……)-->产生中断-->读取数值(键值，坐标……)-->将数值传递给应用程序。一个大致的工作流程就是，input device向上层报告-->input core接收报告，并根据在注册inputdevice时建立好的连接选择哪一类handler来处理事件-->通过handler将数据存放在相应的dev(evdev,mousedev…)实例的缓冲区中，等待应用程序来读取。这三层中的输入核心层和事件处理层都是内核已经完成了的，因此需要我们完成的只有设备驱动层。

（上段来至博客：<http://m.blog.csdn.net/blog/jin615567975/37922023> ）

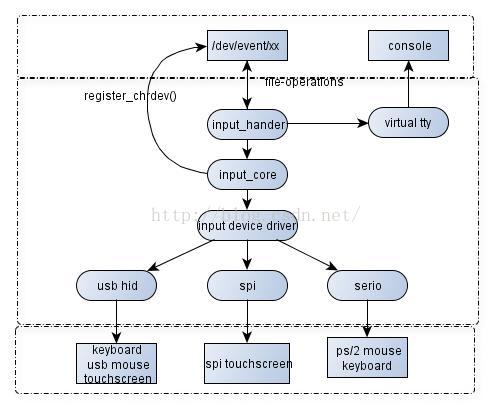
下面网上有张图来解释输入子系统的框架：



（图片来源：<http://www.embedu.org/column/column289.htm> ）

通过上面的图可以看出来，input有像按键等的输入设备event；触摸屏等的输入设备ts；遥控杆等输入设备js；鼠标等输入设备mouse和键盘等不会在/dev/input下产生节点，而是作为ttyn终端（不包括串口终端）的输入。

通过上面的介绍，结合具体的函数或结构体来解释就是设备驱动层为具体用户设备驱动，输入设备由struct input-dev 结构表示，并由input\_register\_device和input\_unregister\_device来注册和卸载；input hander事件处理层主要和用户空间交互，接收用户空间下发的file operation操作命令，生成/dev/input/xx设备节点供用户空间进行file operations操作； input core层负责管理系统中的input dev设备 和input hander事件处理，并起到承上启下作用，负责输入设备和input handler之间信息传输，框架图如下：



（上段来自博客：[http://blog.csdn](http://blog.csdn.net/fanqipin/article/details/8019512)[**.Net**](http://lib.csdn.net/base/dotnet)/fanqipin/article/details/8019512）

大概介绍到这里，下面来分析代码：

按键的驱动定义在Gpio\_keys.c(drivers\input\keyboard)文件中，老规矩首先看到这个宏：

late\_initcall(gpio\_keys\_init);

module\_exit(gpio\_keys\_exit);

这里有个宏late\_initcall，这个宏的跟之前遇见的subsys\_initcall的功能是一样的，只不过其优先级不同，具体的他们全部定义在Init.h (include\linux)中：

#define early\_initcall(fn)         \_\_define\_initcall(fn,early)

#define pure\_initcall(fn)          \_\_define\_initcall(fn,0)

#define core\_initcall(fn)          \_\_define\_initcall(fn,1)

#define core\_initcall\_sync(fn)               \_\_define\_initcall(fn,1s)

#define postcore\_initcall(fn)           \_\_define\_initcall(fn,2)

#define postcore\_initcall\_sync(fn)        \_\_define\_initcall(fn,2s)

#define arch\_initcall(fn)          \_\_define\_initcall(fn,3)

#define arch\_initcall\_sync(fn)               \_\_define\_initcall(fn,3s)

#define subsys\_initcall(fn)             \_\_define\_initcall(fn, 4)

#define subsys\_initcall\_sync(fn)    \_\_define\_initcall(fn,4s)

#define fs\_initcall(fn)                     \_\_define\_initcall(fn,5)

#define fs\_initcall\_sync(fn)            \_\_define\_initcall(fn,5s)

#define rootfs\_initcall(fn)               \_\_define\_initcall(fn,rootfs)

#define device\_initcall(fn)              \_\_define\_initcall(fn, 6)

#define device\_initcall\_sync(fn)    \_\_define\_initcall(fn,6s)

#define late\_initcall(fn)           \_\_define\_initcall(fn, 7)

#define late\_initcall\_sync(fn)         \_\_define\_initcall(fn,7s)

#define module\_init(x)     \_\_initcall(x);

#define \_\_initcall(fn) device\_initcall(fn)

通过上面的宏可以看出他们的后面的参数不同，系数越小优先级越大，可以看出：

优先级由大到小为：subsys\_initcall>module\_init > late\_initcall

接下来就来看gpio\_keys\_init 和gpio\_keys\_exit：

static int \_\_init gpio\_keys\_init(void)

{

      returnplatform\_driver\_register(&gpio\_keys\_device\_driver);

}

static void \_\_exit gpio\_keys\_exit(void)

{

      platform\_driver\_unregister(&gpio\_keys\_device\_driver);

}

他们分别调用了平台驱动注册和注销函数，先看平台驱动注册函数的参数结构体：

static struct platform\_driver gpio\_keys\_device\_driver = {

      .probe           = gpio\_keys\_probe,

      .remove        = gpio\_keys\_remove,

      .driver           = {

             .name     = "gpio-keys",

             .owner   = THIS\_MODULE,

             .pm = &gpio\_keys\_pm\_ops,

             .of\_match\_table =of\_match\_ptr(gpio\_keys\_of\_match),

      }

};

从这个结构体看，跟led的格式就一样的，显示比较name是否都是"gpio-keys"平台设备跟平台驱动的名字比较搭配的话，在注册的时候就会去执行gpio\_keys\_probe函数，在注销的时候就会执行gpio\_keys\_remove的函数，往下看就是对应驱动的名字了;然后gpio\_keys\_pm\_ops,这个由宏

static SIMPLE\_DEV\_PM\_OPS(gpio\_keys\_pm\_ops, gpio\_keys\_suspend,gpio\_keys\_resume);

生成的，该宏定义在Pm.h (include\linux)       ，根据该宏的定义，最终会调用gpio\_keys\_remove和gpio\_keys\_suspend函数控制电源的进入工作状态或者低功耗状态。下一个定义了设备和驱动匹配函数，匹配方法可以用两个字方法：如果driver中定义了of\_device\_id，则通过driver中的of\_device\_id和device中的device\_node内容进行匹配判断，匹配工作由of\_match\_node来完成，该函数会遍历of\_device\_id列表，查找是否有成员与device\_node相匹配，具体由matches的name,type和compatioble来进行对比，如果找到则返回相应的表项，否则返回null.如果没有定义of\_device\_id,device\_node或不能找到对应的匹配项，则通过第二种方式platform\_device\_id来进行对比匹配,通过platform\_match\_id来完成。

接下来详细说明上面提到的几个函数。

先看探测函数gpio\_keys\_probe：

static int gpio\_keys\_probe(struct platform\_device \*pdev)

{

      struct device \*dev =&pdev->dev;

      const structgpio\_keys\_platform\_data \*pdata = dev\_get\_platdata(dev);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

获取平台设备信息，也就是得到JZ2440\_button\_data的数据信息。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      struct gpio\_keys\_drvdata\*ddata;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

structgpio\_keys\_drvdata这个是驱动信息的[**数据结构**](http://lib.csdn.net/base/datastructure)体，这个结构体的定义是解释按键驱动的关键一步，看一下这个结构体的定义：

structgpio\_keys\_drvdata {

      const struct gpio\_keys\_platform\_data\*pdata;

      struct input\_dev \*input;

      struct mutex disable\_lock;

      struct gpio\_button\_data data[0];

};

gpio\_keys\_platform\_data这个就是在Common-smdk.c (arch\arm\mach-s3c24xx)定义的平台信息，包括引脚等的信息；struct input\_dev \*input;这里分配了一个输入设备；struct mutex disable\_lock分配一个互斥锁；struct gpio\_button\_data data[0];分配一个按键数据结构体。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      struct input\_dev \*input;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

structinput\_dev \*input;这里定义了一个输入设备，一个输入设备就是使用input\_dev来表示的，在Input.h (include\linux)       文件中定义了input\_dev结构体，其具体含义为(有背景色的为本次按键定义的)：

structinput\_dev {

        constchar \*name;                       //提供给用户的输入设备的名称

        constchar \*phys;                       //提供给编程者的设备节点的路径及其名称

        const char \*uniq;                       //指定唯一的ID号，就像MAC地址一样

        structinput\_id id;                       //输入设备标识ID，用于和事件处理层进行匹配。

unsignedlong propbit[BITS\_TO\_LONGS(INPUT\_PROP\_CNT)]; //位图，设备属性

unsignedlong evbit[BITS\_TO\_LONGS(EV\_CNT)]; //位图，记录设备支持的事件类型

unsigned long keybit[BITS\_TO\_LONGS(KEY\_CNT)];   //位图，记录设备支持的按键类型

unsignedlong relbit[BITS\_TO\_LONGS(REL\_CNT)]; //位图，记录设备支持的相对坐标

unsignedlong absbit[BITS\_TO\_LONGS(ABS\_CNT)];   //位图，记录设备支持的绝对坐标

unsignedlong mscbit[BITS\_TO\_LONGS(MSC\_CNT)]; //位图，记录设备支持的其他功能

unsignedlong ledbit[BITS\_TO\_LONGS(LED\_CNT)]; //位图，记录设备支持的指示灯

unsignedlong sndbit[BITS\_TO\_LONGS(SND\_CNT)]; //位图，记录设备支持的声音或警报

unsignedlong ffbit[BITS\_TO\_LONGS(FF\_CNT)]; //位图，记录设备支持的作用力功能

unsignedlong swbit[BITS\_TO\_LONGS(SW\_CNT)];   //位图，记录设备支持的开关功能

unsignedint hint\_events\_per\_packet;     //每个数据包记录数据量的个数（上报一次输入事件在EV\_SYN/SYN\_REPORT之间的数据的个数）

unsignedint keycodemax;                //设备支持的最大按键值个数

        unsigned int keycodesize;               //每个按键的字节大小

        void \*keycode;                               //指向按键池，即指向按键值数组首地址

        int (\*setkeycode)(struct input\_dev \*dev, intscancode, int keycode);        //修改按键值

        int (\*getkeycode)(struct input\_dev \*dev, int scancode,int \*keycode);        //获取按键值

        struct ff\_device \*ff;                       //假如设备支持力反馈，这里与里反馈结构相关的东西

        unsigned int repeat\_key;               //支持重复按键

        struct timer\_list timer;               //设置当有连击时的延时定时器

intrep[REP\_CNT]; //记录重复按键的参数值比如延时时间和速率

structinput\_mt \*mt; //指向多触点操作状态

structinput\_absinfo \*absinfo;//对于绝对坐标而言，该指针指向的是桌表包括当前坐标值，最大值最小值，滤波毛刺，分辨率等，具体的可以查看结构体input\_absinfo

unsignedlong key[BITS\_TO\_LONGS(KEY\_CNT)];位图，按键的状态

unsignedlong led[BITS\_TO\_LONGS(LED\_CNT)];位图，led的状态

unsignedlong snd[BITS\_TO\_LONGS(SND\_CNT)];位图，声音的状态

unsignedlong sw[BITS\_TO\_LONGS(SW\_CNT)];位图，开关的状态

        int (\*open)(struct input\_dev \*dev);                       //输入设备打开函数

        void(\*close)(struct input\_dev \*dev);                       //输入设备关闭函数

        int (\*flush)(struct input\_dev \*dev, struct file\*file);        //输入设备断开后清除刷新函数

        int (\*event)(struct input\_dev \*dev, unsigned inttype, unsigned int code, int value);        //事件处理

        structinput\_handle \_\_rcu \*grab;

spinlock\_t event\_lock; //当输入核处理输入时间时候的自旋锁

structmutex mutex; //用于open、close、flush函数的连续访问互斥

unsignedint users;// input handlers打开设备的次数

boolgoing\_away; //输入设备注销的时候的标志位

struct device dev; //输入设备的类信息

structlist\_head       h\_list; //handle链表

structlist\_head       node; //input\_dev链表

unsignedint num\_vals;//当前队列的排队数

unsignedint max\_vals;//允许最大的排队个数

structinput\_value \*vals;//当前队列排队的存储数组

booldevres\_managed;//表明设备正在受到驱动管理，不需要注销或者释放

     };

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      int i, error;

      int wakeup = 0;

      if (!pdata) {

             pdata =gpio\_keys\_get\_devtree\_pdata(dev);

             if (IS\_ERR(pdata))

                    returnPTR\_ERR(pdata);

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

假如conststruct gpio\_keys\_platform\_data \*pdata = dev\_get\_platdata(dev)没有获取平台信息成功，就用设备树获取。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      ddata =kzalloc(sizeof(struct gpio\_keys\_drvdata) +

                    pdata->nbuttons\* sizeof(struct gpio\_button\_data),

                    GFP\_KERNEL);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这里是为pdata申请一块内存，平且将其内容清零。kzalloc实际调用的是kmalloc函数。kmalloc函数返回的是虚拟地址，这里要注意kmalloc（释放内存为kfree）和vmalloc（释放内存为vfree）的区别：kmalloc最大只能开辟128k-16字节，其分配的内存在屋里地址上是线序的，vmalloc是申请的可能是非连续的地址，具体的可参考：

<http://blog.chinaunix.net/uid-20671208-id-3522841.html>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      input = input\_allocate\_device();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

返回值是input\_dev结构体，到这里是申请为输入设备。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (!ddata || !input) {

             dev\_err(dev,"failed to allocate state\n");

             error = -ENOMEM;

             goto fail1;

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

判断是否申请成功，假如不成功的话就打印出失败语句，并且goto到fail1，执行input\_free\_device(input)， kfree(ddata)， kfree(pdata)把申请的资源或者得到的数据释放掉。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      ddata->pdata = pdata;

      ddata->input = input;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

给ddata相应的成员赋值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      mutex\_init(&ddata->disable\_lock);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这是在为ddata初始化互斥锁（互斥体），mutex的使用场合跟信号量基本相同，一般用户那些进程之间竞争，且占用时间较长的场合，当占用时间较短是，一般使用互旋锁。

对于linux锁机制部分可参考http://blog.csdn.net/cyxlxp8411/article/details/8068224

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      platform\_set\_drvdata(pdev,ddata);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

将驱动数据保存到驱动平台数据中，后期将会使用保存的函数。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      input\_set\_drvdata(input,ddata);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

将驱动数据保存到输入设备中中，后期将会使用保存的函数。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      input->name =pdata->name ? : pdev->name;

input->phys ="gpio-keys/input0";

      input->dev.parent =&pdev->dev;

      input->open =gpio\_keys\_open;//在挂起或者唤醒的时候会调用open和close函数

      input->close =gpio\_keys\_close;

      input->id.bustype =BUS\_HOST;

      input->id.vendor =0x0001;

      input->id.product =0x0001;

      input->id.version =0x0100;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这里是在给input赋值，这里主要设备了输入的名字为gpio-keys；设备节点及其路径，驱动父类；然后设置了open和close函数；最后设置了id。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      /\* Enable auto repeatfeature of Linux input subsystem \*/

      if (pdata->rep)

             \_\_set\_bit(EV\_REP,input->evbit);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

给按键设置可重复多次按下的特性。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      for (i = 0; i <pdata->nbuttons; i++) {

             const structgpio\_keys\_button \*button = &pdata->buttons[i];

             structgpio\_button\_data \*bdata = &ddata->data[i];

             error = gpio\_keys\_setup\_key(pdev,input, bdata, button);

             if (error)

                    goto fail2;

             if(button->wakeup)

                    wakeup = 1;

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个循环获取每个按键的信息，并通过gpio\_keys\_setup\_key函数为每个按键初始化引脚，滤波消抖，申请外部中断，申请定时器中断平且设定中断定时器服务函数。gpio\_keys\_setup\_key该函数将会在下面单独介绍。假如创建失败会调用gpio\_remove\_key(&ddata->data[i]);释放掉申请的引脚，取消申请的队列等等

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

error = sysfs\_create\_group(&pdev->dev.kobj,&gpio\_keys\_attr\_group);

      if (error) {

             dev\_err(dev,"Unable to export keys/switches, error: %d\n",

                    error);

             goto fail2;

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

sysfs\_create\_group()在kobj目录下创建一个属性集合，并显示集合中的属性文件。如果文件已存在，会报错。以函数为例，这里将会在gpio-keys/目录下创建一个属性文件gpio\_keys\_attr\_group，gpio\_keys\_attr\_group最终会调用：

staticstruct attribute \*gpio\_keys\_attrs[] = {

      &dev\_attr\_keys.attr,

      &dev\_attr\_switches.attr,

      &dev\_attr\_disabled\_keys.attr,

      &dev\_attr\_disabled\_switches.attr,

      NULL,

};

也就是会生成：

/sys/devices/platform/gpio-keys/disabled\_keys[rw]

/sys/devices/platform/gpio-keys/disables\_switches[rw]

/sys/devices/platform/gpio-keys/keys[ro]

/sys/devices/platform/gpio-keys/switches[ro]

假如创建失败假如创建失败会调用gpio\_remove\_key(&ddata->data[i]);释放掉申请的引脚，取消申请的队列等等

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      error =input\_register\_device(input);

      if (error) {

             dev\_err(dev,"Unable to register input device, error: %d\n",

                    error);

             goto fail3;

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这是是真正的向内核注册一个输入设备。该函数将input\_dev结构体注册到输入子系统核心中，input\_dev由前面介绍的input\_allocate\_device()函数来分配。input\_register\_device()函数如果注册失败，必须调用input\_free\_device()函数释放分配的空间。假如函数注册成功，调用input\_unregister\_device()函数来注销输入设备结构体。 input\_register\_device()函数中的dev\_set\_name设置input\_dev中的device的名字，名字以input0、input1、input2·······等的形式出现在sysfs文件系统中。并且input\_register\_device()函数中list\_add\_tail()函数将input\_dev加入input\_dev\_list链表中，的input\_dev\_list链表中包含了系统中所有的input\_dev设备。

假如注册失败会调用sysfs\_remove\_group(&pdev->dev.kobj,&gpio\_keys\_attr\_group);来删除之前创建的属性文件。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      device\_init\_wakeup(&pdev->dev,wakeup);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这函数跟电源管理有关，在Pm\_wakeup.h (include\linux)文件中：

staticinline int **device\_init\_wakeup**(struct device \*dev, bool val)

{

   device\_set\_wakeup\_capable(dev, val);  **//设置设备能不能被唤醒**

    device\_set\_wakeup\_enable(dev, val);     **//设置设备使不使用唤醒；**

    return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      return 0;

 fail3:

      sysfs\_remove\_group(&pdev->dev.kobj,&gpio\_keys\_attr\_group);

 fail2:

      while (--i >= 0)

             gpio\_remove\_key(&ddata->data[i]);

 fail1:

      input\_free\_device(input);

      kfree(ddata);

      /\* If we have no platformdata, we allocated pdata dynamically. \*/

      if(!dev\_get\_platdata(&pdev->dev))

             kfree(pdata);

      return error;

}

probe函数做完了整个驱动要做的事情，现在总结一下probe都做了些什么事：

首先获取平台设备信息：dev\_get\_platdata(dev)或者gpio\_keys\_get\_devtree\_pdata(dev)

为定义的设备信息申请一块内存：kzalloc

申请分配一个输入设备: input\_allocate\_device

为该输入设备设置属性：input->······

初始化按键相关的引脚、中断、及其有关的定时器信息：gpio\_keys\_setup\_key

为该设备创建一个属性集合：sysfs\_create\_group

  正式申请为输入设备：input\_register\_device

刚才有看到在probe函数中，调用了好多自定义的一些函数，接下来逐个分析一下这些函数的实现。

先看一下设置按键的函数gpio\_keys\_setup\_key：

static int gpio\_keys\_setup\_key(struct platform\_device \*pdev,

                           structinput\_dev \*input,

                           structgpio\_button\_data \*bdata,

                           conststruct gpio\_keys\_button \*button)

{

      const char \*desc =button->desc ? button->desc : "gpio\_keys";

      struct device \*dev =&pdev->dev;

      irq\_handler\_t  isr;

      unsigned long   irqflags;

      int irq, error;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

constchar \*desc = button->desc ? button->desc : "gpio\_keys";这里是获取按键的描述的名字个设备信息应该为Button 1、Button 2、Button 3，假如获取不到其描述符就是gpio\_keys。

并且定义了中断申请需要的变量。

irq\_handler\_t定义在Interrupt.h (include\linux)      文件中：

typedefirqreturn\_t (\*irq\_handler\_t)(int, void \*);

其中为irqreturn\_t定义在Irqreturn.h (include\linux)文件中

/\*\*

 \* enum irqreturn

 \* @IRQ\_NONE            interruptwas not from this device

 \* @IRQ\_HANDLED          interruptwas handled by this device

 \* @IRQ\_WAKE\_THREAD handler requests to wake the handler thread

 \*/

enumirqreturn {

      IRQ\_NONE        =(0 << 0),

      IRQ\_HANDLED             = (1 << 0),

      IRQ\_WAKE\_THREAD          = (1 << 1),

};

typedefenum irqreturn irqreturn\_t;

也就是说返回不同的值代表的含义：

IRQ\_NONE ，还没有发生中断

IRQ\_HANDLED     中断已经处理完毕

IRQ\_WAKE\_THREAD中断需要唤醒处理线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      bdata->input = input;

      bdata->button =button;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

向gpio\_button\_data：：bdata的参数赋值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      spin\_lock\_init(&bdata->lock);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

初始化自旋锁,自旋锁适用于临界区不是很大的情况（临界区的执行时间比较短）

总结自旋锁使用流程：

1、首先定义一个自旋锁：

spinlock\_t lock

2、初始化自旋锁：

   spin\_ lock\_init(lock)

3、获取自旋锁

  (1)  spin\_trylock(lock)//假如获得锁返回真，否则返回假，返回假货不会原地打转的等着获得锁

  (2)  spin\_lock(lock)//假如获得锁返回真，否则返回假，返回假货将会原地打转的等着获得锁

##############################################################

假如获得会执行临界区（要锁存的区域）的操作

。。。。。。。。。

###############################################################

4、释放掉自旋锁

   spin\_unlock(lock);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if(gpio\_is\_valid(button->gpio)) {

             error =gpio\_request\_one(button->gpio, GPIOF\_IN, desc);  //申请一个pgio,配置为输入

             if (error < 0){

                    dev\_err(dev,"Failed to request GPIO %d, error %d\n",

                           button->gpio,error);

                    returnerror;

             }

             if(button->debounce\_interval) {

                    error =gpio\_set\_debounce(button->gpio,

                                  button->debounce\_interval\* 1000); //按键消抖处理

                    /\* usetimer if gpiolib doesn't provide debounce \*/

                    if (error< 0)

                           bdata->timer\_debounce=

                                         button->debounce\_interval;

             }

             irq =gpio\_to\_irq(button->gpio);//申请GPIO中断,正确返回中断编号，错误返回错误编码（负）

             if (irq < 0) {

                    error =irq;

                    dev\_err(dev,

                           "Unableto get irq number for GPIO %d, error %d\n",

                           button->gpio,error);

                    goto fail;

             }

             bdata->irq =irq;//赋值中断号

             INIT\_WORK(&bdata->work,gpio\_keys\_gpio\_work\_func);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

将函数gpio\_keys\_gpio\_work\_func假如工作队列，让中断的第二阶段去执行该函数。实际上&bdata->work是一个描述工作队列的结构体，变量定义为struct work\_struct work，该结构体定义在Workqueue.h (include\linux)文件中：

structwork\_struct {

      atomic\_long\_t data;

      struct list\_head entry;

      work\_func\_t func;

#ifdefCONFIG\_LOCKDEP

      struct lockdep\_map lockdep\_map;

#endif

};

work\_struct结构体中的work\_func\_t func指向要假如中断队列的函数，这样的话函数gpio\_keys\_gpio\_work\_func就和&bdata->work绑定在一起了，一般情况下，gpio\_keys\_gpio\_work\_func会在中断下半部执行，所以在中断服务函数中会调用schedule\_work(&bdata->work)来执行该函数。

最后在卸载的时候会调用cancel\_work\_sync(&bdata->work);来取消加进去的工作队列。

总结工作队列过程(注：这里是使用内核线程，并不是自定义一个线程，假如需要创建队列可参考：

http://blog.chinaunix.net/uid-24148050-id-296982.html)：

1、首先定义一个工作结构体：

struct work\_struct work

2、将定义的工作加入内核工作队列并且绑定放入队列的函数：

    INIT\_WORK(work, work\_func);

或者

INIT\_DELAYED\_WORK(work, work\_func)

3、当需要执行工作函数的时：

   (1)对应的INIT\_WORK

执行schedule\_work(work)会马上调用work\_func函数

   (2) 对应的NIT\_DELAYED\_WORK

      执行schedule\_delayed\_work（time,work）会在time时间后执行work\_func

##############################################################

执行函数work\_func的操作

。。。。。。。。。

###############################################################

4、取消工作队列中的工作：

     (1)对应的INIT\_WORK

cancel\_work\_sync(work)

   (2) 对应的NIT\_DELAYED\_WORK

     cancel\_delayed\_work\_sync//取消延时工作并且等待其完成工作

cancel\_delayed\_work(work)//取消延时工作

<http://blog.csdn.net/bingqingsuimeng/article/details/7891157>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

             setup\_timer(&bdata->timer,

                        gpio\_keys\_gpio\_timer, (unsignedlong)bdata);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

由于按键消抖需要定时器，这是是初始化定时器，首先看一下&bdata->timer，其类型struct timer\_list定义在Timer.h (include\linux)  文件中：

structtimer\_list {

      /\*

       \*All fields that change during normal runtime grouped to the

       \*same cacheline

       \*/

      struct list\_head entry;  //双链表，用来将多个定时器连接成一条双向循环队列。

      unsigned long expires; //定时时间

structtvec\_base \*base;  // 这个tvec\_base是动态定时器的主要数据结构，每个cpu上有一个，它包含相应cpu中处理动态定时器需要的所有数据

      void (\*function)(unsigned long); //超时时要处理的函数

      unsigned long data;  //超时处理函数参数

      int slack; //跟定时器的内核处理方式有关的变量

#ifdefCONFIG\_TIMER\_STATS

      int start\_pid;

      void \*start\_site;

      char start\_comm[16];

#endif

#ifdefCONFIG\_LOCKDEP

      struct lockdep\_map lockdep\_map;

#endif

};

当运行setup\_timer(&bdata->timer,gpio\_keys\_gpio\_timer, (unsigned long)bdata);这个宏就是设定的定时器为&bdata->timer，定时器到时间后运行的函数为gpio\_keys\_gpio\_timer，定时器函数所需要的数据为(unsigned long)bdata。

有关输入子系统的介绍就到这里，下面总结一下定时器的使用方法：

1、定义一个定时器timer\_listmytimer

2、初始化定时器并赋值成员setup\_timer(mytimer, timer\_func, data);

3、增加定时器add\_timer(mytimer)

4、该修改定时器的定时时间expire  mod\_timer(mytimer,expire)

5、取消定时器，有两个可选择

1.       del\_timer(mytimer) 直接删除定时器

2.       del\_timer\_sync(mytimer)等待本次定时器处理完毕再取消（不适用中断上下文）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

             isr =gpio\_keys\_gpio\_isr;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 这是是设置中断服务函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

             irqflags = IRQF\_TRIGGER\_RISING| IRQF\_TRIGGER\_FALLING;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 这是设定中断处理的属性，也就是中断触发方式处理方式等等。这里设置上升沿触发或者下降沿触发。假如设置IRQF\_SHARED表明多个设备共享一个中断，此时会用到dev\_id；当设置IRQF\_DISABLED时候表明，中断为快速中断。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      } else {

             if(!button->irq) {

                    dev\_err(dev,"No IRQ specified\n");

                    return-EINVAL;

             }

             bdata->irq =button->irq;

             if(button->type && button->type != EV\_KEY) {

                    dev\_err(dev,"Only EV\_KEY allowed for IRQ buttons.\n");

                    return-EINVAL;

             }

             bdata->timer\_debounce= button->debounce\_interval;

             setup\_timer(&bdata->timer,

                        gpio\_keys\_irq\_timer, (unsigned long)bdata);

             isr =gpio\_keys\_irq\_isr;

             irqflags = 0;

      }

      input\_set\_capability(input,button->type ?: EV\_KEY, button->code);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

设定该按键具有什么能力，以本例来说，button1 button2 button3具有按键F1 F2 F3的能力

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      /\*

       \* If platform has specified that the buttoncan be disabled,

       \* we don't want it to share the interruptline.

       \*/

      if(!button->can\_disable)

             irqflags |=IRQF\_SHARED;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

设置是否共享中断，这里涉及到中断共享机制：

多个中断共享一个中断线必须用IRQF\_SHARED做标记，以IRQF\_SHARED作为标记的中断假如要向内核申请成功一个中断需要两个条件之一：

该中断还没有被申请

该中断虽然被申请了，但是已经申请的中断也有IRQF\_SHARED做标记

当发生共享中断时，所用挂载到此中断的中断服务函数都会得到响应（遍历所有该中断线上的中断），所以说，当该中断设备为共享中断的时候，中断服务函数首先需要判断是否是自己的dev\_id假如不是自己的dev\_id那么返回IRQ\_NOTE（表明不是本中断），假如检测到时本中断的话就会执行中断里面的函数，最后返回IRQ\_HANDLED。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      error =request\_any\_context\_irq(bdata->irq, isr, irqflags, desc, bdata);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

申请一个中断线，其中bdata->irq为要申请的中断线（中断号）；isr为中断服务函数；irqflags中断的触发方式或者处理方式；要申请中断的描述符；bdata为dev\_id，区分共享中断线线而设定的。

看一下申请外部中断的流程：

1、申请某个引脚为外部中断 intirq = gpio\_to\_irq(button->gpio);

2、设备外部中断函数irq\_handler\_tisr = gpio\_keys\_gpio\_isr;

3、设置中断发出类型 unsignedlong flags =？

4、描述该中断的一个assic字符串的名字  constchar \*name=？

5、设备dev-id 是一个空函数指针  void\*dev\_id =？

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

if (error < 0) {

             dev\_err(dev,"Unable to claim irq %d; error %d\n",

                    bdata->irq,error);

             goto fail;

      }

      return 0;

fail:

      if(gpio\_is\_valid(button->gpio))

             gpio\_free(button->gpio);

      return error;

}

现在总结一下设置按键的函数gpio\_keys\_setup\_key作用：

该函数主要是申请外部中断，设定中断服务函数，由于在消抖的时候会用到定时器，这个函数还初始化了定时器。并且绑定了定时器中断服务函数。

接下来看一下按键服务函数：

static irqreturn\_t gpio\_keys\_gpio\_isr(int irq, void \*dev\_id)

{

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

首先看中断服务函数的返回值：

IRQ\_NONE ，还没有发生中断

IRQ\_HANDLED     中断已经处理完毕

IRQ\_WAKE\_THREAD中断需要唤醒处理线程

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      struct gpio\_button\_data\*bdata = dev\_id;

      BUG\_ON(irq !=bdata->irq);//代码调试用的

      if(bdata->button->wakeup)  //根据wakeup保存非睡眠状态

             pm\_stay\_awake(bdata->input->dev.parent);

      if (bdata->timer\_debounce)

             mod\_timer(&bdata->timer,

                    jiffies +msecs\_to\_jiffies(bdata->timer\_debounce));

      else

             schedule\_work(&bdata->work);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 先看if的条件，条件为消抖的时间，假如需要消抖（消抖的时间非0），那么就执行       mod\_timer(&bdata->timer,jiffies+ msecs\_to\_jiffies(bdata->timer\_debounce));

mod\_timer用于改变或者设定定时器的定时时间，其中&bdata->timer为要修改的定时器，jiffies +msecs\_to\_jiffies(bdata->timer\_debounce)为修改后的时间msecs\_to\_jiffies函数是吧毫秒转换为jiffies，单单看这个函数就是在过msecs\_to\_jiffies(bdata->timer\_debounce)的时间触发定时器，定时器时间到后就会执行定时器服务函数（在按键设置函数gpio\_keys\_setup\_key中已经设置定时器服务函数为gpio\_keys\_gpio\_timer），定时器服务函数的内容为：

static void gpio\_keys\_gpio\_timer(unsigned long\_data)

{

      structgpio\_button\_data \*bdata = (struct gpio\_button\_data \*)\_data;

      schedule\_work(&bdata->work);

}

可以看到时间一到会执行schedule\_work(&bdata->work);执行中断底部的队列部分；

这里可以看到，假如不需要消抖，那么直接执行中断底部的队列部分schedule\_work(&bdata->work)。

（稍后介绍工作工作队列的工作）

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      return IRQ\_HANDLED;

}

对于按键中断服务函数gpio\_keys\_gpio\_isr可以看出来，中断函数确实是分为中断顶半部（top half）和中断底半部（bottom half），对于按键来说，中断顶半部就是消抖（消抖还用到了定时间，所以说需要的执行时间很短），中断的底半部就是用来执行工作队列里面的工作了，接下来看一下队列的工作都做了什么：

static void gpio\_keys\_gpio\_work\_func(struct work\_struct \*work)

{

      struct gpio\_button\_data\*bdata =

             container\_of(work,struct gpio\_button\_data, work);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 container\_of是根据一个结构体变量中的一个域成员变量的指针来获取指向整个结构体变量的指针，就本例而言，就是根据gpio\_button\_data->work这个成员的指针来获取整个结构体gpio\_button\_data的指针首地址。

假如想了解container\_of的原理，可参考：<http://www.embedu.org/Column/Column433.htm>

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      gpio\_keys\_gpio\_report\_event(bdata);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个是整个输入子系统的关键的一个函数（时间上报函数），稍后将会介绍。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if(bdata->button->wakeup)   //电源管理有关

             pm\_relax(bdata->input->dev.parent);

}

可以看到队列工作就是为了调用时间上报函数，看一下时间上报函数的实现：

static void gpio\_keys\_gpio\_report\_event(struct gpio\_button\_data\*bdata)

{

      const structgpio\_keys\_button \*button = bdata->button;

      struct input\_dev \*input =bdata->input;

      unsigned int type =button->type ?: EV\_KEY;

      int state =(gpio\_get\_value\_cansleep(button->gpio) ? 1 : 0) ^ button->active\_low;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

gpio\_get\_value\_cansleep是获取按键的值，其gpio\_get\_value区别是，有些芯片的引脚与cpu是依靠一些总线连接的比如iic总线，那么这些引脚就有可能产生休眠，因此要用gpio\_get\_value\_cansleep来获取按键的值，获取后异或button->active\_low（之前设定的是1），那么当有按键按下时，gpio\_get\_value\_cansleep得到的是低电平也就是0，所以0^1=1，这里也是再说名，当按键按下时，获取的state应该是1。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (type == EV\_ABS) {  //这个类型判断是不是EV\_ABS（绝对坐标）事件，本次是EV\_KEY事件

             if (state)

                    input\_event(input,type, button->code, button->value);

      } else {

             input\_event(input,type, button->code, !!state);

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 这是就是上报事件了，这个为执行input\_event(input, type, button->code,!!state);

input\_event函数的参数为，input上报的设备（也就是input\_dev）；type为上报事件的类型，这里是EV\_KEY（按键事件），button->code按键的编码，就本次而言对应的额是F1 F2 F3；state为上报值

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      input\_sync(input);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

input\_sync(input)实际定义是：

staticinline void input\_sync(struct input\_dev \*dev)

{

      input\_event(dev, EV\_SYN, SYN\_REPORT, 0);

}

用来告诉上层，本次的事件已经完成了.

这里要注意的就是，设置触发方式的时候，设置的是下降沿触发或上升沿触发，也就是当执行一次按键按下松开的动作时，会触发两次中断（一次按下，一次松开），每次中断都会执行本函数，执行一次本函数上报两个输入事件（执行了两次input\_event），也就是说，按下松开一共会上报四个数据。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

到这里上报时间就完工了，回想一下，整个输入子系统的流程可以总结为两部：

首先定义申请注册一个输入设备。

申请按键为外部中断，当按键按下时，在中断服务函数中处理，并上报按键事件。

有关按键的主题部分已经分析完毕，接下来看一下其他的一些函数：

在介绍gpio\_keys\_probe函数中的sysfs\_create\_group时，会生成一些属性文件，看一下这个属性文件是如何使用在驱动程序中的，先看一下生成哪些属性文件

/sys/devices/platform/gpio-keys/disabled\_keys[rw]

/sys/devices/platform/gpio-keys/disables\_switches[rw]

/sys/devices/platform/gpio-keys/keys[ro]

/sys/devices/platform/gpio-keys/switches[ro]

这里就以disabled\_keys文件为例介绍如何使用这些属性文件来设置驱动程序。

由上节的led的分析我们知道，对于sysfs下的属性文件，当要读文件的实收会执行\*show函数，当要写文件的时候会执行\*store程序，接下来分别看下一下这两个函数，首先看show函数：

static ssize\_t gpio\_keys\_show\_##name(struct device \*dev,          \

                                struct device\_attribute \*attr, \

                                char \*buf)                       \

{                                                            \

      struct platform\_device\*pdev = to\_platform\_device(dev);              \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 先看下定义：

#define to\_platform\_device(x)container\_of((x), struct platform\_device, dev)

可以看到to\_platform\_device就是通过platform\_device的成员dev来得到platform\_device结构体的首地址。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      struct gpio\_keys\_drvdata\*ddata = platform\_get\_drvdata(pdev);    \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在probe有执行platform\_set\_drvdata(pdev, ddata);将ddata的保存起来，这里是获取保存的数据。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

                                                              \

      returngpio\_keys\_attr\_show\_helper(ddata, buf,                \

                                    type, only\_disabled);            \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个函数就是控制使能与否的关键函数了。下面介绍本函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

static ssize\_t gpio\_keys\_attr\_show\_helper(structgpio\_keys\_drvdata \*ddata,

                                    char \*buf, unsigned int type,

                                    bool only\_disabled)

{

      int n\_events =get\_n\_events\_by\_type(type);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

get\_n\_events\_by\_type这是是返回事件类型的最大值（因为后面要申请内存，所以这里需要对应本类时间的最大值）。可以查看实际上返回的书是0x3000。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      unsigned long \*bits;

      ssize\_t ret;

      int i;

      bits = kcalloc(BITS\_TO\_LONGS(n\_events), sizeof(\*bits), GFP\_KERNEL);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

kcalloc申请一个数组的内存空间,并把申请得到的内存都初始化为零，其参数为要申请的数量，第二个参数为单位数量占有的字节数，第三个参数分配内存时的控制方式。

BITS\_TO\_LONGS函数是求一个数是几个long型的长度，sizeof(\*bits)就是sizeof(long)=4，这个函数要是申请的字节数就是BITS\_TO\_LONGS(n\_events)乘以sizeof(\*bits)得到的字节数。

实际上，可用下面的例子来测试：

#include <stdio.h>

#define DIV\_ROUND\_UP(n, d) (((n)+(d)-1) / (d))

#define BITS\_PER\_BYTE           8

#define BITS\_TO\_LONGS(nr)       DIV\_ROUND\_UP(nr, BITS\_PER\_BYTE \*sizeof(long))

void main()

{

      intj = 0x300;

      unsignedlong \*bits;

      printf("sizeof long is %ld\n", sizeof(\*bits));

      printf("result is%ld\n", BITS\_TO\_LONGS(j));

while(1);

}

运行结果为：size of long is 4      result is 24

对于第三个参数的标志位，这些标志位定义在Gfp.h (include\linux)文件中：

具体的介绍这在里摘录博客的一段话：<http://m.blog.csdn.net/blog/gs_119/23698667>

GFP的标记有两种:带双下划线前缀的和不带双下划线前缀的;

不带双下划线前缀的GFP标志:

GFP\_ATOMIC:用于在中断上下文和进程上下文之外的其它代码中分配内存;从不睡眠;

GFP\_KERNEL:内核正常分配内存;可能睡眠;

GFP\_USER  :用于为用户空间页分配内存;可能睡眠;

GFP\_HIGHUSER:如同GFP\_USER,但它是从高端内存中申请;

GFP\_NOIO和GFP\_NOFS:功能如同GFP\_KERNEL,但是它俩增加限制到内核能做的来满足请求;GFP\_NOFS分配不允许进行任何文件系统调用,而GFP\_NOIO分配根本不允许进行任何IO初始化;它俩主要用于文件系统和虚拟内存代码,那里允许一个分配睡眠,但是递归的文件系统调用会是个坏主意;

带有双下划线前缀的GFP标志:

\_\_GFP\_DMA:这个标志要求分配的内存在能够进行DMA的内存区;平台依赖的;

\_\_GFP\_HIGHMEM:这个标志指示分配的内存可以位于高端内存区;平台依赖的;

\_\_GFP\_COLD:正常地,内存分配器尽力返回"缓冲热"的页---可能在处理器缓冲中找到的页;相反,这个标志请求一个"冷"页---在一段时间内没被使用的页;它对分配页做DMA读是很有用的,此时在处理器缓冲中出现是没用的;

\_\_GFP\_NOWARN:这个标志用于分配内存时阻止内核发出警告,当一个分配请求无法满足时;

\_\_GFP\_HIGH:这个标志标识了一个高优先级请求,它被允许来消耗甚至被内核保留给紧急状况的最后的内存页;

\_\_GFP\_REPEAT:分配器的动作;当分配器有困难满足一个分配请求时,通过重复尝试的方式来"尽力尝试",但是分配操作仍然有可能失败;

\_\_GFP\_NOFAIL:分配器的动作;当分配器有困难满足一个分配请求时,这个标志告诉分配器不要失败,尽最大努力来满足分配请求;

\_\_GFP\_NORETRY:分配器的动作;当分配器有困难满足一个分配请求时,这个标志告诉分配器立即放弃,不再做任何尝试;

通常,一个或多个带双下划线前缀的标记相或,即可得到对应的不带双下划线前缀的标记;

最常用的标记就是GFP\_KERNEL,它的意思就是当前的这个分配代表运行在内核空间的进程而进行的;换句话说,这意味着调用函数是代表一个进程在执行一个系统调用;使用GFP\_KERNEL标记,就意味着kmalloc能够使当前进程在少内存的情况下通过睡眠来等待一个内存页;因此,一个使用GFP\_KERNEL的函数必须是可重入的,且不能在原子上下文中运行;当当前进程睡眠,内核采取正确的动作来定位一些空闲的内存页,或者通过刷新缓存到磁盘或者交换出去一个用户进程的内存页;

如果一个内存分配动作发生在中断处理或内核定时器的上下文中时,当前进程就不能被设置为睡眠,也就不能再使用GFP\_KERNEL标志了,此时应该使用GFP\_ATOMIC标志来代替;正常地,内核试图保持一些空闲页以便来满足原子的分配;当使用GFP\_ATOMIC标志时,kmalloc标志能够使用甚至最后一个空闲页;如果这最后一个空闲页不存在,那分配就会失败;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (!bits)

             return -ENOMEM;

      for (i = 0; i <ddata->pdata->nbuttons; i++) {

             structgpio\_button\_data \*bdata = &ddata->data[i];

             if(bdata->button->type != type)  //假如不是预定类型跳过本循环

                    continue;

             if (only\_disabled&& !bdata->disabled) //假如不使能跳过本循环

                    continue;

             \_\_set\_bit(bdata->button->code,bits);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\_\_set\_bit是非原子型的按位操作（set\_bit函数是原子操作），这里有点难理解，就是将申请内存的是bits相应的按键编码位置1，就以本函数的实际代表意思来解释，就是在判定本按键有效的情况下，就是把刚才申请的bits所指向的内存对应的F1或者F2或者F3相应的位置1。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      }

      ret =bitmap\_scnlistprintf(buf, PAGE\_SIZE - 2, bits, n\_events);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

将位图转花纹字符串。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      buf[ret++] = '\n';

      buf[ret] = '\0';

      kfree(bits);

      return ret; //返回值是换成的字符串的长度

}

这个函数的实际含义就是当你对/sys/devices/platform/gpio-keys/disabled\_keys 文件进行读的时候就用调用open函数，举一个直观些的例子就是：

int strong read(fd,char \*buf,&ev\_key, only\_disabled); //fd为打开的设备文件

上面的函数是读/sys/devices/platform/gpio-keys/disabled\_keys要执行的函数，下面看下写该文件要执行的函数：

static ssize\_tgpio\_keys\_store\_##name(struct device \*dev,           \

                                struct device\_attribute \*attr,      \

                                const char \*buf,                   \

                                size\_t count)                 \

{                                                            \

      struct platform\_device \*pdev = to\_platform\_device(dev);              \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 先看下定义：

#define to\_platform\_device(x)container\_of((x), struct platform\_device, dev)

可以看到to\_platform\_device就是通过platform\_device的成员dev来得到platform\_device结构体的首地址。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata(pdev);    \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

在probe有执行platform\_set\_drvdata(pdev,ddata);将ddata的保存起来，这里是获取保存的数据。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      ssize\_t error;                                           \

                                                              \

      error = gpio\_keys\_attr\_store\_helper(ddata, buf, type);            \

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这个函数就是控制使能与否的关键函数了。下面介绍本函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (error)                                           \

             return error;                                     \

                                                              \

      return count;                                           \

}

static ssize\_tgpio\_keys\_attr\_store\_helper(struct gpio\_keys\_drvdata \*ddata,

                                    const char \*buf, unsigned int type)

{

      int n\_events = get\_n\_events\_by\_type(type);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

get\_n\_events\_by\_type这是是返回事件类型的最大值（因为后面要申请内存，所以这里需要对应本类时间的最大值）。可以查看实际上返回的书是0x3000。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      unsigned long \*bits;

      ssize\_t error;

      int i;

      bits = kcalloc(BITS\_TO\_LONGS(n\_events), sizeof(\*bits),GFP\_KERNEL);

      if (!bits)

             return -ENOMEM;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

申请内存，跟上个函数一样，不清楚的查看上个函数的解析。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      error = bitmap\_parselist(buf, bits, n\_events);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bitmap\_parselist就是吧buf的数据，写入到bits所指向的内存中，n\_events是写入的数量。

bitmap\_parselist函数定义在Bitmap.c (lib)     文件中，原型为：

int bitmap\_parselist(const char \*bp, unsignedlong \*maskp, int nmaskbits)

{

      char\*nl  = strchr(bp, '\n');//

（

功能：查找[字符](http://baike.baidu.com/view/263416.htm)串bp中首次出现字符'\n'的位置

说明：返回首次出现'\n'的位置的[指针](http://baike.baidu.com/view/159417.htm)，返回的地址是被查找字符串指针开始的第一个与'\n'相同字符的指针，如果bp中不存在'\n'则返回[NULL](http://baike.baidu.com/view/329484.htm)。

返回值：成功则返回要查找字符第一次出现的位置，失败返回NULL.

）

      intlen;

      if(nl)

             len= nl - bp; //字符串低一个字符到字符'\n'的长度

      else

             len= strlen(bp);

      return\_\_bitmap\_parselist(bp, len, 0, maskp, nmaskbits);//该函数将在下面分析

}

\_\_bitmap\_parselist函数定义在Bitmap.c (fs\reiserfs)中，现在只列出该参数：

static int \_\_bitmap\_parselist(const char \*buf,unsigned int buflen,int is\_user, unsigned long \*maskp,

             intnmaskbits)

该函数的作用是将asccic的字符串转换成为位图形似，看一下其参数的意义：

第一个参数const char \*buf是要转换的字符串

第二个参数unsigned int buflen要转化的字符串的长度

第三个参数int is\_user,用户，0代表是内核空间

第四个参数unsigned long \*maskp：将要写入的数据地址

第五个参数int nmaskbits：要写入数据的数量

返回0表示成功

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (error)

             goto out;

      /\* First validate \*/

      for (i = 0; i < ddata->pdata->nbuttons; i++) {

             struct gpio\_button\_data \*bdata = &ddata->data[i];

             if (bdata->button->type != type)

                    continue;

             if (test\_bit(bdata->button->code, bits) &&

                !bdata->button->can\_disable) {

                    error = -EINVAL;

                    goto out;

             }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

test\_bit是为测试，就是测试bits第bdata->button->code位是的值，返回的数据bits第bdata->button->code位的值（0or1）。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      }

      mutex\_lock(&ddata->disable\_lock);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这里是申请互斥体，在前面的probe函数里面有mutex\_init(&ddata->disable\_lock)初始化了互斥体。

互斥锁的使用步骤如下：

1、定义互斥体 struct mutex  my\_mutex

2、初始化互斥体 mutex\_init（my\_mutex）

3、尝试获取锁 mutex\_lock（my\_mutex）

#########################################################################

临界区操作

#########################################################################

4、释放掉互斥体

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      for (i = 0; i < ddata->pdata->nbuttons; i++) {

             struct gpio\_button\_data \*bdata = &ddata->data[i];

             if (bdata->button->type != type)

                    continue;

             if (test\_bit(bdata->button->code, bits))

                    gpio\_keys\_disable\_button(bdata);

             else

                    gpio\_keys\_enable\_button(bdata);

      }

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

根据输入的数据选择是执行gpio\_keys\_disable\_button函数还是执行gpio\_keys\_enable\_button函数，其中gpio\_keys\_disable\_button关掉外部中断和取消定时器（这里的取消是指上次设定的时间没到的那次定时器作用），gpio\_keys\_enable\_button函数就是使能外部按键中断，这里

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      mutex\_unlock(&ddata->disable\_lock); //释放掉互斥体

out:

      kfree(bits); //释放掉申请的bits的内存

      return error;

}

有关按键的驱动的驱动初始化及其上报事件读写部分代码已经分析完毕，解析来看一下是卸载函数要执行的代码：

static intgpio\_keys\_remove(struct platform\_device \*pdev)

{

      struct gpio\_keys\_drvdata \*ddata = platform\_get\_drvdata(pdev);

      struct input\_dev \*input = ddata->input;

      int i;

      sysfs\_remove\_group(&pdev->dev.kobj,&gpio\_keys\_attr\_group);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

相对于sysfs\_create\_group(&pdev->dev.kobj,&gpio\_keys\_attr\_group);这里是吧初始化时建立的属性文件删除掉

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      device\_init\_wakeup(&pdev->dev, 0);

      for (i = 0; i < ddata->pdata->nbuttons; i++)

             gpio\_remove\_key(&ddata->data[i]);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

static void gpio\_remove\_key(structgpio\_button\_data \*bdata)

{

      free\_irq(bdata->irq,bdata);

      if(bdata->timer\_debounce)

             del\_timer\_sync(&bdata->timer);

      cancel\_work\_sync(&bdata->work);

      if(gpio\_is\_valid(bdata->button->gpio))

             gpio\_free(bdata->button->gpio);

}

该函数主要是删除定时器，取消工作队列的工作，释放申请的引脚

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      input\_unregister\_device(input);//注销输入设备

      /\* If we have no platform data, we allocated pdata dynamically.\*/

      if (!dev\_get\_platdata(&pdev->dev))

             kfree(ddata->pdata); //释放pdata

      kfree(ddata);//释放在probe申请的ddata的内存

      return 0;

}

上面的是设备驱动层的核心代码，已经做出详细的介绍，在说明文档里面的Documentation\input\input-programming.txt，文件中有实例代码，有时间，可以去参考分析一下，这里就不在赘述。

有关输入子系统的介绍就到这里，在文章一开始的时候就有介绍过，Input子系统分为三层，从下至上分别是设备驱动层，输入核心层以及事件处理层，上面的介绍是围绕着设备驱动层进行的，接下来看一下事件驱动层。核心层的文件定义在Input.c (drivers\input)文件中，这里仅仅对该文件做大概的介绍，看初始化函数：

static int \_\_init input\_init(void)

{

      int err;

      err =class\_register(&input\_class);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

注册input类，这里也就是sysfs/class/目录下生成input目录。

这里，class\_register注册类和class\_create是一样的,实际上class\_create调用的是class\_register来实现类的注册的，class\_register对应的注销函数是class\_unregister，class\_create对应的注销函数是class\_destroy

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (err) {

             pr\_err("unableto register input\_dev class\n");

             return err;

      }

      err = input\_proc\_init();

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

创建proc/bus/input文件路径，并且在该路径下devices和handlers设备文件及其他们分别对应的file\_operations结构体。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (err)

             goto fail1;

      err =register\_chrdev\_region(MKDEV(INPUT\_MAJOR, 0),  //

                                INPUT\_MAX\_CHAR\_DEVICES,"input");

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

一个字符驱动获取一个或多个设备编号，

第一个参数MKDEV(INPUT\_MAJOR, 0)：是你要分配的起始设备编号，次设备号一般为0，dev\_t是个 32 位量，其中 12位用来表示主设备号，20位用来表示次设备号。

第二个参数INPUT\_MAX\_CHAR\_DEVICES,：请求的连续设备编号的总数

第三个参数"input"：设备的名字;它会出现在 /proc/devices和 sysfs 中

假如成功返回0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      if (err) {

             pr\_err("unableto register char major %d", INPUT\_MAJOR);

             goto fail2;

      }

      return 0;

 fail2:   input\_proc\_exit();

 fail1:   class\_unregister(&input\_class);

      return err;

}

这个初始化代码比较简单，就是申请一个输入字符设备，在sysfs/class/目录下生成input目录，在proc/bus/input生成目录并且生成相应的属性文件。主要是该文件还有许多定义的代码，当用户驱动代码的时候会调用这些程序，在这里不做介绍，待用到哪个函数时在去花时间去分析它。

接下来，看一下事件处理层，事件处理层文件主要是用来支持输入设备并与用户空间交互，这部分代码一般不需要我们自己去编写，因为Linux内核已经自带有一些事件处理器，可以支持大部分输入设备，比如Evdev.c（按键等对应的处理函数）、mousedev.c（鼠标等对应的处理函数）、joydev.c（摇杆等对应的处理函数）等。对按键来说，用到的是Evdev.c文件（drivers/input）。

首先看该文件对应的初始化函数：

static int \_\_initevdev\_init(void)

{

   return input\_register\_handler(&evdev\_handler);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

这里是注册一个新的input handler，evdev\_handler就是要注册的handler，接下俩就要看一下这个时间处理（接口）结构体。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

}

对于evdev\_handler，其为input\_handler类型的结构体，先看一下input\_handler（Input.h (include\linux)）结构体：

struct input\_handler {

      void \*private;  //指向对应每一个驱动的所特有的数据

      void (\*event)(structinput\_handle \*handle, unsigned int type, unsigned int code, int value);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

输入事件向内核报告会，内核需要调用的函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      void (\*events)(structinput\_handle \*handle,

                    const struct input\_value \*vals, unsignedint count);

      bool (\*filter)(structinput\_handle \*handle, unsigned int type, unsigned int code, int value);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

类似event，负责从filters分离出一般时间处理函数

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      bool (\*match)(structinput\_handler \*handler, struct input\_dev \*dev); //比较设备和处理函数的id

      int (\*connect)(structinput\_handler \*handler, struct input\_dev \*dev, const struct input\_device\_id\*id);

// 连 接 handler 和 input\_dev

      void (\*disconnect)(structinput\_handle \*handle);

      void (\*start)(structinput\_handle \*handle);

      bool legacy\_minors;

      int minor;

      const char \*name;  //时间处理函数的名字

      const struct input\_device\_id\*id\_table;

      struct list\_head    h\_list;///用于链接和此input\_handler相关的input\_handle

      struct list\_head    node;//用于将该input\_handler链入input\_handler\_list

}

我们知道对应每一个输入设备，上报事件后最终内核用调用相应的函数去处理，在这里就是每一个input\_event都会有对应的input\_handler（可以是一对多，也可以是多对一）来处理相应的事件。那么input\_event要找到与之对应的input\_handler就需要一个东西来确定两者的对应关系，这个东西就是input\_handle。其定义在Input.h(include\linux\usb)中:

struct input\_handle {

      void \*private;  //处理者（handler）的特有数据

      int open;//记录句柄（handle）是否被打开，这样话就决定是否传递驱动事件

      const char \*name;  //由处理者（handler）给句柄（handle）创建的名字

      struct input\_dev \*dev;  //输入设别结构体

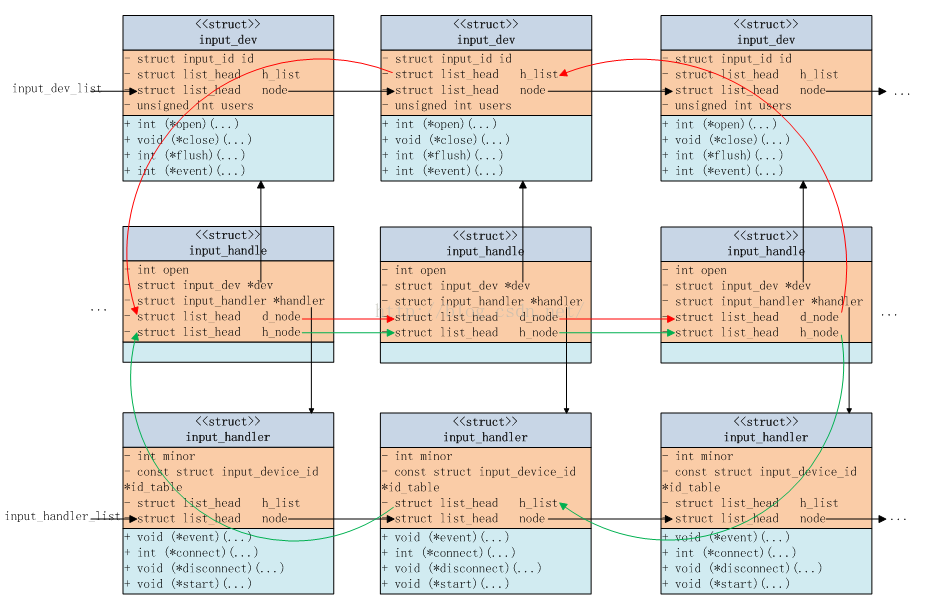
      struct input\_handler\*handler; //指向handler

      struct list\_head    d\_node; // 用于将此input\_handle链入所属input\_dev的h\_list链表

      struct list\_head    h\_node;//用于将此input\_handle链入所属input\_handler的h\_list链表

};

下面来缕缕input\_handler  input\_handle input\_dev它们三个之间的关系，input\_handler  拥有struct list\_head    d\_node（用于链接和此input\_handler相关的input\_handle）；input\_dev也有 struct list\_head    h\_list;（用于链接和此input\_handler相关的input\_handle）；相应的input\_handle拥有分别连接它们俩的structlist\_head       d\_node（用于将此input\_handle链入所属input\_dev的h\_list链表），struct list\_head    h\_node;（用于将此input\_handle链入所属input\_handler的h\_list链表）。这样的话，input\_handle 就把input\_dev和input\_handler绑定到一块了。现在的疑问是问什么它们俩不直接相互绑定，而需要一个中介input\_handle来绑定它们俩呢，这里涉及到两个方面，其一是因为一个输入设备input\_dev可能对应多个input\_handler，对调用驱动的用户来说，用户可以选择本input\_dev根据自身的需要调用哪一个input\_handler；其二，一个输入处理函数input\_handler可以对应多个输入设备input\_dev，也就是说多个input\_dev可共享一个input\_handler。下面借鉴一张图：



（图片来源：<http://www.cnblogs.com/jason-lu/articles/3155228.html>）

最后在分析一个内核源码目录的提供的申请输入设备框架程序，该例位于Documentation\input\input-programming.txt中

#include <linux/input.h>

#include <linux/module.h>

#include <linux/init.h>

#include <asm/irq.h>

#include <asm/io.h>

static struct input\_dev \*button\_dev;

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

外部中断服务函数，当按键按下时，会触发中断，该函数就是上报按键事件

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

static irqreturn\_t button\_interrupt(int irq, void \*dummy)

{

      input\_report\_key(button\_dev,BTN\_0, inb(BUTTON\_PORT) & 1);

      input\_sync(button\_dev);

      return IRQ\_HANDLED;

}

static int \_\_init button\_init(void)

{

      int error;

      if (request\_irq(BUTTON\_IRQ,button\_interrupt, 0, "button", NULL)) {

               printk(KERN\_ERR "button.c: Can't allocate irq %d\n",button\_irq);

                return -EBUSY;

        }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

申请外部中断

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

      button\_dev =input\_allocate\_device();//申请为输入设备

      if (!button\_dev) {

             printk(KERN\_ERR"button.c: Not enough memory\n");

             error = -ENOMEM;

             goto err\_free\_irq;

      }

      button\_dev->evbit[0] =BIT\_MASK(EV\_KEY);

      button\_dev->keybit[BIT\_WORD(BTN\_0)]= BIT\_MASK(BTN\_0);

      error = input\_register\_device(button\_dev); //注册为输入设备

      if (error) {

             printk(KERN\_ERR"button.c: Failed to register device\n");

             goto err\_free\_dev;

      }

      return 0;

 err\_free\_dev:

      input\_free\_device(button\_dev);

 err\_free\_irq:

      free\_irq(BUTTON\_IRQ,button\_interrupt);

      return error;

}

static void \_\_exit button\_exit(void)

{

       input\_unregister\_device(button\_dev);

      free\_irq(BUTTON\_IRQ,button\_interrupt);

}

module\_init(button\_init);

module\_exit(button\_exit);