**linux内核input子系统解析**

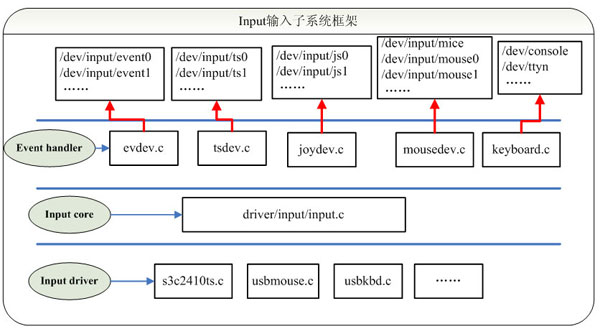
**作者：刘老师,**[华清远见嵌入式学院](http://www.embedu.org/)讲师。

Android、X windows、qt等众多应用对于linux系统中键盘、鼠标、触摸屏等输入设备的支持都通过、或越来越倾向于标准的input输入子系统。

因为input子系统已经完成了字符驱动的文件操作接口，所以编写驱动的核心工作是完成input系统留出的接口，工作量不大。但如果你想更灵活的应用它，就需要好好的分析下input子系统了。

一、input输入子系统框架

下图是input输入子系统框架，输入子系统由输入子系统核心层（ Input Core ），驱动层和事件处理层（Event Handler）三部份组成。一个输入事件，如鼠标移动，键盘按键按下，joystick的移动等等通过 input driver -> Input core -> Event handler -> userspace 到达用户空间传给应用程序。



注意：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。

二、Input driver编写要点

**1、分配、注册、注销input设备**

struct input\_dev \*input\_allocate\_device(void)  
int input\_register\_device(struct input\_dev \*dev)  
void input\_unregister\_device(struct input\_dev \*dev)

**2、设置input设备支持的事件类型、事件码、事件值的范围、input\_id等信息**

参见usb键盘驱动：usbkbd.c

usb\_to\_input\_id(dev, &input\_dev->id);//设置bustype、vendo、product等  
input\_dev->evbit[0] = BIT(EV\_KEY) | BIT(EV\_LED) | BIT(EV\_REP);//支持的事件类型  
input\_dev->ledbit[0] = BIT(LED\_NUML) | BIT(LED\_CAPSL) | BIT(LED\_SCROLLL) | BIT(LED\_COMPOSE) | BIT(LED\_KANA);// EV\_LED事件支持的事件码  
for (i = 0; i < 255; i++)  
  set\_bit(usb\_kbd\_keycode[i], input\_dev->keybit); //EV\_KEY事件支持的事件码

include/linux/input.h中定义了支持的类型（下面列出的是2.6.22内核的情况）

#define EV\_SYN           0x00  
        #define EV\_KEY           0x01  
        #define EV\_REL           0x02  
        #define EV\_ABS           0x03  
        #define EV\_MSC           0x04  
        #define EV\_SW            0x05  
        #define EV\_LED           0x11  
        #define EV\_SND          0x12  
        #define EV\_REP          0x14  
        #define EV\_FF            0x15  
        #define EV\_PWR         0x16  
        #define EV\_FF\_STATUS        0x17  
        #define EV\_MAX           0x1f

一个设备可以支持一个或多个事件类型。每个事件类型下面还需要设置具体的触发事件码。比如：EV\_KEY事件，需要定义其支持哪些按键事件码。

**3、如果需要，设置input设备的打开、关闭、写入数据时的处理方法**

参见usb键盘驱动：usbkbd.c

input\_dev->open = usb\_kbd\_open;  
input\_dev->close = usb\_kbd\_close;  
input\_dev->event = usb\_kbd\_event;

**4、在发生输入事件时，向子系统报告事件**

用于报告EV\_KEY、EV\_REL、EV\_ABS等事件的函数有：

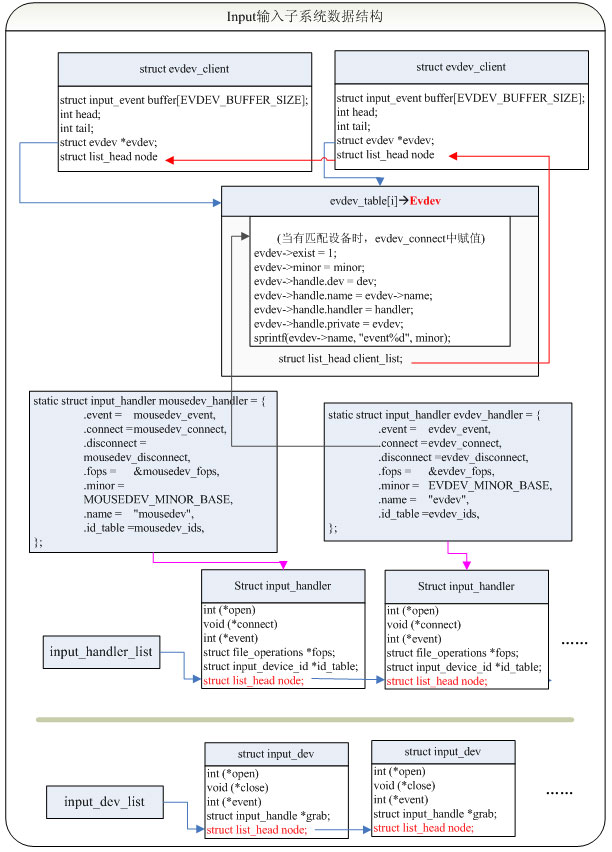
void input\_report\_key(struct input\_dev \*dev, unsigned int code, int value)  
void input\_report\_rel(struct input\_dev \*dev, unsigned int code, int value)  
void input\_report\_abs(struct input\_dev \*dev, unsigned int code, int value)

如果你觉得麻烦，你也可以只记住1个函数（因为上述函数都是通过它实现的）

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

三、Event Handler层解析

**1、Input输入子系统数据结构关系图**



**2、input\_handler结构体**

以evdev.c中的evdev\_handler为例：

static struct input\_handler evdev\_handler = {  
    .event = evdev\_event, //向系统报告input事件，系统通过read方法读取  
    .connect = evdev\_connect, //和input\_dev匹配后调用connect构建  
    .disconnect = evdev\_disconnect,  
    .fops = &evdev\_fops, //event设备文件的操作方法  
    .minor = EVDEV\_MINOR\_BASE, //次设备号基准值  
    .name = "evdev",  
    .id\_table = evdev\_ids, //匹配规则  
  };

**3、input字符设备注册过程**

drivers/input/input.c中：  
static int \_\_init input\_init(void)  
{  
   int err;  
   err = class\_register(&input\_class);  
   ……  
   err = register\_chrdev(INPUT\_MAJOR, "input", &input\_fops);  
   ……  
}

input\_fops定义：

static const struct file\_operations input\_fops = {  
   .owner = THIS\_MODULE,  
   .open = input\_open\_file,  
};

Input\_dev和input\_handler匹配后调用input\_handler的connect。以evdev\_handler为例：

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;

   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;  
   }

   evdev = kzalloc(sizeof(struct evdev), GFP\_KERNEL);//为每个匹配evdev\_handler的设备创建一个evdev。  
    if (!evdev)  
      return -ENOMEM;

    INIT\_LIST\_HEAD(&evdev->client\_list);  
    init\_waitqueue\_head(&evdev->wait);

    evdev->exist = 1;  
    evdev->minor = minor;  
    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;//记录evdev的位置，字符设备/dev/input/evnetx访问时根据次设备号及EVDEV\_MINOR\_BASE最终在evdev\_open中找到对应的evdev  
    **devt = MKDEV(INPUT\_MAJOR, EVDEV\_MINOR\_BASE + minor),  
    cdev = class\_device\_create(&input\_class, &dev->cdev, devt,dev->cdev.dev, evdev->name);//创建了event字符设备节点  
    ……**  
 }

**4、input字符设备的打开过程**

static int input\_open\_file(struct inode \*inode, struct file \*file)  
{  
    struct input\_handler \*handler = input\_table[iminor(inode) >> 5];  
                //得到对应的input\_handler  
    const struct file\_operations \*old\_fops, \*new\_fops = NULL;  
    int err;  
    if (!handler || !(new\_fops = fops\_get(handler->fops)))  
                //取出对应input\_handler的file\_operations  
        return -ENODEV;  
    if (!new\_fops->open) {  
        fops\_put(new\_fops);  
        return -ENODEV;  
    }  
    old\_fops = file->f\_op;  
    file->f\_op = new\_fops;//重定位打开的设备文件的操作方法  
    err = new\_fops->open(inode, file);  
    if (err) {  
        fops\_put(file->f\_op);  
        file->f\_op = fops\_get(old\_fops);  
    }  
    fops\_put(old\_fops);  
      return err;  
}

**5、input字符设备的其它操作**

由于在open阶段已经把设备文件的操作操作方法重定位了到了具体的input\_handler，所以其它接口操作（read、write、ioctl等），由各个input\_handler的fops方法决定。如evdev.c中的：evdev\_fops。