网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

输入子系统框架:

回顾：以中断方式处理的“按键驱动”程序。
1，确定主设备号。

```
int major;
static int third_drv_init(void)
{
    major = register_chrdev(0, "third_drv", &sencod_drv_fops);
```

2，构造一个“file_operations”结构体。

```
static struct file_operations sencod_drv_fops = {
    .owner   =  THIS_MODULE,    /* 这是一个宏，推向编译模块时自动创建的__this_module变量 */
    .open    =  third_drv_open,
    .read    =  third_drv_read,
    .release =  third_drv_close,
};
```

3，open 函数中申请中断：

```
static int third_drv_open(struct inode *inode, struct file *file)
{
    /* 配置GPF0,2为输入引脚 */
    /* 配置GPG3,11为输入引脚 */
    request_irq(IRQ_EINT0,  buttons_irq, IRQT_BOTHEDGE, "S2", &pins_desc[0]);
    request_irq(IRQ_EINT2,  buttons_irq, IRQT_BOTHEDGE, "S3", &pins_desc[1]);
    request_irq(IRQ_EINT11, buttons_irq, IRQT_BOTHEDGE, "S4", &pins_desc[2]);
    request_irq(IRQ_EINT19, buttons_irq, IRQT_BOTHEDGE, "S5", &pins_desc[3]);

    return 0;
}
```

4，read 函数中，若没有按键按下就休眠：

```
/* 如果没有按键动作，休眠 */
wait_event_interruptible(button_waitq, ev_press);
```

有按键按下后，中断程序被调用：在中断服务程序里确定是哪个按键按下。最后唤醒按键程序：

```
ev_press = 1;                          /* 表示中断发生了 */
wake_up_interruptible(&button_waitq);   /* 唤醒休眠的进程 */
```

而这个驱动的测试程序是：
1，先 open 某个设备文件：

```
fd = open("/dev/buttons", O_RDWR);
```

2，接着读：

```
while (1)
{
    read(fd, &key_val, 1);
    printf("key_val = 0x%x\n", key_val);
}
```

上面的驱动程序和测试程序的缺点：
上面打开了一个特定的设备文件“/dev/buttons”。而一般写的应用程序不会去打开这个“/dev/buttons”。一般打开的都是原有的文件，如“dev/tty*”,还有可能是不需要打开什么设备文件，而 是直接“scanf()”就去获得了按键的输入。

以前写的那些驱动程序只能自己使用而非通用。要写一个通用的驱动程序，让其他应用程序“无缝”的使用， 就是说不需要修改人家的应用程序。这需要使用现成的驱动，把自己的设备相关的驱动放到内核中这种驱动架构 中去。这个现成的驱动就是“输入子系统--input 子系统”。
要将自己的驱动整合进“input 子系统”，就得把它的框架理清楚：

一、 输入子系统框架：

1. 最上层（核心层）：

drivers/input.c 所以输入子系统的代码在这个 c 文件中。
看一个驱动程序是从“入口函数”开始查看。

```
static int __init input_init(void)
```

以前注册字符设备驱动的函数是自己写的。上面的注册是内核有的。

```
err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
```

```
#define INPUT_MAJOR     13
```

```
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“INPUT_MAJOR”为 13 的字符设备，名字为“input”，它的 file_operations 结构是“input_fops”。这个结构中只有一个“.open”函数。
输入子系统，如这里想读按键，而这里只有一个“open”函数，那么这个 Open 函数中应该做了某些工作。

分析“int input_open_file(struct inode *inode, struct file *file)”：

```
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;

    /* No load-on-demand here? */
    if (!handler || !(new_fops = fops_get(handler->fops)))
        return -ENODEV;

    /*
     * That's _really_ odd. Usually NULL ->open means "nothing special",
     * not "no device". Oh, well...
     */
    if (!new_fops->open) {
        fops_put(new_fops);
        return -ENODEV;
    }
    old_fops = file->f_op;
    file->f_op = new_fops;

    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;
} ? end input_open_file ?
```

其中有一个“input_handler *”(“输入处理器”或“输入处理句柄”):

```
struct input_handler {

    void *private;

    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
    int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);
    void (*disconnect)(struct input_handle *handle);
    void (*start)(struct input_handle *handle);

    const struct file_operations *fops;
    int minor;
    const char *name;

    const struct input_device_id *id_table;
    const struct input_device_id *blacklist;

    struct list_head    h_list;
    struct list_head    node;
};
```

```
struct input_handler *handler = input_table[iminor(inode) >> 5];
```

这里这个“输入处理句柄”结构指向一个“input_table[]”数组。从这个数组里面根据这个“次设 备号 iminor(inode) >> 5”把打开的文件，根据它的次设备号找到一项。

```
const struct file_operations *old_fops, *new_fops = NULL;
int err;

/* No load-on-demand here? */
if (!handler || !(new_fops = fops_get(handler->fops)))
    return -ENODEV;
```

定义了一个新的file_operation结构变量

将input_handler结构中的file_operations赋给new_fops。

接着新的“file_operations”结构"new_fops"等于上面的“input_handler *”指针变量

handler 的成员“ops”（这是一个 file_operations 结构。Input_handler 结构中有这个 file_operations 成员）。

```
file->f_op = new_fops;
```

接着把这个新的 file_operations 结构赋给此函数“input_open_file”的形参“file”的 f_op。然后再调用这个新的 file_operations 结构“new_fops”的 open（inode,file）函数，如下：

```
err = new_fops->open(inode, file);
```

这样以后要来读按键时，用到的是“struct input_handler *handler”中的“new_fops”.

Input.c 只是一个“中转”作用。最终还会用到“input_table[]”.

(1) **流程如下：**input.c

(2) int __init input_init(void)：

-> err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
其中“input_fops”结构中只有一个".open"函数：
static const struct file_operations input_fops = {
.owner = THIS_MODULE,
.open = input_open_file,
};

问题：怎么读按键？
这里只有一个 Open 函数而没有读函数。则可能是这个 “input_open_file”函数中做了什么工作。

(3) **分析“**input_open_file**”函数：**

int input_open_file(struct inode *inode, struct file *file)：
->input_handler *handler = input_table[iminor(inode) >> 5];
根据传进来的“inode” 这个打开的文件的次设备号（“iminor(inode) >> 5”）得到一个
“input_handler *handler”。
->new_fops = fops_get(handler->fops):
然后的新的 file_operations 结构体（new_fops）等于这个“input_handler *handler”结构里面的“file_operations”结构“handler->fops”。
->file->f_op = new_fops:
然后所打开的这个 file 中的 f_op(file_operations）等于这个“new_fops”。
->err = new_fops->open(inode, file)：
接着调用这个 new_fops 的“open”函数。

以后 APP 来读的时候，是最终调用“file->f_op”中的 read 函数。最终是要明白“input_handler”是如何定义的，input_table 数组由谁构造。

(4) "input_table[]"的构造：

“static struct input_handler *input_table[8]”是个静态变量，所以只能用在这个文件里面，所以只 在这个 Input.c 中找到使用了这个数组的地方，如下：
int input_register_handler(struct input_handler *handler)这个函数中构造了这个 input_table[]数 组项：

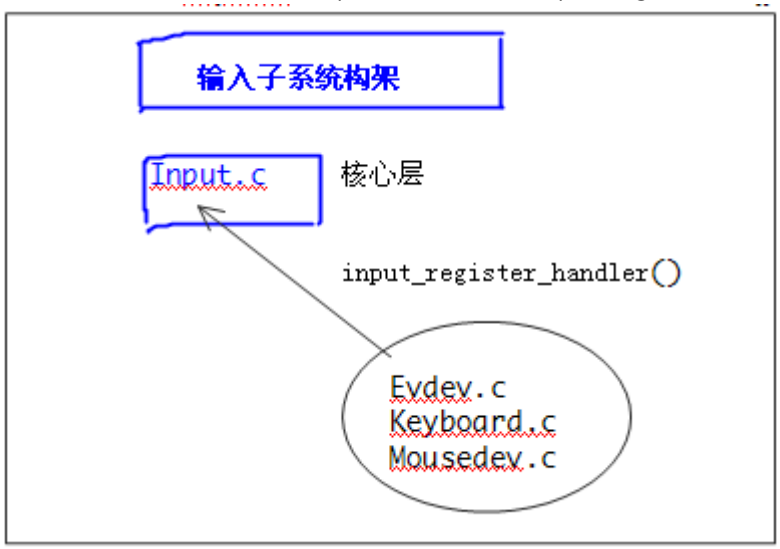
```
if (handler->fops != NULL) {
     if (input_table[handler->minor >> 5])
          return -EBUSY;

     input_table[handler->minor >> 5] = handler;
}
```

(5) 这个“input_register_handler ()”函数被谁调用过：

搜索源码工程，见到如下：

```
Evdev.c (drivers\input):    return input_register_handler(&evdev_handler);
Input.c (drivers\input):int input_register_handler(struct input_handler *handler)
Input.c (drivers\input):EXPORT_SYMBOL(input_register_handler);
Input.h (include\linux):int input_register_handler(struct input_handler *);
Joydev.c (drivers\input):   return input_register_handler(&joydev_handler);
Keyboard.c (drivers\char):  error = input_register_handler(&kbd_handler);
Mousedev.c (drivers\input): error = input_register_handler(&mousedev_handler);
```

有游戏手柄，键盘和鼠标等的源代码调用过它。这些使用就离开了“input.c”这个核心层了。它们就向上 面的核心层“input.c”注册了“input_register_handler()”.



打开这个“evdev.c”源码。查看调用“input_register_handler()”的调用：

```
static int __init evdev_init(void)
{
    return input_register_handler(&evdev_handler);
}
```

只是在这个入口函数“evdev_init()”中调用了"input_register_handler()"。
以上是具体的设备与核心层 input.c 的层次调用关系。

（6）**查看“read”的过程：**

```
return input_register_handler(&evdev_handler);
```

在“evdev.c”的入口函数“evdev_init()”中注册了“evdev_handler”这个结构。它的原型如下：

```
struct input_handler {

    void *private;

    void (*event)(struct input_handle *handle, unsigned int type, unsigned int code, int value);
    int (*connect)(struct input_handler *handler, struct input_dev *dev, const struct input_device_id *id);
    void (*disconnect)(struct input_handle *handle);
    void (*start)(struct input_handle *handle);

    const struct file_operations *fops;
    int minor;
    const char *name;

    const struct input_device_id *id_table;
    const struct input_device_id *blacklist;

    struct list_head    h_list;
    struct list_head    node;
};
```

它是一个“input_handler”结构体。它的定义如下：

```
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,
};
```

从上面的"input_handler"结构变量“evdev_handler”的定义可以看到一个“file_operations”结构“.fops = &evdev_fops”。在这个“evdev_fops”中便有相关的 read,write 等。

```
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
};
```

以前自己写驱动时，这个 file_operations 结构体是自己构造的，这里是由系统构造了。

```
.fops =     &evdev_fops,
.minor =    EVDEV_MINOR_BASE,
```

```
#define EVDEV_MINOR_BASE    64
#define EVDEV_MINORS        32
#define EVDEV_BUFFER_SIZE   64
```

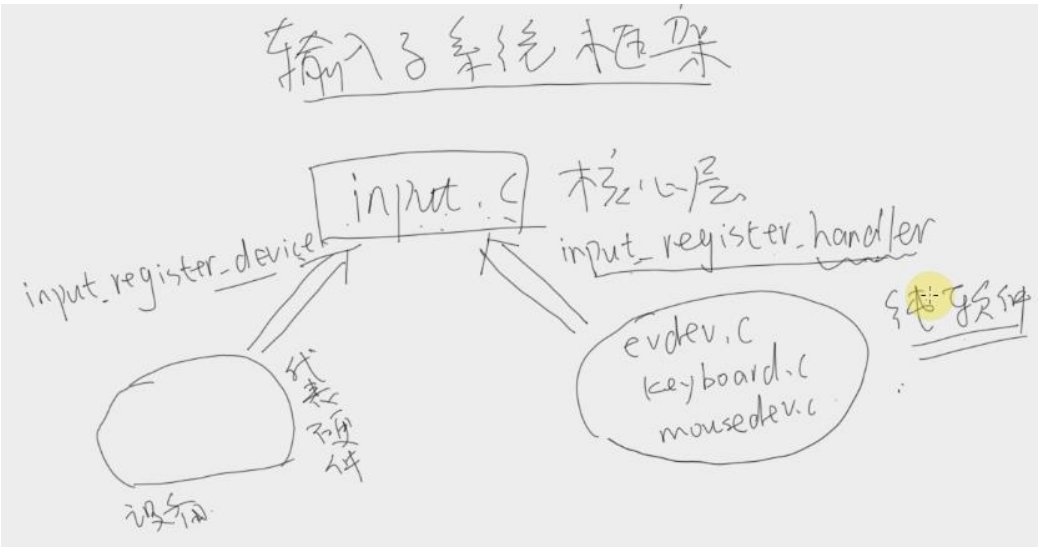
次设备号是“64”。
通过“input_register_handler(&evdev_handler)”注册后，就放到：

```
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;
    }
```

64右移5位

通过传值后，形参“handler”就是“&evdev_handler”，则“handler->minor”就是“evdev_handler->minor”即“EVDEV_MINOR_BASE”（即次设备号 64）。
就相当于 64 除以 2 的 5 次方，即 64 除以 32 为 2.
则这个“handler”是放在“input_table[2]”这个第二项处。handler 是纯软件的概念。

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”是否支持另一边的“device”，中间会有一个联系：

```
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,
};
```

“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)：

```
+--- input_register_device Matches (129 in 115 files) ----
A3d.c (drivers\input\joystick): err = input_register_device(a3d->dev);
Acecad.c (drivers\input\tablet):    err = input_register_device(acecad->input);
Adbhid.c (drivers\macintosh):   err = input_register_device(input_dev);
Adi.c (drivers\input\joystick):         err = input_register_device(port->adi[i].dev);
Ads7846.c (drivers\input\touchscreen):  err = input_register_device(input_dev);
Aiptek.c (drivers\input\tablet):    err = input_register_device(aiptek->inputdev);
Alps.c (drivers\input\mouse):   if (input_register_device(priv->dev2))
Amijoy.c (drivers\input\joystick):      err = input_register_device(amijoy_dev[i]);
Amikbd.c (drivers\input\keyboard):  err = input_register_device(amikbd_dev);
Amimouse.c (drivers\input\mouse):   err = input_register_device(amimouse_dev);
Ams-input.c (drivers\hwmon\ams):    if (input_register_device(ams_info.idev)) {
Analog.c (drivers\input\joystick):  error = input_register_device(analog->dev);
Applesmc.c (drivers\hwmon): ret = input_register_device(applesmc_idev);
Appletouch.c (drivers\input\mouse): error = input_register_device(dev->input);
Atakbd.c (drivers\input\keyboard):  input_register_device(atakbd_dev);
Atarimouse.c (drivers\input\mouse): input_register_device(atamouse_dev);
Ati_remote.c (drivers\input\misc):  err = input_register_device(ati_remote->idev);
Ati_remote2.c (drivers\input\misc): retval = input_register_device(idev);
```

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。

(2) 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：
input_register_handler
    // 放入数组
    input_table[handler->minor >> 5] = handler;

    // 放入链表
    list_add_tail(&handler->node, &input_handler_list);

    // 对于每个input_dev，调用input_attach_handler
    list_for_each_entry(dev, &input_dev_list, node)
        input_attach_handler(dev, handler); // 根据input_handler的id_table判断能否支持这个input_dev

注册输入设备：
input_register_device
    // 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_handler的id_table判断能否支持这个input_dev
```

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);//1.根据handler->id_table与形参dev这个输入设备比较.
    if (!id) //1.1.看dev与处理方式中的id_table比较是否有匹配.
        return -ENODEV;

    error = handler->connect(handler, dev, id);//1.2.若匹配则调用处理方式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;
} ? end input_attach_handler ?
```

即：

```
input_attach_handler //调用input_match_device检测id匹配情况,然后调用handler中的connect函数.
    id = input_match_device(handler->id_table, dev);

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

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 都有自己不同的方式。

实例分析“evdev.c”中的“connect”：

```
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,
};
```

(3) 进入“evdev_connect 函数”分析：

```
evdev = kzalloc(sizeof(struct evdev), GFP_KERNEL); //分配一个input_handle结构体.
```

1) ，分配了一个“input_handle evdev”结构变量

（不是 input_handler 结构）。

查看这个 evdev 结构中的成员：

```
struct evdev {
    int exist;
    int open;
    int minor;
    char name[16];
    struct input_handle handle;
    wait_queue_head_t wait;
    struct evdev_client *grab;
    struct list_head client_list;
};
struct input_handle {

    void *private;

    int open;
    const char *name;

    struct input_dev *dev;有输入设备
    struct input_handler *handler; 有处理方式

    struct list_head    d_node;
    struct list_head    h_node;
};
```

成员中有一个“input_handle handle”结构。

2) 再对这个“input_handle evdev”进行设置：

```
evdev->handle.dev = dev; //指向左边的input_dev结构体(输入设备)
evdev->handle.name = evdev->name;
evdev->handle.handler = handler; //指向右边的input_handler结构体(处理方式)
evdev->handle.private = evdev;
```


3) 最后注册这个 handle：

```
error = input_register_handle(&evdev->handle); //注册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);
    list_add_tail(&handle->h_node, &handler->h_list);

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

    return 0;
}
```

将形参“handle”放到一个输入设备的链表里面：

```
list_add_tail(&handle->d_node, &handle->dev->h_list);
```

再把“handler”放到右边一个“h_list”链表里面。

```
list_add_tail(&handle->h_node, &handler->h_list);
```

怎么建立连接？

1. 分配一个input_handle结构体
2.
```
    input_handle.dev = input_dev;  // 指向左边的input_dev
    input_handle.handler = input_handler;  // 指向右边的input_handler
```
3. 注册：
```
   input_handler->h_list = &input_handle; //input_handler中的h_list指向结构体“input_handle”
   input_dev->h_list     = &input_handle; //input_dev中的h_list指向结构体“input_handle”
  /* 找到连接过程中的“input_handle”结构后通过此结构中的“dev”和“handler”指向左边的输入
   设备“input_dev”和右边的“input_handler”.这样最终建立连接。 */
```

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”

```
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,
};
```

是file_operations结构，其中有读函数。

```
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
};
```

怎么读按键？

app: read

```
-------------------------
   .......
        evdev_read
            // 无数据并且是非阻塞方式打开，则立刻返回
            if (client->head == client->tail && evdev->exist && (file->f_flags & O_NONBLOCK))
                return -EAGAIN;

            // 否则休眠
            retval = wait_event_interruptible(evdev->wait,
                client->head != client->tail || !evdev->exist);
```

在“evdev_read()”中：

```
    if (count < evdev_event_size()) //evdev_event_size指事件的大小.
        return -EINVAL;
//若client缓冲区的头部等于它的尾部(环形缓冲区)则表示缓冲区里面数据是空的。
//(file->f_flags & O_NONBLOCK)是指这个文件是以"非阻塞方式"打开.
if (client->head == client->tail && evdev->exist && (file->f_flags & O_NONBLOCK))
        return -EAGAIN; //以上条件就返回一个"-EAGAIN"让你再次尝试.这和以前的阻塞非阻塞对应起来.
//上面判断后若不是非阻塞而是阻塞的，则下面马上要"休眠"。
    retval = wait_event_interruptible(evdev->wait,
        client->head != client->tail || !evdev->exist);
```

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。

```
谁来唤醒？
evdev_event()
    wake_up_interruptible(&evdev->wait);
```

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()”：**

```
evdev_event被谁调用？
猜：应该是硬件相关的代码，input_dev那层调用的
在设备的中断服务程序里，确定事件是什么，然后调用相应的input_handler的event处理函数：
下面是一个例子：
gpio_keys_isr
    // input_event()是用来上报事件
    input_event(input, type, button->code, !!state);
    input_sync(input);

input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)
    struct input_handle *handle;
    //在input_event()中有一个“input_dev *dev”形参，在此函数中有一个“input_handle *
handle”，设备和处理都在这里。
    //下面是对这个链接“h_list”里的每一项(有input_handle和input_dev)handle扫描，若这个handle-
>open打开过，则这个handle(相当于上面所说的ipnut_handle这个连接过程结构体)的handler指向右边“软件部分”的event函
数。
    list_for_each_entry(handle, &dev->h_list, d_node)
        if (handle->open)
            handle->handler->event(handle, type, code, value);
```

总结“输入子系统”：

(1) 分为上下两层：

核心层“input.c”.它里面有“register_chdev”（input_init()中），但它简单：

```
err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
```

因为这个 register_chrdev（）中的“input_fops”file_operations 结构很简单：

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};
```

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：

```
input_open_file(struct inode *inode, struct file *file)
```

根据打开的设备节点的次设备号找到一个“handler”：

```
struct input_handler *handler = input_table[iminor(inode) >> 5];
```

并且把这个文件的 f_op 指向新的“input_handler *handler”里面的“fops”：

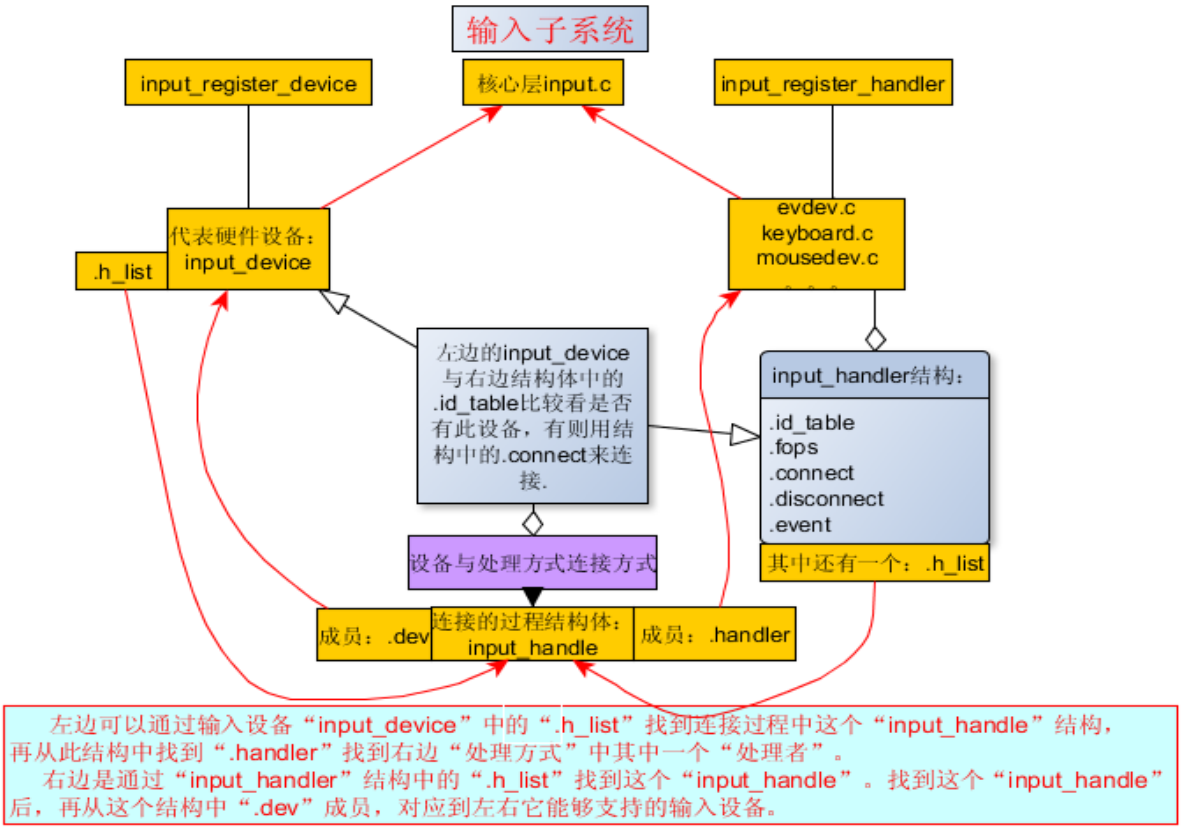
```
old_fops = file->f_op;
file->f_op = new_fops;
```

然后再调用这个“handler”中的 open 函数：

```
err = new_fops->open(inode, file);
```

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）.

“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。

写输入子系统实例：

怎么写符合输入子系统框架的驱动程序？
1. 分配一个input_dev结构体
2. 设置
3. 注册
4. 硬件相关的代码，比如在中断服务程序里上报事件

一个实例：gpio_keys.c （分析驱动从“init（）”函数开始）

（1）首先注册一个“平台驱动”：

```
struct platform_driver gpio_keys_device_driver = {
    .probe      = gpio_keys_probe,
    .remove     = __devexit_p(gpio_keys_remove),
    .driver     = {
        .name   = "gpio-keys",
    }
};
```

先不关心平台驱动，只关心平台驱动中的“.probe”函数：

```
static int __devinit gpio_keys_probe(struct platform_device *pdev)
```

a) 先定义了一个“input_dev”结构体：

```
struct input_dev *input;
```

b) 再设置这个结构体：

```
input->evbit[0] = BIT(EV_KEY);

input->name = pdev->name;
input->phys = "gpio-keys/input0";
input->dev.parent = &pdev->dev;

input->id.bustype = BUS_HOST;
input->id.vendor = 0x0001;
input->id.product = 0x0001;
input->id.version = 0x0100;
```

c) 注册：

```
error = input_register_device(input);
```


d) 其他的操作就是“硬件”相关的操作：

```
static irqreturn_t gpio_keys_isr(int irq, void *dev_id) //GPIO按键的中断处理函数.
```

e) 在“gpio_keys_isr()”中上报事件：

```
input_event(input, type, button->code, !!state);
input_sync(input);
```

二、自己写驱动：

下面是字符设备驱动程序的框架和步骤。

1，确定主设备号：major

2，构造一个 file_operations 结构体：

.open
.read
.write
、、、、其中有很多函数指针

3，上面的信息构造出来后要告诉内核（要使用）：

4，注册这个字符设备驱动程序：register.chrdev

5，谁来调用这个注册的字符设备驱动程序：入口函数。

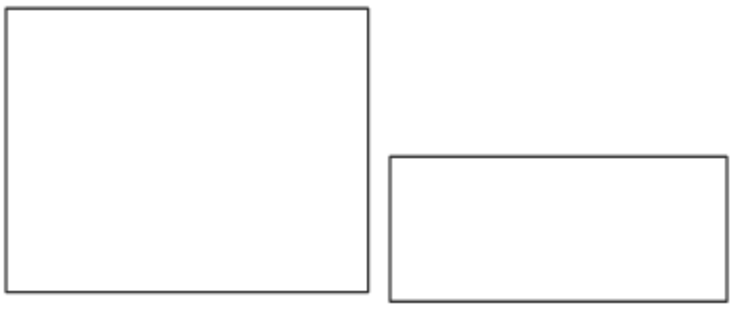
6，有入口函数，就会有“出口函数”。

现在用内核原有的框架,左边的步骤肯定也有（这是整个系统的基础）
Input 输入子系统。

输入子系统框架

核心层：input.c

Input_register_device
Input_register_handler



代表硬件设备概念：
Input_device
纯软件处理者概念：
Input_handler
如：
Evdev.c

Keyboard.c
Mousedev.c
比较看 id_table 是否支持些硬件设备

Input_handler 结构成员：
.id_table
.fops
.connect
.disconnect
.event
Input_device

若 id_table 支持此硬件设备，
则调用：.connect 函数。
当系统发现有一个新的输入设备时，如有“evdev.c”、“keyboard.c”等 handler 时，这些 handler 是否支持左边的代表硬件设备概念的硬件，就要看“input_handler”结构中的“.id_table”成员。就是让“input_device”与“id_table”作比较。若 id_table 表示能支持这个硬件设备时，就会调用 input_handler 中的“connect”函数，给“硬件”和“软件”作一个连接。

输入子系统例程编写：

字符驱动程序编写：

APP: open,　　read,　　write.　----> 对应提供驱动程序的读写等函数。

驱动：drv_open, drv_read,　drv_write

硬件

代码步骤：

1，确定主设备号：
可以自己确定，也可让内核分配。

2，要构造驱动中的“open,read,write 等”
是将它们放在一个“file_operations”结构体中.
File_operations==> .open, .read, .write, .poll 等。
这里"open"函数会去配置硬件的相关引脚等，还有注册中断。

3，register_chrdev 注册字符设备构造的“file_operations”结构：
使用这个 file_operations 结构体。是把这个结构放到内核的某个以此设备的“主设备号”为下标的数组中去。
Register_chrdev(主设备号，设备号，file_operations 结构）.

4，入口函数：
调用这个“register_chrdev()”函数。内核装载某个模块时，会自动调用这个“入口函数”。

5，出口函数

Input.c 中的如上步骤：

1，**主设备号：**

```
#define INPUT_MAJOR     13
```

2，file_operations **结构：**

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};
```

区别：

这个"file_operations"结构中只有一个“.open”函数。这里不像自己写字符设备驱动程序时在 open 函数中定义硬件引脚和注册中断等，这里的 open 函数是作为“中转”的作用。

在某个数组中找到所谓的“input_handler”结构，用这个结构中的“.fops”去读或写。

3，注册结构体：

```
err = register_chrdev(INPUT_MAJOR, "input", &input_fops);
```

4，入口函数：

```
static int __init input_init(void)
```

5，出口函数：

```
static void __exit input_exit(void)
```

实例编写：

①，参考：linux-2.6.22.6\drivers\input\keyboard\gpio_keys.c

②，先包含头文件 和 初略框架：

头文件：

```
/* 输入子系统参考: gpio_keys.c */
#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 <linux/gpio_keys.h>

#include <asm/gpio.h>
```

初略代码框架：

```c
//1. 先写入口函数:
static int buttons_init(void)
{
    return 0;
}

//2.再写出口函数:
static void buttons_exit(void)
{

}

//3.入口、出口函数只是普通的函数，驱动中要修饰它们:
module_init(buttons_init);
module_exit(buttons_exit);
MODULE_LICENSE("GPL");
```

③，入口函数的框架:

```c
static int buttons_init(void)
{
    //1.1.分配一个 input_dev 结构体:

    //1.2.设置 input_dev 结构体:

    //1.3.注册

    //1.4.硬件相关的操作:
    return 0;
}
```


a，定义 input_dev 结构变量:

```c
//1.1.1, 定义input_dev结构变量 buttons_dev
static struct input_dev *buttons_dev;
```

b，用函数来分配这个 input_dev *buttons_dev 变量:

```c
struct platform_driver gpio_keys_device_driver = {
    .probe      = gpio_keys_probe,
    .remove     = __devexit_p(gpio_keys_remove),
    .driver     = {
        .name   = "gpio-keys",
    }
};

static int __init gpio_keys_init(void)
{
    return platform_driver_register(&gpio_keys_device_driver);
}

input = input_allocate_device();
```

只关心其中的枚举"probe"

入口函数

参考“gpio_keys.c”这个例子。看它的入口函数“”，只关心其中的“枚举”.probe = gpio_keys_probe.跳到“gpio_keys_probe()”后，可以看到是由“input_allocate_device()”分配一个“input_dev”结构体。

c，从上面的“gpio_keys_init()”分析

，分配“input_dev”结构体是“input_allocate_device()”,则这里自己民写分配一个“input_dev”结构是用此函数。

```
//1.1.2.分配这个 buttons_dev （input_dev结构指针变量）:
buttons_dev = input_allocate_device();
```

正常情况下要判断此“Input_dev”结构是否分配成功，但这里为简化代码不予判断。但一般都会成功的。

d,设置这个“input_dev”结构体：

看实例中的用法。

先看看这个结构体的原型：

```
struct input_dev {//子系统中用此结构体来描述一个输入设备.

    void *private;
    //导出到用户空间的相关信息，在sys文件可以看到.
    const char *name;
    const char *phys;
    const char *uniq;
    struct input_id id;

    unsigned long evbit[NBITS(EV_MAX)];
    unsigned long keybit[NBITS(KEY_MAX)];
    unsigned long relbit[NBITS(REL_MAX)];
    unsigned long absbit[NBITS(ABS_MAX)];
    unsigned long mscbit[NBITS(MSC_MAX)];
    unsigned long ledbit[NBITS(LED_MAX)];
    unsigned long sndbit[NBITS(SND_MAX)];
    unsigned long ffbit[NBITS(FF_MAX)];
    unsigned long swbit[NBITS(SW_MAX)];
```

下面还很长。

“unsigned long evbit[NBITS(EV_MAX)];”是：表示能产生哪 类 事件。这里是类，所以会有很多宏表示的事件：

```
/*
 * Event types
 */

#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
```

#define EV_SYN 0x00 //同步类

#define EV_KEY 0x01 //按键类，如键盘上的 a,b 等按键事件。

#define EV_REL 0x02 //relation 相对位移事件（如鼠标的位移是基于上一个位置的）。

#define EV_ABS 0x03 //ABS 是绝对位移（如触摸屏是 XY 坐标绝对位置）。

#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
```

开始设置，首先设置它能产生哪类事件：
如这里产生按键类事件。
下面是“gpio_keys.c”中的设置“input_dev”结构体：
static int __devinit gpio_keys_probe(struct platform_device *pdev)

```
platform_set_drvdata(pdev, input);

input->evbit[0] = BIT(EV_KEY);
```

直接设置了产生哪类事件，没有理会名字什么的。但上面的“BIT(EV_KEY)”方法不大好。
而用“set_bit()”比较好。下面是“gpio_keys.c”中的一个函数，其中实现就是用了“set_bit()”：
设置某一位。
static int __devinit gpio_keys_probe(struct platform_device *pdev)
-->input_set_capability(input, type, button->code);
void input_set_capability(struct input_dev *dev, unsigned int type, unsigned int code)
{
switch (type) {
case EV_KEY:
__set_bit(code, dev->keybit);
break;

case EV_REL:
__set_bit(code, dev->relbit);
break;

case EV_ABS:
__set_bit(code, dev->absbit);
break;

case EV_MSC:
__set_bit(code, dev->mscbit);
break;

case EV_SW:
__set_bit(code, dev->swbit);
break;

case EV_LED:
__set_bit(code, dev->ledbit);
break;

```
case EV_SND:
__set_bit(code, dev->sndbit);
break;

case EV_FF:
__set_bit(code, dev->ffbit);
break;

default:
printk(KERN_ERR
"input_set_capability: unknown type %u (code %u)\n",
type, code);
dump_stack();
return;
}

__set_bit(type, dev->evbit);
}
```

“unsigned long keybit[NBITS(KEY_MAX)];”:表示能产生哪些事件.这是“哪些”不同于“哪类”。

"unsigned long relbit[NBITS(REL_MAX)];"：表示能产生哪些相对位移事件（如鼠标 x,y 和滚轮）

"unsigned long absbit[NBITS(ABS_MAX)];"：表示能产生哪些绝对位移事件。

"unsigned long mscbit[NBITS(MSC_MAX)];"

"unsigned long ledbit[NBITS(LED_MAX)];"

"unsigned long sndbit[NBITS(SND_MAX)];"

"unsigned long ffbit[NBITS(FF_MAX)];"

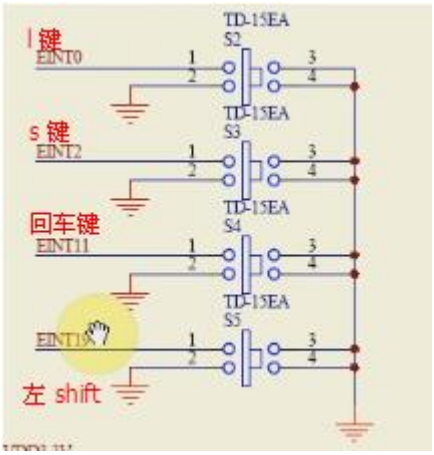
"unsigned long swbit[NBITS(SW_MAX)];"

e,自己的操作：用“set_bit()”设置某一位：

```
//1.2.1,能产生哪类事件
set_bit(EV_KEY, buttons_dev->evbit); //设置成按键类事件。设置evbit数组里的某一位为EV_KEY,
                                     //EV_KEY事件表示能产生按键类事件。
```

f,上面用 set_bit()设置了 evbit 数组中的“EV_KEY”按键类事件。但按键有 26 个字母还有其他符号按键，那么能产生这一类“EV_KEY”里的哪些按键事件？

要产生哪些按键事件，要看具体的原理图：



想产生“L,S,ENTER,左 shift”这 4 个按键事件。

以前写按键驱动时，按键值只有应用程序知道是什么意思：

```
/* 键值：按下时，0x01，0x02，0x03，0x04 */
/* 键值：松开时，0x81，0x82，0x83，0x84 */
```

而现在要写一个通用的“按键”驱动程序。在为这里开发板上只有 4 个键，要是有很多就可能设置成普通键盘那样了。但现在单板上只有 4 个按键，则这里定义成“L,S,Enter 和左 shift 键”。

```
//1.2.2,能产生"按键类事件"中的哪些具体的事件:L,S,Enter和左shift键。
set_bit(KEY_L, buttons_dev->keybit); //设置buttons_dev这个input_dev结构体的keybit按键事件具体按键之一为L字母
set_bit(KEY_S, buttons_dev->keybit); //S按键
set_bit(KEY_ENTER, buttons_dev->keybit); //回车按键
set_bit(KEY_LEFTSHIFT, buttons_dev->keybit); //左shift按键
```

设置这个“keybit”数组中某一位表示它能产生哪种按键事件。

g，注册“input_dev”结构：

先看其他例子中的注册：

int __devinit gpio_keys_probe(struct platform_device *pdev)

->error = input_register_device(input);

则注册“buttons_dev”结构如下：

```
//1.3,注册input_dev结构:
  input_register_device(buttons_dev);
```

注册了“input_dev”结构体之后，就会把这里具体的“buttons_dev”设备结构体挂到内核的“input_dev”输入设备的链表中去。接着就从右边的“input_handler”链表中一个个取出来具体的 id_table[]与“buttons_dev”进行比较。若是能匹配说明“input_handler”结构中的“id_table”能支持这个“input_dev”设备“buttons_dev”，这时支持后，就会接着调用“input_handler”结构中的“.connect”函数。

到了“.connect”建立连接这个过程时，就会创建一个新的结构“input_handle”,这个结构中有两个成员“.handler”（具体处理方式）和“.dev”（具体设备）。“.dev”指向左边的设备层，“.handle”指向右边的“处理方式”层。这个“input_handle”结构会分别放到左边设备层的“input_dev”这个结构成员“h_list”链表中，也挂到处理方式层的“input_handler”结构的成员“h_list”链表中去。

h，硬件相关的操作：

参考以前的代码：

首先，定义一个结构体“pin”脚描述。

```
struct pin_desc{
    unsigned int pin; 芯片手册中的引脚定义
    unsigned int key_val;按键值。
};
```

接着，定义 4 个按键的具体引脚定义和按键值。

```
//定义4个按键。
struct pin_desc pins_desc[4] = {
    {S3C2410_GPF0,  0x01},
    {S3C2410_GPF2,  0x02},  对应上面pins_desc结构数组中的引脚
    {S3C2410_GPG3,  0x03},  定义和按键值。
    {S3C2410_GPG11,  0x04},
};
```

然后，注册 4 个中断，引脚定义和按键值对应的名字和中断号。

```
request_irq(IRQ_EINT0,  buttons_irq, IRQT_BOTHEDGE, "S2", &pins_desc[0]);
request_irq(IRQ_EINT2,  buttons_irq, IRQT_BOTHEDGE, "S3", &pins_desc[1]);
request_irq(IRQ_EINT11, buttons_irq, IRQT_BOTHEDGE, "S4", &pins_desc[2]);
request_irq(IRQ_EINT19, buttons_irq, IRQT_BOTHEDGE, "S5", &pins_desc[3]);
```


自己实现的代码：

硬件操作中的硬件相关结构定义：

```
//1.4.1.定义pin脚描述结构体:中断号，名字，引脚定义，按键值.
struct pin_desc{
    int irq;        //中断号为IRQ_EINT0, IRQ_EINT2, IRQ_EINT11, IRQ_EINT19.
    char *name;     //名字为 s2, s3, s4, s5.
    unsigned int pin; //哪一个引脚.
    unsigned int key_val; //按键值.
};
//1.4.2，定义4个按键。
struct pin_desc pins_desc[4] = { //每个按键分别对应:中断号，名字，引脚定义和按键值.
    {IRQ_EINT0, "s2",  S3C2410_GPF0,  KEY_L}, //当按键下s2按键时就对应"KEY_L".
    {IRQ_EINT2, "s3",  S3C2410_GPF2,  KEY_S},
    {IRQ_EINT11, "s4", S3C2410_GPG3,  KEY_ENTER},
    {IRQ_EINT19, "s5", S3C2410_GPG11, KEY_LEFTSHIFT},
};
```

硬件相关的操作：

首先，注册 4 个中断：

```
  //1.4.3，注册4个中断:
for(i = 0; i < 4; i++)
{
  request_irq(pin_desc[i].irq,  buttons_irq, IRQT_BOTHEDGE, pin_desc[i].name, &pins_desc[i]);
}
  return 0;
```

中断号　中断处理函数　按键的名字

上面为减少代码，判断“request_irq()”的返回值代码省略去了。

其次，中断处理函数的编写:

之前的中断处理函数“buttons_irq()”是启动一个定时器：

```c
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);
}
```

下面是自己写的部分：

定义“定时器”和“中断函数”：

```c
//1.4.4.1:定义"定时器":
static struct pin_dese *irq_pd;
static struct timer_list buttons_timer; //定义一个定时器.定时器要初始化。

//1.4.4.编写中断函数"buttons_irq()":
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);
}
```

定时器初始化 和 定时器初始化函数定义：

```c
//1.4.硬件相关的操作:
  //1.4.5, 定时器要初始化:
  init_timer(&buttons_timer);
  //1.4.6, 定时器要大要素:超时时间(先不理会, 可以中断中修改超时时间) 和 处理函数。
  buttons_timer.function = buttons_timer_function; //定时器处理函数.
  //1.4.7, add_timer();
  add_timer(buttons_timer);
```

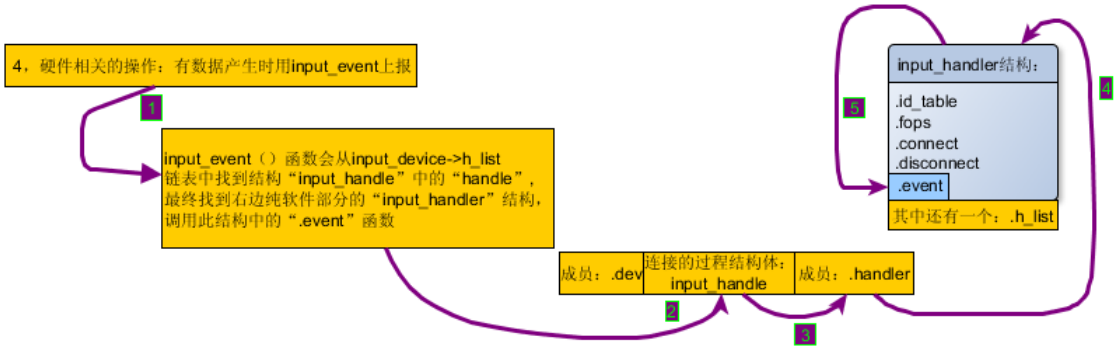
定时器初始化函数“buttons_timer_function()”的编写：

“确定按键值”--->“唤醒应用程序” 或 “发送信号”。

这里只需要上报"按键值":上报事件--- input_event 上报事件.

在“硬件相关的操作”中，有数据产生时用“input_event()”上报事件：

--->input_event()函数会从“input_device->h_list"链表中找到结构“input_handle”中的“handle”成员，以此最终找到右边纯软件的“处理方式”层中的“input_handler”结构，调用此结构中的“.event”函数。



void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)

参考

参 1，struct input_dev *dev：input_dev 结构变量指针。这里为“buttons_dev”.
参 2，unsigned int type：哪类事件。这里为“EV_KEY”按键类事件。
参 3，unsigned int code：哪个值。这里是指哪一个按键，为“pin_desc *pindesc->key_val”（KEY_L,KEY_S,KEY_ENTER,KEY_LEFTSHIFT）.
参 4,int value: 这里表示“按下”或是“松开”。

```
//定时器处理函数"buttons_timer_function()"定义:
static void buttons_timer_function(unsigned long data)
{    //以前是:  "确定按键值" --->"唤醒应用程序" 或 "发送信号":
     //这里只需要上报"按键值":上报事件--- input_event 上报事件.
     struct pin_desc * pindesc = irq_pd;
     unsigned int pinval;

     if (!pindesc)
          return;

     pinval = s3c2410_gpio_getpin(pindesc->pin);

     if (pinval)
     {
          /* 松开:最后一参数int value,0表示松开，1表示按下*/
          input_event(buttons_dev,EV_KEY, pindesc->key_val, 0);
     }
     else
     {
          /* 按下 */
          input_event(buttons_dev, EV_KEY, pindesc->key_val, 1);
     }
} ? end buttons_timer_function ?
```

上报事件后，还有一个上报同步事件：看“gpio_keys.c”的中断处理函数“”中：

```
input_event(input, type, button->code, !!state);
input_sync(input);
```

irqreturn_t gpio_keys_isr(int irq, void *dev_id) //GPIO 按键的中断处理函数.
--->input_event(input, type, button->code, !!state);
--->input_sync(input);

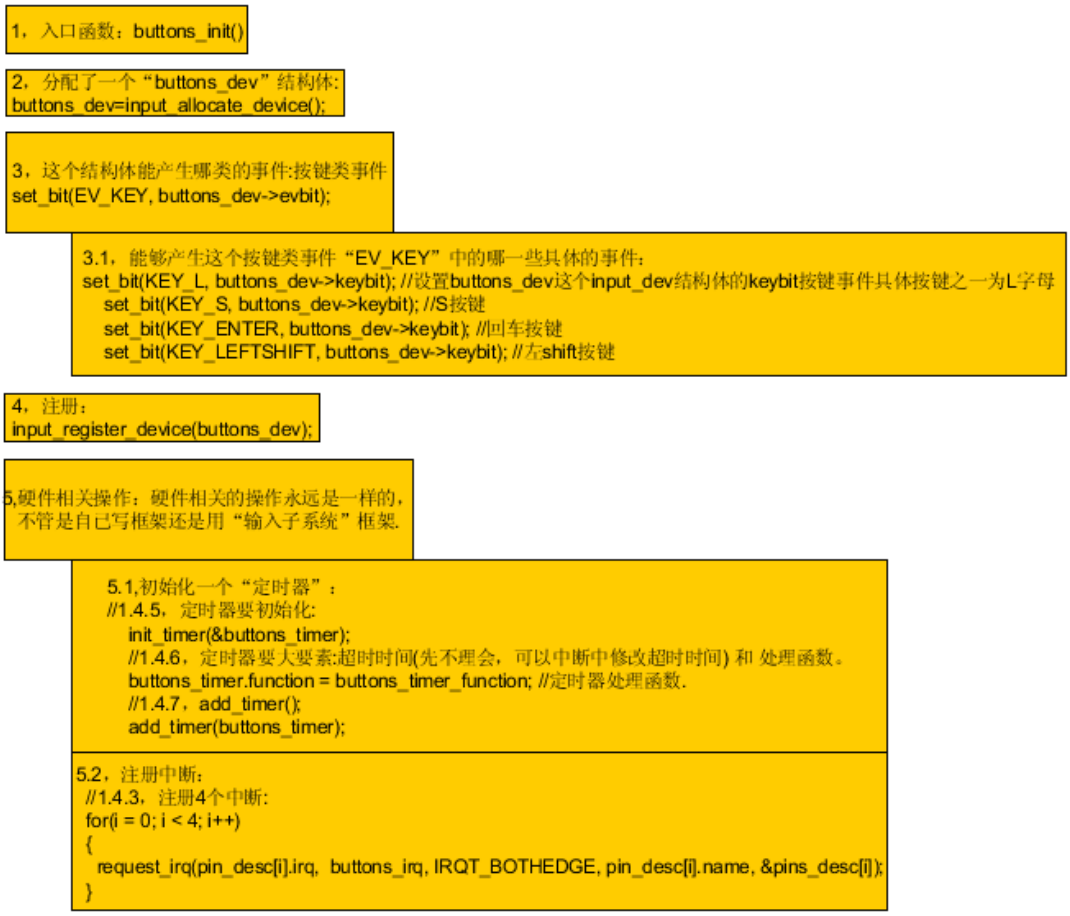
```
//定时器处理函数"buttons_timer_function()"定义:
static void buttons_timer_function(unsigned long data)
{    //以前是:  "确定按键值" --->"唤醒应用程序" 或 "发送信号";
     //这里只需要上报"按键值":上报事件--- input_event 上报事件.
     struct pin_desc * pindesc = irq_pd;
     unsigned int pinval;

     if (!pindesc)
          return;

     pinval = s3c2410_gpio_getpin(pindesc->pin);

     if (pinval)
     {
          /* 松开:最后一参数int value,0表示松开, 1表示按下*/
          input_event(buttons_dev,EV_KEY, pindesc->key_val, 0);
          input_sync(buttons_dev); //上报同步事件。
     }
     else
     {
          /* 按下 */
          input_event(buttons_dev, EV_KEY, pindesc->key_val, 1);
          input_sync(buttons_dev); //上报同步事件。
     }
} ? end buttons_timer_function ?
```

所有的应用程序都根据这个“上报同步事件” input_sync(buttons_dev);，

```
static inline void input_sync(struct input_dev *dev)
{
     input_event(dev, EV_SYN, SYN_REPORT, 0);
}
```

原型：void input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)。
事件类：是同步类事件“EV_SYN”。
后面 code 形参“SYN_REPORT”和 value 形参值为“0”时表示这个事件已经上报完了。

小结：



6， 假设驱动程序装载好了，

按下一个“按键”后，那么中断处理函数“buttons_irq()”就会被调用。在中断处理函数中，是先将哪一个按键“dev_id”记录下来（“irq_pd = (struct pin_desc *)dev_id;”），然后修改定时器（“mod_timer(&buttons_timer, jiffies+HZ/100);”将它 10ms“jiffies+HZ/100”再启动“&buttons_timer”这个函数），假设 10ms 时间到了，那么“定时器处理”函数（buttons_timer_function）初调用。

7， buttons_timer_function（）定时器处理函数的工作：

读引脚值（pinval = s3c2410_gpio_getpin(pindesc->pin);），再确实是松开还是按下：

```
if (pinval)
{
/* 松开:最后一参数 int value,0 表示松开，1 表示按下*/
input_event(buttons_dev, pindesc->EV_KEY, 0);
}
else
{
/* 按下 */
input_event(buttons_dev, pindesc->EV_KEY, 1);
}
```

8，input_event()上报事件：

input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)。

看最后一行:

```
list_for_each_entry(handle, &dev->h_list, d_node)
if (handle->open)
handle->handler->event(handle, type, code, value);
```

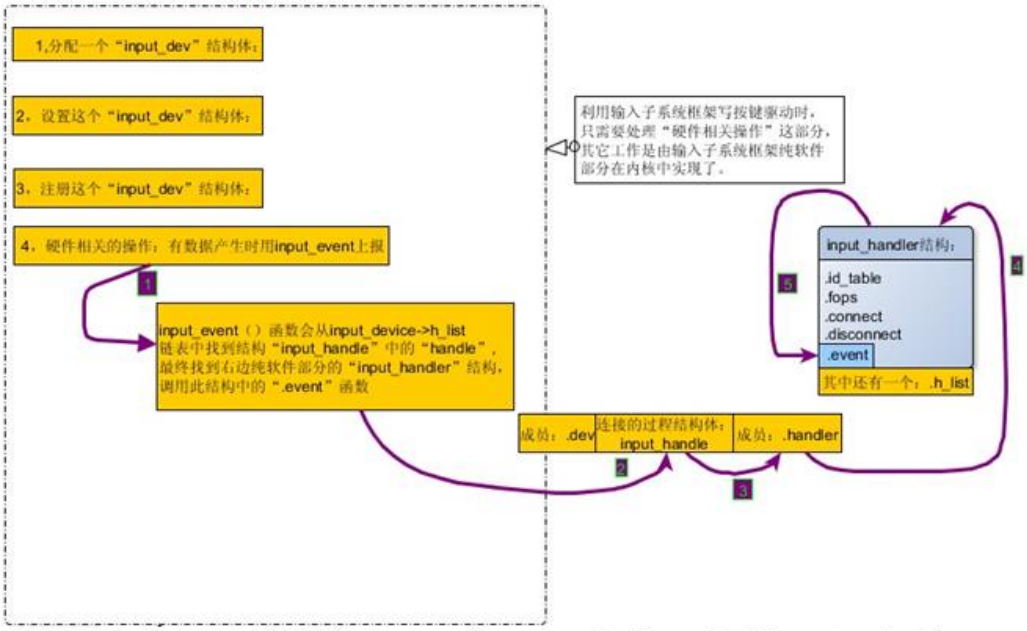
对输入设备的“h_list” 链表里的每一个成员（就是中间作连接的只有".dev"和".handler"成员的“input_handle”结构）。把这些“input_handle”结构取出来到“handle”.

若它们是打开时，就调用这个具体“input_handle”结构变量的成员“.handler”的“event()”函数。

以前需要自己构造“open,read,write”等函数，而现在是输入子系统框架中定义好的。参看“evdev.c”中：

```
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
};
```

就是说输入子系统右边的纯软件部分“处理方式”层由内核提供好了，我们只需要做左边“硬件相关的操作”这部分，即：



8，input_event()上报事件：input_event(struct input_dev *dev, unsigned int type, unsigned int code, int value)。

看最后一行：

```
list_for_each_entry(handle, &dev->h_list, d_node)
    if (handle->open)
        handle->handler->event(handle, type, code, value);
```

对输入设备的“h_list”链表里的每一个成员（就是中间作连接的只有“.dev”和“.handler”成员的“input_handle”结构），把这些“input_handle”结构取出来到“handle”。

若它们是打开时，就调用这个具体“input_handle”结构变量的成员“.handler”的“event()”函数。

④，出口函数：

a，添加两个头文件：

```
#include <asm/gpio.h>
//2.1，出口函数添加如下两个头文件
#include <asm/io.h>
#include <asm/arch/regs-gpio.h>
```

b，释放中断：

原型：void free_irq(unsigned int irq, void *dev_id)

参 1，unsigned int irq：中断号。

参 2，void *dev_id：pin 脚描述结构数组组元。

```
//2.2,释放中断:
    int i;
    for(i = 0; i < 4; i++){
        free_irq(pin_desc[i].irq, &pins_desc[i]);
    }
```

c，消除注册到内核定时器目录上的内容：

名称： del_timer()

功能： 消除注册到内核定时器目录上的内容

原型：

#include <linux/timer.h>

int del_timer (struct timer_list *timer);

说明：

从内核定时器目录消除结构体。内核定时器的目录为连接 list 结构，不是消除结构体的内

容，而是修改结构体的连接信息，因此该函数不参与结构体变量的分配和消除。
变量：
timer ： 将要消除的内核定时器注册结构体的数据地址。
返回值：
正常运行返回 1 ，否则返回 0 。

```
//2.3.消除注册到内核定时器目录上的内容.
del_timer(&buttons_timer);
```

d,卸载“input_dev”结构：

```
//2.4.卸载input_dev结构：
input_unregister_device (buttons_dev);
```

e,释放“input_dev”结构分配的空间：

```
//2.5.释放“input_dev”结构分配的空间：
input_free_device(buttons_dev);
```

编译和实验：

```
warning: ignoring return value of `request_irq', declared with attribute warn
```

因为忽略了“request_irq()”返回值，所以有警告，一般是要判断，但为简化代码没有添加判断。

1，挂载 NFS 文件系统：

```
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt/
# ls buttons.ko
buttons.ko
```

2，查看当前的“/dev/event*”设备：

```
# ls -l /dev/event*
crw-rw----    1 0        0          13,  64 Jan  1 00:00 /dev/event0
```

3，加载驱动程序：

```
# insmod buttons.ko
input: Unspecified device as /class/input/input1
# ls -l /dev/event*
crw-rw----    1 0        0          13,  64 Jan  1 00:00 /dev/event0
crw-rw----    1 0        0          13,  65 Jan  1 00:44 /dev/event1
#
```

现在多了一个“event1”，对应“buttons.ko”.它的主设备号是“13”，次设备号为“65”。注册时“input_register_device(buttons_dev)”会在右边的纯软件处理方式层取出来一个一个的比较，若是能够支持这个“buttons_dev”时，就从右边的“input_handler”结构中取出“.connect”函数建立联系。右边有很多文件看能不能够支持（如evdev.c,keyboard.c,mousedev.c 等），这时看 even.c 是否支持我们这里的按键驱动（buttons_dev）.

查看“evdev.c”中的：input_handler 结构：

```
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,
};
```

其中的“.id_table”为“evdev_ids”是指支持的设备：看它支持哪些设备。

```
static const struct input_device_id evdev_ids[] = {
    { .driver_info = 1 },    /* Matches all devices */
    { },                /* Terminating zero entry */
};
```

“Matches all devices”注释说明这个“evdev_ids[]”支持所有的设备。那么也就会支持我们这个按键设备。这时，evdev_ids[]支持这个“buttons_dev”后，就会调用"evdev_handler"这个“input_handler”结构中“.connect = evdev_connect,”。

可以分析“evdev_connect()”函数：

```
static int evdev_connect(struct input_handler *handler, struct input_dev *dev,
                         const struct input_device_id *id)
```

int evdev_connect(struct input_handler *handler, struct input_dev *dev,
const struct input_device_id *id)

--->for (minor = 0; minor < EVDEV_MINORS && evdev_table[minor]; minor++);寻找次设备号。

（#define EVDEV_MINORS　32）

构造“input_dev”结构体后，在类下创建某些设备：

cdev = class_device_create(&input_class, &dev->cdev, devt,
dev->cdev.dev, evdev->name);

回头看“input.c”中有创建类：err = class_register(&input_class);

注册“file_operations”结构体：err = register_chrdev(INPUT_MAJOR, "input", &input_fops);

但是没有在“类”下面创建设备。

再看以前写的一个驱动程序：

注册“file_operations”结构：major = register_chrdev(0, "sixth_drv", &sencod_drv_fops);

创建类：sixthdrv_class = class_create(THIS_MODULE, "sixth_drv");

在类下创建设备：sixthdrv_class_dev = class_device_create(sixthdrv_class, NULL, MKDEV(major, 0), NULL, "buttons"); /* /dev/buttons */

这样文件系统下的“mdev”或“udev”才能自动的为我们创建设备节点。

而“input.c”中只有注册 file_operations 结构和创建类而没有在类下创建设备。那么是什么时在类下创建设备，是在注册了左边的“input_dev”后，调用右边的“input_handler”由“.connect”函数来帮我们在类下创建设备。所以看“evdev.c”下的“.connect”函数“.connect =evdev_connect,”

```
cdev = class_device_create(&input_class, &dev->cdev, devt,
                           dev->cdev.dev, evdev->name); //在类下创建设备.
```

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

在类下创建的设备名字叫作“event%d”这样的名字。从上面在单板上加载驱动 buttons.ko 后，看到设备节点名为“/dev/event1”,那么次设备号就是以前存在的一个次设备号“”加上这个“dev/event1”中的数字“1”.

```
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,
};
```

此结构中的次设备号就是从64开始的。

这个 evdev_handler 结构中的次设备号就是从“EVDEV_MINOR_BASE”64 开始的。

```
#define EVDEV_MINOR_BASE    64
```

```
devt = MKDEV(INPUT_MAJOR, EVDEV_MINOR_BASE + minor),
```

Sprintf()打印出来的“event%d”中的“%d”为“minor”是“1”，则“EVDEV_MINOR_BASE + minor”为 64+1 为 65.

4，做测试：

```
# cat /dev/tty1
```

这时按下单板上的按键“s2,s3,s4”.可是下面出现错误：

```
# cat /dev/tty1
&? ? ? ? ? ?
             #
```

第一种测试方法：

用“hexdump”来看：下面的结果是正确的。

```
# hexdump /dev/event1
0000000 0bb2 0000 0e48 000c 0001 0026 0001 0000
0000010 0bb2 0000 0e54 000c 0000 0000 0000 0000
0000020 0bb2 0000 5815 000e 0001 0026 0000 0000
0000030 0bb2 0000 581f 000e 0000 0000 0000 0000
```


hexdump 是 16 进制显示 open 后的/dev/event1。读里面的数据以 16 进制显示出来。

hexdump /dev/event1 (open(/dev/event1), read(), 最后以 16 进制显示)

秒 微秒 类 code value

0000000 0bb2 0000 0e48 000c 0001 0026 0001 0000

0000010 0bb2 0000 0e54 000c 0000 0000 0000 0000

0000020 0bb2 0000 5815 000e 0001 0026 0000 0000

0000030 0bb2 0000 581f 000e 0000 0000 0000 0000

看 open 这个“event1”时对应哪个文件。看“input.c”中代码：

```
static const struct file_operations input_fops = {
    .owner = THIS_MODULE,
    .open = input_open_file,
};
```

open 时肯定会调用到“input_open_file”函数。进入此函数看代码：

struct input_handler *handler = input_table[iminor(inode) >> 5];

```
crw-rw----    1 0        0          13, 65 Jan  1 00:44 /dev/event1
```

这里次设备号是“65”，“iminor[inode]>>5”65 左移 5 位，就等于 2.即：

struct input_handler *handler = input_table[2];第二项就是：&evdev_handler

即：struct input_handler *handler = input_table[2]=&evdev_handler.

再看“evdev.c”中的注册过程：

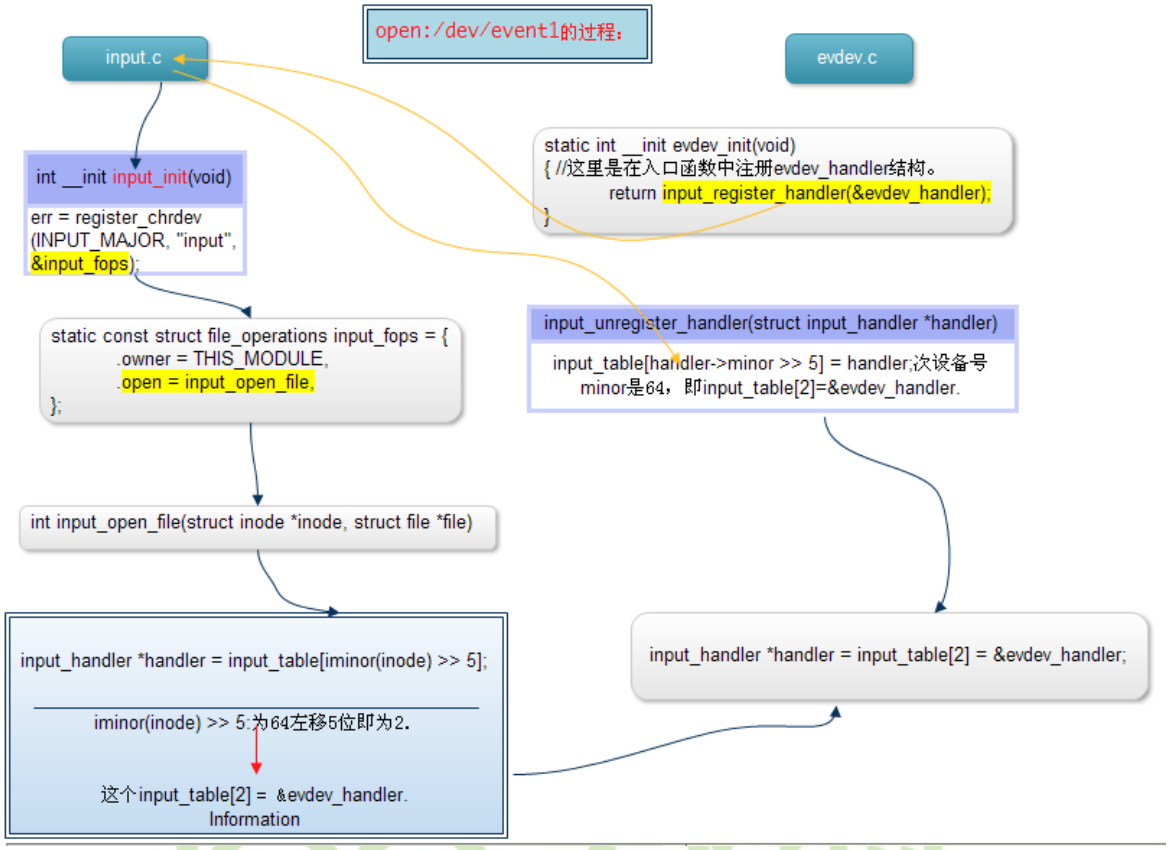
static int __init evdev_init(void)

-->return input_register_handler(&evdev_handler);

-->input_table[handler->minor >> 5] = handler;

即：

input_table[2] = &evdev_handler;之前注册“evdev_handler”结构时就把这个数组配上去了。



接着 input.c--->input_open_file()中“new_fops = fops_get(handler->fops)”就等于 evdev.c-->input_handler evdev_handler 结构中的“.fops=&evdev_fops”.因为从上面可知：
input_handler *handler = input_table[iminor(inode) >> 5] = &evdev_handler;
-->new_fops = fops_get(handler->fops)为：
new_fops = fops_get(&evdev_handler->fops);
-->new_fops = fops_get(&evdev_handler->(fops=&evdev_fops));

看完整的“evdev_handler”结构定义：

```
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,
};
```

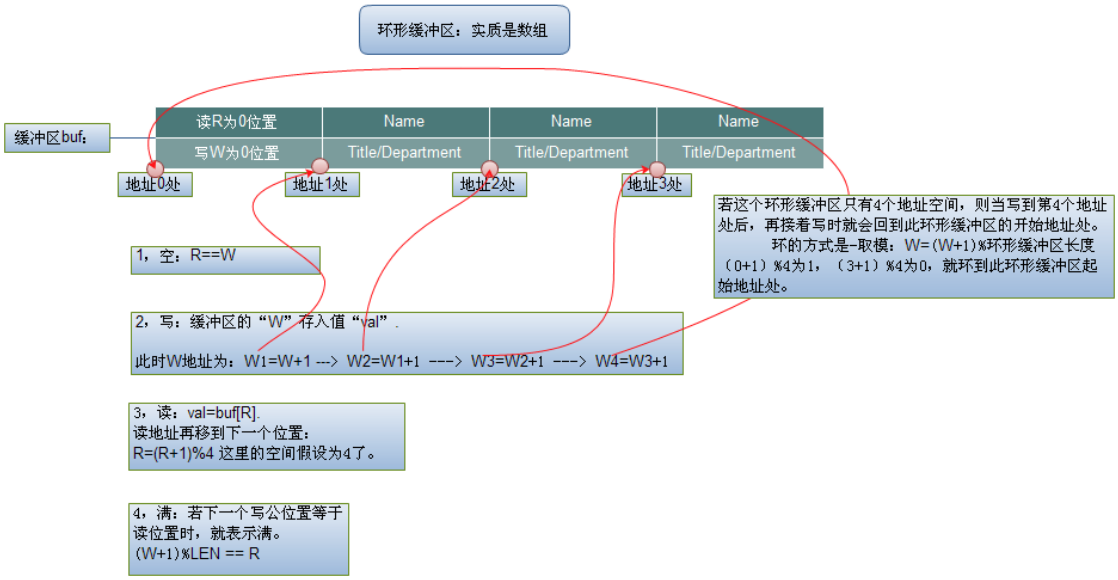
当 new_fops 为“evdev_fops”时，则读，写等函数都用了：

```c
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
};
```

所以，最后“open(/dev/event1)”就用到了“evdev_fops”结构中的读、写、IO 复用等函数。
如读时就用到：
“.read = evdev_read,”--->ssize_t evdev_read(struct file *file, char __user *buffer, size_t count, loff_t *ppos)

```c
if (count < evdev_event_size()) //evdev_event_size指事件的大小.
    return -EINVAL;

//若client缓冲区的头部等于它的尾部(环形缓冲区)则表示缓冲区里面数据是空的。
//(file->f_flags & O_NONBLOCK)是指这个文件是以"非阻塞方式"打开.
if (client->head == client->tail && evdev->exist && (file->f_flags & O_NONBLOCK)
        return -EAGAIN; //以上条件就返回一个"-EAGAIN"让你再次尝试.这和以前的阻塞非阻塞对应起来.

//上面判断后若不是非阻塞而是阻塞的，则下面马上要"休眠"。
    retval = wait_event_interruptible(evdev->wait,
        client->head != client->tail || !evdev->exist);
```

补充“环形缓冲区”：
环形缓冲区实质是一个数组。



//若 client 缓冲区的头部等于它的尾部(环形缓冲区)则表示缓冲区里面数据是空的。
//(file->f_flags & O_NONBLOCK)是指这个文件是以"非阻塞方式"打开.
if (client->head == client->tail && evdev->exist && (file->f_flags & O_NONBLOCK))
return -EAGAIN; //以上条件就返回一个"-EAGAIN"让你再次尝试.这和以前的阻塞非阻塞对应起来.

所以“client->head == client->tail”是指头等于尾时，就表示缓冲区是“空”，这时“file->f_flags & O_NONBLOCK”文件是“非阻塞”时，就返回了错误“-EAGAIN”

否则，就“休眠”，直到“头不等于尾”--表示缓冲区里有数据了：
retval = wait_event_interruptible(evdev->wait,client->head != client->tail || !evdev->exist);

然后就可以 copy_to_user 数据：evdev_event_to_user(buffer + retval, event)--->实际是封装至：
copy_to_user(buffer, &compat_event, sizeof(struct input_event_compat)。

```
static int evdev_event_to_user(char __user *buffer, const struct input_event *event)
{
    if (copy_to_user(buffer, event, sizeof(struct input_event)))
        return -EFAULT;
                                        实际上是把"input_event"结构体拷贝到用户空间。
    return 0;
}
```

struct input_event {
struct timeval time; // 事件发生的时间.4 字节的秒和 4 字节的微秒
__u16 type; // 2 字节表示“类”。
__u16 code; // 2 字节表示“code”。
__s32 value; // 4 字节表示“value”.
};
上面这个结构体支持所有的输入事件，这里是按键事件，还能支持触摸屏事件。

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

所以：
hexdump /dev/event1 (open(/dev/event1), read(),)
秒 微秒 类 code value
0000000（这 4 字节不用管） 0bb2 0000（这 4 字节指秒） 0e48 000c（这 4 字节指微秒）
0001 0026 0001 0000

类：这里是“按键”类：宏定义为“0x001”

```
/* 松开:最后一参数int value,0表示松开，1表示按下*/
input_event(buttons_dev,EV_KEY, pindesc->key_val, 0);
input_sync(buttons_dev); //上报同步事件。
#define EV_KEY          0x01
```

Code:是按键值。

```
//1.4.2，定义4个按键。
struct pin_desc pins_desc[4] = { //每个按键分别对应:中断号，名字，引脚定义和按键值.
    {IRQ_EINT0, "s2",  S3C2410_GPF0,  KEY_L}, //当按键下s2按键时就对应"KEY_L".
    {IRQ_EINT2, "s3",  S3C2410_GPF2,  KEY_S},
    {IRQ_EINT11, "s4", S3C2410_GPG3,  KEY_ENTER},
    {IRQ_EINT19, "s5", S3C2410_GPG11, KEY_LEFTSHIFT},
};
```

code

在“input.h”中定义了按键：KEY_L 为十进制的“38”，换成十六进制为“0x26”.

```
00151: #define KEY_S            31
00152: #define KEY_D            32
00153: #define KEY_F            33
00154: #define KEY_G            34
00155: #define KEY_H            35
00156: #define KEY_J            36
00157: #define KEY_K            37
00158: #define KEY_L            38
00159: #define KEY_SEMICOLON        39
00160: #define KEY_APOSTROPHE       40
00161: #define KEY_GRAVE        41
00162: #define KEY_LEFTSHIFT        42
00163: #define KEY_BACKSLASH        43
00164: #define KEY_Z            44
```

Value：是指按键按下或松开：

```
if (pinval)
{
    /* 松开:最后一参数int value,0表示松开，1表示按下*/
    input_event(buttons_dev,EV_KEY, pindesc->key_val, 0);
    input_sync(buttons_dev); //上报同步事件。
}
else
{
    /* 按下 */
    input_event(buttons_dev, EV_KEY, pindesc->key_val, 1);
    input_sync(buttons_dev); //上报同步事件。
}
```

hexdump中的value部分

秒 微秒 类 code value

0000000（这 4 字节不用管） 0bb2 0000（这 4 字节指秒） 0e48 000c（这 4 字节指微秒） 0001 0026 0001 0000

0000010 0bb2 0000 0e54 000c 0000 0000 0000 0000

类“0000”为宏“EV_SYN”同步事件。

0000020 0bb2 0000 5815 000e 0001 0026 0000 0000

类“0001”为宏“EV_KEY”按键类事件，value 为“0”是指“松开”。

0000030 0bb2 0000 581f 000e 0000 0000 0000 0000

最后是一个跟着的“同步事件”。

按照用“hexdump”的测试方法，上面的结果是正确的。

第二种测试方法：cat /dev/tty1

```
# ls /dev/tty1 -l
crw-rw----    1 0        0           4,   1 Jan  1 00:00 /dev/tty1
# cat /dev/tty1
&? ? ? ? ? ?
            #
#
```

本来是输入“ls”后应该可以显示出根文件系统的结构。但这里测试时没有成功。可能是“keyboard.c”没有编译进内核。“linux/drivers/char/keyboard.c”：

```
book@book-desktop:/work/system/linux-2.6.22.6$ ls drivers/char/keyboard.
keyboard.c  keyboard.o
```

可见有被编译进内核。

不能用是因为里面有 Qt:

```
785 0          9044 S <  /opt/Qtopia/bin/qss
786 0         11388 S N  /opt/Qtopia/bin/quicklauncher
789 0             SW<  [rpciod/0]
```

如果没有启动 QT：

cat /dev/tty1

按:s2,s3,s4

这种方式最后一定要按“回车”。

就可以得到 ls

或者：

exec 0</dev/tty1

然后可以使用按键来输入

3. 如果已经启动了 QT：

先在触摸屏上新建一个文件，可以点开记事本。

然后按:s2,s3,s4

这里测试时，先杀掉 QT：

```
# kill -9 771
```

```
785 0          9044 S <  /opt/Qtopia/bin/qss
786 0         11764 R    textedit
789 0             SW<  [rpciod/0]
814 0         11392 S N  /opt/Qtopia/bin/quicklauncher
816 0          3096 R    ps
# kill -9 785
#
```

```
789 0             SW<  [rpciod/0]
814 0         11392 S N  /opt/Qtopia/bin/quicklauncher
817 0          3096 R    ps
# kill -9 814
```

最后发现这种杀进程的方式不行。

修改“/etc/inittab”：

```
# /etc/inittab
::sysinit:/etc/init.d/rcS
s3c2410_serial0::askfirst:-/bin/sh
::ctrlaltdel:/sbin/reboot
::shutdown:/bin/umount -a -r
```

修改里面的“/etc/init.d/rcS”.

```
#!/bin/sh
ifconfig eth0 192.168.1.17

mount -a
mkdir /dev/pts
mount -t devpts devpts /dev/pts
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
if [ ! -e /etc/pointercal ]
then
/bin/ts_cal.sh
fi
#/bin/qpe.sh &
```

先注释掉这行

注释掉上面一行后，reboot 单板。

Cat /dev/tty1 时，就用到了“tty”，这个更为复杂。
tty1 的主设备号是 4，次设备号为 1.是通过“tty_io.c”中的驱动程序访问到“keyboard.c”，“tty_io.c”远远比输入子系统要复杂。这里不分析。还是看下“keyboard.c”。
从“入口函数”开始分析：
__init kbd_init(void)
->error = input_register_handler(&kbd_handler); //注册一个 input_handler 结构.

```
static struct input_handler kbd_handler = {
    .event      = kbd_event,
    .connect    = kbd_connect,
    .disconnect = kbd_disconnect,
    .start      = kbd_start,
    .name       = "kbd",
    .id_table   = kbd_ids,
};
```

看这个“kbd_handler”中的“id_table”指明了它支持哪些设备。

```
static const struct input_device_id kbd_ids[] = {
    {
        .flags = INPUT_DEVICE_ID_MATCH_EVBIT,
        .evbit = { BIT(EV_KEY) },
    },

    {
        .flags = INPUT_DEVICE_ID_MATCH_EVBIT,
        .evbit = { BIT(EV_SND) },
    },

    { },    /* Terminating entry */
};
```

只要能产生EV_KEY"按键"类事件keyboard.c就能支持。

只要能产生EV_SND"声音"类事件keyboard.c就能支持。

.flags = INPUT_DEVICE_ID_MATCH_EVBIT，中“MATCH”是指匹配哪些位，这里比较“EVBIT”。
只要这个输入设备，能够产生“EV_KEY”按键类事件，这个“keyboard.c"就能支持。
我们自己写的 buttons 驱动中支持“按键”类事件：

```
//1.2.1.能产生哪类事件
set_bit(EV_KEY, buttons_dev->evbit); //设置成按键类事件。设置evbit数组里的某一位为EV_KEY,
                                     //EV_KEY事件表示能产生按键类事件。
```

所以这个“keyboard.c”也支持我们的 buttons 驱动。当我们按下按键后，上按事件：

```
/* 松开:最后一参数int value,0表示松开，1表示按下*/
input_event(buttons_dev,EV_KEY, pindesc->key_val, 0);
input_sync(buttons_dev); //上报同步事件。
```

这时就会从 input_device 结构的 h_list“链表”里面（会有多项 input_handle 结构）。
找出“evdev.c”的 handler 调用它的“input_handler”结构中的“event”函数。
找出“keyboard.c”的 handler 调用它的“input_handler”结构中的“event”函数。

```
static struct input_handler kbd_handler = {
    .event      = kbd_event,
    .connect    = kbd_connect,
    .disconnect = kbd_disconnect,
    .start      = kbd_start,
    .name       = "kbd",
    .id_table   = kbd_ids,
};
```

如上为“keyboard.c”中 input_handler 结构变量 kbd_handler 中的"event"函数。

```
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();
}
```

在其中的“kbd_keycode（）”或“kbd_rawcode()”代码中就和“tty”有了关系。可以看里面的代码有 tty.

Cat /dev/tty1 时，不是从“输入子系统”中过来的，而是从 tty 关的部分进来的。

单板重启后：

```
Please press Enter to activate this console.
starting pid 770, tty '/dev/s3c2410_serial0': '/bin/sh'
#
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt
# insmod buttons.ko
input: Unspecified device as /class/input/input1
# cat /dev/tty1
ls
```

去掉Qt后，这里按下ls键再按回车后，就可以看到"ls"。

标准输入改为“dev/tty1”，之前是从串口上得到文件，这里是把标准输入改成“dev/tty1”

```
#
# exec 0</dev/tty1
```

这里按按键“ls”和“回车”就可能当成在键盘上在终端上敲写“ls"时显示目录下的文件与目录。

当按下“l”时不松开，却并不能像在 PC 上按下某个键不松开会重复键入的效果，这是因为没有产生“重复”类事件:

```
//1.2.1.能产生哪类事件
set_bit(EV_KEY, buttons_dev->evbit); //设置成按键类事件。设置evbit数组里的某一位为EV_KEY,
                                     //EV_KEY事件表示能产生按键类事件。
//1.2.1_1，测试时加上的产生"重复"类事件：
set_bit(EV_REP, buttons_dev->evbit);
```

按下不松开会重复上报事件的过程：会启动定时器。看“input_event()”

```
case EV_KEY: //若是按键类事件。

    if (code > KEY_MAX || !test_bit(code, dev->keybit) || !!test_bit(code, dev->key) == value)
        return;

    if (value == 2)
        break;

    change_bit(code, dev->key);
    //若是能产生"重复"类事件。记录按键值后，会修改定时器的时间mod_timer()。
    if (test_bit(EV_REP, dev->evbit) && dev->rep[REP_PERIOD] && dev->rep[REP_DELAY] && dev->timer.data && valu
        dev->repeat_key = code; //记录按键值。
        mod_timer(&dev->timer, jiffies + msecs_to_jiffies(dev->rep[REP_DELAY]));
    }

    break;
```

看“定时器”函数的设置：dev->timer.function = input_repeat_key;

```
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;
}
```

```
static void input_repeat_key(unsigned long data)
{
    struct input_dev *dev = (void *) data;

    if (!test_bit(dev->repeat_key, dev->key))
        return;

    input_event(dev, EV_KEY, dev->repeat_key, 2);
    input_sync(dev);

    if (dev->rep[REP_PERIOD])
        mod_timer(&dev->timer, jiffies + msecs_to_jiffies(dev->rep[REP_PERIOD]));
}
```

上报事件，repeat_key为之前上报的一个值。

2表示重复类。

接着在“input_repeat_key()”中再次修改定时器：

```
if (dev->rep[REP_PERIOD])
    mod_timer(&dev->timer, jiffies + msecs_to_jiffies(dev->rep[REP_PERIOD]));
```

代码中加入"上报重复"类事件后，重新编译模块再加载测试：

再重启单板：

```
Please press Enter to activate this console.
starting pid 770, tty '/dev/s3c2410_serial0': '/bin/sh'
#
# mount -t nfs -o nolock,vers=2 192.168.1.5:/work/nfs_root/first_fs /mnt
# cd /mnt
# insmod buttons.ko
input: Unspecified device as /class/input/input1
#
#
# cat /dev/tty1
ls                  按"l"后回车就有很多重复出来。
lllllllllllllll
```

0，1，2 都是对应的是“串口”设备“/dev/s3c2410_serial0”.

这时:exec 0</dev/tty1 后，按下 4 个按键中的“l”后重复显示“l”.

测试左 shift 键，按下 shift 的同时再按下“l”就会是“L”。