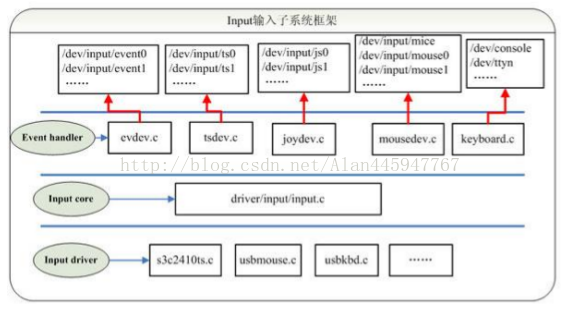
<http://blog.csdn.net/alan445947767/article/details/41047757>

### Linux Input 子系统

什么是 Linux Input 子系统呢?顾名思义,就是输入子系统。这不是废话吗?说了等于没说。换句话说,Linux 系统下把键盘、鼠标、触摸屏、游戏句柄等等设备都当做是输入设备。作为一个输入设备,好处可大了。因为 Linux 输入子系统已经帮输入设备完成很大一部分通用的处理功能了。就比如说,以前我们写字符设备驱动时的读写函数,这些都不用驱动工程师去编写了,输入子系统已经帮我们写好了。这种思想,在Linux内核多了去呢!

#### Input 子系统层次框架

输入(Input)子系统是分层架构的,总共分为 3 层,从上到下分别是:事件处理层(Event Handler)、输入子系统核心层(Input Core)、硬件驱动层(Input Driver)。如图 7.1:



(1) 硬件驱动层负责操作具体的硬件设备,这层的代码是针对具体的驱动程序的,比如你的设备是触摸输入设备,还是鼠标输入设备,还是键盘输入设备,这些不同的设备,自然有不同的硬件操作,驱动工程师往往只需要完成这层的代码编写。

(2) 输入子系统核心层是链接其他两层之间的纽带与桥梁,向下提供硬件驱动层的接口,向上提供事件处理层的接口。

(3) 事件处理厂负责与用户程序打交道,将硬件驱动层传来的事件报告给用户程序。

各层之间通信的基本单位就是事件,任何一个输入设备的动作都可以抽象成一种事件,如键盘的按下,触摸屏的按下,鼠标的移动等。事件有三种属性:**类型(type),编码(code),值(value),**Input 子系统支持的所有事件都定义在 input.h中,包括所有支持的类型,所属类型支持的编码等。事件传送的方向是 硬件驱动层-->子系统核心-->事件处理层-->用户空间。

#### 三个重要的结构体

##### 输入设备(input\_dev)

Linux 系统里,使用 input\_dev 结构体抽象一个输入设备,编写硬件驱动就是主要围绕这个结构体而进行的。由于它的成员比较多,这里只挑最重点的讲解。其他成员请参考 include/linux/input.h

struct input\_dev {

const char \*name;

/\* 标识设备驱动特征,如总线类型、生产厂商、产品类型、版本 \*/

struct input\_id id;

/\* 表示能产生哪类事件 \*/

unsigned long evbit[NBITS(EV\_MAX)];

/\* 表示能产生哪些按键 \*/

unsigned long keybit[NBITS(KEY\_MAX)];

/\* 表示能产生哪些相对位移事件, x,y,滚轮 \*/

unsigned long relbit[NBITS(REL\_MAX)];

/\* 表示能产生哪些绝对位移事件, x,y \*/

unsigned long absbit[NBITS(ABS\_MAX)];

struct device dev;

/\* 用来链接他所支持的 input\_handle 结构,然后用

\* input\_handle 找到里面的 input\_handler

\*/

struct list\_head h\_list;

/\* 链接到 input\_handler\_list,这个链表

\* 链接了所有注册到内核的事件处理器

\*/

struct list\_head node;

...

}

input\_dev 结构体里有几个数组也是非常重要的,都是我们写硬件驱动需要进行设置的,所以我们务必认识一下它们。

(1) evbit[BITS\_TO\_LONGS(EV\_CNT)]数组,这个数组以位掩码的形式,代表了这个设备支持哪类事件,比如:

#define EV\_SYN 0x00 //同步类

#define EV\_KEY 0x01 //按键类

#define EV\_REL 0x02 //相对位移类

#define EV\_ABS 0x03 //绝对位移类

#define EV\_MSC 0x04

#define EV\_SW 0x05

#define EV\_LED 0x11

#define EV\_SND 0x12

#define EV\_REP 0x14 //重复类

#define EV\_FF 0x15

#define EV\_PWR 0x16

#define EV\_FF\_STATUS 0x17

#define EV\_MAX 0x1f

#define EV\_CNT (EV\_MAX+1)

(2) keybit[BITS\_TO\_LONGS(KEY\_CNT)]数组,这个数组也是以位掩码的形式,代表这个设备支持哪些按键,比如:

#define KEY\_ESC 1

#define KEY\_1 2

#define KEY\_2 3

#define KEY\_3 4

#define KEY\_TAB 15

#define KEY\_ENTER 28

#define KEY\_A 30

#define KEY\_B 48

#define KEY\_C 46

......

(3) relbit[BITS\_TO\_LONGS(REL\_CNT)]数组,这个数组也是以位掩码的形式,代表这个设备支持哪些相对位移事件,比如:

#define REL\_X 0x00

#define REL\_Y 0x01

#define REL\_Z 0x02

#define REL\_WHEEL 0x08 //滚轮

......

(4) absbit[BITS\_TO\_LONGS(ABS\_CNT)]数组,这个数组也是以位掩码的形式,代表这个设备支持哪些绝对位移事件,比如:

#define ABS\_X 0x00

#define ABS\_Y 0x01

##### 事件处理器(input\_handler)

input\_handler 属于输入子系统三层中的事件处理层的一个重要结构体。主要成员有:其他成员请参考 include/linux/input.h

struct input\_handler {

/\* 当事件处理器接收到了来自 input 设备传来的

\* 事件时调用的处理函数,负责处理事件。

\*/

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

/\* 当一个 input 设备注册到内核的时候被调用,将事件处理器与输入设备

\* 联系起来的函数,也就是将 input\_dev 和 input\_handler 配对的函数。

\*/

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

/\* 与 connect 相反 \*/

void (\*disconnect)(struct input\_handle \*handle);

/\* 文件操作集,因为事件处理器要完成读写功能 \*/

const struct file\_operations \*fops;

/\* 事件处理器所支持的 input 设备 \*/

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

/\* 链接他所支持的 input\_handle 结构,然后用

\* input\_handle 找到里面的 input\_dev

\*/

struct list\_head h\_list;

/\* 链接到 input\_handler\_list,这个链表

\* 链接了所有注册到内核的事件处理器

\*/

struct list\_head node;

};

##### 事件沟通者(input\_handle)

之所以称 input\_handle(注意了,不是事件处理器 input\_handler)为事件沟通者,是因为它代表一个成功配对的 input\_dev 和 input\_handler。主要成员有:其他成员请参考 include/linux/input.h

struct input\_handle {

/\* 每个配对的事件处理器都会分配一个对应的设备结构,

\* 如 evdev 事件处理器的 evdev 结构,注意这个结构与

\* 设备驱动层的 input\_dev 不同,初始化 handle 时,保存到这里。

\*/

void \*private;

/\* 指向 input\_dev 结构体实例 \*/

struct input\_dev \*dev;

/\* 指向 input\_handler 结构体实例 \*/

struct input\_handler \*handler;

/\* input\_handle 通过 d\_node 连接到了 input\_dev 上的 h\_list 链表上 \*/

struct list\_head d\_node;

/\* input\_handle 通过 h\_node 连接到了 input\_handler 的 h\_list 链表 \*/

struct list\_head h\_node;

};

##### 三个结构体之间的关系

input\_dev 是硬件驱动层,代表一个 input 设备。通过全局的 input\_dev\_list 链接在一起。设备注册的时候实现这个操作。

input\_handler 是事件处理层,代表一个事件处理器。通过全局的 input\_handler\_list 链接在一起。事件处理器注册的时候实现这个操作(事件处理器一般内核自带,一般不需要我们来写)

input\_handle 个人认为属于核心层,代表一个配对的 input 设备与 input 事件处理器。它没有一个全局的链表,它注册的时候将自己分别挂在了 input\_dev 和input\_handler 的 h\_list 上了。通过 input\_dev 和 input\_handler 就可以找到input\_handle 在设备注册和事件处理器, 注册的时候都要进行配对工作,配对后就会实现链接。通过 input\_handle 也可以找到 input\_dev 和 input\_handler。

##### Input 子系统核心层分析

前面我们说过,子系统核心的功能是:向下提供硬件驱动层的接口,向上提供事件处理层的接口。子系统的核心文件在 drivers/input/input.c,这文件代码量有 2000+行,我们也不可能每一个函数都去分析一遍,只把重点抓出来。前面讲的三个重要的数据结构都会对应一个注册函数,他们都定义在子系统核心的 input.c 文件中。

主要有三个注册函数:

/\* 向内核注册一个 input 设备 \*/

int input\_register\_device(struct input\_dev \*dev)

/\* 向内核注册一个事件处理器 \*/

int input\_register\_handler(struct input\_handler \*handler)

/\* 向内核注册一个 handle 结构 \*/

int input\_register\_handle(struct input\_handle \*handle)

###### input\_register\_device

/\* 详细请参考 drivers/input/input.c \*/

int input\_register\_device(struct input\_dev \*dev)

{

struct input\_handler \*handler;

...

/\* 设置输入设备支持同步类事件 \*/

\_\_set\_bit(EV\_SYN, dev->evbit);

/\* 将 device 加入到 linux 设备模型中去 \*/

device\_add(&dev->dev);

...

/\* 把 input\_dev 放入 input\_dev\_list 链表 \*/

list\_add\_tail(&dev->node, &input\_dev\_list);

...

/\* 对于 input\_handler\_list 链表的每一项,都调用 input\_attach\_handler

\* 它根据 input\_handler 的 id\_table 判断能否支持这个 input\_dev

\*/

list\_for\_each\_entry(handler, &input\_handler\_list, node)

input\_attach\_handler(dev, handler);

...

}

input\_register\_device 完成的主要功能就是:初始化一些默认的值,将自己的 device 结构添加到 linux 设备模型当中,将 input\_dev 添加到 input\_dev\_list

链表中,对于 input\_handler\_list 链表的每一项,都调用 input\_attach\_handler它根据 input\_handler 的 id\_table 判断能否支持这个输入设备。

###### input\_attach\_handler

/\* 详细请参考 drivers/input/input.c \*/

static int input\_attach\_handler(struct input\_dev \*dev,struct input\_handler \*handler)

{

const struct input\_device\_id \*id;

int error;

/\* 根据 input\_handler 的 id\_table 判断能否支持这个输入设备 \*/

id = input\_match\_device(handler, dev);

if (!id)

return -ENODEV;

/\* 若支持,则调用 input\_handler 的 connect 函数,建立连接

\* 对于 evdev.c 里,则调用 evdev\_connect 函数

\*/

error = handler->connect(handler, dev, id);

if (error && error != -ENODEV)

pr\_err("failed to attach handler %s to device %s, error: %d\n",

handler->name, kobject\_name(&dev->dev.kobj), error);

return error;

}

###### evdev\_connect

evdev\_connect 函数在 drivers/input/evdev.c 文件里实现。对于 connect 函数,每种事件处理器的实现都有差异,但原理都相同,按键、触摸屏用的事件处理器都为 evdev,下面分析 evdev 的 connect 函数 evdev\_connect。

static int evdev\_connect(struct input\_handler \*handler,struct input\_dev \*dev, const struct input\_device\_id \*id)

{

struct evdev \*evdev;

...

/\* 分配一个 evdev 结构体,它里面包括 input\_handle 结构体 \*/

evdev = kzalloc(sizeof(struct evdev), GFP\_KERNEL);

...

/\* 设备节点,如/dev/event0 \*/

dev\_set\_name(&evdev->dev, "event%d", dev\_no);

...

/\* handle 里的 input\_dev 指向的实例化的 input\_dev \*/

evdev->handle.dev = input\_get\_device(dev);

evdev->handle.name = evdev->name;

/\* handle 里的 input\_handler 指向的实例化的 input\_handler \*/

evdev->handle.handler = handler;

evdev->handle.private = evdev;

......

/\* 注册 \*/

input\_register\_handle(&evdev->handle);

/\* 初始化 cdev \*/

cdev\_init(&evdev->cdev, &evdev\_fops);

evdev->cdev.kobj.parent = &evdev->dev.kobj;

/\* 注册字符设备 \*/

error = cdev\_add(&evdev->cdev, evdev->dev.devt, 1);

...

}

###### input\_register\_handle

/\* 详细请参考 drivers/input/input.c \*/

int input\_register\_handle(struct input\_handle \*handle)

{

struct input\_handler \*handler = handle->handler;

struct input\_dev \*dev = handle->dev;

...

/\* 把 handle->d\_node 添加到 dev->h\_list

\* 这样,就可以从 dev->h\_list 找到 handle,进而找到 handler

\*/

list\_add\_tail\_rcu(&handle->d\_node, &dev->h\_list);

...

/\* 把 handle->h\_node 添加到 handler->h\_list

\* 这样,就可以从 handler->h\_list 找到 handle,进而找到 dev

\*/

list\_add\_tail(&handle->h\_node, &handler->h\_list);

...

return 0;

}

input\_register\_handle 比较简单,主要把 handle 结构体通过 d\_node 链表项链接到 input\_dev 的 h\_list;把 handle 结构体通过 h\_node 链表项链接到

input\_handler 的 h\_list。这样一来,就可以通过 input\_dev 的 h\_list 找到 handle,进而找到 handle里的 handler;或者通过 input\_handler 的 h\_list 找到 handle,进而找到 handle里的 input\_dev。

###### input\_register\_handler

/\* 详细请参考 drivers/input/input.c \*/

int input\_register\_handler(struct input\_handler \*handler)

{

struct input\_dev \*dev;

......

INIT\_LIST\_HEAD(&handler->h\_list);

/\* 将 handler 放入 input\_handler\_list 链表 \*/

list\_add\_tail(&handler->node, &input\_handler\_list);

/\* 对于每个 input\_dev,调用 input\_attach\_handler(前面讲过)

\* 它根据 input\_handler 的 id\_table 判断能否支持这个输入设备

\* 如果支持,则调用 handler->connect 函数,建立“连接”

\*/

list\_for\_each\_entry(dev, &input\_dev\_list, node)

input\_attach\_handler(dev, handler);

......

return 0;

}

###### input 子系统接口总结

再一次强调 Input 子系统核心的功能是:向下提供硬件驱动层的接口,向上提供事件处理层的接口。

向下对硬件驱动层的接口主要有:input\_allocate\_device 这个函数主要是分配一个 input\_dev 输入设备接口,并初始化一些基本的成员,这就是我们不能简单用 kmalloc 分配 input\_dev 结构的原因,因为缺少了一些初始化。input\_register\_device 注册一个 input 设备,input\_unregister\_device 注销一个 input 设备。input\_event()函数是硬件驱动层向 input 子系统核心报告事件的函数,这 4 个函数一般用在驱动工程师需要完成的具体硬件驱动程序中。

向上对事件处理层接口主要有:input\_register\_handler 注册一个事件处理器,input\_register\_handle注册一个 input\_handle 结构,它将输入设备和事件处理器联系起来。

##### 事件处理层分析

进入 drivers/input/目录可以发现,Linux 系统中有很多事件处理器, joydev、如mousedev、evdev 等等。因为 evdev 事件处理器可以处理所有的事件,按键、

触摸屏设备驱动用的时间处理器就是 evdev。比较通用,我们这里以 evdev 为例,分析事件处理器做了什么。

还记得前面说的吗?事件处理层与用户程序和输入子系统核心打交道,是他们两层的桥梁。如果忘了,就回去看看图 7.1 呗~~先看它的入口函数(参考 drivers/input/evdev.c):

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

{

/\* 向内核注册 evdev\_handler \*/

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

}

evdev\_handler 是一个 input\_handler 型结构体,它代表一个事件处理器,既然是一个事件处理器,当然有重要的成员,我们来看看。

static struct input\_handler evdev\_handler = {

.event= evdev\_event,

/\* 用来处理事件 \*/

.events= evdev\_events,

/\* 用来连接 input\_dev 和 input\_handler \*/

.connect= evdev\_connect,

.disconnect= evdev\_disconnect,

.legacy\_minors = true,

.minor= EVDEV\_MINOR\_BASE,

.name= "evdev",

/\* 表示 evdev\_handler 能够支持哪一些输入设备 \*/

.id\_table= evdev\_ids,

};

前面我们说过, 一个事件 处理器能不能支持一个输入设备,主要通过input\_attach\_handler 函数来判定,它根据 input\_handler 的 id\_table 判断能否支持这个 input\_dev。那我们来看看 evdev\_handler 的 id\_table:

static const struct input\_device\_id evdev\_ids[] = {

{

.driver\_info = 1 }, /\* Matches all devices \*/

{ },/\* Terminating zero entry \*/

}

evdev\_ids 没有定义 flags,也没有定义匹配属性值。这个 evdev\_ids 的意思就是:evdev\_handler 可以匹配所有 input\_dev 设备,也就是所有的 input\_dev 发

出的事件,都可以由 evdev\_handler 来处理。

###### evdev\_connect

evdev\_connect 函 数 在 前 面 已 经 分 析 过 了 , 这 里 简 单 讲 述 一 下 。 当input\_register\_handler 向内核注册事件处理器时,就会对于每个 input\_dev,调

用 input\_attach\_handler(前面讲过) ,它根据 input\_handler 的 id\_table 判断能否支持这个输入设备如果支持,则调用 handler->connect 函数,建立“连接” 。

而对于 evdev.c 里,则调用 evdev\_connect 函数。evdev\_connect 主要工作是:分配设置一个包含 input\_handle 的 evdev 结构,关联 input\_dev 和input\_handler,向内核注册一个 handle,最后注册一个主设备号为 13,file\_operations 为 evdev\_fops 的字符设备。这样不就又回到字符设备的那套框架了?

static const struct file\_operations evdev\_fops = {

.owner= THIS\_MODULE,

.read= evdev\_read,

.write= evdev\_write,

.poll= evdev\_poll,

.open= evdev\_open,

.release= evdev\_release,

.unlocked\_ioctl = evdev\_ioctl,

#ifdef CONFIG\_COMPAT

.compat\_ioctl= evdev\_ioctl\_compat,

#endif

.fasync= evdev\_fasync,

.flush= evdev\_flush,

.llseek= no\_llseek,

};

后面写硬件驱动时,你会发现并没有编写相关的读写函数,是因为事件处理层已经帮我们做好了,只需要硬件设备把产生的事件传递给事件处理层就可以了,

事件处理层会帮我们完成。这就大大减少了驱动工程师的工作量啦。其实,除了输入子系统,Linux 内核还有很多利用这种思想的。比如:总线设备驱动模型,

I2C 模型等等。

###### evdev\_open

当事件处理器找到支持的输入设备时,就会建立连接,然后注册一个主设备号为 13,file\_operations 为 evdev\_fops 的字符设备。用户程序可以通过设备节点/dev/input/event0,/dev/input/event1 来访问输入设备。比如用户程序使用open(“/dev/input/event0”,RDWR),最终就会调用到 evdev\_open 函数。在分析 evdev\_open 函数前,我们来看两个重要的结构。

/\* 参考 drivers/input/evdev.c \*/

struct evdev {

/\* 打开标志 \*/

int open;

/\* 通过 handle 可以找到输入设备和事件处理器 \*/

struct input\_handle handle;

/\* 等待队列,当进程读取设备,而没有事件

\* 产生的时候,进程将在队列里休眠

\*/

wait\_queue\_head\_t wait;

/\* 强制绑定的 evdev\_client 结构 \*/

struct evdev\_client \_\_rcu \*grab;

/\* evdev\_client 链表,这说明一个 evdev 设备可以处理

\* 多个 evdev\_client,可以有多个进程访问 evdev 设备

\*/

struct list\_head client\_list;

spinlock\_t client\_lock; /\* protects client\_list \*/

struct mutex mutex;

struct device dev;

/\* 字符设备结构 \*/

struct cdev cdev;

bool exist;

};

/\* 参考 drivers/input/evdev.c \*/

struct evdev\_client {

/\* input\_event 代表一个事件,基本成员:

\* 类型(type)

,编码(code)

,值(value)

\*/

struct input\_event buffer[];

/\* 针对 buffer 数组的索引 \*/

unsigned int head;

/\* 针对 buffer 数组的索引,当 head 与 tail 相等的时候,说明没有事件 \*/

unsigned int tail;

unsigned int packet\_head;

spinlock\_t buffer\_lock;

/\* 异步通知函数 \*/

struct fasync\_struct \*fasync;

/\* evdev 设备 \*/

struct evdev \*evdev;

/\* evdev\_client 链表项 \*/

struct list\_head node;

int clkid;

unsigned int bufsize;

};

evdev\_client 这个结构在进程打开 event0 设备的时候调用 evdev 的 open方法,在 open 中创建这个结构,并初始化。

static int evdev\_open(struct inode \*inode, struct file \*file)

{

struct evdev \*evdev = container\_of(inode->i\_cdev,struct evdev, cdev);

unsigned int bufsize =evdev\_compute\_buffer\_size(evdev->handle.dev);

struct evdev\_client \*client;

int error;

/\* 分配并初始化 evdev\_client 结构 \*/

client = kzalloc(sizeof(struct evdev\_client) +bufsize \* sizeof(struct input\_event),GFP\_KERNEL);

...

client->bufsize = bufsize;

spin\_lock\_init(&client->buffer\_lock);

/\* 使 client 与 evdev 设备相关联 \*/

client->evdev = evdev;

/\* 把 client 挂到 evdev->client\_list \*/

evdev\_attach\_client(evdev, client);

/\* 当设备是第一次被打开时,则调用 input\_open\_device 函数 \*/

error = evdev\_open\_device(evdev);

...

/\* 把 evdev\_client 放到文件的私有数据,

\* 以备其他函数取出,如 evdev\_read

\*/

file->private\_data = client;

...

return 0;

}

###### evdev\_open\_device

/\* 参考 drivers/input/evdev.c \*/

static int evdev\_open\_device(struct evdev \*evdev)

{

int retval;

/\* 获得互斥锁 \*/

retval = mutex\_lock\_interruptible(&evdev->mutex);

if (retval)

return retval;

/\* 检查设备是否存在 \*/

if (!evdev->exist)

retval = -ENODEV;

/\* 判断设备是否已经被打开?如果没有打开,

\* 则调用 input\_open\_device(&evdev->handle)

\* 打开 evdev 对应的 handle;否则不做任何操作返回

\*/

else if (!evdev->open++) {

retval = input\_open\_device(&evdev->handle);

if (retval)

evdev->open--;

}

mutex\_unlock(&evdev->mutex);

return retval;

}

###### input\_open\_device

/\* 参考 drivers/input/input.c

\*

\* This function should be called by input handlers when they

\* want to start receive events from given input device.

\*/

int input\_open\_device(struct input\_handle \*handle)

{

/\* 从 handle 中获得 input\_dev 结构体 \*/

struct input\_dev \*dev = handle->dev;

/\* 递增 handle 的打开计数 \*/

handle->open++;

/\* 如果是第一次打开,并且 dev 存在 open 函数,

\* 则调用 input\_dev 的 open 函数

\*/

if (!dev->users++ && dev->open)

retval = dev->open(dev);

...

}

###### evdev\_read

当用户程序 read 时,最终会调用到 evdev\_read。

static ssize\_t evdev\_read(struct file \*file, char \_\_user \*buffer,size\_t count, loff\_t \*ppos)

{

/\* 这个客户端结构在打开的时候分配并保存在 file->private\_data 中 \*/

struct evdev\_client \*client = file->private\_data;

struct evdev \*evdev = client->evdev;

struct input\_event event;

size\_t read = 0;

int error;

/\* 用户进程每次读取设备的字节数,不要少于 input\_event 结构的大小 \*/

if (count != 0 && count < input\_event\_size())

return -EINVAL;

for (;;) {

if (!evdev->exist)

return -ENODEV;

/\* 如果缓存中没有数据可读而设备又存在,

\* 并且被设置为 O\_NONBLOCK 方式,则退出

\*/

if (client->packet\_head == client->tail &&(file->f\_flags & O\_NONBLOCK))

return -EAGAIN;

/\*

\* count == 0 is special - no IO is done but we check

\* for error conditions (see above).

\*/

if (count == 0)

break;

while (read + input\_event\_size() <= count &&evdev\_fetch\_next\_event(client, &event)) {

/\* 最终使用 copy\_to\_user 函数读取数据 \*/

if (input\_event\_to\_user(buffer + read, &event))

return -EFAULT;

read += input\_event\_size();

}

if (read)

break;

/\* 如果设备没有被设为 O\_NONBLOCK,并且缓存中没有数据可读

\* 而设备又存在,就调用 wait\_event\_interruptible 休眠

\*/

if (!(file->f\_flags & O\_NONBLOCK)) {

error = wait\_event\_interruptible(evdev->wait,client->packet\_head != client->tail ||!evdev->exist);

if (error)

return error;

}

}

return read;

}

evdev\_fops 的其他函数成员也是这么分析,这里就不继续分析下去了。有兴趣的读者,请自行阅读更多的源码。

##### 硬件驱动层分析

Linux 系统自带了很多类型的输入设备驱动,打开 drivers/input/目录,确实可以看到有很多类型的输入设备,比如:keyboard、mouse、joystick、touchscreen等等。进入这些目录后,发现还有很多相应类型下的具体设备,比如 keyboard目录下有 atkbds.c、 mcs\_touchkey.c、gpio\_keys.c 文件,这些文件都是某一种输入设备类型下的具体的输入设备。这里,我们以 gpio\_keys.c 文件来分析,它是 Linux 系统自带的通用的,基于 GPIO 的按键驱动程序。先看它的入口函数(参考 drivers/input/keyboard/gpio\_keys.c):

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

{

/\* 注册一个平台驱动 \*/

return platform\_driver\_register(&gpio\_keys\_device\_driver);

}

在 gpio\_keys\_init 入口函数里,注册了一个平台驱动,关于什么是平台驱动,将在第八章学习,这里可以暂且跳过,暂且不需要理会它。这个平台驱动注册函数需要一个 platform\_driver 型的参数,它是一个结构体。 在这里,注册 的platform\_driver 是这样的 driver:

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),

}

};

如 果内核中 存在有 与 gpio-keys 同 名的平台设备的话,最终 就会调用gpio\_keys\_probe 函数。这里你暂且记住这句话,以后你将会看到大量类似的调用,因为 Linux 内核大量的使用了总线设备驱动模型,这里暂且先说一下,后面章节再详细分析。

###### gpio\_keys\_probe

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

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

{

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

/\* 设备相关的信息会保存在 pdev->dev.platform\_data 中 \*/

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

struct gpio\_keys\_drvdata \*ddata;

struct input\_dev \*input;

int i, error;

int wakeup = 0;

...

/\* 为驱动数据 ddata 开辟一段内存空间,并清零 \*/

ddata = kzalloc(sizeof(struct gpio\_keys\_drvdata) +pdata->nbuttons \* sizeof(struct gpio\_button\_data),GFP\_KERNEL);

/\* 分配设置 input\_dev 结构体 \*/

input = input\_allocate\_device();

...

/\*把 ddata 保存到 pdev->dev->p->driver\_data,供其他函数中使用

\*(比如:remove),通过 platform\_get\_drvdata 函数来获取。

\*/

platform\_set\_drvdata(pdev, ddata);

/\* 把驱动数据 ddata 以私有数据形式存放在

\* 输入设备 input 中,以备日后使用

\*/

input\_set\_drvdata(input, ddata);

/\* 初始化 input\_dev 结构体的部分成员,加载驱动后可以

\* 通过 cat /proc/bus/input/devices 命令查看

\*/

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

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

input->dev.parent = &pdev->dev;

input->open = gpio\_keys\_open;

input->close = gpio\_keys\_close;

input->id.bustype = BUS\_HOST;

input->id.vendor = 0x0001;

input->id.product = 0x0001;

input->id.version = 0x0100;

...

/\* 为每个按键进行初始化,及设置属性 \*/

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

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

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

/\* 初始化按键,后面分析 \*/

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

}

...

/\* 注册输入设备 input\_dev,把输入设备加到输入设备链表中,

\* 并寻找合适的 input\_handle 中的 handler 与 input\_handler 配对

\*/

error = input\_register\_device(input);

return 0;

...

}

gpio\_keys\_probe 函数的代码量将近 100 多行,为了便于分析输入子系统中硬件驱动层主要工作是什么,我们只抓重点分析。详细的请看源码,源码里也有详细的注释。gpio\_keys\_probe 函数的主要工作是:

(1) 分配设置一个 input\_dev 结构体,这个结构体前面说过了,它代表一个输入设备。如果你已经忘了,请回头看 7.1.2.1 小节的内容。

(2) 获取平台数据

(3) 初始化部分非必要的 input\_dev 结构体成员,剩下重要的 input\_dev 结构体成员在后面设置。

(4) 初始化按键

(5) 向内核注册一个 input\_dev

###### gpio\_keys\_setup\_key

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

static int gpio\_keys\_setup\_key(struct platform\_device \*pdev,struct input\_dev \*input,struct gpio\_button\_data \*bdata,const struct 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;

....

/\* 判断按键所对应的 gpio 是否有效 \*/

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

/\* 向系统申请 gpio \*/

error = gpio\_request\_one(button->gpio, GPIOF\_IN, desc);

...

/\* 提取 gpio 所对应的外部中断 \*/

irq = gpio\_to\_irq(button->gpio);

...

bdata->irq = irq;

/\* 初始化工作队列,在 bdata->work 的工作队列中

\* 增加一个任务 gpio\_keys\_gpio\_work\_func

\*/

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

/\* 设置定时器,当定时器的时间到了,就执行 gpio\_keys\_gpio\_timer

\* 回调函数,回调函数调度工作队列 schedule\_work(&bdata->work),

\* 最终调用 gpio\_keys\_gpio\_work\_func

\*/

setup\_timer(&bdata->timer,gpio\_keys\_gpio\_timer, (unsigned long)bdata);

/\* 设置按键中断处理函数为 gpio\_keys\_gpio\_isr \*/

isr = gpio\_keys\_gpio\_isr;

/\* 设置外部中断上升沿或下降沿触发 \*/

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

}

...

/\* 设置这个输入设备支持按键类(EV\_KEY)事件 \*/

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

...

/\* 分配一条中断线,即当按键中断(bdata->irq)发生时,

\* 进入中断函数(isr,即 gpio\_keys\_irq\_isr 函数内)

\*/

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

...

return 0;

...

}

gpio\_keys\_setup\_key 这个函数的工作内容就是完成一系列的硬件初始化工作,主要有:

(1) 判断 GPIO 引脚是否有效,有效的话获取 GPIO 对应的外部中断号。

(2) 初 始 化 工 作 队 列 , 在 bdata->work 的 工 作 队 列 中 增 加 一 个 任 务gpio\_keys\_gpio\_work\_func。

(3) 定义设置一个定时器,定时器中断处理函数为 gpio\_keys\_gpio\_timer

(4) 定义按键中断处理函数:gpio\_keys\_gpio\_isr

(5) 设置 input\_dev,让它支持按键类事件。

(6) 分配一条中断线,即当按键中断(bdata->irq)发生时,进入中断函数(isr,即 gpio\_keys\_irq\_isr 函数内)

###### gpio\_keys\_gpio\_isr

当有按键事件发生时,就会进入 gpio\_keys\_gpio\_isr 按键中断处理函数

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

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

{

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

...

/\* 如果有设置按键去抖动,则修改定时器时间 \*/

if (bdata->timer\_debounce)

mod\_timer(&bdata->timer,jiffies + msecs\_to\_jiffies(bdata->timer\_debounce));

/\* 否则执行调度队列 \*/

else

schedule\_work(&bdata->work);

return IRQ\_HANDLED;

}

在 后 面 的 移 植 中 , 并 没 有 设 置 去 抖 动 标 志 , 也 就 是 没 有 设 置 struct gpio\_button\_data \*bdata 结 构 体 中 的 timer\_debounce , 所 以 不 会 调 用mod\_timer 修改定时器时间。暂且认为,这里就调用 schedule\_work 函数,它的作用是将 gpio\_keys\_gpio\_work\_func 挂在工作队列上,利用它来上报事件。

###### gpio\_keys\_gpio\_timer

假设定时时间到达,就会调用 gpio\_keys\_gpio\_timer 函数。

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

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

{

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

schedule\_work(&bdata->work);

}

它也是调用 schedule\_work 函数,它的作用是将 gpio\_keys\_gpio\_work\_func挂在工作队列上,利用它来上报事件。也就说,当定时时间到了,它就会调用gpio\_keys\_gpio\_work\_func 函数进行上报事件。跟 gpio\_keys\_gpio\_isr 按键中断处理函数的作用是一样的。显然,这里是多余的。因为我们在移植按键平台设备时,没有设置去抖动标志,也就相当于定时器是无效的。如果你想实现定时器去抖动的话,就需要设置 bdata->timer\_debounce。

###### gpio\_keys\_gpio\_work\_func

当有按键事件发生时,就会触发按键中断,从而进入 gpio\_keys\_gpio\_isr 函数,由于没有设置 bdata->timer\_debounce 去抖动标志,所以就直接调用了

schedule\_work(&bdata->work)。又因为前面 gpio\_keys\_setup\_key 函数里初始化了一个 bdata->work 工作队列:

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

并 将 gpio\_keys\_gpio\_work\_func 放 入 了 工 作 队 列 , 所 以 当 有 人 调 用schedule\_work(&bdata->work)时,就相当于调用 gpio\_keys\_gpio\_work\_func函数。

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

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

{

/\* container\_of 实现了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的功能。

\*/

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

/\* 向系统上报事件 \*/

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

if (bdata->button->wakeup)

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

}

###### gpio\_keys\_gpio\_report\_event

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

{

const struct gpio\_keys\_button \*button = bdata->button;

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

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

if()

...

/\* 把按键类输入事件上报给系统 \*/

else {

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

}

/\* 用于事件的同步,即告知系统设备已传递完一组完整的数据 \*/

input\_sync(input);

}

gpio\_keys\_gpio\_report\_event 函数使用输入子系统核心提供的接口函数input\_event 来上报事件,input\_sync 其实也是调用 input\_event 来实现的。

###### 事件传递过程分析

以 gpio\_keys 为例子,它是 Linux 系统自带的通用的,基于 GPIO 的按键驱动程序。evdev 处理器支持这个输入设备,结合 evdev 来大概梳理一下事件的传递过程。

当用户程序打开/dev/event0 后,接着 read 这个输入设备,最终就会调用到事件处理层的 evdev\_read 函数,在 7.3.5 小节已经将过。如果没有数据可读进程就会休眠,什么时候被唤醒呢?有事件发生的时候才会唤醒进程。说到事件,那自然是硬件设备产生的。  
当按键被按下或松开时,就会进入到硬件设备驱动层,以 gpio\_key 为例,它 就 会 进 入 到 gpio\_keys\_gpio\_isr 按 键 中 断 处 理 函 数 , 它 最 终 调 用gpio\_keys\_gpio\_work\_func 函数,进而使用 gpio\_keys\_gpio\_report\_event 来上报 事 件 , 而 gpio\_keys\_gpio\_report\_event 利 用 了 输 入 子 系 统 核 心 提 供 的input\_event 函数来上报事件。好啦,接着我们用函数流程来看看 input\_event是怎么上报事件的。

input\_event  
   -->input\_handle\_event  
        -->input\_pass\_values  
            -->input\_to\_handler  
                 -->handler->events(handle, vals, count);或者  
                 -->handler->event(handle, v->type, v->code, v->value);

看到最后一个,是事件处理器 handler 里的 events 函数。对于 gpio\_keys这个输入设备对应的事件处理器是 evdev。换句话说,input\_event 函数最终调用到 evdev.c 里的 evdev\_events 函数来上报事件。

###### evdev\_events

/\* 参考 drivers/input/keyboard/gpio\_keys.c

\* Pass incoming events to all connected clients.

\*/

static void evdev\_events(struct input\_handle \*handle,const struct input\_value \*vals, unsigned int count)

{

struct evdev \*evdev = handle->private;

struct evdev\_client \*client;

...

/\* 在 evdev\_pass\_values 函数里把在 evdev\_read 时休眠的进程唤醒 \*/

if (client)

evdev\_pass\_values(client, vals, count, time\_mono, time\_real);

else

list\_for\_each\_entry\_rcu(client, &evdev->client\_list, node)

evdev\_pass\_values(client, vals, count,

time\_mono, time\_real);

...

}

###### evdev\_pass\_values

/\* 参考 drivers/input/keyboard/gpio\_keys.c \*/

static void evdev\_pass\_values(struct evdev\_client \*client,const struct input\_value \*vals, unsigned int count,ktime\_t mono, ktime\_t real)

{

struct evdev \*evdev = client->evdev;

const struct input\_value \*v;

struct input\_event event;

...

/\* 把在 evdev\_read 时休眠的进程唤醒 \*/

wake\_up\_interruptible(&evdev->wait);

}

唤醒进程后,evdev\_read 函数把事件(数据)通过 input\_event\_to\_user 函数传递给用户程序。而 input\_event\_to\_user 正是调用了 copy\_to\_user 将事件(数据)从内核空间拷贝到用户空间。这样,用户程序就读取到了按键所产生的动作。

#### 移植 gpio\_key

前面说了那么多知识,把 Linux 内核的 Input 子系统说的那么牛叉,那是不是不用怎么移植就能够用上了呢?对,没错。linux 系统提供了很好的支持,只要把按键对应的 IO 端口配置好,按键就可以工作了。

如果你学习过本教程的内核移植篇,那么你对下面这个文件一定不会陌生!

##### 添加按键平台设备

arch/arm/mach-s5pv210/mach-smdkv210.c

在 smdkv210\_devices 结构体指针数组里添加最后一项(红色那项)

static struct platform\_device \*smdkv210\_devices[] \_\_initdata = {

&s3c\_device\_nand,

&s3c\_device\_adc,

&s3c\_device\_cfcon,

&s3c\_device\_fb,

&s3c\_device\_hsmmc0,

&s3c\_device\_hsmmc1,

&s3c\_device\_hsmmc2,

&s3c\_device\_hsmmc3,

&s3c\_device\_i2c0,

&s3c\_device\_i2c1,

&s3c\_device\_i2c2,

&s3c\_device\_rtc,

&s3c\_device\_ts,

&s3c\_device\_usb\_hsotg,

&s3c\_device\_wdt,

&s5p\_device\_fimc0,

&s5p\_device\_fimc1,

&s5p\_device\_fimc2,

&s5p\_device\_fimc\_md,

&s5p\_device\_jpeg,

&s5p\_device\_mfc,

&s5p\_device\_mfc\_l,

&s5p\_device\_mfc\_r,

&s5pv210\_device\_ac97,

&s5pv210\_device\_iis0,

&s5pv210\_device\_spdif,

&samsung\_asoc\_idma,

&samsung\_device\_keypad,

&smdkv210\_dm9000,

&smdkv210\_lcd\_lte480wv,

&webee210\_button\_device,

/\* Add by Webee

\*/

};

然后在定义 smdkv210\_devices 结构体指针数组的前面,定义下面这些代码。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Add by Webee\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <linux/gpio\_keys.h> /\* Add by Webee \*/

static struct gpio\_keys\_button webee210\_buttons[] = {

{

.gpio= S5PV210\_GPH2(0),

/\* S1 \*/

.code= KEY\_A,

.desc= "Button 1",

.active\_low = 1,

},

{

.gpio= S5PV210\_GPH2(1),

/\* S2 \*/

.code= KEY\_B,

.desc= "Button 2",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH2(2), /\* S3 \*/

.code = KEY\_C,

.desc = "Button 3",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH2(3), /\* S4 \*/

.code = KEY\_L,

.desc = "Button 4",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH3(0), /\* S5 \*/

.code = KEY\_S,

.desc = "Button 5",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH3(1), /\* S6 \*/

.code = KEY\_ENTER,

.desc = "Button 6",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH3(2), /\* S7 \*/

.code = KEY\_LEFTSHIFT,

.desc = "Button 7",

.active\_low = 1,

},

{

.gpio = S5PV210\_GPH3(3), /\* S8 \*/

.code = KEY\_DELETE,

.desc = "Button 8",

.active\_low = 1,

},

};

static struct gpio\_keys\_platform\_data webee210\_button\_data = {

.buttons= webee210\_buttons,

.nbuttons = ARRAY\_SIZE(webee210\_buttons),

};

/\* 设备相关的信息会保存在 pdev->dev.platform\_data 中 \*/

static struct platform\_device webee210\_button\_device = {

.name= "gpio-keys",

.id= -1,

.dev={

.platform\_data = &webee210\_button\_data,

}

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Add by Webee\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

把上面的内容添加好后,通过 make meuconfig,将 gpio\_keys.c 配置上,就可以使用这个 GPIO 按键了。

##### 配置支持 gpio\_key

要想系统支持这个 gpio\_key 硬件驱动,那自然要去编译这个 gpio\_key.c 文件,这里通过 make menuconfig 命令,将 gpio\_key.c 编译进内核,这样就不用我们手工加载驱动了。

进入 make menuconfig 界面后,搜索 KEYBOARD\_GPIO,就可以发现配置gpio\_key 的驱动路径了,如图 7.2。至于如何搜索,这是一个非常基础的问题,这里就不说了,不会的话,请自行百度吧。



也就是按下下面的配置要求,将配置项配置上。

-> Device Drivers

-> Input device support

-> Generic input layer (needed for keyboard, mouse, ...)

[\*] Keyboards --->

<\*>GPIO Buttons

配置好后,保存退出 make menuconfig 配置界面,然后使用下面的命令编译内核。

#make uImage

编译好后,uImage 的存放路径是 arch/arm/boot/uImage,进行测试时,就需要将新内核烧到 NAND FLASH 里去。(使用网蜂自制 u-boot 菜单栏,选择 2烧写)。

#### 测试程序

在分析测试程序前,我们先来看一个内核的结构体,Linux 输入子系统里,将事件抽象成 input\_event 结构体,它代表一个事件,比如按键按下、或者按键松开,这些都是一个事件。

struct input\_event {

struct timeval time;/\* 时间 \*/

\_\_u16 type;/\* 事件类型 \*/

\_\_u16 code;/\* 事件类下的值 \*/

\_\_s32 value;/\* 0-松开, 1-按下,2-重复 \*/

};

struct timeval {

\_\_kernel\_time\_t

tv\_sec;/\* 秒 \*/

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

};

gpio\_key.c 按 键 驱 动 测 试 程 序 源 码 在 : webee210\_drivers\6th\_buttons\_input\ 1th\_Transplant\_ok\ gpio\_key\_test.c

/\* 某些头文件 \*/

int main(int argc,char \*argv[])

{

int fd;

struct input\_event ev\_key;

if (argc != 2)

{

printf("Usage:\n");

printf("%s /dev/event0 or /dev/event1\n",argv[0]);

return 0;

}

fd= open(argv[1], O\_RDWR);

if(fd < 0)

{

perror("open device buttons");

exit(1);

}

while(1)

{

read(fd,&ev\_key,sizeof(struct input\_event));

printf("type:%d,code:%d,value:%d\n",ev\_key.type,

ev\_key.code,ev\_key.value);

}

close(fd);

return 0;

}

编译+拷贝到开发板。

[root@localhost 1th\_Transplant\_ok]# arm-linux-gcc -o gpio\_key\_test gpio\_key\_test.c

[root@localhost 1th\_Transplant\_ok]# cp gpio\_key\_test /home/webee210v2/rootfs/webee210\_driver\_and\_test/

测试程序首先打开/dev/event0 或者/dev/event1 设备文件,这个设备文件正是 evdev\_connect 函数创建的。/dev/event0 设备节点的创建,说明了 evdev 事件处理器支持 gpio\_key 输入设备。为什么是/dev/evnet0 或/dev/event1 呢?那event1 又是指什么设备呢?

如果使用移植 gpio\_key 后的内核启动的话,那么 event0 指的的是 gpio\_key设备,而 event1 指的是触摸屏设备,关于触摸屏的内容,后面再详细介绍,这里暂且告诉大家,触摸屏也使用了输入子系统的框架。但是,如果使用后面 7.6节,自己从零编写按键设备驱动,手动加载驱动后,event0 代表触摸屏设备,event1 代表按键设备。很简单,这是时间先后的问题。

打开设备后,测试程序就一直在读/dev/event0 设备,最终就会调用到事件处理层的 evdev\_read 函数,如果没有数据可读进程就会休眠,什么时候被唤醒呢?有事件发生的时候才会唤醒进程。说到事件,那自然是硬件设备产生的。

当按键被按下或松开时,就会进入到硬件设备驱动层,以 gpio\_key 为例,它 就 会 进 入 到 gpio\_keys\_gpio\_isr 按 键 中 断 处 理 函 数 , 它 最 终 调 用gpio\_keys\_gpio\_work\_func 函数,进而使用 gpio\_keys\_gpio\_report\_event 来上报 事 件 , 而 gpio\_keys\_gpio\_report\_event 利 用 了 输 入 子 系 统 核 心 提 供 的input\_event 函数来上报事件。 input\_event 函数最终调用到 evdev.c 里 的evdev\_events 函 数 来 上 报 事 件 。 它 又 调 用 evdev\_pass\_values 函 数 ,而evdev\_pass\_values 函数的工作就是唤醒在 evdev\_read 时休眠的进程,唤醒进程后,evdev\_read 函数把事件(数据)通过 input\_event\_to\_user 函数传递给用户程序。而 input\_event\_to\_user 正是调用了 copy\_to\_user 将事件(数据)从内核空间拷贝到用户空间。这样,用户程序就读取到了按键所产生的动作。