# 字符设备驱动之input按键

driver

## 灯板引脚

| Key | Pin | GPIO | No. | set pullup |
|---|---|---|---|---|
| 1 | P7 | B18 | 50 |  |
| 2 | P8 | C27 | 91 | PAD_PULLCTL1:1 = 1 |
| 3 | P10 | C26 | 90 | PAD_PULLCTL1:2 = 1 |
| 4 | P11 | C0 | 64 |  |
| 5 | P13 | C1 | 65 | PAD_PULLCTL0:26 = 1 |
| 6 | P15 | C4 | 68 | PAD_PULLCTL1:31 = 1 |
| 7 | P16 | A25 | 25 | PAD_PULLCTL0:(13:12) = 01 |
| 8 | P18 | C6 | 70 |  |
| 9 | P19 | C25 | 89 |  |
| 10 | P21 | C24 | 88 |  |
| 11 | P22 | C5 | 69 | PAD_PULLCTL1:30 = 1 |
| 12 | P23 | C22 | 86 |  |
| 13 | P24 | C23 | 87 |  |
| 14 | P26 | B19 | 51 |  |
| 15 | P29 | B15 | 47 |  |

## 定义按键

```
static struct pin_desc{
    unsigned int pin;
    unsigned int key_val;
    char *name;
};

static struct pin_desc pins_desc[] = {
    {OWL_GPIO_PORTC(27), KEY_2,  "SW2"},
    {OWL_GPIO_PORTC(26), KEY_3,  "SW3"},
    {OWL_GPIO_PORTC(1),  KEY_5,  "SW5"},
    {OWL_GPIO_PORTC(4),  KEY_6,  "SW6"},
    {OWL_GPIO_PORTA(25), KEY_7,  "SW7"},
    {OWL_GPIO_PORTC(5),  KEY_B, "SW11"},
};
```

## 通过寄存器设置上拉

因为灯板上按键没有上拉电阻, 所以在这里通过寄存器设置相应的GPIO引脚为上拉模式. 根据上面表格,设置引脚GPIO C27, C26, C1, C4, A25, C5(SW2, SW3, SW5, SW6, SW7, SW11)为上拉:

```
    act_writel(act_readl(PAD_PULLCTL0) | (1 << 26) | (1 << 12) & 0xffffdfff, PAD_PULLCTL0);
    act_writel(act_readl(PAD_PULLCTL1) | (3 << 30) | 6 , PAD_PULLCTL1);
```

## 申请与释放gpio

在设置gpio引脚为上拉状态后, 申请相应的gpio并设置为输入模式:

```
gpio_request(pins_desc[i].pin, pins_desc[i].name);
gpio_direction_input(pins_desc[i].pin);
```

## 获取中断号

在申请与释放中断时会用到中断号,如:

```c
request_irq(gpio_to_irq(pins_desc[i].pin), buttons_irq,
    IRQF_TRIGGER_FALLING, pins_desc[i].name, &pins_desc[i]);
```

这里使用了`gpio_to_irq(unsigned gpio)`函数, 将GPIO映射为IRQ中断, 参数为相应的引脚, 如**OWL_GPIO_PORTC(26)**.

> **NOTE:** S500的GPIO口不支持双边沿触发中断方式.

# input设备相关操作

- 声明input设备:

```c
static struct input_dev *button_dev;
```

- 为input设备申请内存空间:

```c
button_dev = input_allocate_device();
```

- 填充设备信息(可以省略):

```c
button_dev->name = "gpio-keys";
button_dev->id.bustype = BUS_HOST;
...
```

- 设置事件类型与事件:

```c
// 设置按键产生哪类事件
et_bit(EV_KEY, button_dev->evbit);
//set_bit(EV_REP, button_dev->evbit); //重复报告

// 设置能产生这类操作的哪些事件
for (i = 0; i < sizeof(pins_desc)/sizeof(pins_desc[0]); i++) {
    set_bit(pins_desc[i].key_val, button_dev->keybit);
}
```

**注意:**

如果设置了`set_bit(EV_REP, input->evbit);`也就是重复报告, 它的工作机制是这样的:

如果按键报告了`input_event(input, type, button->code,1);`之后, 在**250ms**(可以改)后, 依然没有报告`input_event(input, type, button->code,0);` 则 input 会每隔**33ms**继续报告一次`input_event(input, type, button->code,2);` 直到报告了`input_event(input, type, button->code,0);` 才停止, 这就是我们按住一个按键不松开时会一直打印键值的原因
这段代码在`drivers/input/input.c`中:

```
/*
* If delay and period are pre-set by the driver, then autorepeating
* is handled by the driver itself and we don't do it in input.c.
*/

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_DELAY] = 2500;
    dev->rep[REP_PERIOD] = 33;
}
```

这里要注意注释中的说明文字, 也就说如果我们自己的驱动里自己定义了`dev->rep[REP_DELAY] = 2500;` 那么就不会使用input的timer, 而要使用自己编写的timer.

- 注册设备:

```
input_register_device(button_dev);
```

- 注销与释放input设备:

```
input_unregister_device(button_dev);
input_free_device(button_dev);
```

- 上报事件并同步:

```
input_report_key(button_dev, button_irqs->key_val, 1);
input_report_key(button_dev, button_irqs->key_val, 0);
input_sync(button_dev);
```

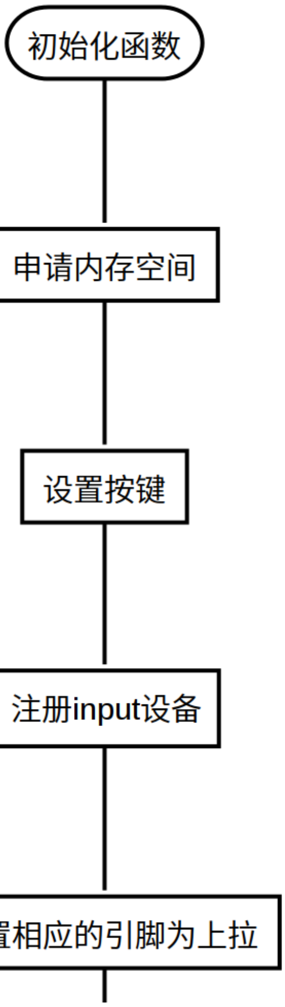
**注意:**

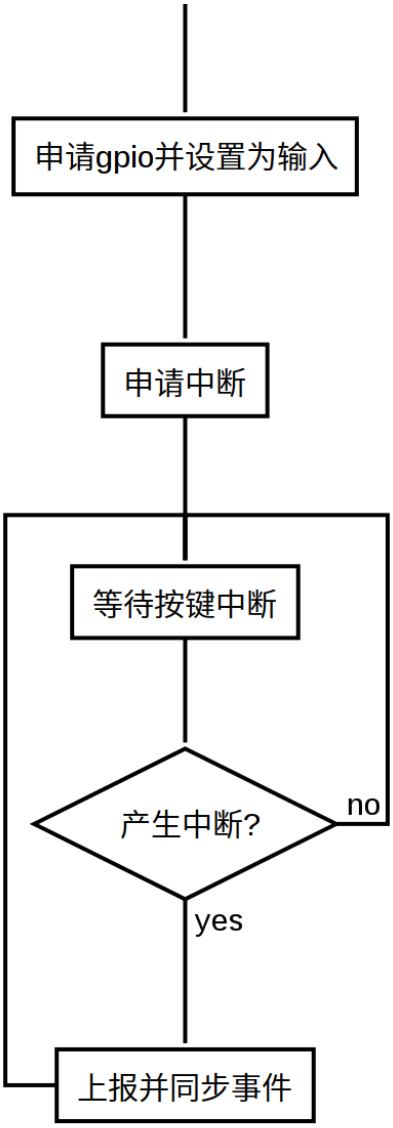
`input_event(input, type, button->code, !!state);`

如果第一次报告了`input_event(input, type, button->code,1);`第二次又报告了`input_event(input, type, button->code,1);`那么**第二次是报告不上的**，也就是说: 只有**键值变化**了报告才有效. 这也是按键驱动为什么都是**双边延触发**, 就是为了产生按键按下和按键抬起, 如果每次只报告一次按键按下, 那么驱动只会报告一次按键.

但是S500的GPIO口不支持双边沿触发中断方式, 所以在这里, 按键上报一次`input_event(input, type, button->code,1);`之后立即上报次`input_event(input, type, button->code,0);`. 这样就没有必要设置`set_bit(EV_REP, button_dev->evbit);`(重复报告)了.

## 流程图





# 源代码

## 驱动

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <linux/input/mt.h>
#include <linux/sched.h>
#include <linux/gpio.h>
#include <linux/platform_device.h>
#include <mach/gpio.h>
#include <mach/hardware.h>
#include <mach/irqs.h>
```

```
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <asm/io.h>

static struct pin_desc{
    unsigned int pin;
    unsigned int key_val;
    char *name;
};

static struct pin_desc pins_desc[] = {
//  {OWL_GPIO_PORTB(18), KEY_1,  "SW1"},
    {OWL_GPIO_PORTC(27), KEY_2,  "SW2"},
    {OWL_GPIO_PORTC(26), KEY_3,  "SW3"},
//  {OWL_GPIO_PORTC(0),  KEY_4,  "SW4"},
    {OWL_GPIO_PORTC(1),  KEY_5,  "SW5"},
    {OWL_GPIO_PORTC(4),  KEY_6,  "SW6"},
    {OWL_GPIO_PORTA(25), KEY_7,  "SW7"},
//  {OWL_GPIO_PORTC(6),  KEY_8,  "SW8"},
//  {OWL_GPIO_PORTC(25), KEY_9,  "SW9"},
//  {OWL_GPIO_PORTC(24), KEY_A, "SW10"},
    {OWL_GPIO_PORTC(5),  KEY_B, "SW11"},
//  {OWL_GPIO_PORTC(22), KEY_C, "SW12"},
//  {OWL_GPIO_PORTC(23), KEY_D, "SW13"},
//  {OWL_GPIO_PORTB(19), KEY_E, "SW14"},
//  {OWL_GPIO_PORTB(15), KEY_F, "SW15"},
};

static struct input_dev *button_dev;

static irqreturn_t button_interrupt(int irq, void *dev_id)
{
    struct pin_desc *button_irqs = (struct pin_desc*)dev_id;

    input_report_key(button_dev, button_irqs->key_val, 1);
    input_report_key(button_dev, button_irqs->key_val, 0);
    input_sync(button_dev);

    return IRQ_RETVAL(IRQ_HANDLED);
}

static int __init button_init(void)
{
    int err = 0;
    int i;

    button_dev = input_allocate_device();
    if (!button_dev) {
        printk("no enough memort\n");
        return -ENOMEM;
    }

    button_dev->name = "gpio-keys";
    button_dev->id.bustype = BUS_HOST;

    set_bit(EV_KEY, button_dev->evbit);
//   set_bit(EV_REP, button_dev->evbit);

    for (i = 0; i < sizeof(pins_desc)/sizeof(pins_desc[0]); i++) {
        set_bit(pins_desc[i].key_val, button_dev->keybit);
    }

    err = input_register_device(button_dev);
    if (err) {
        printk("failed to register device \n");
        goto err_register_dev;
    }

    act_writel(act_readl(PAD_PULLCTL0) | (1 << 26) | (1 << 12) & 0xffffdfff, PAD_PULLCTL0);
    act_writel(act_readl(PAD_PULLCTL1) | (3 << 30) | 6 , PAD_PULLCTL1);

    for (i = 0; i < sizeof(pins_desc)/sizeof(pins_desc[0]); i++) {
        gpio_request(pins_desc[i].pin, pins_desc[i].name);
        gpio_direction_input(pins_desc[i].pin);

        request_irq(gpio_to_irq(pins_desc[i].pin), button_interrupt,
            IRQF_SHARED|IRQF_TRIGGER_FALLING, pins_desc[i].name, &pins_desc[i]);
    }
    return 0;

err_register_dev:
    input_unregister_device(button_dev);
    input_free_device(button_dev);

    return err;
}

static void __exit button_exit(void)
```

```
{
    int i;
    input_unregister_device(button_dev);
    input_free_device(button_dev);
    for (i = 0; i < sizeof(pins_desc)/sizeof(pins_desc[0]); i++) {
        free_irq(gpio_to_irq(pins_desc[i].pin),&pins_desc[i]);
        gpio_free(pins_desc[i].pin);
    }
}

module_init(button_init);
module_exit(button_exit);

MODULE_AUTHOR("Tab Liu");
MODULE_DESCRIPTION("Just for Demon");
MODULE_LICENSE("GPL");
```

## 测试

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <linux/input.h>

int main(void)
{
    struct input_event ev_key;
    int fd;
    fd = open("/dev/input/event2", O_RDWR);

    if (fd < 0) {
        perror("open device buttons");
        exit(1);
    }
    while(1) {
        read(fd, &ev_key, sizeof(struct input_event));
        if (EV_KEY == ev_key.type)
            printf("type:%d,code:%d,value:%d\n", ev_key.type,ev_key.cod
e,ev_key.value);
    }
    close(fd);
    return 0;
}
```