# 中断使用

Linux中断的使用非常简便，通常只需要完成申请中断、实现中断服务即可，使能中断已经在申请中断时完成。

## 申请/释放

int request\_irq(unsigned int irq,irq\_handler\_t handler, unsigned long flags, const char \*name,void \*dev);

中断申请。

irq：中断号。

handler：中断服务函数，中断触发后执行。

flags：中断标志，用于设置中断的触发方式与特性，不同标志可以或起来用，常用标志如下：

|  |  |
| --- | --- |
| IRQF\_TRIGGER\_NONE | 无触发 |
| IRQF\_TRIGGER\_RISING | 上升沿触发 |
| IRQF\_TRIGGER\_FALLING | 下降沿触发 |
| IRQF\_TRIGGER\_HIGH | 高电平触发 |
| IRQF\_TRIGGER\_LOW | 低电平触发 |
| IRQF\_SHARED | 共享中断 |
| IRQF\_ONESHOT | 单次中断 |

name: 中断名称，申请成功后会出现在/proc/interrupts下。

dev：传递到中断服务函数，是中断服务函数的第二个参数。

void free\_irq(unsigned int irq, void \*dev);

irq：中断号。

dev：如果是共享中断，这个参数用于查找，只有所有使用此共享中断的设备都进行了释放，中断才会被禁用。

## 中断服务

中断服务的格式为：

irqreturn\_t (\*irq\_handler\_t) (int, void \*)

第一个参数就是中断号，第二个参数就是上面提到的dev。

返回值有三种：

IRQ\_NONE 中断不是来自于此设备或者未处理

IRQ\_HANDLED 中断处理完毕

IRQ\_WAKE\_THREAD 用于当使用request\_threaded\_irq进行中断申请时，返回此值内核就会执行申请中断时填充的thread\_fn。

## 使能/禁用

虽然使能/禁用中断的使用机会很少，但是毕竟还是会接触到，比如在调用spin\_lock\_irqsave()的时候就会禁用中断。

使能/禁用单个中断

disable\_irq(int irq);

disable\_irq\_nosync(int irq);

enable\_irq(int irq);

disable\_irq会等待当前正在执行的中断处理完成才会返回，所以如果在中断上半部中调用此函数就会造成死锁，而disable\_irq\_nosync会立即返回，在中断上半部调用也没有问题。

使能/禁用本CPU中断

local\_irq\_save(flags)

local\_irq\_disable()

禁用本CPU中断，前者会将目前的中断状态进行保存。

local\_irq\_restore(flags)

local\_irq\_enable()

前者会恢复到local\_irq\_save保存的状态，后者无条件打开中断。

# 中断下半部

中断会打断内核的调度，只有中断处理完成才会重新调度，这就需要我们的中断服务快速完成，那么对于那种需要在中断服务中做很多工作的情况就需要用到中断下半部机制。Linux目前提供的下半部机制包括tasklet、工作队列、软中断、中断线程化。

## softirq

软中断一般不直接使用，下面的tasklet其实就是基于软中断实现的。

## tasklet

tasklet的执行的上下文是软中断，它的使用需要定义tasklet以及其处理函数，然后初始化：

void tasklet\_init(struct tasklet\_struct \*t,

void (\*func)(unsigned long), unsigned long data)

func: tasklet触发时的处理函数。

data：传递到func的输入参数。

可以在中断上半部中调用如下函数进行调度：

tasklet\_schedule(struct tasklet\_struct \*t)

如下函数会禁止tasklet的调度:

void tasklet\_kill(struct tasklet\_struct \*t)

## 工作队列

工作队列的执行上下文为内核线程，所以与tasklet最大的区别就是可以调度和睡眠。

初始化工作队列：

INIT\_WORK(\_work, \_func)

\_work：由struct work\_struct定义

\_func：处理函数

调度工作队列的执行：

schedule\_work(struct work\_struct \*work)

注意工作队列并不能像tasklet一样向处理函数中传递参数，这个时候就可以使用我们在字符设备驱动中提到的container\_of(ptr, type, member)函数，因为处理函数还是有一个参数是work本身的指针，如果work是一个结构的成员，则可以通过container\_of获取到此结构。

## thread\_irq

上面在讲中断返回值得时候已经提到过thread\_irq，我们可以通过如下方式进行中断申请：

int request\_threaded\_irq(unsigned int irq,irq\_handler\_t handler, irq\_handler\_t thread\_fn, unsigned long flags, const char \*name,void\*dev);

与request\_irq的区别就是多了thread\_fn，用此方式申请中断时，内核就会为这个中断号分配内核线程。中断处理函数handler返回值如果为IRQ\_WAKE\_THREAD，那么thread\_fn就会被执行。

同样因为thread\_fn的上下文是内核线程，那么thread\_fn也可以像工作队列一样可以睡眠。

## 区别

上面提到了4种中断下半部处理方式，主要介绍了三种，他们都有什么特点，又有什么区别，又该如何选择呢？

**tasklet:**

* 不能睡眠
* 运行于软中断上下文
* 因为运行于软中断上下文，也就是说运行于原子上下文，所以同一时间哪怕是不同处理器上也只有一个相同的tasklet运行，但是不同的tasklet还是可以并发的。
* tasklet运行时还是能被中断抢占，所以下半部和中断处理函数的锁还是需要的

**工作队列：**

* 可以睡眠
* 运行于内核线程上下文
* 非原子上下文，同一时间同一个工作队列可以运行于不同的处理器上

**thread\_irq：**

* 可以睡眠
* 运行于内核线程上下文

因为tasklet还是属于软中断，是在硬件中断处理程序的退出路径上运行，优先级更高，此时进程抢占被禁止，所以如果我们希望下半部更快被执行，那么就可以使用tasklet。

如果下半部中可能睡眠，比如有申请内存操作，就不能使用tasklet了，此时可以使用工作队列或者thread\_irq。那么工作队列和thread\_irq怎么选择，thread\_irq的内核线程的调度类别是SCHED\_FIFO，属于实时进程，而工作队列则属于普通进程，所以如果有实时需求肯定优先选择thread\_irq，如果没有实时需求那么两个任意选。

# 示例说明

示例中使用GPIO中断进行说明，insmod gpio\_interrupt.ko type=0，模块挂载时的type输入参数控制中断下半部机制，type=1为tasklet，type=2为工作队列，type=3为thread irq，其他输入参数都为不使用下半部机制。示例中通过打印信息就可判断是否正确进入中断下半部，也通过打印type值判断参数传递是否成功。

## 设备树

&am33xx\_pinmux {//引脚复用

interrupt\_gpio: interrupt\_gpio {

pinctrl-single,pins = <

AM33XX\_IOPAD(0x8ec, PIN\_INPUT | MUX\_MODE7) /\* (R6) \*/

>;

};

};

gpio\_interrupt: gpio\_interrupt@0 {

compatible = "gpio-interrupt";

gpios = <&gpio2 23 GPIO\_ACTIVE\_LOW>;

pinctrl-names = "default";

pinctrl-0 = <&interrupt\_gpio>;

};