# Linux 多按键中断驱动流程详解

## 一、核心功能概述

本驱动程序实现了对多个 GPIO 按键的中断管理，通过结构体封装、统一中断处理、等待队列与用户空间交互，实现了一个具备阻塞读功能的字符设备。

## 二、结构体设计

### 1. gpio\_irq\_t：描述一个按键的中断信息

struct gpio\_irq\_t{  
 unsigned int gpio; // GPIO 编号  
 unsigned int irq\_n; // 中断号  
 irq\_handler\_t fun; // 中断处理函数  
 unsigned long irq\_f; // 中断触发方式  
 const char\* name; // 中断名字  
 void\* dev; // 传给中断处理函数的参数  
};

### 2. key\_irq\_t：整体按键信息管理器

#define KEY\_AMOUNT 4  
  
struct key\_irq\_t{  
 struct gpio\_irq\_t gi[KEY\_AMOUNT]; // 所有按键信息  
 wait\_queue\_head\_t wq; // 等待队列  
 int cond; // 条件变量，1 表示有中断  
 unsigned char val; // 当前按键值，按位表示  
};

## 三、结构体初始化说明

在定义 keys\_irq 全局变量时就进行了结构体初始化：

struct key\_irq\_t keys\_irq={  
 .gi[0] = {(PAD\_GPIO\_A+28), PB\_PIO\_IRQ(PAD\_GPIO\_A+28), mykey\_irq\_handler, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING, "gpioa\_28\_irq", &keys\_irq},  
 .gi[1] = {(PAD\_GPIO\_B+30), PB\_PIO\_IRQ(PAD\_GPIO\_B+30), mykey\_irq\_handler, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING, "gpiob\_30\_irq", &keys\_irq},  
 .gi[2] = {(PAD\_GPIO\_B+31), PB\_PIO\_IRQ(PAD\_GPIO\_B+31), mykey\_irq\_handler, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING, "gpiob\_31\_irq", &keys\_irq},  
 .gi[3] = {(PAD\_GPIO\_B+9 ), PB\_PIO\_IRQ(PAD\_GPIO\_B+9 ), mykey\_irq\_handler, IRQF\_TRIGGER\_FALLING|IRQF\_TRIGGER\_RISING, "gpiob\_9\_irq" , &keys\_irq},  
 .cond = 0,  
};

### 为什么 gi[].dev = &keys\_irq

因为 .dev 会传给 request\_irq 的 dev\_id， 中断触发时会作为参数传入中断处理函数 mykey\_irq\_handler()， 可以通过它访问整个按键系统的状态。

这是实现“一个中断处理函数服务多个按键”的关键！

**重点体现**：

* dev\_id == &keys\_irq
* 所以 ki 拿到的是整个结构体 keys\_irq
* gi = ki->gi 就遍历了所有按键
* 通过 irq == gi[i].irq\_n 找出**是哪一个按键触发的中断**

## 四、初始化流程概述

在模块初始化函数 mykey\_init 中，完成以下操作：

1. 初始化等待队列头 init\_waitqueue\_head(&keys\_irq.wq);
2. 注册中断：遍历结构体数组 gi[]，调用 request\_irq()
3. 注册混杂设备 misc\_register()

这样，整个设备系统就具备了对外接口与中断响应能力。

## 五、中断处理流程

### 中断处理函数 mykey\_irq\_handler()：

irqreturn\_t mykey\_irq\_handler(int irq, void \*dev\_id);

#### 主要功能：

1. 判断是哪个按键触发

* if(irq==gi->irq\_n){
* ki->val|=gpio\_get\_valuer(gi->gpio)?0:1<<i;
* }

1. 读取 GPIO 电平状态（高/低）

* gpio\_get\_valuer(gi->gpio)

1. 设置对应按键 bit 值

* ki->val|=

1. 唤醒等待队列，设置 cond = 1

* wake\_up(&ki->wq);
* ki->cond=1;

## 六、read 函数阻塞机制

### 函数名：mykey\_read()

wait\_event\_interruptible(keys\_irq.wq, keys\_irq.cond);

#### 流程：

* 用户态调用 read()，若无中断，则阻塞。
* 中断触发后：
  + 设置 val 按键值
  + 设置 cond=1，并唤醒等待队列
* 继续往下执行，将按键值 copy\_to\_user

## 七、模块加载流程 mykey\_init

for 每个按键:  
 request\_irq(gi->irq\_n, gi->fun, gi->irq\_f, gi->name, gi->dev);

* 注册所有中断
* 注册混杂设备 misc\_register
* 初始化等待队列头

### 卸载函数 mykey\_exit()：

* deregister 混杂设备
* 释放所有中断 free\_irq

## 八、设备注册

static struct miscdevice mykey\_misc = {  
 .minor = MISC\_DYNAMIC\_MINOR,  
 .name = "MY\_KEY",  
 .fops = &mykey\_fops,  
};

/dev/MY\_KEY 用户空间访问接口

## 九、整体流程图

+---------------------+  
 | 用户调用 read() |  
 +---------------------+  
 |  
 wait\_event\_interruptible  
 |  
 +------------------+  
 | 等待中断唤醒 |  
 +------------------+  
 |  
 中断触发 --> mykey\_irq\_handler()  
 |  
 设置 val、cond=1  
 |  
 wake\_up()  
 |  
 +---------------------------+  
 | 继续 read -> copy\_to\_user |  
 +---------------------------+

## 十、常见易错点

* .dev = &keys\_irq，否则中断服务函数拿不到全局状态
* 中断中不能直接 copy\_to\_user，只能通过唤醒驱动主线程
* 中断注册失败记得逐个释放，防止内核崩溃
* init\_waitqueue\_head 要在注册设备之前初始化

## 十一、总结

本驱动充分体现了 Linux 驱动开发中“中断处理 + 等待队列 + 字符设备 + 封装结构体”的典型模式，非常适合作为模板用于多个类似场景，如多个 GPIO 输入设备、传感器事件等。

建议学完本驱动后动手复写，并逐步加入：

* 防抖机制
* input 子系统适配
* 按键时长识别（短按/长按）等功能。