# Linux 中断下半部 tasklet 应用文档（以多按键驱动为例）

## 一、任务背景

在嵌入式 Linux 驱动开发中，处理外部按键等中断信号时，为了提高中断响应效率，常使用中断下半部机制（如 tasklet）将耗时操作延后处理。

本文档以实际代码为基础，讲解如何使用 tasklet 实现多按键中断处理，并分析 unsigned long irq\_r 字段的引入原因、类型使用理由及转换的必要性。

## 二、结构体定义

### 1. gpio\_irq\_t：每个按键的信息结构体

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

### 2. key\_irq\_t：用于统一管理所有按键

struct key\_irq\_t {  
#define KEY\_AMOUNT 4  
 struct gpio\_irq\_t gi[KEY\_AMOUNT]; // 多个 GPIO 中断结构体  
 wait\_queue\_head\_t wq; // 等待队列  
 int cond; // 条件变量，是否唤醒读进程  
 unsigned char val; // 每个位表示按键状态  
 unsigned long irq\_r; // 当前触发的中断号  
};

🔍 为什么增加 unsigned long irq\_r？

* 多个按键共用一个中断函数时，为了区分到底是哪个按键触发了中断，需要记录 irq。
* irq\_r 即 recent irq，用来保存中断上半部 irq 参数的值，传给 tasklet 下半部使用。
* tasklet 函数参数类型必须是 unsigned long，因此选择使用 unsigned long 类型的 irq\_r 能更高效地传递并使用。

## 三、tasklet 使用流程

### 1. 声明 tasklet

DECLARE\_TASKLET(mystasklet, mystasklet\_handler, 0);

* mystasklet\_handler 是 tasklet 的处理函数
* 0 是初始化 data 参数，最终会被中断函数覆盖为结构体指针（强转为 unsigned long）

📌 **为什么写在 mykey\_irq\_handler 和 mystasklet\_handler 中间？**

为了保证先声明后使用，tasklet 必须在中断函数和 tasklet 函数之间声明，这样中断函数中才能调用 tasklet\_schedule，而 tasklet 函数又不受声明顺序影响。

### 2. 中断服务函数

static irqreturn\_t mykey\_irq\_handler(int irq, void \*dev\_id) {  
 struct key\_irq\_t \*ki = (struct key\_irq\_t \*)dev\_id;  
 ki->irq\_r = irq; // 保存当前中断号  
 mystasklet.data = (unsigned long)dev\_id; // 强制转换传给下半部  
 tasklet\_schedule(&mystasklet); // 调度 tasklet 执行  
 return IRQ\_HANDLED;  
}

⚠️ 注意：tasklet\_struct.data 类型是 unsigned long，传结构体指针时必须强制转换。

### 3. tasklet 执行函数

void mystasklet\_handler(unsigned long data) {  
 int i;  
 struct key\_irq\_t \*ki = (struct key\_irq\_t \*)data;  
 struct gpio\_irq\_t \*gi = ki->gi;  
  
 for (i = 0; i < KEY\_AMOUNT; i++) {  
 if (ki->irq\_r == gi->irq\_n) {  
 ki->val |= gpio\_get\_value(gi->gpio) ? 0 : (1 << i);  
 }  
 gi++;  
 }  
  
 ki->cond = 1;  
 wake\_up(&ki->wq);  
 printk(KERN\_INFO "mystasklet\_handler exit\n");  
}

* 使用 irq\_r 比对具体是哪个按键被按下
* 读取 GPIO 电平并更新 val
* 唤醒等待队列，通知用户空间有按键按下

### 4. 用户空间读取函数

ssize\_t mykey\_read(...) {  
 wait\_event\_interruptible(keys\_irq.wq, keys\_irq.cond);  
 keys\_irq.cond = 0;  
 copy\_to\_user(buf, &keys\_irq.val, sizeof(keys\_irq.val));  
 keys\_irq.val = 0;  
 return len;  
}

* 阻塞等待按键事件（由下半部唤醒）
* 将按键状态传给用户空间

## 四、整体执行流程

1. DECLARE\_TASKLET(...)
   * 注册 tasklet，设置 handler
2. request\_irq(...)
   * 注册中断处理函数 mykey\_irq\_handler
3. mykey\_irq\_handler()
   * 获取中断号 irq，保存至 irq\_r
   * 设置 tasklet.data = (unsigned long)dev\_id
   * 调度执行下半部 tasklet
4. mystasklet\_handler()
   * 恢复 key\_irq\_t 指针（强转回来）
   * 根据 irq\_r 确定是哪个按键触发
   * 读取电平值，更新 val
   * 设置 cond 并唤醒等待队列
5. mykey\_read()
   * 阻塞等待 cond=1
   * 读取并清除 val
   * 返回给用户

## 五、注意事项总结

* tasklet\_struct.data 是 unsigned long，必须强制转换结构体指针。
* irq\_r 是为了标记到底是哪个中断触发，必须由上半部传给下半部。
* 中断处理越快越好，耗时操作（如 GPIO 读、判断、唤醒）放入下半部执行。
* 多个按键共用中断函数时，一定要对中断来源进行确认与分发。

## 六、经验总结附录（用户理解补充）

我刚才又看了几遍代码，我发现在上半部传递给下半部的时候的类型转换 mystasklet.data = (unsigned long)dev\_id; 是必须的，**但是设计 ki 是为了代码的可读性**，不然就需要写很多嵌套结构体访问的长代码。

✅ 这体现了驱动开发中： - 封装结构体传参的规范性； - 减少冗余、提升可读性； - 明确分工——上半部采集基本信息，下半部处理逻辑。

## 七、总结语

通过 tasklet 机制，我们成功将多个按键的中断处理延后到软中断阶段完成，既提升了中断响应速度，也降低了中断处理负载。

使用 irq\_r 明确事件来源，是应对多个共享中断设备的必备设计。

强制类型转换为 unsigned long 是与 tasklet 接口兼容的标准方式，理解其背后的机制有助于更清晰地掌握中断下半部的设计原理。