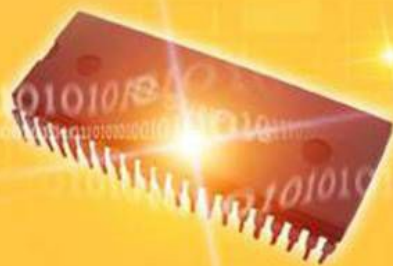


嵌入式系统工程师



linux设备驱动工程实例

- 杂项设备
- cdev方式注册字符驱动
- Platform总线
- Input子系统

- 杂项设备
- cdev方式注册字符驱动
- Platform总线
- Input子系统

- Linux下有些字符设备不符合预先确定的字符设备范畴，所有这些设备都统一采用主设备号为10，次设备动态分配的方式注册到内核
- 杂项设备内部就是封装了字符设备的一系列数据结构和操作接口，其本质还是字符设备类型
- 下面来看杂项设备和字符设备是如何联系和区别.

➤字符设备注册和生成节点过程

```
①static struct file_operations dev_fops = {  
    .owner    = THIS_MODULE,  
    .open     = dev_open,  
    .release  = dev_close,  
    .read     = dev_read,  
};
```



```
②register_chrdev() { ... };
```

```
③class_create() { ... };
```

```
④device_create() { ... };
```

杂项设备：

①填充file_operations



②填充miscdevice



③misc_register()

- 设备注册进内核
- 自动生成设备节点

字符设备：

①填充file_operations



②register_chrdev()



③class_create()



④device_create()

字符设备的注册需要:

file_operations

主设备号

次设备号

设备名字

那miscdevice会有什么的成员呢?

```
struct miscdevice {  
    int minor; → 次设备号  
    const char *name; → 设备名字  
    const struct file_operations *fops; → file_operations 指针  
    struct list_head list;  
};
```


➤ 杂项设备注册和生成节点过程:

① static struct file_operations dev_fops = {

```
.owner  = THIS_MODULE,  
.open   = dev_open,  
.release = dev_close,  
.read   = dev_read, };
```

② struct miscdevice dev_misc = { //填充miscdevice结构体

```
.minor=MISC_DYNAMIC_MINOR, //动态获取次设备号
```

```
.name=DRIVER_NAME,
```

```
.fops = &dev_fops };
```

③ misc_register(&dev_misc); //注册并自动生成节点

详见01_led_simple_misc例子...

- 杂项设备
- cdev方式注册字符驱动
- Platform总线
- Input子系统

- Linux内核对字符驱动模型的实现主要依赖：
 - **cdev结构体**: 驱动中用来描述一个字符设备
 - **设备号**: 实现设备文件和驱动程序的关联
- 之前所学的register_chrdev(...)函数内部封装了设备号申请以及cdev创建并注册的过程
- 内核为我们提供了字符设备的手动注册过程，即自己动手实现上面的过程，具有很好的灵活性，更合理的利用内核资源。

➤ 字符驱动cdev注册流程

1. 申请并注册主从设备号
2. 初始化已定义的cdev变量，cdev变量指定file_operations接口
3. 添加cdev变量到内核，完成驱动注册，添加cdev时需要一个已申请成功的主从设备号

➤ 字符驱动cdev注销流程

- 删除已添加的cdev
- 注销申请的主从设备号

➤设备号:

设备号注册两种办法:

- 指定主从设备号并告知内核
- 从内核中动态申请主从设备号

➤指定主从设备号并告知内核

```
int register_chrdev_region( dev_t from,  
                           unsigned count,   const char *name)
```


参数from: 指定的主设备号+起始从设备号

参数count: 占用从设备号数目

参数name: 驱动的名字, 通过/proc/device查看

返回值: 成功: 0; 失败: 负数。

- 从内核中动态申请主从设备号

```
int alloc_chrdev_region(  
    dev_t *dev,  dev_t from  
    unsigned baseminor,  
    unsigned count,  
    const char *name)
```

参数dev: 用来存储申请成功的主设备号+起始从设备号

参数baseminor: 指定申请时的起始从设备号

参数count: 指定申请时从设备号的数目

参数name: 驱动的名字

返回值: 成功: 0; 失败: 非0值。

➤ 设备号的释放接口函数

```
void unregister_chrdev_region(  
    dev_t from,  
    unsigned count)
```

参数dev: 注册成功的主设备号+起始从设备号

参数count: 从设备号的数目

➤ 字符驱动cdev注册流程

1. 申请并注册主从设备号
2. 初始化已定义的cdev变量，cdev变量指定file_operations接口
3. 添加cdev变量到内核，完成驱动注册，添加cdev时需要一个已申请成功的主从设备号

➤ 字符驱动cdev注销流程

- 删除已添加的cdev
- 注销申请的主从设备号

➤ cdev

cdev结构体描述了一个字符设备

```
struct cdev {  
    struct kobject kobj; /*用于管理*/  
    struct module *owner; /*用于管理*/  
    ★ const struct file_operations *ops;  
    struct list_head list; /*用于管理*/  
    ★ dev_t dev; /*设备号*/  
    unsigned int count; /*引用次数*/  
};
```

注：内核为我们提供了标准的接口进行cdev相关操作，不建议用户自行操作，如：cdev的初始化

➤ cdev相关操作接口

- 初始化cdev变量，并设置fops

```
void cdev_init(struct cdev *cdev,  
               const struct file_operations *fops)
```

添加cdev到linux内核，完成驱动注册

```
int cdev_add(struct cdev *p,  
             dev_t dev, unsigned count)
```

参数dev：申请好的主设备号+起始从设备号

参数count：为该驱动所占用从设备号的数目

- 从内核中删除cdev数据

```
void cdev_del(struct cdev *p)
```

➤ 详细参看代码...

字符驱动注册小结:

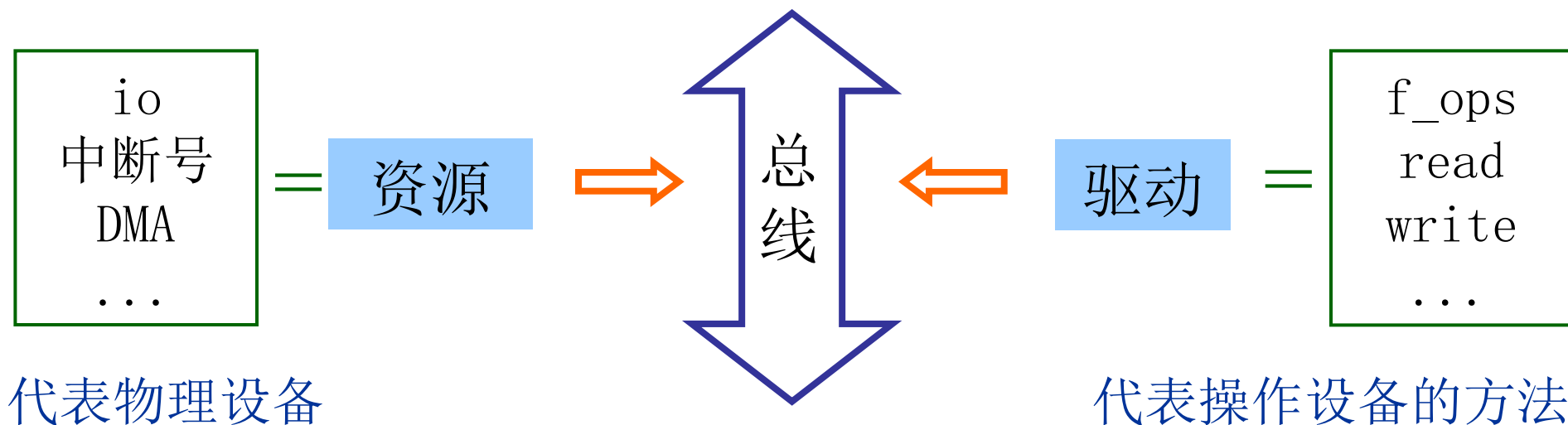
功能	函数
申请注册设备号（指定）	<code>register_chrdev_region(dev_t from, unsigned count, const char *name);</code>
申请注册设备号（动态）	<code>int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);</code>
初始化cdev	<code>cdev_init(struct cdev *cdev, const struct file_operations *fops);</code>
注册cdev	<code>cdev_add(struct cdev *p, dev_t dev, unsigned count);</code>
删除cdev	<code>cdev_del(struct cdev *p);</code>
注销设备号	<code>unregister_chrdev_region(dev_t from, unsigned count)</code>

- 杂项设备
- cdev方式注册字符驱动
- Platform总线
- Input子系统

- 总线：各部件之间传递信息的公共通道。
- 一个现实的linux设备和驱动通常需要挂载在一种总线上，这总线可以是：
 - 物理总线：USB/PCI/I2C/SPI总线。
 - 虚拟总线：Platform总线（触摸屏、LCD…）
- 之前我们的程序都把资源和驱动放在一起，这样会导致资源和驱动缺乏相互独立性，给管理和移植带来诸多不便。现在分别把它们放在platform总线统一管理，问题就迎刃而解。

- Platform总线是连接设备和驱动的桥梁，把device资源（代表物理设备）和driver（驱动程序）连系在一起的。

分层思想>>



➤采用总线的方式好处:

资源: platform机制将设备的资源单独注册进内核, 由内核进行统一管理。

驱动: 使用这些资源时通过提供的标准接口对资源进行申请并使用, 实现驱动与资源的分离与统一。

- 提高了驱动和资源管理的独立性
- 拥有更好的可移植性和安全性。

- platform机制开发的并不复杂，由两部分组成：
 - platform_device
 - platform_driver
- platform_device是用来描述当前驱动使用的平台硬件信息，一般情况定义在厂家提供的板级支持包中. 我们当前平台的硬件资源位于：
arch/arm/mach-s5pv210/mach-smdkc110.c
- platform_driver是驱动具体的操作接口，和file_operations形同意不同，也是一些函数接口

➤platform驱动开发流程:

实现platform_device



注册platform_device

实现platform_driver



注册platform_driver

```
struct platform_device {  
    const char *name;           /*资源名字*/  
    int id;                     /*一般写0或-1*/  
    struct device dev;  
    u32 num_resources;          /*资源大小*/  
    struct resource *resource;  /*资源*/  
  
    const struct platform_device_id *id_entry;  
    struct pdev_archdata archdata;  
};
```

```
struct device {  
    void (*release)(struct device *dev);  
    ... ..  
}
```

```
struct resource {  
    resource_size_t start; /*资源起始的物理地址*/  
    resource_size_t end; /*资源结束的物理地址*/  
    const char *name;  
    unsigned long flags; /*资源类型*/  
    struct resource *parent, *sibling, *child;  
};
```

flags:

I/O资源: IORESOURCE_MEM

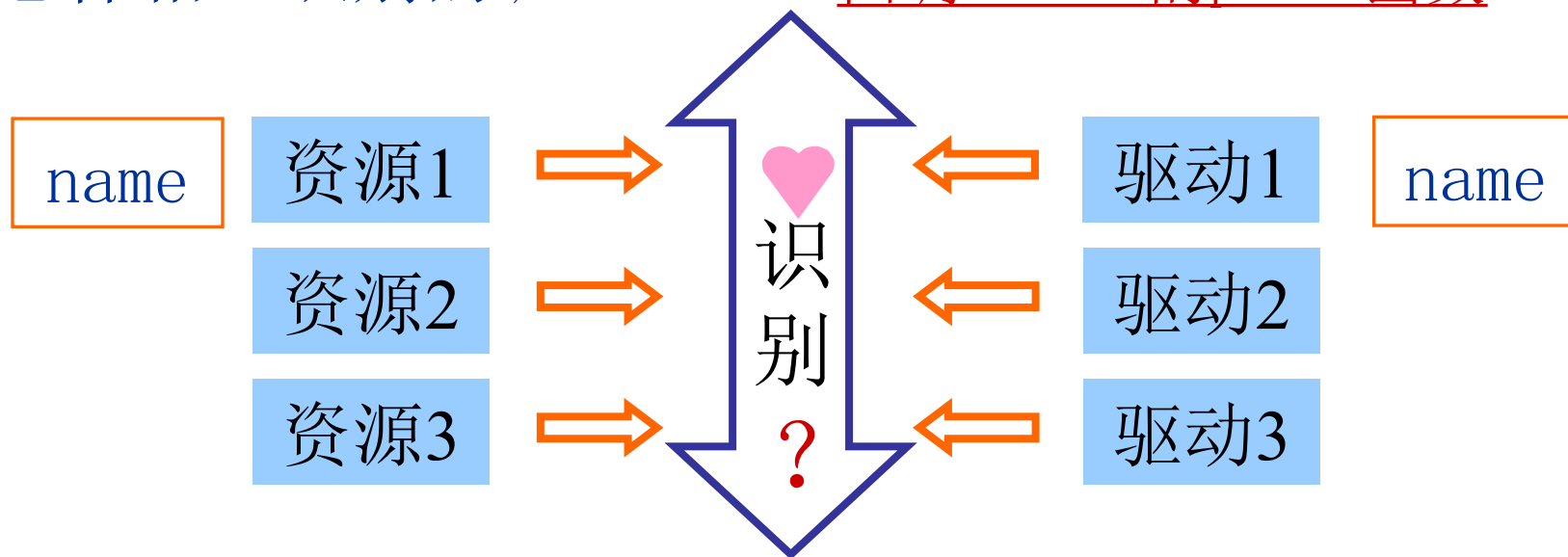
中断号: IORESOURCE_IRQ

```
struct platform_driver {  
    ★ int (*probe)(struct platform_device *);    /*匹配后回调函数*/  
    ★ int (*remove)(struct platform_device *);  
    void (*shutdown)(struct platform_device *);  
    int (*suspend)(struct platform_device *, pm_message_t state);  
    int (*resume)(struct platform_device *);  
    ★ struct device_driver driver;  
    const struct platform_device_id *id_table;  
};
```

```
struct device_driver {  
    struct module *owner;    /*填写THIS_MODULE*/  
    const char *name;    /*驱动名字*/  
};
```

- 注册进platform的device和driver不止一个，它们是怎样相互识别的呢？

回调driver的probe函数



总线上的各个device和driver都有一个name，platform_bus总线有一个专门的match函数通过name完成两者的匹配。

➤platform驱动开发流程:

实现platform_device

注册platform_device

实现platform_driver

注册platform_driver

内核自动匹配

匹配成功回调probe



·编写read/write函数
·实现file_operations



在probe里面:
·IO/资源初始化
·注册驱动
·生成设备节点
.....

➤ platform相关接口函数：资源+驱动

资源添加有两种方式：

- 直接在板级支持包mach-smdkv210.c添加，编译内核。
- ★• 单独编写一个.ko文件，模块添加。

•注册platform device到platform_bus

```
int platform_device_register(struct  
                                platform_device *pdev)
```

参数：填充好的platform_device

返回值：成功返回0.

➤ 驱动接口函数

- 注册platform driver到platform_bus

```
int platform_driver_register(struct platform_driver *drv)
```

参数drv:填充好的platform_driver

- 设备驱动端获取platform device

```
struct resource *platform_get_resource(  
    struct platform_device *dev,  
    unsigned int type,  
    unsigned int num)
```

参数dev: 内核传过来platform_device的指针

参数type:资源类型, 与device的flag对应

参数unm: 同类资源序号

- 杂项设备
- Proc接口访问驱动
- cdev方式注册字符驱动
- Platform总线
- input子系统

➤我们之前写按键驱动，使用`copy_to_user(buf, &key, len)`把键码返回给应用，但输入设备五花八门，存入buf的值各不相同，没有统一规范使得驱动极不通用，上层应用不知道定义什么数据类型去接驱动返回的值。



➤为了解决这个问题，input以“兼容并包”的大一统思想，把这些输入设备的键码、键值、上传方式都分类统一起来，提高驱动通用性(linux和andriod都适用)，减少应用和驱动开发者的沟通成本。

➤ input子系统:

为输入设备(按键、键盘、触摸屏)的驱动规范完成上报输入信息任务的子系统。input是对字符设备驱动的另一种封装。

➤ input子系统在驱动中不是必须的，它的存在只是规范了上报输入信息这一任务，减少驱动与应用开发工程师的沟通成本。

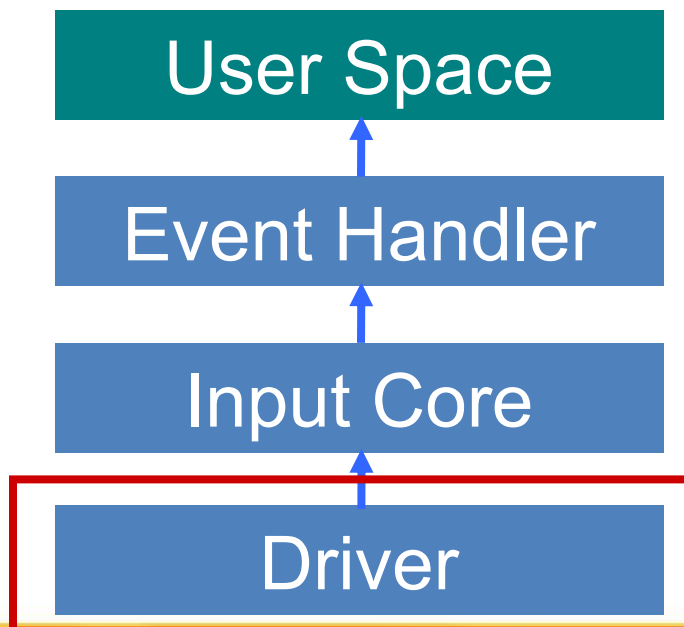
➤ input子系统是输入设备驱动一个标准，一个约定俗成的规范，几乎所有输入设备驱动都是使用input来上报输入信息的。

➤ input子系统的框架结构

子系统由三部分组成:



上报的路径:



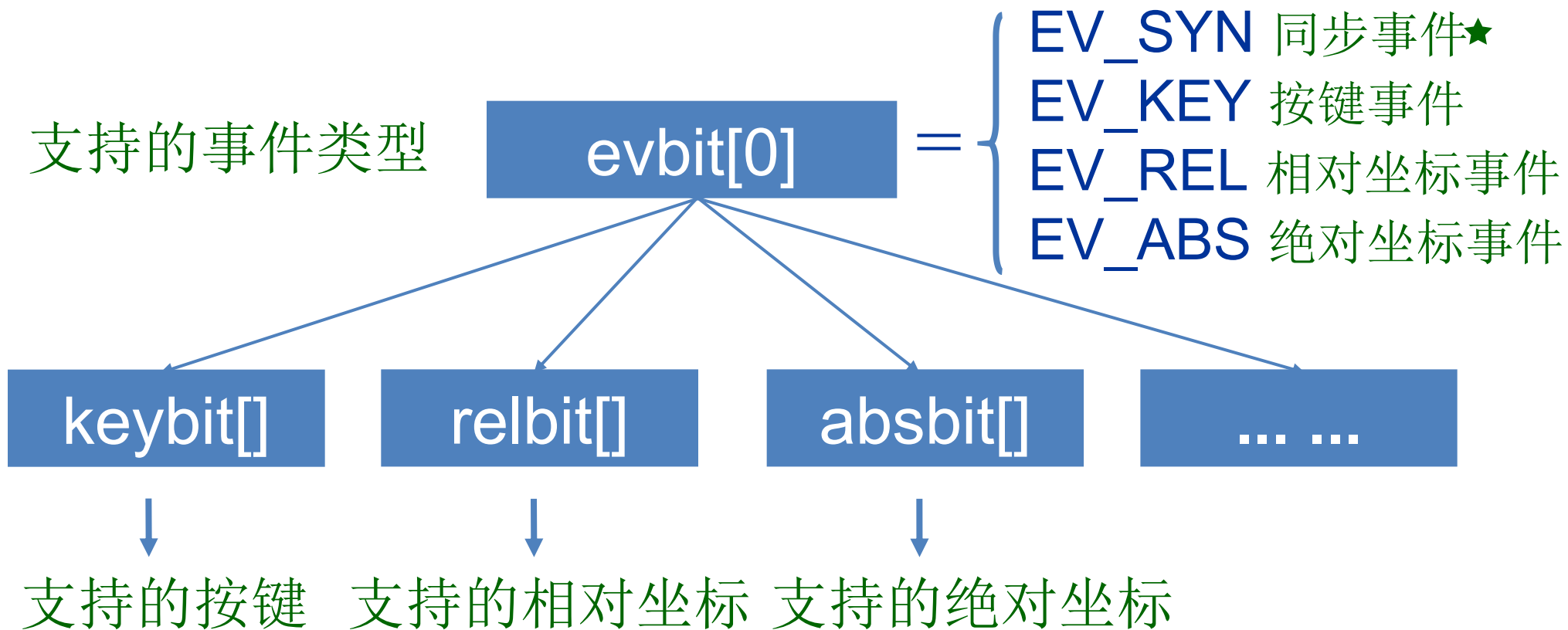
Driver的编写流程:

- ① 定义一个input_dev结构体
- ↓
- ② 申请input_dev内存空间并初始化
- ↓
- ③ 填充input_dev部分成员
- ↓
- ④ 向core注册一个input_dev
- ↓
- ⑤ 获取键码并上报按键（一般在中断里）

注：input是对字符设备驱动的封装，它在底层实现了file_operations这一套机制而不用我们去填充了，只需按照以上流程即可完成驱动。

input_dev重要成员 (input.h)

支持的事件类型



input_dev重要成员 (input.h)

keybit[] = {

#define	KEY_Q	16
#define	KEY_W	17
#define	KEY_E	18
#define	KEY_R	19
	

absbit[] = {

#define	ABS_X	0x00
#define	ABS_Y	0x01
#define	ABS_Z	0x02
	

input_dev赋值(以按键驱动为例)

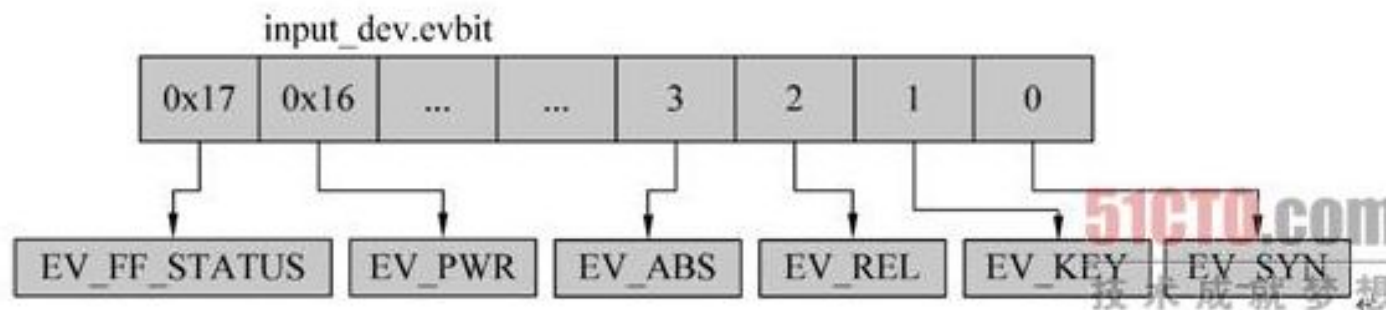
赋值有两种方法:

- ① 直接赋值
- ② 函数赋值

➤ 直接赋值:

```
evbit[0] = BIT_MASK(EV_SYN) | BIT_MASK(EV_KEY);
```

$\text{BIT_MASK}(x)$: 把32位中第x位置1



➤ 直接赋值:

```
keybit[BIT_WORD(KEY_D)] |= BIT_MASK(KEY_D);
```

表示KEY_D在keybit数组的第几个元素

BIT_WORD
定位所在行

ESC	KEY_1	KEY_2	KEY_2	KEY_2
KEY_Q	KEY_W	KEY_E	KEY_R	KEY_T
KEY_A	KEY_S	KEY_D	KEY_F	KEY_G
KEY_Z	KEY_X	KEY_C	KEY_V	KEY_B

BIT_MASK 定位所在列 (32位=32列) 并置1

➤ 函数赋值

set_bit(nr,addr);

- 参数nr: 要置1的那个位;
- 参数addr: 数组首地址

① `evbit[0]= BIT_MASK(EV_SYN) | BIT_MASK(EV_KEY);`




`set_bit(EV_SYN , evbit);`
`set_bit(EV_KEY , evbit);`

② `keybit[BIT_WORD(KEY_D)] |= BIT_MASK(KEY_D);`



`set_bit(KEY_D , keybit);`

Driver的流程:

- ① 定义一个input_dev结构体
- ↓
- ② 申请input_dev内存空间并初始化
- ↓
- ③ 填充input_dev部分成员 
- ↓
- ④ 向core注册一个input_dev
- ↓
- ⑤ 获取键码并上报按键（一般在中断里）

注：input是对字符设备驱动的封装，它在底层实现了file_operations这一套机制而不用我们去填充了，只需按照以上流程即可完成驱动。

➤ 函数接口

申请、注册与释放:

- 申请、初始化input_dev

```
struct input_dev *input_allocate_device(void);
```

- 注册input_dev

```
int input_register_device(struct input_dev *dev);
```

- 注销input_dev

```
void input_unregister_device(struct input_dev *dev);
```

➤ 上报按键(中断里面)

• 上报

```
input_report_key(struct input_dev *dev,  
                unsigned int code,  
                int value);
```

参数code: 键码(填充结构体时已经注册支持)

参数value: 按下1 或 抬起0

• 同步

```
input_sync(struct input_dev *dev);
```



凌阳教育官方微信：Sunplusedu

Tel: 400-705-9680, BBS: www.51develop.net, QQ群: 241275518

