我们写的驱动程序：

fd = open("/dev/buttons",O\_RDWR |O\_NONBLOCK)

别人不可能去打开你的"/dev/buttons"那如何写出通用的驱动程序，让别人的应用程序去调用？

引出：

**输入子系统**：现成的驱动程序

我们自己写的时候：框架

确定主设备号

file\_operations(.open .read ….)

Register\_chardev()

入口

出口

我们要做的就是把我们的融合进去输入子系统：

一：输入子系统框架

在input.c //drives/input所有的输入子系统代码在这个c文件中

看一个驱动程序从他的入口函数（我们自己写的时候用module\_init()修饰）开始看：

static int \_\_init input\_init(void) err = register\_chrdev(INPUT\_MAJOR, "input", &input\_fops);

以前的字符设备驱动的时候是自己写的，上面的注册是内核的

#define INPUT\_MAJOR 13

err = register\_chrdev(INPUT\_MAJOR, "input", &input\_fops);

static const struct file\_operations input\_fops = {

.owner = THIS\_MODULE,

.open = input\_open\_file,

};

这里注册了一个主设备号为INPUT\_MAJOR 13的设备，名字为input.他的file\_operations结构体是input\_fops。但是里面只有一个open函数。想要读按键的话，那肯定open里面有做一些工作

接下来分析input\_open\_file：

static int input\_open\_file(struct inode \*inode, struct file \*file)

{

struct input\_handler \*handler = input\_table[iminor(inode) >> 5];

//根据此次设备号得到一个handler结构体

//里面有一个input\_handler \*的输入处理句柄指向了input\_table数组

const struct file\_operations \*old\_fops, \*new\_fops = NULL;//创建新的file\_operations结构体new\_fops

if (!handler || !(new\_fops = fops\_get(handler->fops)))// 将input\_handle 中的file\_operations \*fops赋给new\_fops

return -ENODEV;

//这样新的file\_operations new\_fops等于input\_handler \*指针变量handler 成员的ops

old\_fops = file->f\_op;

file->f\_op = new\_fops;

//这样的操作将file\_operations 结构赋给input\_open\_file函数的形参f\_op 通过这样的操作后当我们调用//input\_open\_file 时打开file->f\_op

//然后再调用新的file\_operations的new\_fops的open函数

err = new\_fops->open(inode, file);

}

通过上述的操作我们读取按键的时候只是用到input\_handler \*handler中的const struct file\_operations \*fops;intput.c只是一个中转过程最终还是用到input\_table[].以后app来读的时候最终会调用file->f\_op的read但是最终都是跟input\_handler input\_table有关

input\_table 如何得到？

static struct input\_handler \*input\_table[8];//是个静态数组只有再本文件中用到

int input\_register\_handler(struct input\_handler \*handler)//再这里用到

if (handler->fops != NULL) {

if (input\_table[handler->minor >> 5])

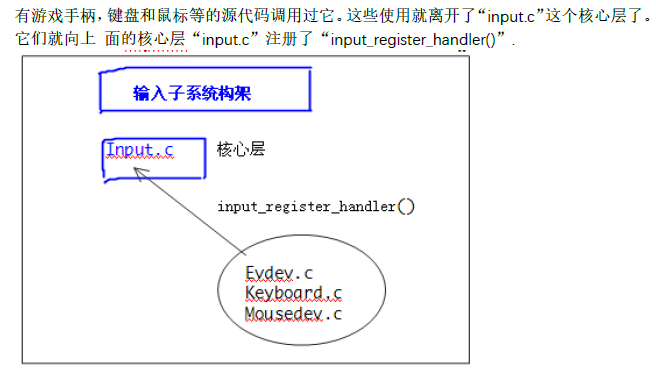
return -EBUSY;

input\_table[handler->minor >> 5] = handler;

}

input\_register\_handler被谁调用？

edev.c keyboard.c mouse.c



打开edev.c查看源码

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

{

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

}

evdev\_handler结构体如下：

static struct input\_handler evdev\_handler = {

.event = evdev\_event,

.connect = evdev\_connect,

.disconnect = evdev\_disconnect,

.fops = &evdev\_fops,

.minor = EVDEV\_MINOR\_BASE,//64

.name = "evdev",

.id\_table = evdev\_ids,

};

evdev\_fops里面有read write 函数

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

};

以前是我们自己写的，现在是系统构造了file\_operations.

通过edev\_init 的input\_register\_handler（）

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

{

struct input\_dev \*dev;

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

if (handler->fops != NULL) {

if (input\_table[handler->minor >> 5])

return -EBUSY;

input\_table[handler->minor >> 5] = handler;//所以handler放在input\_table[2]里面

}

}

软件方面（evdev.c/keyboard.c/mousedev.c）是向核心层“input.c”注册“handler”（input\_register\_handler），这一边代表“软件”；还有另一边另一个层“设备”，是向“核心层input.c”注册“input\_register\_device”，这一边代表硬件.

一边的handler软件处理者是否支持一边的device中间有一个联系：

static struct input\_handler evdev\_handler = {

.event = evdev\_event,

.connect = evdev\_connect,

.disconnect = evdev\_disconnect,

.fops = &evdev\_fops,

.minor = EVDEV\_MINOR\_BASE,

.name = "evdev",

.id\_table = evdev\_ids,//表示edev\_handler能够支持那些输入设备

};

“id\_table”表示这个“evdev\_handler”能够支持哪些“输入设备”。当我们注册上图中的“handler”和“device”时，这两者就会比较（handler和设备比较），看handler是否支持这个设备。若能支持则，则从上面的“evdev\_handler”结构中知道，应该会调用其中的“.connect =evdev\_connect”.

看看谁会调用“input\_register\_device”,如鼠标：Amimouse.c (drivers\input\mouse)，

如键盘：Amikbd.c (drivers\input\keyboard)；

注册输入设备：

分析“int input\_register\_device(struct input\_dev \*dev)”所做的事情：看“input\_c”中此函数源码。

注册函数，将输入设备注册到核心层中，注册前需要输入设备需要调用input\_allocate\_device来分配，然后设置设备处理能力.

1. 放入一个链表：

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

input\_dev:子系统中用此结构体来描述一个输入设备.这里是：

将设备加入全局链表中→

然后遍历input\_handler\_list中的每一个handler,调用input\_attach\_handler进行attach。

1. 对链表里的每个条目

，“input\_handler\_list(注册handler时加入的链表）”都会调用“input\_attach\_handler()”函数来对“硬件设备device”与“处理方式handler”进行关联。

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

input\_attach\_handler(dev, handler);

List\_for\_each\_entry函数遍历Input\_handler\_list(全局链表，连接所有的input\_handler)上的handler，并调用Input\_attach\_handler来进行输入设备和处理方法的关联。

//注册input\_handler

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

{

//放入数组

input\_table[handler->minor >> 5] = handler;

//放入链表

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

//对于每一个input\_dev 都调用input\_attach\_handler

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

input\_attach\_handler(dev, handler);

}

//注册输入设备

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

{ //放入链表

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

//对于每一个input\_handler 都调用input\_attach\_handler  
list\_for\_each\_entry(handler, &input\_handler\_list, node)

input\_attach\_handler(dev, handler);//根据input\_headler 的id\_table 看是否支持设备

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

{ // input\_attach\_handler 主要完成如下的内容

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

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

}

}

从上图可知，不管是先注册加载右边的“handler”还是左边的“device”。最终都会成对的调用“input\_attach\_handler()”。看看源码“input\_attach\_handler()”：

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

{

const struct input\_device\_id \*id;

int error;

if (handler->blacklist && input\_match\_device(handler->blacklist, dev))

return -ENODEV;

id = input\_match\_device(handler->id\_table, dev);//根据handler->id\_table于形参这个输入设备比较

if (!id)//看dev与处理方式id\_table有匹配吗

return -ENODEV;//没有直接返回

error = handler->connect(handler, dev, id);//有匹配的话调用handler中的connect函数

if (error && error != -ENODEV)

printk(KERN\_ERR

"input: failed to attach handler %s to device %s, "

"error: %d\n",

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

return error;

}

注册input\_dev或input\_handler时，会两两比较左边的input\_dev和右边的input\_handler,根据input\_handler的id\_table判断这个input\_handler能否支持这个input\_dev，如果能支持，则调用input\_handler的connect函数建立"连接。

如何建立连接，可能不同的handler都有自已不同的方式。

实例分析“evdev.c”中的“connect”：

static int evdev\_connect(struct input\_handler \*handler, struct input\_dev \*dev,

const struct input\_device\_id \*id)

{

//分配了一个“input\_handle evdev”结构变量

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

// 再对这个“input\_handle evdev”进行设置：

evdev->exist = 1;

evdev->minor = minor;

evdev->handle.dev = dev;//指向左边的input\_edev结构体

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

evdev->handle.handler = handler;//直向右边的input\_handler结构

evdev->handle.private = evdev;

sprintf(evdev->name, "event%d", minor);

//最后注册这个 handle：

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

}

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

{

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

list\_add\_tail(&handle->d\_node, &handle->dev->h\_list);// 将形参“handle”放到一个输入设备的链表里面：

list\_add\_tail(&handle->h\_node, &handler->h\_list);// 再把“handler”放到右边一个“h\_list”链表里面。

if (handler->start)

handler->start(handle);

return 0;

}



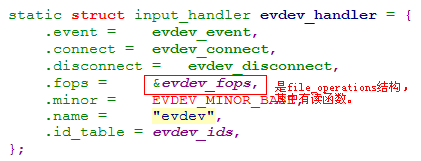
如何读数据“按键”：

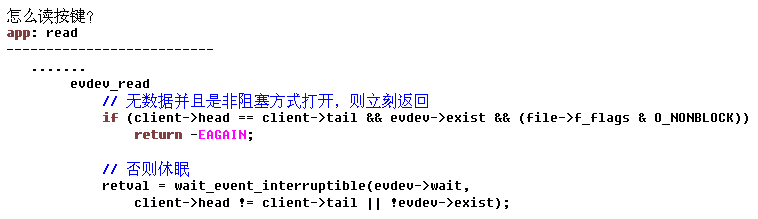
App:read

应用程序来读，最终会导致“handler”里面的新的“.fops”里面的“读函数”被调用。如：

“evdev.c”中的“evdev\_handler”结构里面的成员“.fops=&evdev\_fops”，在“evdev\_fops”

结构中有一个“读”函数“evdev\_read”





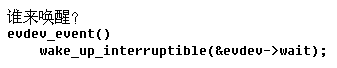
在“evdev\_read()”中：



可以从上面的“wait\_event\_interruptible（）”知道是在“evdev”上休眠，则唤醒也会是在它上面。可以搜索它。

1. 谁来“唤醒”：

如按下一个按键后，中断处理函数就会被调用。在中断处理函数里面先确定按键值。然后才来唤醒。

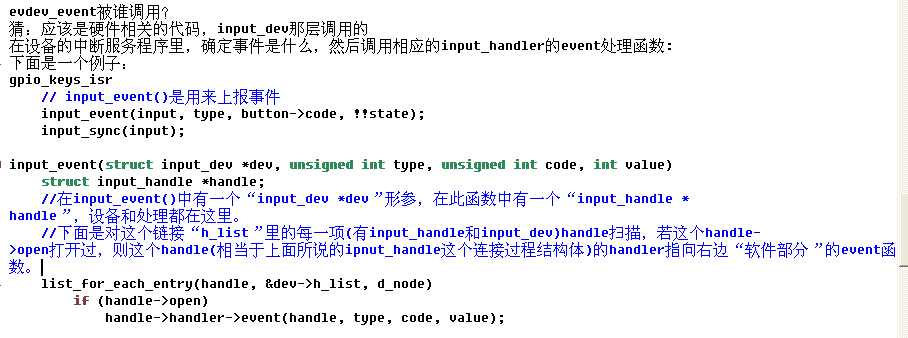


在代码中搜索“evdev->wait”后找到在“evdev\_event()”中有唤醒操作。这个“evdev->wait”是结构“evdev\_handler”的“.event”事件成员。

分析事件处理函数“evdev\_event()”:

主图中右边是“纯软件”的部分，按键按下时应该是由左边输入设备这一层触发的。

(2) 谁调用“evdev\_event()”:

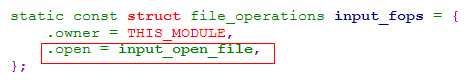


总结“输入子系统”：

核心层“input.c”.它里面有“register\_chdev”（input\_init()中），但它简单：



因为这个register\_chrdev（）中的“input\_fops”file\_operations结构很简单：



只有一个“.open”函数，所以让它去读去写不行，里面还有个“中转”的过程：



根据打开的设备节点的次设备号找到一个“handler”：



并且把这个文件的f\_op指向新的“input\_handler \*handler”里面的“fops”:



然后再调用这个“handler”中的open函数：



其中的“iput\_table[]”数组由下面的各个“纯软件”代码构建（如：evdev.c,keyboard.c）.

input\_register\_device（）//input.c中宏定义EXPORT\_SYMBOL(input\_register\_device); 如鼠标：Amimouse.c (drivers\input\mouse)，如键盘调用



Amikbd.c (drivers\input\keyboard)

input\_register\_handler（）//在edev.c keyboard.c Mousedev.c等有调用 通过在这个入口函数module\_init(evdev\_init);

“纯软件（input\_handler）”部分和“硬件部分（input\_dev）”联系起来，纯软件部分是由

“ input\_register\_handler ” 向上“ input.c ” 核心层注册处理方式； 硬件部分是由

“input\_register\_device”向上“input.c”核心层注册硬层。这样注册后，会使它们两两比较，

看看其中的某个“handler”是否支持其中的某个“dev”。若是“handler”能支持某个“dev”，

则会接着调用“input\_handler”结构下的“.connect”函数，这个函数一般会创建一个

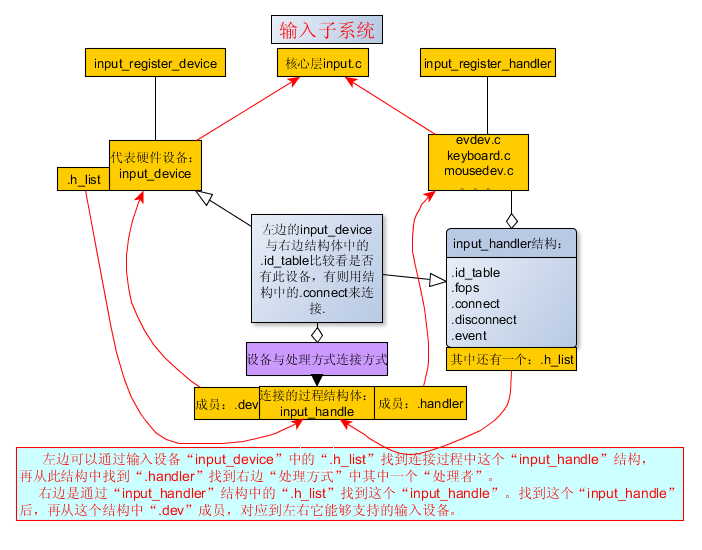
“input\_handle”结构（是handle 而非handler），并且这个结构“input\_handle”会分别放

在两边的“h\_list”链表中去。这个结构体中有“.dev”和“.handler”，让它两分别具体指向

右边的“纯软件处理部分”和左右的硬件部分，这样具体的某硬件就和纯软件联系起来了。

可以从任何一边通过“h\_list”找到这个“input\_handle”结构，再通过成员找到另一端的“.dev”

或“.handler”。



举例是如何读按键：

最终是应用程序读，最终会导致handler中的“read”函数。读的过程中，没有数据可读时就休眠，有休眠就会唤醒，搜索的结果是“event”函数来唤醒。分析到这里就没有接着去分析而是猜测是由“硬件”调用的“event”函数，硬件则是指“input\_dev”层的设备中断服务程序调用了“event”函数。通过这个“event”函数可以最终追踪到“纯软件”部分的“input\_handler”结构体中的“.event”成员。

以gpio\_kyes为例

gpio\_keys\_isr（）//在中段服务程序中

input\_event（）//上报事件

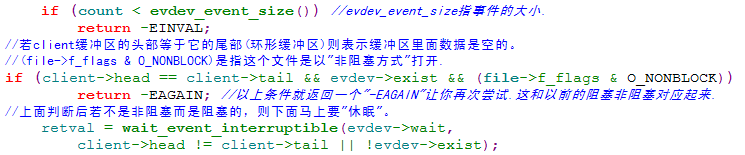
list\_for\_each\_entry(handle, &dev->h\_list, d\_node)

if (handle->open)

handle->handler->event(handle, type, code, value);//调用event函数



在“evdev\_read()”中：



可以从上面的“wait\_event\_interruptible（）”知道是在“evdev”上休眠，则唤醒也会是在它上面。可以搜索它。

(1) 谁来“唤醒”：

如果按下一个按建中断处理函数就会被调用，在中断处理函数里面先确定按键值然后才会唤醒。

谁来唤醒?

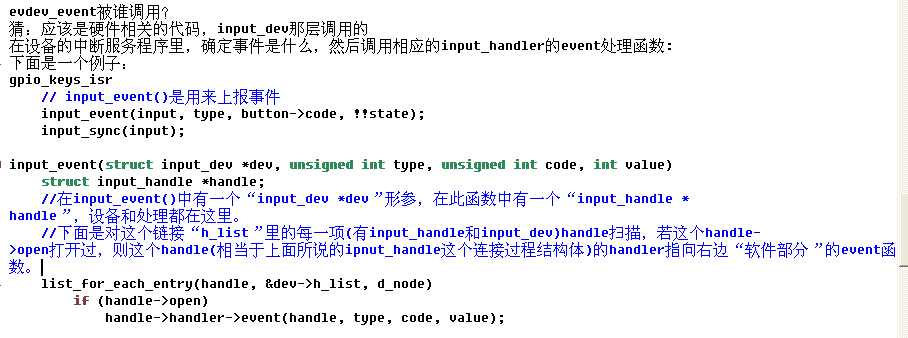
(1)evdev\_event();

在代码中搜索“evdev->wait”后找到在“evdev\_event()”中有唤醒操作。这个“evdev->wait”是结构“evdev\_handler”的“.event”事件成员。

分析事件处理函数“evdev\_event()”:

主图中右边是“纯软件”的部分，按键按下时应该是由左边输入设备这一层触发的。

谁调用“evdev\_event()”:



input\_dev （硬件） input\_handler (纯软件)

这俩向上注册最终两两比较。有一个input\_hande 是一个中间的桥梁

//edev.c中

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

{

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

}

输入输出子系统框架：

intpu.c //核心层

input\_register\_device （） intput\_register\_handler（）//纯软件

代表硬件的设备

input\_device

Edev.c

Keyboard.c mouse.c