链滴

Linux INPUT 子系统

作者：zhang-ke-wei

原文链接：https://ld246.com/article/1598093237235

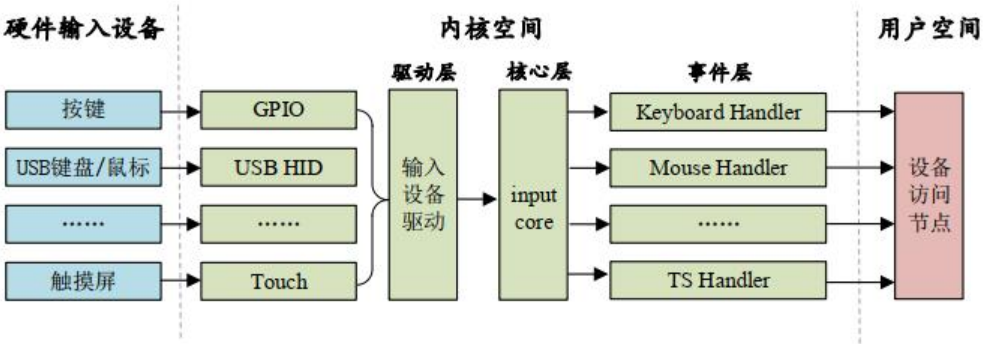
来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

按键、鼠标、键盘、触摸屏等都属于输入(input)设备， Linux 内核为此专门做了一个叫做 input子系的框架来处理输入事件。输入设备本质上还是字符设备，只是在此基础上套上了 input 框架，用户只要负责上报输入事件，比如按键值、坐标等信息， input 核心层负责处理这些事件。

一、input 子系统简介

input 子系统是管理输入的子系统，是 Linux 内核针对输入设备而创建的框架。比如按键输入、键盘鼠标、触摸屏等等这些都属于输入设备，不同的输入设备所代表的含义不同，按键和键盘就是代表按信息，鼠标和触摸屏代表坐标信息，因此在应用层的处理就不同，对于驱动编写者而言不需要去关心用层的事情，只需要按照要求上报这些输入事件即可。为此 input 子系统分为 input 驱动层、 input 心层、 input 事件处理层，最终给用户空间提供可访问的设备节点， input 子系统框架如图 所示：



驱动层：输入设备的具体驱动程序，比如按键驱动程序，向内核层报告输入内容。

核心层：承上启下，为驱动层提供输入设备注册和操作接口。通知事件层对输入事件进行处理。

事件层：主要和用户空间进行交互。

二、input 驱动编写流程

input 核心层会向 Linux 内核注册一个字符设备，大家找到 drivers/input/input.c 这个文件，input.c 就是 input 输入子系统的核心层，此文件里面有如下所示代码：

```
struct class input_class = {
    .name = "input",
    .devnode = input_devnode,
};
.....
static int __init input_init(void)
{
    int err;

    err = class_register(&input_class);
    if (err) {
    pr_err("unable to register input_dev class\n");
    return err;
    }

    err = input_proc_init();
    if (err)
        goto fail1;

    err = register_chrdev_region(MKDEV(INPUT_MAJOR, 0),
    INPUT_MAX_CHAR_DEVICES, "input");
    if (err) {
        pr_err("unable to register char major %d", INPUT_MAJOR);
        goto fail2;
    }

    return 0;

fail2: input_proc_exit();
fail1: class_unregister(&input_class);
    return err;
}
```

注册一个字符设备，主设备号为 INPUT_MAJOR， INPUT_MAJOR 定义在 include/uapi/linux/major h 文件中，定义如下：

```
#define INPUT_MAJOR 13
```

input 子系统的所有设备主设备号都为 13，我们在使用 input 子系统处理输入设备的时候就不需要注册字符设备了，我们只需要向系统注册一个 input_device 即可。

1.注册 input_dev

在使用 input 子系统的时候我们只需要注册一个 input 设备即可， input_dev 结构体表示 input设备此结构体定义在 include/linux/input.h 文件中，定义如下(有省略)：

```
struct input_dev {
    const char *name;
    const char *phys;
    const char *uniq;
```

```
    struct input_id id;

    unsigned long propbit[BITS_TO_LONGS(INPUT_PROP_CNT)];

    unsigned long evbit[BITS_TO_LONGS(EV_CNT)]; /* 事件类型的位图 */
    unsigned long keybit[BITS_TO_LONGS(KEY_CNT)]; /* 按键值的位图 */
    unsigned long relbit[BITS_TO_LONGS(REL_CNT)]; /* 相对坐标的位图 */
    unsigned long absbit[BITS_TO_LONGS(ABS_CNT)]; /* 绝对坐标的位图 */
    unsigned long mscbit[BITS_TO_LONGS(MSC_CNT)]; /* 杂项事件的位图 */
    unsigned long ledbit[BITS_TO_LONGS(LED_CNT)]; /*LED 相关的位图 */
    unsigned long sndbit[BITS_TO_LONGS(SND_CNT)];/* sound 有关的位图 */
    unsigned long ffbit[BITS_TO_LONGS(FF_CNT)]; /* 压力反馈的位图 */
    unsigned long swbit[BITS_TO_LONGS(SW_CNT)]; /*开关状态的位图 */
    ......
    bool devres_managed;
};
```

evbit 表示输入事件类型，可选的事件类型定义在 include/uapi/linux/input.h 文件中，事件类型如：

```
#define EV_SYN 0x00 /* 同步事件 */
#define EV_KEY 0x01 /* 按键事件 */
#define EV_REL 0x02 /* 相对坐标事件 */
#define EV_ABS 0x03 /* 绝对坐标事件 */
#define EV_MSC 0x04 /* 杂项(其他)事件 */
#define EV_SW 0x05 /* 开关事件 */
#define EV_LED 0x11 /* LED */
#define EV_SND 0x12 /* sound(声音) */
#define EV_REP 0x14 /* 重复事件 */
#define EV_FF 0x15 /* 压力事件 */
#define EV_PWR 0x16 /* 电源事件 */
#define EV_FF_STATUS 0x17 /* 压力状态事件 */
```

evbit、 keybit、 relbit 等等都是存放不同事件对应的值.

keybit 是按键事件使用的位图， Linux 内核定义了很多按键值，这些按键值定义在include/uapi/linu/input.h 文件中，按键值如下：

```
#define KEY_RESERVED 0
#define KEY_ESC 1
#define KEY_1 2
#define KEY_2 3
#define KEY_3 4
#define KEY_4 5
#define KEY_5 6
#define KEY_6 7
#define KEY_7 8
#define KEY_8 9
#define KEY_9 10
#define KEY_0 11
......
#define BTN_TRIGGER_HAPPY39 0x2e6
#define BTN_TRIGGER_HAPPY40 0x2e7
```

在编写 input 设备驱动的时候我们需要先申请一个 input_dev 结构体变量，使用input_allocate_devi

e 函数来申请一个 input_dev，此函数原型如下所示：

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

参数：无。

返回值： 申请到的 input_dev。

如果要注销的 input 设备的话需要使用 input_free_device 函数来释放掉前面申请到的input_dev， input_free_device 函数原型如下：

```
void input_free_device(struct input_dev *dev)
```

函数参数和返回值含义如下：

dev：需要释放的 input_dev。

返回值： 无。

申请好一个 input_dev 以后就需要初始化这个 input_dev，需要初始化的内容主要为事件类型(evbit)事件值(keybit)这两种。 input_dev 初始化完成以后就需要向 Linux 内核注册 input_dev了，需要用到nput_register_device 函数，此函数原型如下：

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

函数参数和返回值含义如下：

dev：要注册的 input_dev 。

返回值： 0， input_dev 注册成功；负值， input_dev 注册失败。

同样的，注销 input 驱动的时候也需要使用 input_unregister_device 函数来注销掉前面注册的 input dev， input_unregister_device 函数原型如下：

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

函数参数和返回值含义如下：

dev：要注销的 input_dev 。

返回值： 无。

综上所述， input_dev 注册过程如下：

①、使用 input_allocate_device 函数申请一个 input_dev。

②、初始化 input_dev 的事件类型以及事件值。

③、使用 input_register_device 函数向 Linux 系统注册前面初始化好的 input_dev。

④、卸载input驱动的时候需要先使用input_unregister_device函数注销掉注册的input_deve

example

```
struct input_dev *inputdev; /* input 结构体变量 */

/* 驱动入口函数 */
static int __init xxx_init(void)
{
    ......
    inputdev = input_allocate_device(); /* 申请 input_dev */
```

```
    inputdev->name = "test_inputdev"; /* 设置 input_dev 名字 */

    /*********第一种设置事件和事件值的方法***********/
    __set_bit(EV_KEY, inputdev->evbit); /* 设置产生按键事件 */
    __set_bit(EV_REP, inputdev->evbit); /* 重复事件 */
    __set_bit(KEY_0, inputdev->keybit); /*设置产生哪些按键值 */
    /**********************************************/

    /*********第二种设置事件和事件值的方法***********/
    keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
    keyinputdev.inputdev->keybit[BIT_WORD(KEY_0)] |= BIT_MASK(KEY_0);
    /**********************************************/

    /*********第三种设置事件和事件值的方法***********/
    keyinputdev.inputdev->evbit[0] = BIT_MASK(EV_KEY) | BIT_MASK(EV_REP);
    input_set_capability(keyinputdev.inputdev, EV_KEY, KEY_0);
    /**********************************************/

    /* 注册 input_dev */
    input_register_device(inputdev);
    ......
    return 0;
}

/* 驱动出口函数 */
static void __exit xxx_exit(void)
{
    input_unregister_device(inputdev); /* 注销 input_dev */
    input_free_device(inputdev); /* 删除 input_dev */
}
```

2.上报输入事件

当我们向 Linux 内核注册好 input_dev 以后仅仅只是完成了第一步， input 设备都是具有输入功能，但是具体是什么样的输入值 Linux 内核是不知道的，我们需要获取到具体的输入值，或者说是输入件，然后将输入事件上报给 Linux 内核。比如按键，我们需要在按键中断处理函数，或者消抖定时器断函数中将按键值上报给 Linux 内核，这样 Linux 内核才能获取到正确的输入值。不同的事件，其上事件的 API 函数不同。

一些常用的事件上报 API 函数：

input_event 函数，此函数用于上报指定的事件以及对应的值，函数原型如下：

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

dev：需要上报的 input_dev。

type: 上报的事件类型，比如 EV_KEY。

code： 事件码，也就是我们注册的按键值，比如 KEY_0、 KEY_1 等等。

value：事件值，比如 1 表示按键按下，0 表示按键松开。

返回值： 无。

input_event 函数可以上报所有的事件类型和事件值， Linux 内核也提供了其他的针对具体事件的上函数，这些函数其实都用到了 input_event 函数。比如上报按键所使用的input_report_key 函数，函数内容如下：

```
static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_KEY, code, !!value);
}
```

其他的事件上报函数：

```
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)
void input_report_ff_status(struct input_dev *dev, unsigned int code, int value)
void input_report_switch(struct input_dev *dev, unsigned int code, int value)
void input_mt_sync(struct input_dev *dev)
```

当我们上报事件以后还需要使用 input_sync 函数来告诉 Linux 内核 input 子系统上报结束，

input_sync 函数本质是上报一个同步事件，此函数原型如下所示：

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

函数参数和返回值含义如下：

dev：需要上报同步事件的 input_dev。

返回值： 无。

三、 input_event 结构体

Linux 内核使用 input_event 这个结构体来表示所有的输入事件， input_envent 结构体定义在includ/uapi/linux/input.h 文件中，结构体内容如下：

```
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```

time：时间，也就是此事件发生的时间，为 timeval 结构体类型， timeval 结构体定义如下：

```
typedef long __kernel_long_t;
typedef __kernel_long_t __kernel_time_t;
typedef __kernel_long_t __kernel_suseconds_t;

struct timeval {
    __kernel_time_t tv_sec; /* 秒 */
    __kernel_suseconds_t tv_usec; /* 微秒 */
}
```

type： 事件类型，比如 EV_KEY，表示此次事件为按键事件，此成员变量为 16 位。

code： 事件码，比如在 EV_KEY 事件中 code 就表示具体的按键码，如： KEY_0、 KEY_1等等这些键。此成员变量为 16 位。

value： 值，比如 EV_KEY 事件中 value 就是按键值，表示按键有没有被按下，如果为 1 的话说明按按下，如果为 0 的话说明按键没有被按下或者按键松开了。

input_envent 这个结构体非常重要，因为所有的输入设备最终都是按照 input_event 结构体呈现给户的，用户应用程序可以通过 input_event 来获取到具体的输入事件或相关的值，比如按键值等。