# Linux驱动\_输入子系统

## 1.基本概念

以前我们写一些输入设备（键盘、鼠标等）的驱动都是采用字符设备、混杂设备处理的。问题由此而来，Linux开源社区的大神们看到了这大量输入设备如此分散不堪，有木有可以实现一种机制，可以对分散的、不同类别的输入设备进行统一的驱动，所以才出现了输入子系统。

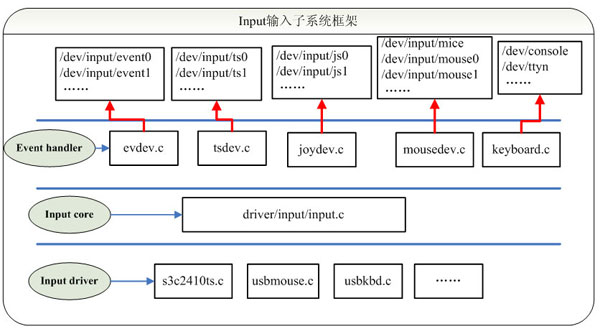
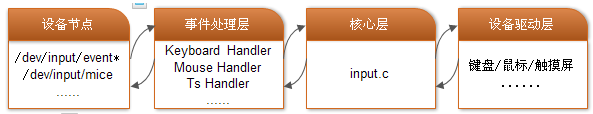
输入子系统引入的好处：

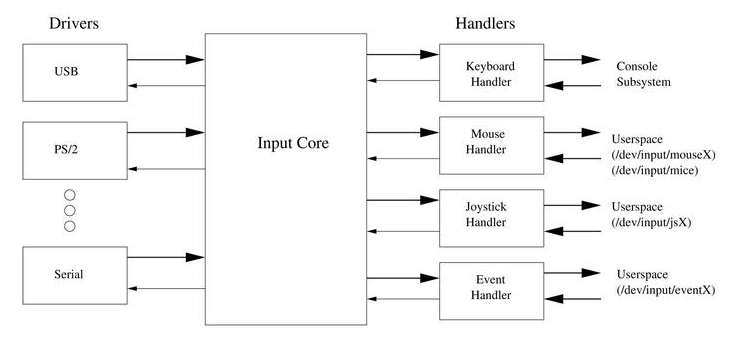
(1)统一了物理形态各异的相似的输入设备的处理功能。例如，各种鼠标，不论PS/2、USB、还是蓝牙，都被同样处理。

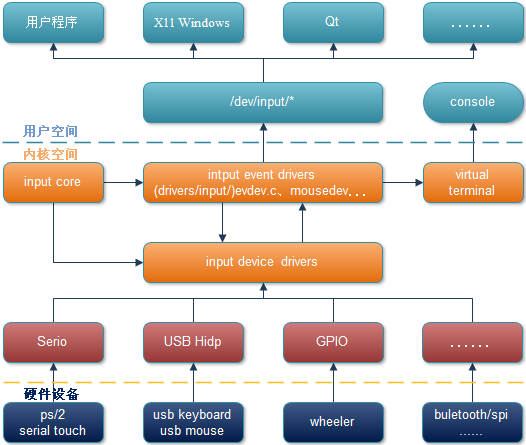
(2)提供了用于分发输入报告给用户应用程序的简单的事件（event）接口。你的驱动不必创建、管理/dev节点以及相关的访问方法。因此它能够很方便的调用输入API以发送鼠标移动、键盘按键，或触摸事件给用户空间。X windows这样的应用程序能够无缝地运行于输入子系统提供的event接口之上。

(3)抽取出了输入驱动的通用部分，简化了驱动，并提供了一致性。例如，输入子系统提供了一个底层驱动（成为serio）的集合，支持对串口和键盘控制器等硬件输入的访问。

**输入子系统的基本框架图：**





## 2. 实现一个输入子系统框架的按键驱动

### 2.1 输入设备的驱动开发步骤

看到上面的输入子系统框架图还是有一些困惑，那么先来实现一个输入子系统框架的按键驱动，看看输入子系统是如何适配，如何使用。

作为输入设备的驱动开发者，需要做以下几步：

(1)在驱动加载模块中，设置你的input设备支持的事件类型，类型参见表1设置

(2)注册中断处理函数，例如键盘设备需要编写按键的抬起、放下，触摸屏设备需要编写按下、抬起、绝对移动，鼠标设备需要编写单击、抬起、相对移动，并且需要在必要的时候提交硬件数据（键值/坐标/状态等等）。

(3)将输入设备注册到输入子系统中。

在写代码之前，需要设置下mdev的配置文件/etc/mdev.conf：

# Move input devices to input directory

event**.\*** 0**:**0 0660 @**(**mkdir **-**p input**&&**mv $MDEV input**)**

mice 0**:**0 0660 @**(**mkdir **-**p input**&&**mv $MDEV input**)**

mouse**.\*** 0**:**0 0660 @**(**mkdir **-**p input**&&**mv $MDEV input**)**

ts**.\*** 0**:**0 0660 @**(**mkdir **-**p input**&&**mv $MDEV input**)**

js**.\*** 0**:**0 0660 @**(**mkdir **-**p input**&&**mv $MDEV input**)**

### 2.2 输入设备的驱动代码

**符合输入子系统框架的按键驱动程序代码：**

#include <linux/module.h>

#include <linux/version.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/interrupt.h>

#include <linux/irq.h>

#include <linux/sched.h>

#include <linux/pm.h>

#include <linux/sysctl.h>

#include <linux/proc\_fs.h>

#include <linux/delay.h>

#include <linux/platform\_device.h>

#include <linux/input.h>

#include <linux/irq.h>

#include <asm/gpio.h>

#include <asm/io.h>

#include <asm/arch/regs-gpio.h>

unsigned char input\_name**[]** **=** "s3c240\_keys"**;**

struct pin\_desc**{**

int irq**;**

char **\***name**;**

unsigned int pin**;**

unsigned int key\_val**;**

**};**

struct pin\_desc pins\_desc**[**4**]** **=** **{**

**{**IRQ\_EINT0**,** "K1"**,** S3C2410\_GPF0**,** KEY\_L**},**

**{**IRQ\_EINT2**,** "K2"**,** S3C2410\_GPF2**,** KEY\_S**},**

**{**IRQ\_EINT11**,** "K3"**,** S3C2410\_GPG3**,** KEY\_ENTER**},**

**{**IRQ\_EINT19**,** "K4"**,** S3C2410\_GPG11**,** KEY\_LEFTSHIFT**},**

**};**

static struct input\_dev **\***buttons\_dev**;**

static struct pin\_desc **\***irq\_pd**;**

static struct timer\_list buttons\_timer**;**

static irqreturn\_t buttons\_irq**(**int irq**,** void **\***dev\_id**)**

**{**

/\* 10ms后启动定时器 \*/

irq\_pd **=** **(**struct pin\_desc **\*)**dev\_id**;**

mod\_timer**(&**buttons\_timer**,** jiffies**+**HZ**/**100**);**

**return** IRQ\_RETVAL**(**IRQ\_HANDLED**);**

**}**

static void buttons\_timer\_function**(**unsigned long data**)**

**{**

struct pin\_desc **\*** pindesc **=** irq\_pd**;**

unsigned int pinval**;**

**if** **(!**pindesc**)**

**return;**

pinval **=** s3c2410\_gpio\_getpin**(**pindesc**->**pin**);**

**if** **(**pinval**)**

**{**

/\* 松开 : 最后一个参数: 0-松开, 1-按下 \*/

input\_event**(**buttons\_dev**,** EV\_KEY**,** pindesc**->**key\_val**,** 0**);**

input\_sync**(**buttons\_dev**);**

**}**

**else**

**{**

/\* 按下 \*/

input\_event**(**buttons\_dev**,** EV\_KEY**,** pindesc**->**key\_val**,** 1**);**

input\_sync**(**buttons\_dev**);**

**}**

**}**

static int buttons\_init**(**void**)**

**{**

int i**;**

//1. 分配一个input\_dev结构体

buttons\_dev **=** input\_allocate\_device**();;**

buttons\_dev**->**name **=** input\_name**;**

//2. 设置

//2.1 能产生哪类事件 \*/

set\_bit**(**EV\_KEY**,** buttons\_dev**->**evbit**);**

set\_bit**(**EV\_REP**,** buttons\_dev**->**evbit**);**

//2.2 能产生这类操作里的哪些事件: L,S,ENTER,LEFTSHIT

set\_bit**(**KEY\_L**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_S**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_ENTER**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_LEFTSHIFT**,** buttons\_dev**->**keybit**);**

//3. 注册

input\_register\_device**(**buttons\_dev**);**

//4. 硬件相关的操作

init\_timer**(&**buttons\_timer**);**

buttons\_timer**.**function **=** buttons\_timer\_function**;**

add\_timer**(&**buttons\_timer**);**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)**

**{**

request\_irq**(**pins\_desc**[**i**].**irq**,** buttons\_irq**,** IRQT\_BOTHEDGE**,** pins\_desc**[**i**].**name**,** **&**pins\_desc**[**i**]);**

**}**

**return** 0**;**

**}**

static void buttons\_exit**(**void**)**

**{**

int i**;**

**for** **(**i **=** 0**;** i **<** 4**;** i**++)**

**{**

free\_irq**(**pins\_desc**[**i**].**irq**,** **&**pins\_desc**[**i**]);**

**}**

del\_timer**(&**buttons\_timer**);**

input\_unregister\_device(buttons\_dev);

input\_free\_device(buttons\_dev);

**}**

module\_init**(**buttons\_init**);**

module\_exit**(**buttons\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

输入子系统关键代码：

//1. 分配一个input\_dev结构体

buttons\_dev **=** input\_allocate\_device**();;**

//2. 设置

//2.1 能产生哪类事件 \*/

set\_bit**(**EV\_KEY**,** buttons\_dev**->**evbit**);**

set\_bit**(**EV\_REP**,** buttons\_dev**->**evbit**);**

//2.2 能产生这类操作里的哪些事件: L,S,ENTER,LEFTSHIT

set\_bit**(**KEY\_L**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_S**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_ENTER**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_LEFTSHIFT**,** buttons\_dev**->**keybit**);**

//3. 注册

input\_register\_device**(**buttons\_dev**);**

注册按键设备到输入子系统中，并告知输入子系统该按键设备能产生KEY和REP事件，键值为KEY\_L, KEY\_S, KEY\_ENTER, KEY\_LEFTSHIFT。

当有实际的按键动作时，将键值上报给输入子系统。

input\_event**(**buttons\_dev**,** EV\_KEY**,** pindesc**->**key\_val**,** 0**);**

input\_sync**(**buttons\_dev**);**

### 2.3 测试方法

**下面来看下如何测试该驱动：**

加载该驱动后，在/dev/input目录下会多出一个event1的文件。

# ls -l /dev/input/

crw**-**rw**----** 1 0 0 13**,** 64 Jan 1 00**:**00 event0

crw**-**rw**----** 1 0 0 13**,** 65 Jan 1 00**:**00 event1

crw**-**rw**----** 1 0 0 13**,** 63 Jan 1 00**:**00 mice

crw**-**rw**----** 1 0 0 13**,** 32 Jan 1 00**:**00 mouse0

crw**-**rw**----** 1 0 0 13**,** 128 Jan 1 00**:**00 ts0

event1到底是什么呢？下面来查询下：

# cat /proc/bus/input/devices

I**:** Bus**=**0013 Vendor**=**dead Product**=**beef Version**=**0101

N**:** Name**=**"s3c2410 TouchScreen"

P**:** Phys**=**

S**:** Sysfs**=/class/input/**input0

U**:** Uniq**=**

H**:** Handlers**=**mouse0 event0 ts0

B**:** EV**=**b

B**:** KEY**=**400 0 0 0 0 0 0 0 0 0 0

B**:** ABS**=**1000003

I**:** Bus**=**0000 Vendor**=**0000 Product**=**0000 Version**=**0000

N**:** Name**=**"s3c240\_keys"

P**:** Phys**=**

S**:** Sysfs**=/class/input/**input3

U**:** Uniq**=**

H**:** Handlers**=**kbd event1

B**:** EV**=**100003

B**:** KEY**=**440 90000000

从查询结果看TouchScreen的handlers为mouse0，event0，ts0，说明touchsreen属于鼠标，event0，和触摸屏。而刚才加入的按键属于kbd(键盘)和event1。一个设备可以与多个平台驱动进行连接。

执行如下命令，当按键按下并松开一次时，打印如下：

# cat /dev/input/event1 | hexdump

0000000 00bc 0000 5fe1 0001 **0001** 002a 0001 0000

0000010 00bc 0000 5fec 0001 0000 0000 0000 0000

0000020 00bc 0000 0d78 0003 0001 002a 0000 0000

0000030 00bc 0000 0d81 0003 0000 0000 0000 0000

每一行数据是按键发生时候input\_event上报到输入子系统的数据。

struct input\_event **{**

struct timeval time**;**

\_\_u16 type**;**

\_\_u16 code**;**

\_\_s32 value**;**

**};**

根据对应的结构体，分析上面数据的内容：

第一行第六列开始：  
0001(EV\_KEY) 002a(#define KEY\_LEFTSHIFT 42) 0001 0000(按下)

第二行代表input\_sync动作产生的值  
0000(EV\_SYN)

第三行第六列开始：  
0001(EV\_KEY) 002a(#define KEY\_LEFTSHIFT 42) 0000 0000(松开)

第四行代表input\_sync动作产生的值  
0000(EV\_SYN)

当按键按住不松开时，触发了输入子系统的REPEAT事件，此时的键值如下：

# cat /dev/input/event1 | hexdump

0000000 079f 0000 4f1e 0009 0001 002a 0001 0000

0000010 079f 0000 4f2e 0009 0000 0000 0000 0000

0000020 079f 0000 1fa2 000d 0001 002a 0002 0000

0000030 079f 0000 1fad 000d 0000 0000 0000 0000

0000040 079f 0000 a864 000d 0001 002a 0002 0000

0000050 079f 0000 a872 000d 0000 0000 0000 0000

0000060 079f 0000 311f 000e 0001 002a 0002 0000

0000070 079f 0000 312a 000e 0000 0000 0000 0000

0000080 079f 0000 b9d4 000e 0001 002a 0002 0000

0000090 079f 0000 b9e0 000e 0000 0000 0000 0000

00000a0 07a0 0000 0052 0000 0001 002a 0002 0000

00000b0 07a0 0000 005e 0000 0000 0000 0000 0000

00000c0 07a0 0000 88fa 0000 0001 002a 0002 0000

00000d0 07a0 0000 8906 0000 0000 0000 0000 0000

00000e0 07a0 0000 11be 0001 0001 002a 0002 0000

00000f0 07a0 0000 11c9 0001 0000 0000 0000 0000

0000100 07a0 0000 9a75 0001 0001 002a 0002 0000

0000110 07a0 0000 9a82 0001 0000 0000 0000 0000

0000120 07a0 0000 2330 0002 0001 002a 0002 0000

0000130 07a0 0000 233b 0002 0000 0000 0000 0000

0000140 07a0 0000 abe8 0002 0001 002a 0002 0000

0000150 07a0 0000 abf4 0002 0000 0000 0000 0000

0000160 07a0 0000 bf90 0002 0001 002a 0000 0000

0000170 07a0 0000 bf9b 0002 0000 0000 0000 0000

从数据看长按的键变成了0002 0000。

### 2.4 tty console的一些概念

有两种控制台：串口控制台，虚拟控制台(vga+键盘)，linux启动选择其中一种作为控制台。  
/dev/console：当前使用的控制台(串口，vga+键盘二选一)。   
/dev/ttyN：虚拟控制台，(串口，vga+键盘二选一)。  
/dev/pts/N：xwindow下的虚拟控制台(vga+键盘)。

tty命令，可以查看当前的控制台或虚拟终端，如果在x-window下面使用，查看到的是伪终端设备。这里为串口设备。

# tty

**/**dev**/**s3c2410\_serial0

当执行cat /dev/tty1命令后，顺序按下K1，K2，K3时，串口终端则会显示ls。读出tty1中的数据。

下面将当前shell的标准输入重定向为/dev/tty1。

# exec 0<dev/tty1

重定向前，串口终端的输入设备为串口，重定向后，输入设备为tty1，tty1的输入设备会是键盘。下面为重定向前的fd指向。

# ls -l /proc/782/fd/

lrwx**------** 1 0 0 64 Jan 1 00**:**02 0 **->** **/**dev**/**s3c2410\_serial0

lrwx**------** 1 0 0 64 Jan 1 00**:**02 1 **->** **/**dev**/**s3c2410\_serial0

lrwx**------** 1 0 0 64 Jan 1 00**:**02 10 **->** **/**dev**/**tty

lrwx**------** 1 0 0 64 Jan 1 00**:**02 2 **->** **/**dev**/**s3c2410\_serial0

重定向后，可以接收按键发出的键值。

### 2.5 应用程序

也可以通过read /dev/input/event1设备读取按键内容，代码如下：

#include <stdio.h>

#include <linux/input.h>

#include <fcntl.h>

int main**(**void**)**

**{**

int buttons\_fd**;**

int key\_value**,**i**=**0**,**count**;**

struct input\_event ev\_key**;**

buttons\_fd **=** open**(**"/dev/input/event1"**,** O\_RDWR**);**

**if** **(**buttons\_fd **<** 0**)** **{**

perror**(**"open device buttons"**);**

exit**(**1**);**

**}**

**for** **(;;)** **{**

count **=** read**(**buttons\_fd**,&**ev\_key**,sizeof(**struct input\_event**));**//读取事件的个数

**for(**i**=**0**;** i**<(**int**)**count**/sizeof(**struct input\_event**);** i**++)**

**if(**EV\_KEY**==**ev\_key**.**type**)**

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

**if(**EV\_SYN**==**ev\_key**.**type**)**

printf**(**"syn event\n\n"**);**

**}**

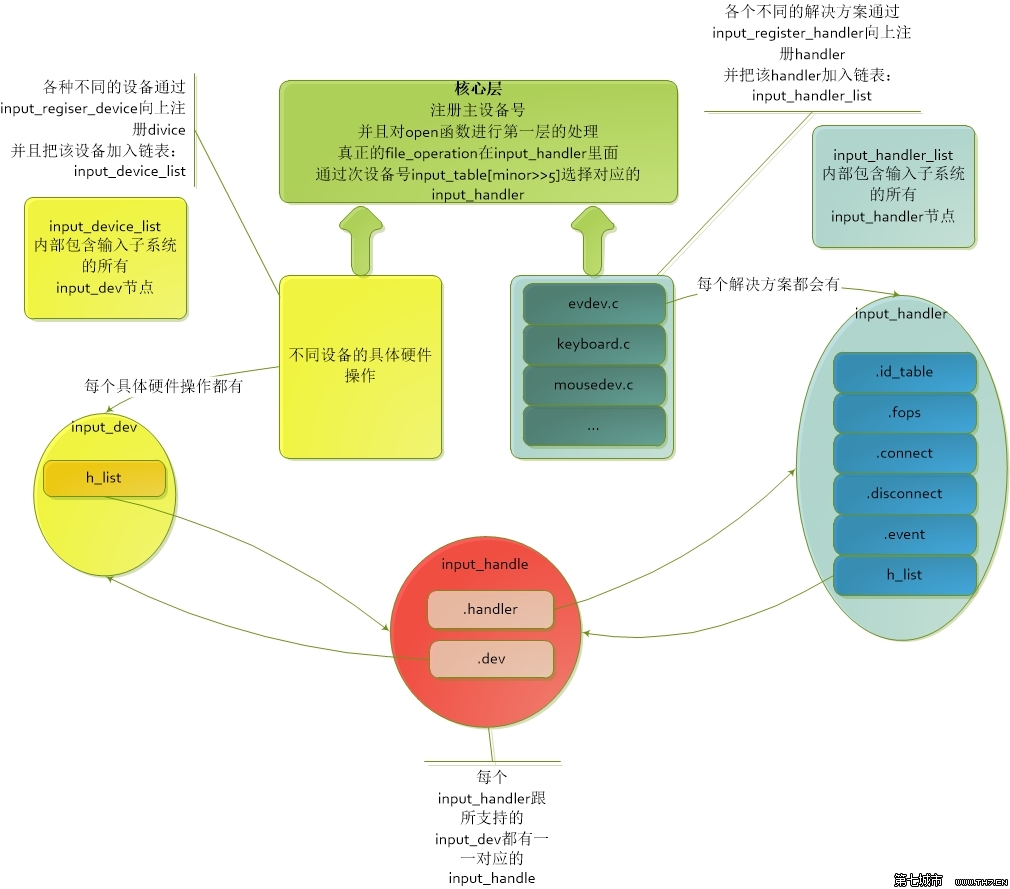
close**(**buttons\_fd**);**

**return** 0**;**

**}**

## 3.输入子系统框架分析

### 3.1 整体分析



**输入子系统中重要的数据结构：**

input\_dev是硬件驱动层，代表一个input设备。  
input\_handler是事件处理层，代表一个事件处理器。  
input\_handle 属于核心层，代表一个配对的input设备与input事件处理器。

input\_dev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\_hande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。

### 3.2 核心层input.c

输入子系统核心对应与/drivers/input/input.c文件,这个也是作为一个模块注册到内核的。所以首先分析模块初始化函数。

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

**{**

……………

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

err **=** input\_proc\_init**();**

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

**if** **(**err**)** **{**

printk**(**KERN\_ERR "input: unable to register char major %d"**,** INPUT\_MAJOR**);**

**goto** fail2**;**

**}**

**return** 0**;**

……………

**}**

很简单input也是一个字符设备，主设备号为13。但是只有一个open操作。

# cat /proc/devices

Character devices**:**

13 input

同时class\_register生成/sys/class/input目录，input\_proc\_init会生成/proc/bus/input/devices和/proc/bus/input/handlers等信息，可以读取这两个文件。

当打开一个input设备时，首先会调用input\_open\_file，但是并没有看到read函数啊？

static const struct file\_operations input\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**open **=** input\_open\_file**,**

**};**

继续分析input\_open\_file函数：

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

**{**

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

const struct file\_operations **\***old\_fops**,** **\***new\_fops **=** **NULL;**

int err**;**

**if** **(!**handler **||** **!(**new\_fops **=** fops\_get**(**handler**->**fops**)))**

**return** **-**ENODEV**;**

**......**

file**->**f\_op **=** new\_fops**;**

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

**......**

**return err;**

**}**

根据从设备号，从input\_table中找到对应的input\_handler，常用的handler由evdev,mousedev,tsdev,joydev,keboard等创建，为什么子设备号要除以32，再从table里面寻找handler呢？因为每一类输入设备的子设备号占用了32，所以除以32可以获得该子设备号属于哪个输入设备的大类。

#define JOYDEV\_MINOR\_BASE 0

#define JOYDEV\_MINORS 16

#define MOUSEDEV\_MINOR\_BASE 32

#define MOUSEDEV\_MINORS 32

#define EVDEV\_MINOR\_BASE 64

#define EVDEV\_MINORS 32

#define TSDEV\_MINOR\_BASE 128

#define TSDEV\_MINORS 32

接下来会获取input\_handler的fops，并赋给file的f\_op结构体，最终调用input\_handler中的fops的open函数打开设备。以后当调用read,write等操作时，操作的依然是input\_handler中的fops。

### 3.3 事件处理层

是谁在input\_table中注册了input\_handler结构体呢？在事件处理层中相应的模块会调用input\_register\_handler进行注册。

#### 3.2.1 evdev

evdev会调用input\_register\_handler(&evdev\_handler)进行注册，下面是注册相关内容。

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**,**

**};**

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

**{**

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

**}**

**input\_register\_handler**源码如下：

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

**{**

struct input\_dev **\***dev**;**

//初始化handler的handle链表头

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

**if** **(**handler**->**fops **!=** **NULL)** **{**

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

**return** **-**EBUSY**;**

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

**}**

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

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

input\_attach\_handler**(**dev**,** handler**);**

input\_wakeup\_procfs\_readers**();**

**return** 0**;**

**}**

上述源码完成的功如下：  
(1)将evdev\_handler添加到input\_table中  
(2)将evdev\_handler添加到input\_handler\_list队列中  
(3)遍历input\_dev\_list链表，建立dev和该handler之间关系，一个handler可以对应多个device  
(4)唤醒input\_devices\_poll\_wait等待队列

这里有一个关键函数**input\_attach\_handler**，源码如下：

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**);**

**if** **(!**id**)**

**return** **-**ENODEV**;**

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

**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**;**

**}**

首先调用input\_match\_device(handler->id\_table, dev)检查device是否符合evdev类型的要求。如果具备则调用evdev的connect函数evdev\_connect。下面分析下这两个函数。

input\_match\_device函数代码：

static const struct input\_device\_id **\***input\_match\_device**(**const struct input\_device\_id **\***id**,** struct input\_dev **\***dev**)**

**{**

int i**;**

**for** **(;** id**->**flags **||** id**->**driver\_info**;** id**++)** **{**

//在id->flags中定义了要匹配的项。定义INPUT\_DEVICE\_ID\_MATCH\_BUS。

//则是要比较input device和input handler的总线类型。

//INPUT\_DEVICE\_ID\_MATCH\_VENDOR，INPUT\_DEVICE\_ID\_MATCH\_PRODUCT，

//INPUT\_DEVICE\_ID\_MATCH\_VERSION 分别要求设备厂商。设备号和设备版本.

**if** **(**id**->**flags **&** INPUT\_DEVICE\_ID\_MATCH\_BUS**)**

**if** **(**id**->**bustype **!=** dev**->**id**.**bustype**)**

**continue;**

**if** **(**id**->**flags **&** INPUT\_DEVICE\_ID\_MATCH\_VENDOR**)**

**if** **(**id**->**vendor **!=** dev**->**id**.**vendor**)**

**continue;**

**if** **(**id**->**flags **&** INPUT\_DEVICE\_ID\_MATCH\_PRODUCT**)**

**if** **(**id**->**product **!=** dev**->**id**.**product**)**

**continue;**

**if** **(**id**->**flags **&** INPUT\_DEVICE\_ID\_MATCH\_VERSION**)**

**if** **(**id**->**version **!=** dev**->**id**.**version**)**

**continue;**

//如果id->flags定义的类型匹配成功。或者是id->flags没有定义，

//就会进入到MATCH\_BIT的匹配项了

//从MATCH\_BIT宏的定义可以看出。

//只有当iput device和input handler的id成员在evbit, keybit,… swbit项相同

//才会匹配成功。而且匹配的顺序是从evbit, keybit到swbit.

//只要有一项不同，就会循环到id中的下一项进行比较.

//总结下：input\_handler中声明的能力，如果device有一项不满足，

//则认为两者不匹配。

MATCH\_BIT**(**evbit**,** EV\_MAX**);**

MATCH\_BIT**(**keybit**,** KEY\_MAX**);**

MATCH\_BIT**(**relbit**,** REL\_MAX**);**

MATCH\_BIT**(**absbit**,** ABS\_MAX**);**

MATCH\_BIT**(**mscbit**,** MSC\_MAX**);**

MATCH\_BIT**(**ledbit**,** LED\_MAX**);**

MATCH\_BIT**(**sndbit**,** SND\_MAX**);**

MATCH\_BIT**(**ffbit**,** FF\_MAX**);**

MATCH\_BIT**(**swbit**,** SW\_MAX**);**

**return** id**;**

**}**

**return** **NULL;**

**}**

evdev的input\_device\_id结构体内容：

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

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

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

**};**

假设是device是第二节中的按键设备，按键设备设置的标志位为：

set\_bit**(**EV\_KEY**,** buttons\_dev**->**evbit**);**

set\_bit**(**EV\_REP**,** buttons\_dev**->**evbit**);**

set\_bit**(**KEY\_L**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_S**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_ENTER**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_LEFTSHIFT**,** buttons\_dev**->**keybit**);**

回到input\_match\_device函数，当evdev和按键device进行对比，发现任何设备都会适配evdev handler。

evdev\_connect函数代码如下：

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

**{**

struct evdev **\***evdev**;**

struct class\_device **\***cdev**;**

dev\_t devt**;**

int minor**;**

int error**;**

//evdev最多支持32个设备关联

**for** **(**minor **=** 0**;** minor **<** EVDEV\_MINORS **&&** evdev\_table**[**minor**];** minor**++);**

**if** **(**minor **==** EVDEV\_MINORS**)** **{**

printk**(**KERN\_ERR "evdev: no more free evdev devices\n"**);**

**return** **-**ENFILE**;**

**}**

//创建evdev设备结构体

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

**if** **(!**evdev**)**

**return** **-**ENOMEM**;**

//初始化client\_list链表头,代表了有多少应用读写这个evdev设备

INIT\_LIST\_HEAD**(&**evdev**->**client\_list**);**

//初始化等待队列,当evdev设备没有数据可读时,就在该等待队列上睡眠

init\_waitqueue\_head**(&**evdev**->**wait**);**

evdev**->**exist **=** 1**;**

//evdev设备的次设备号

evdev**->**minor **=** minor**;**

//初始化evdev里面的handle，也就是dev和handler之间的连接关系

evdev**->**handle**.**dev **=** dev**;**

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

evdev**->**handle**.**handler **=** handler**;**

evdev**->**handle**.**private **=** evdev**;**

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

evdev\_table**[**minor**]** **=** evdev**;**

devt **=** MKDEV**(**INPUT\_MAJOR**,** EVDEV\_MINOR\_BASE **+** minor**),**

//自动创建设备节点

cdev **=** class\_device\_create**(&**input\_class**,** **&**dev**->**cdev**,** devt**,**

dev**->**cdev**.**dev**,** evdev**->**name**);**

**if** **(**IS\_ERR**(**cdev**))** **{**

error **=** PTR\_ERR**(**cdev**);**

**goto** err\_free\_evdev**;**

**}**

/\* temporary symlink to keep userspace happy \*/

error **=** sysfs\_create\_link**(&**input\_class**.**subsys**.**kobj**,**

**&**cdev**->**kobj**,** evdev**->**name**);**

**if** **(**error**)**

**goto** err\_cdev\_destroy**;**

//注册handle

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

**if** **(**error**)**

**goto** err\_remove\_link**;**

**return** 0**;**

err\_remove\_link**:**

sysfs\_remove\_link**(&**input\_class**.**subsys**.**kobj**,** evdev**->**name**);**

err\_cdev\_destroy**:**

class\_device\_destroy**(&**input\_class**,** devt**);**

err\_free\_evdev**:**

kfree**(**evdev**);**

evdev\_table**[**minor**]** **=** **NULL;**

**return** error**;**

**}**

evdev\_connect主要完成了如下功能：  
(1)创建了对应的设备。  
(2)建立了handler和device之间的关联handle，通过handler可以找到device，通过device可以找到对应的handler，handle结构体会挂靠在device和handler的h\_list链表下，通过device或者handler可以寻找到其对应的另一半。

input\_register\_handle源码和示意图如下：

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

**{**

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

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

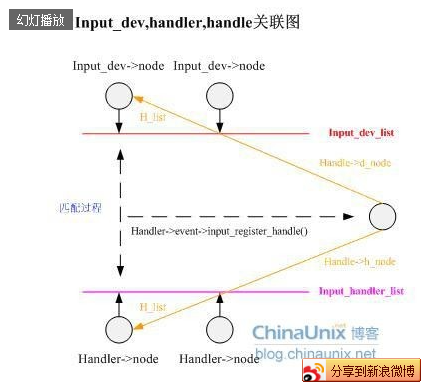
list\_add\_tail**(&**handle**->**h\_node**,** **&**handler**->**h\_list**);**

**if** **(**handler**->**start**)**

handler**->**start**(**handle**);**

**return** 0**;**

**}**



### 3.4 设备层驱动层

以第二节的按键为例分析设备驱动层，按键驱动通过input\_register\_device进行设备注册。

//1. 分配一个input\_dev结构体

buttons\_dev **=** input\_allocate\_device**();;**

//2. 设置

//2.1 能产生哪类事件 \*/

set\_bit**(**EV\_KEY**,** buttons\_dev**->**evbit**);**

set\_bit**(**EV\_REP**,** buttons\_dev**->**evbit**);**

//2.2 能产生这类操作里的哪些事件: L,S,ENTER,LEFTSHIT

set\_bit**(**KEY\_L**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_S**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_ENTER**,** buttons\_dev**->**keybit**);**

set\_bit**(**KEY\_LEFTSHIFT**,** buttons\_dev**->**keybit**);**

//3. 注册

input\_register\_device**(**buttons\_dev**);**

input\_register\_device源码分析如下：

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

**{**

//为生成设备节点序号做准备。

//下面的代码会调用dev\_set\_name()来设置设备节点的名字

//会在sysfs系统中以input0、input1、input2.....出现

static atomic\_t input\_no **=** ATOMIC\_INIT**(**0**);**

struct input\_handler **\***handler**;**

const char **\***path**;**

int error**;**

//设置evbit的EV\_SYN位，支持同步事件

set\_bit**(**EV\_SYN**,** dev**->**evbit**);**

//下面的定时器用于重复按键的检测

init\_timer**(&**dev**->**timer**);**

**if** **(!**dev**->**rep**[**REP\_DELAY**]** **&&** **!**dev**->**rep**[**REP\_PERIOD**])** **{**

dev**->**timer**.**data **=** **(**long**)** dev**;**

dev**->**timer**.**function **=** input\_repeat\_key**;**

dev**->**rep**[**REP\_DELAY**]** **=** 250**;**

dev**->**rep**[**REP\_PERIOD**]** **=** 33**;**

**}**

//key mapping设置函数

**if** **(!**dev**->**getkeycode**)**

dev**->**getkeycode **=** input\_default\_getkeycode**;**

**if** **(!**dev**->**setkeycode**)**

dev**->**setkeycode **=** input\_default\_setkeycode**;**

//将device加入input\_dev\_list链表

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

//在class下注册input0 input1等设备到内核

snprintf**(**dev**->**cdev**.**class\_id**,** **sizeof(**dev**->**cdev**.**class\_id**),**

"input%ld"**,** **(**unsigned long**)** atomic\_inc\_return**(&**input\_no**)** **-** 1**);**

**if** **(!**dev**->**cdev**.**dev**)**

dev**->**cdev**.**dev **=** dev**->**dev**.**parent**;**

error **=** class\_device\_add**(&**dev**->**cdev**);**

**if** **(**error**)**

**return** error**;**

path **=** kobject\_get\_path**(&**dev**->**cdev**.**kobj**,** GFP\_KERNEL**);**

printk**(**KERN\_INFO "input: %s as %s\n"**,**

dev**->**name **?** dev**->**name **:** "Unspecified device"**,** path **?** path **:** "N/A"**);**

kfree**(**path**);**

//遍历input\_handler\_list的handler，建立device和handler之间的联系

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

input\_attach\_handler**(**dev**,** handler**);**

input\_wakeup\_procfs\_readers**();**

**return** 0**;**

**}**

通过上面的代码将device加入到input\_dev\_list链表中，并和相应的handler建立联系，这里会和两个handler建立关系，分别为keyboard和evdev，也就是有两个handle挂在device的h\_list上。

继续分析，当有按键事件发生时，通过input\_event上报键值。

input\_event**(**buttons\_dev**,** EV\_KEY**,** pindesc**->**key\_val**,** 0**);**

input\_sync**(**buttons\_dev**);**

input\_event源码如下：

//以按键为例type=EV\_KEY code=KEY\_L value=0/1

void input\_event**(**struct input\_dev **\***dev**,** unsigned int type**,** unsigned int code**,** int value**)**

**{**

struct input\_handle **\***handle**;**

**if** **(**type **>** EV\_MAX **||** **!**test\_bit**(**type**,** dev**->**evbit**))**

**return;**

add\_input\_randomness**(**type**,** code**,** value**);**

**switch** **(**type**)** **{**

**case** EV\_SYN**:**

**switch** **(**code**)** **{**

**case** SYN\_CONFIG**:**

**if** **(**dev**->**event**)**

dev**->**event**(**dev**,** type**,** code**,** value**);**

**break;**

**case** SYN\_REPORT**:**

**if** **(**dev**->**sync**)**

**return;**

dev**->**sync **=** 1**;**

**break;**

**}**

**break;**

**case** EV\_KEY**:**

//测试设备的键值是否注册了，如果没有注册或者按键没有改变，不应该发送出去

**if** **(**code **>** KEY\_MAX **||** **!**test\_bit**(**code**,** dev**->**keybit**)** **||** **!!**test\_bit**(**code**,** dev**->**key**)** **==** value**)**

**return;**

//value不能为2，2表示重复按键事件

**if** **(**value **==** 2**)**

**break;**

//改变按键状态

change\_bit**(**code**,** dev**->**key**);**

//检查是否定义了repeat事件,如果定义了repeat事件，启动定时器计时

**if** **(**test\_bit**(**EV\_REP**,** dev**->**evbit**)** **&&** dev**->**rep**[**REP\_PERIOD**]** **&&** dev**->**rep**[**REP\_DELAY**]** **&&** dev**->**timer**.**data **&&** value**)** **{**

dev**->**repeat\_key **=** code**;**

mod\_timer**(&**dev**->**timer**,** jiffies **+** msecs\_to\_jiffies**(**dev**->**rep**[**REP\_DELAY**]));**

**}**

**break;**

**......**

**if** **(**type **!=** EV\_SYN**)**

dev**->**sync **=** 0**;**

**if** **(**dev**->**grab**)**

dev**->**grab**->**handler**->**event**(**dev**->**grab**,** type**,** code**,** value**);**

**else**

//从device的handle链表中，查询对应的handler，一个device可以对应多个

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

**if** **(**handle**->**open**)** //如果对应的handler设备打开了

//执行handler的event函数，这里为evdev\_event函数

handle**->**handler**->**event**(**handle**,** type**,** code**,** value**);**

**}**

这个设备既属于evdev设备，又属于kbd设备，所以分别会调用evdev\_event和kbd\_event。

evdev\_event将键值相关数据存放到对应的环形缓冲中，如果一个evdev设备被打开了多次，那么每个环形队列里面都存放了相同的数据。

static void evdev\_event**(**struct input\_handle **\***handle**,** unsigned int type**,** unsigned int code**,** int value**)**

**{**

struct evdev **\***evdev **=** handle**->**private**;**

struct evdev\_client **\***client**;**

    //如果该evdev有个专用的client,那么就将事件发给它

    //如果该evdev不存在专用的client,那个就把该事件发送给evdev上client\_list链表上所有的client

**if** **(**evdev**->**grab**)** **{**

client **=** evdev**->**grab**;**

do\_gettimeofday**(&**client**->**buffer**[**client**->**head**].**time**);**

client**->**buffer**[**client**->**head**].**type **=** type**;**

client**->**buffer**[**client**->**head**].**code **=** code**;**

client**->**buffer**[**client**->**head**].**value **=** value**;**

client**->**head **=** **(**client**->**head **+** 1**)** **&** **(**EVDEV\_BUFFER\_SIZE **-** 1**);**

kill\_fasync**(&**client**->**fasync**,** SIGIO**,** POLL\_IN**);**

**}** **else**

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

//将按键的时间/type/code/value保存到环形队列中

do\_gettimeofday**(&**client**->**buffer**[**client**->**head**].**time**);**

client**->**buffer**[**client**->**head**].**type **=** type**;**

client**->**buffer**[**client**->**head**].**code **=** code**;**

client**->**buffer**[**client**->**head**].**value **=** value**;**

client**->**head **=** **(**client**->**head **+** 1**)** **&** **(**EVDEV\_BUFFER\_SIZE **-** 1**);**

//发送异步信号

kill\_fasync**(&**client**->**fasync**,** SIGIO**,** POLL\_IN**);**

**}**

//唤醒读操作

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

**}**

kbd\_event源码如下：

static void kbd\_event**(**struct input\_handle **\***handle**,** unsigned int event\_type**,**unsigned int event\_code**,** int value**)**

**{**

**if** **(**event\_type **==** EV\_MSC **&&** event\_code **==** MSC\_RAW **&&** HW\_RAW**(**handle**->**dev**))**

kbd\_rawcode**(**value**);**

**if** **(**event\_type **==** EV\_KEY**)**

kbd\_keycode**(**event\_code**,** value**,** HW\_RAW**(**handle**->**dev**));**

tasklet\_schedule**(&**keyboard\_tasklet**);**

do\_poke\_blanked\_console **=** 1**;**

schedule\_console\_callback**();**

**}**

### 3.5 从read开始分析

在应用程序中我们可以读取/dev/input/eventX和/dev/tty来读取按键的键值，下面来分析下read /dev/input/eventX到底发生了什么？

还是以第二节中的按键驱动为例，设备为/dev/input/event1。

#### 3.5.1 open过程

open /dev/input/event1实际调用的是input字符设备，也就是会调用到input\_open\_file函数，在该函数中先通过从设备号(65)，得到input\_handler是evdev类型。接下来将file的f\_op设置为evdev的fops，最后调用evdev handler的open函数evdev\_open。

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

**{**

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

const struct file\_operations **\***old\_fops**,** **\***new\_fops **=** **NULL;**

int err**;**

**if** **(!**handler **||** **!(**new\_fops **=** fops\_get**(**handler**->**fops**)))**

**return** **-**ENODEV**;**

**......**

file**->**f\_op **=** new\_fops**;**

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

**......**

**return err;**

**}**

evdev\_open主要创建了evdev\_client，并添加到evdev的链表中。

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

**{**

struct evdev\_client **\***client**;**

struct evdev **\***evdev**;**

int i **=** iminor**(**inode**)** **-** EVDEV\_MINOR\_BASE**;**

int error**;**

**if** **(**i **>=** EVDEV\_MINORS**)**

**return** **-**ENODEV**;**

//通过从设备号在evdev\_table中找到对应的evdev结构体

evdev **=** evdev\_table**[**i**];**

**if** **(!**evdev **||** **!**evdev**->**exist**)**

**return** **-**ENODEV**;**

//每open该设备，创建一个evdev\_client

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

**if** **(!**client**)**

**return** **-**ENOMEM**;**

//将evdev\_clinet增加到evdev中的client\_list链表中

client**->**evdev **=** evdev**;**

list\_add\_tail**(&**client**->**node**,** **&**evdev**->**client\_list**);**

**if** **(!**evdev**->**open**++** **&&** evdev**->**exist**)** **{**

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

**if** **(**error**)** **{**

list\_del**(&**client**->**node**);**

kfree**(**client**);**

**return** error**;**

**}**

**}**

//设置私有数据，方便读取数据

file**->**private\_data **=** client**;**

**return** 0**;**

**}**

#### 3.5.2 read过程

read /dev/input/event1实际调用的是evdev\_read函数。

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

**{**

//获取私有数据

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

struct evdev **\***evdev **=** client**->**evdev**;**

int retval**;**

**if** **(**count **<** evdev\_event\_size**())**

**return** **-**EINVAL**;**

**if** **(**client**->**head **==** client**->**tail **&&** evdev**->**exist **&&** **(**file**->**f\_flags **&** O\_NONBLOCK**))**

**return** **-**EAGAIN**;**

//如果环形队列没数据，则休眠，等待input\_event的发生。

retval **=** wait\_event\_interruptible**(**evdev**->**wait**,**

client**->**head **!=** client**->**tail **||** **!**evdev**->**exist**);**

**if** **(**retval**)**

**return** retval**;**

**if** **(!**evdev**->**exist**)**

**return** **-**ENODEV**;**

**while** **(**client**->**head **!=** client**->**tail **&&** retval **+** evdev\_event\_size**()** **<=** count**)** **{**

//上报数据给用户空间

struct input\_event **\***event **=** **(**struct input\_event **\*)** client**->**buffer **+** client**->**tail**;**

**if** **(**evdev\_event\_to\_user**(**buffer **+** retval**,** event**))**

**return** **-**EFAULT**;**

//更新环形队列

client**->**tail **=** **(**client**->**tail **+** 1**)** **&** **(**EVDEV\_BUFFER\_SIZE **-** 1**);**

retval **+=** evdev\_event\_size**();**

**}**

**return** retval**;**

**}**

## 4.其他

再看输入子系统的作用，以mouse为例，硬件驱动只负责上报mouse相关的键值，至于键值如何处理，那是输入子系统完成的事情，将驱动和行为完全区分开来了，假如有多个mouse设备，这些键值都会上报到/dev/mice总接口处，统一管理。

在目前的嵌入式开发中，很多按键都不是标准的按键作用，所以实际上和字符设备相比，借用输入子系统并没有多大的好处，只是输入子系统帮我们完成了read/poll/open等操作，让开发者关注驱动本身的实现，如果一些非常规按键需要上报，可以采用EV\_MSC。

EV\_SYN 0x00 同步事件

EV\_KEY 0x01 按键事件，如KEY\_VOLUMEDOWN

EV\_REL 0x02 相对坐标**,** 如鼠标上报的坐标

EV\_ABS 0x03 绝对坐标，如触摸屏上报的坐标

EV\_MSC 0x04 其它

EV\_LED 0x11 LED

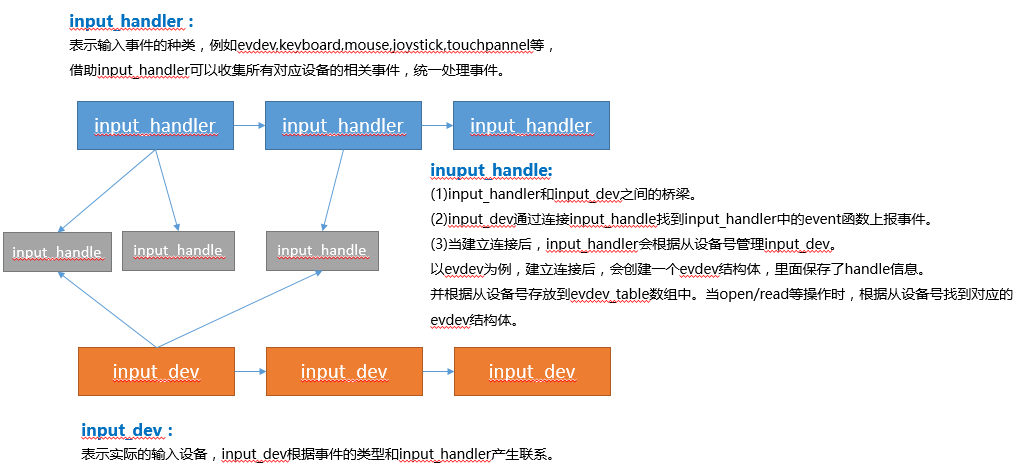
EV\_SND 0x12 声音

EV\_REP 0x14 Repeat

EV\_FF 0x15 力反馈

EV\_PWR 0x16     电源

EV\_FF\_STATUS 0x17 状态



keyboard.c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。