8 Linux 的中断框架

中断是很常用的功能，Linux 内核中也实现了完善的中断框架，这一章来学习一下 Linux 内核中中断的简单用法。

8.1 Linux 中断框架简介

8.1.1 接口函数

Linux 内核中的中断框架使用已经相当便捷，一般需要做三件事申请中断、实现中断服务函数、使能中断。对应的接口函数如下：

(一) 中断申请和释放函数

1) 中断申请函数 request_irq，该函数可能会导致睡眠，在申请成功后会使能中断，函数原型：

```
Int request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags, const char *name, void *dev);
```

参数说明：

irq：申请的中断号，中断号也可以叫中断线，是中断的唯一标识，也是内核找到对应中断服务函数的依据。

handler：中断服务函数，中断触发后会执行的函数。

flags：中断标志，用于设置中断的触发方式和其他特性，常用的标识有：

```
/* 无触发 */
IRQF_TRIGGER_NONE
/* 上升沿触发 */
IRQF_TRIGGER_RISING
/* 下降沿触发 */
IRQF_TRIGGER_FALLING
/* 高电平触发 */
IRQF_TRIGGER_HIGH
/* 低电平触发 */
IRQF_TRIGGER_LOW
/* 单次中断 */
IRQF_ONESHOT
/* 作为定时器中断 */
IRQF_TIMER
/* 共享中断，多个设备共享一个中断号时需要此标志 */
IRQF_SHARED
```

可在/include/linux/interrupt.h 文件中可以查看全部的 flag 和英文释义。中断标志可以使用|号来组合，如 IRQF_TRIGGER_RISING| IRQF_ONESHOT 意为上升沿触发的单次中断。

name：中断名称，申请中断成功后，在/proc/interrupts 文件中可以找到这个名字。

dev：flag 设置 IRQF_SHARED 时，使用 dev 来区分不同的设备，dev 的值会传递给中断服务函数的第二个参数。

返回值：0-申请成功，-EBUSY-中断已被占用，其他值都表示申请失败。

2) 和中断申请相对的中断释放函数 free_irq，如果目标中断不是共享中断，那么 free_irq 函数在释放中断后，会禁止中断并删除中断服务函数，原型如下：

```
void free_irq(unsigned int irq, void *dev);
```

参数说明：

irq：需要释放的中断。

dev：释放的中断如果是共享中断，用这个参数来区分具体的中断，只有共享中断下所有的 dev 都被释放时，free_irq 函数才会禁止中断并删除中断服务函数。

(二) 实现服务申请函数

中断服务函数的格式为：

```
irqreturn_t (*irq_handler_t) (int, void *)
```

第一个参数 int 型时中断服务函数对应的中断号。

第二个参数是通用指针，需要和 request_irq 函数的参数 dev 一致。

返回值 irqreturn_t 为美剧类型，一般在服务函数中用下面的方式返回值：

```
return IRQ_RETVAL(IRQ_HANDLED);
```

(三) 中断使能和禁止函数

1) enable_irq(unsigned int irq) 、disable_irq(unsigned int irq) 和 disable_irq_nosync(unsigned int irq)

enable_irq 和 disable_irq 分别是中断使能和禁止函数，irq 是目标中断号。disable_irq 会等待目标中断的中断服务函数执行结束才会禁止中断，如果想要立即禁止中断则可以使用 disable_irq_nosync()函数。

2) local_irq_enable()和 local_irq_disable()

local_irq_enable()函数用于使能当前处理器的中断系统。

local_irq_disable()函数用于禁止当前处理器的中断系统。

3) local_irq_save(flags)和 local_irq_restore(flags)

local_irq_save(flags)函数也是用于禁止当前处理器中断，但是会把进之前的中断状态保存在输入参数 flags 中。而 local_irq_restore(flags)函数则是把中断恢复到 flags 中记录的状态。

8.1.2 Linux 的下半部机制

上半部下半部是为了尽量缩短中断服务函数的处理过程而引入的机制，上半部就是中断触发后立即执行的中断服务函数，而下半部就是对中断服务函数的延时处理。因为中断的优先级较高，如果处理内容过多，长时间占用处理器会影响其他代码的运行，所以上半部要尽量的短。裸机程序里，在中断处理函数中树 flag，然后到主程序大循环中去轮询判断这个 flag 再做相应的操作，就是一种上半部下半部的思想。Linux 中的上半部就是指中断服务函数 irq_handler_t。至于那些任务放在上半部哪些放在下半部，没有明确的界限，一般我们把对时间敏感、不能被打断的任务放到上半部中，其他的都可以考虑放到下半部。

Linux 针对下半部也提供了一些完善的机制：

1) 软中断

软中断结构体定义在 include/linux/interrupt.h 文件中，具体如下：

```
1.  struct softirq_action
```

```
2.  {
3.      void (*action)(struct softirq_action *);
4.  };
```

内核在 kernel/softirq.c 文件中定义了全局的软中断向量表：

```
static struct softirq_action softirq_vec[NR_SOFTIRQS];
```

NR_SOFTIRQS 为枚举类型的最大值，该枚举类型定义在 include/linux/interrupt.h 中：

```
1.  enum
2.  {
3.      HI_SOFTIRQ=0,
4.      TIMER_SOFTIRQ,
5.      NET_TX_SOFTIRQ,
6.      NET_RX_SOFTIRQ,
7.      BLOCK_SOFTIRQ,
8.      IRQ_POLL_SOFTIRQ,
9.      TASKLET_SOFTIRQ,
10.     SCHED_SOFTIRQ,
11.     HRTIMER_SOFTIRQ,
12.     RCU_SOFTIRQ,
13.
14.     NR_SOFTIRQS
15. };
```

代表了十个软中断，要使用软中断只能向内核定义的软中断向量表注册，注册软中断需要使用函数：

```
void (int nr, void (*action)(struct softirq_action *));
```

参数说明：

nr：小于 NR_SOFTIRQS 的枚举值。

action：对应的软中断服务函数。

软中断必须在编译时静态注册，注册完成就需要使用 raise_softirq(unsigned int nr)触发，nr 即为需要触发的软中断。

但下半部机制通常不用软中断，而是使用下面要讲的 tasklets 机制。

2)　tasklets 机制

Linux 内核在 softirq_int 函数中初始化软中断，其中 HI_SOFTIRQ 和 TASKLET_SOFTIRQ 是默认打开的。tasklets 机制就是在这两个软中断基础上实现的。

tasklets 的结构体也定义在头文件 include/linux/interrupt.h 中，定义如下：

```
1.  struct tasklet_struct
2.  {
3.      struct tasklet_struct *next;
4.      unsigned long state;
5.      atomic_t count;
6.      void (*func)(unsigned long);
7.      unsigned long data;
```

```
};
```

其中 func 就是相当于是 tasklet 的中断服务函数，tasklet 的定义和初始化可以直接用下面的宏定义来完成：

```
DECLARE_TASKLET(name, func, data)
```

name：tasklet 的名字。

func：tasklet 触发时的处理函数。

data：传递给 func 的输入参数。

初始化完成后，调用以下函数即可激活 tesklet：

```
tasklet_schedule(struct tasklet_struct *t)
```

激活后 tesklet 的服务函数就会在合适的时间运行。用作中断的下半段时，就在上半段中调用该函数。如果要用优先级较高的 tasklet，就使用 tasklet_hi_schedule(struct tasklet_struct *t)函数激活。

tasklet 的下半段机制使用示例：

```
/* 定义 taselet */
struct tasklet_struct example;
/* tasklet 处理函数 */
void testtasklet_func(unsigned long data)
{
    /* tasklet 具体处理内容 */
}
/* 中断处理函数 */
irqreturn_t test_handler(int irq, void *dev_id)
{
    /* 调度 tasklet */
    tasklet_schedule(&example);
}
/* 驱动入口函数 */
static int __init xxxx_init(void)
{
    /* 初始化 tasklet */
    tasklet_init(&example, testtasklet_func, data);
    /* 注册中断处理函数 */
    request_irq(irq, test_handler, 0, "name", &dev);
}
```

3)　工作队列

工作队列也是下半部的实现方案。与 tasklet 相对的，工作队列是可阻塞的，因此不能在中断上下文中运行。工作队列的队列实现我们可以不用去管，要使用工作队列，只要定一个工作即可。

工作结构体为 work_struct，定义在/include/linux/workqueue.h 文件中：

```
struct work_struct {
    atomic_long_t data;
```

```
    struct list_head entry;
    work_func_t func;
#ifdef CONFIG_LOCKDEP
    struct lockdep_map lockdep_map;
#endif
};
```

func 即为需要处理的函数。可使用以下宏定义来创建并初始化工作：

```
DECLARE_WORK(n, f)
```

n：需要创建并初始化的工作结构体 work_struct 的名称。

f：工作队列需要处理的函数。

初始化完成后，使用下面的函数来调用工作队列：

```
bool schedule_work(struct work_struct *work)
```

work：需要调用的工作。

返回值：0 成功，1 失败。

workqueue 的下半段机制使用示例：

```
/* 定义工作(work) */
struct work_struct example;
/* work 处理函数 */
void work_func_t(struct work_struct *work);
{
    /* work 具体处理内容 */
}
/* 中断处理函数 */
irqreturn_t test_handler(int irq, void *dev_id)
{
    /* 调度 work */
    schedule_work(&example);
}
/* 驱动入口函数 */
static int __init xxxx_init(void)
{
    ......
    /* 初始化 work */
    INIT_WORK(&example, work_func_t);
    /* 注册中断处理函数 */
    request_irq(irq, test_handler, 0, "name", &dev);
}
```

8.1.3 设备树中的中断

设备树中，通用的中断设置方法可参考文档 Documentation/devicetree/bindings/arm/arm,gic.txt。Xilinx 的设备树中断控制器的设置与 Linux 内核的通用设置稍有区别，可以查看文

档 Documentation/devicetree/bindings/arm/xilinx,intc.txt 了解详情。看这个文件最后的一段例子：

```
1.  axi_intc_0: interrupt-controller@41800000 {
2.      #interrupt-cells = <2>;
3.      compatible = "xlnx,xps-intc-1.00.a";
4.      interrupt-controller;
5.      interrupt-parent = <&ps7_scugic_0>;
6.      interrupts = <0 29 4>;
7.      reg = <0x41800000 0x10000>;
8.      xlnx,kind-of-intr = <0x1>;
9.      xlnx,num-intr-inputs = <0x1>;
10. };
```

回头看一下 gpio 子系统的章节，那时候讲 gpio 的设备树时，就已经出现了这几个中断相关的属性。

第 2 行的"#interrupt-cells"是中断控制器节点的属性，用来描述子节点中"interrupts"属性值的数量。一般父节点的"#interrupt-cells"值为 3，则子节点的"interrupts"一个 cell 的三个 32 位整数的值为<中断域 中断号 触发方式>，如果父节点的该属性是 2，则是<中断号 触发方式>。

第 4 行的属性"interrupt-controller"代表这个节点是一个中断控制器。

第 5 行的"interrupt-parent"属性表明这个设备属于哪个中断控制器，如果没有这个属性会自动依附于父节点的"interrupt-parent"。

第 6 行的"interrupts"，第一个值为 0 表示 SPI 中断，1 表示 PPI 中断。在 zynq 中，第一个值如果是 0，则中断号等于第二个值加 32。

第 8 行的"xlnx,kind-of-intr"表示为每个可能的中断指定中断类型，1 表示 edge，0 表示 level。

第 9 行"xlnx,num-intr-inputs"属性指定控制器的特定实现支持的中断数，范围是 1~32。

我们需要从设备树中获取设备号信息，以向内核注册中断，of 函数中有对应的函数：

```
unsigned int irq_of_parse_and_map(struct device_node *dev, int index);
```

dev 是设备节点

index 是对属性"interrupts"元素的索引，因为中断号的位置有可能不同。

返回值就是中断号。

要使用这个函数的话，需要我们在对应的设备中设置好"interrupts"属性。

对于 gpio，内核提供了更方便的函数获取中断号：

```
int  gpio_to_irq(unsigned  int  gpio);
```

gpio 为需要申请中断号的 gpio 编号。

返回值就是中断号。

zynq 下 gpio 是共享的一个中断，针对单个 io 去设置"interrupts"属性比较麻烦，gpio_to_irq 函数帮我们做了很多事，后面的 gpio 中断实验，我们就直接使用这个函数。

8.2 实验

这章写一个通过按键中断驱动，按下按键触发中断，触发中断后，在中断服务函数中开

启一个 50ms 的定时器来实现按键去抖。

8.2.1 原理图

led 部分和章节 1.3.1 相同。
key 部分和章节 6.1 相同。

8.2.2 设备树

和章节 6.2 相同。

8.2.3 驱动程序

使用 petalinux 新建名为”ax-irq-drv”的驱励程序，并执行 petalinux-config -c rootfs 命令选上新增的驱动程序。

在 ax- irq-drv.c 文件中输入下面的代码：

```
1.  #include <linux/module.h>
2.  #include <linux/kernel.h>
3.  #include <linux/init.h>
4.  #include <linux/ide.h>
5.  #include <linux/types.h>
6.  #include <linux/errno.h>
7.  #include <linux/cdev.h>
8.  #include <linux/of.h>
9.  #include <linux/of_address.h>
10. #include <linux/of_gpio.h>
11. #include <linux/device.h>
12. #include <linux/delay.h>
13. #include <linux/init.h>
14. #include <linux/gpio.h>
15. #include <linux/semaphore.h>
16. #include <linux/timer.h>
17. #include <linux/of_irq.h>
18. #include <linux/irq.h>
19. #include <asm/uaccess.h>
20. #include <asm/mach/map.h>
21. #include <asm/io.h>
22.
23. /* 设备节点名称 */
24. #define DEVICE_NAME       "interrupt_led"
25. /* 设备号个数 */
26. #define DEVID_COUNT       1
```

```
/* 驱动个数 */
#define DRIVE_COUNT       1
/* 主设备号 */
#define MAJOR_U
/* 次设备号 */
#define MINOR_U           0

/* 把驱动代码中会用到的数据打包进设备结构体 */
struct alinx_char_dev {
/** 字符设备框架 **/
    dev_t               devid;              //设备号
    struct cdev         cdev;               //字符设备
    struct class        *class;             //类
    struct device       *device;            //设备
    struct device_node *nd;                 //设备树的设备节点
/** 并发处理 **/
    spinlock_t          lock;               //自旋锁变量
/** gpio **/
    int                 alinx_key_gpio;     //gpio 号
    int                 key_sts;            //记录按键状态，为 1 时被按下
/** 中断 **/
    unsigned int        irq;                //中断号
/** 定时器 **/
    struct timer_list   timer;              //定时器
};
/* 声明设备结构体 */
static struct alinx_char_dev alinx_char = {
    .cdev = {
        .owner = THIS_MODULE,
    },
};

/** 回掉 **/
/* 中断服务函数 */
static irqreturn_t key_handler(int irq, void *dev)
{
    /* 按键按下或抬起时会进入中断 */
    /* 开启 50 毫秒的定时器用作防抖动 */
    mod_timer(&alinx_char.timer, jiffies + msecs_to_jiffies(50));
    return IRQ_RETVAL(IRQ_HANDLED);
}

/* 定时器服务函数 */
void timer_function(unsigned long arg)
```

```
{
    unsigned long flags;
    /* 获取锁 */
    spin_lock_irqsave(&alinx_char.lock, flags);

    /* value 用于获取按键值 */
    unsigned char value;
    /* 获取按键值 */
    value = gpio_get_value(alinx_char.alinx_key_gpio);
    if(value == 0)
    {
        /* 按键按下，状态置 1 */
        alinx_char.key_sts = 1;
    }
    else
    {
        /* 按键抬起 */
    }

    /* 释放锁 */
    spin_unlock_irqrestore(&alinx_char.lock, flags);
}

/** 系统调用实现 **/
/* open 函数实现，对应到 Linux 系统调用函数的 open 函数 */
static int char_drv_open(struct inode *inode_p, struct file *file_p)
{
    printk("gpio_test module open\n");
    return 0;
}


/* read 函数实现，对应到 Linux 系统调用函数的 write 函数 */
static ssize_t char_drv_read(struct file *file_p, char __user *buf, size_t len, loff_t *lof
f_t_p)
{
    unsigned long flags;
    int ret;
    /* 获取锁 */
    spin_lock_irqsave(&alinx_char.lock, flags);

    /* keysts 用于读取按键状态 */
    /* 返回按键状态值 */
    ret = copy_to_user(buf, &alinx_char.key_sts, sizeof(alinx_char.key_sts));
```

```
114.     /* 清除按键状态 */
115.     alinx_char.key_sts = 0;
116. 
117.     /* 释放锁 */
118.     spin_unlock_irqrestore(&alinx_char.lock, flags);
119.     return 0;
120. }
121. 
122. /* release 函数实现，对应到 Linux 系统调用函数的 close 函数 */
123. static int char_drv_release(struct inode *inode_p, struct file *file_p)
124. {
125.     printk("gpio_test module release\n");
126.     return 0;
127. }
128. 
129. /* file_operations 结构体声明，是上面 open、write 实现函数与系统调用函数对应的关键 */
130. static struct file_operations ax_char_fops = {
131.     .owner   = THIS_MODULE,
132.     .open    = char_drv_open,
133.     .read    = char_drv_read,
134.     .release = char_drv_release,
135. };
136. 
137. /* 模块加载时会调用的函数 */
138. static int __init char_drv_init(void)
139. {
140.     /* 用于接受返回值 */
141.     u32 ret = 0;
142. 
143. /** 并发处理 **/
144.     /* 初始化自旋锁 */
145.     spin_lock_init(&alinx_char.lock);
146. 
147. /** gpio 框架 **/
148.     /* 获取设备节点 */
149.     alinx_char.nd = of_find_node_by_path("/alinxkey");
150.     if(alinx_char.nd == NULL)
151.     {
152.         printk("alinx_char node not find\r\n");
153.         return -EINVAL;
154.     }
155.     else
156.     {
157.         printk("alinx_char node find\r\n");
```

```
158.     }
159.
160.     /* 获取节点中 gpio 标号 */
161.     alinx_char.alinx_key_gpio = of_get_named_gpio(alinx_char.nd, "alinxkey-gpios", 0);
162.     if(alinx_char.alinx_key_gpio < 0)
163.     {
164.         printk("can not get alinxkey-gpios");
165.         return -EINVAL;
166.     }
167.     printk("alinxkey-gpio num = %d\r\n", alinx_char.alinx_key_gpio);
168.
169.     /* 申请 gpio 标号对应的引脚 */
170.     ret = gpio_request(alinx_char.alinx_key_gpio, "alinxkey");
171.     if(ret != 0)
172.     {
173.         printk("can not request gpio\r\n");
174.         return -EINVAL;
175.     }
176.
177.     /* 把这个 io 设置为输入 */
178.     ret = gpio_direction_input(alinx_char.alinx_key_gpio);
179.     if(ret < 0)
180.     {
181.         printk("can not set gpio\r\n");
182.         return -EINVAL;
183.     }
184.
185. /** 中断 **/
186.     /* 获取中断号 */
187.     alinx_char.irq = gpio_to_irq(alinx_char.alinx_key_gpio);
188.     /* 申请中断 */
189.     ret = request_irq(alinx_char.irq,
190.                       key_handler,
191.                       IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING,
192.                       "alinxkey",
193.                       NULL);
194.     if(ret < 0)
195.     {
196.         printk("irq %d request failed\r\n", alinx_char.irq);
197.         return -EFAULT;
198.     }
199.
200. /** 定时器 **/
201.     alinx_char.timer.function = timer_function;
```

```
202.     init_timer(&alinx_char.timer);
203. 
204. /** 字符设备框架 **/
205.     /* 注册设备号 */
206.     alloc_chrdev_region(&alinx_char.devid, MINOR_U, DEVID_COUNT, DEVICE_NAME);
207. 
208.     /* 初始化字符设备结构体 */
209.     cdev_init(&alinx_char.cdev, &ax_char_fops);
210. 
211.     /* 注册字符设备 */
212.     cdev_add(&alinx_char.cdev, alinx_char.devid, DRIVE_COUNT);
213. 
214.     /* 创建类 */
215.     alinx_char.class = class_create(THIS_MODULE, DEVICE_NAME);
216.     if(IS_ERR(alinx_char.class))
217.     {
218.         return PTR_ERR(alinx_char.class);
219.     }
220. 
221.     /* 创建设备节点 */
222.     alinx_char.device = device_create(alinx_char.class, NULL,
223.                                       alinx_char.devid, NULL,
224.                                       DEVICE_NAME);
225.     if (IS_ERR(alinx_char.device))
226.     {
227.         return PTR_ERR(alinx_char.device);
228.     }
229. 
230.     return 0;
231. }
232. 
233. /* 卸载模块 */
234. static void __exit char_drv_exit(void)
235. {
236. /** gpio **/
237.     /* 释放 gpio */
238.     gpio_free(alinx_char.alinx_key_gpio);
239. 
240. /** 中断 **/
241.     /* 释放中断 */
242.     free_irq(alinx_char.irq, NULL);
243. 
244. /** 定时器 **/
245.     /* 删除定时器 */
```

```
246.     del_timer_sync(&alinx_char.timer);
247. 
248. /** 字符设备框架 **/
249.     /* 注销字符设备 */
250.     cdev_del(&alinx_char.cdev);
251. 
252.     /* 注销设备号 */
253.     unregister_chrdev_region(alinx_char.devid, DEVID_COUNT);
254. 
255.     /* 删除设备节点 */
256.     device_destroy(alinx_char.class, alinx_char.devid);
257. 
258.     /* 删除类 */
259.     class_destroy(alinx_char.class);
260. 
261.     printk("timer_led_dev_exit_ok\n");
262. }
263. 
264. /* 标记加载、卸载函数 */
265. module_init(char_drv_init);
266. module_exit(char_drv_exit);
267. 
268. /* 驱动描述信息 */
269. MODULE_AUTHOR("Alinx");
270. MODULE_ALIAS("alinx char");
271. MODULE_DESCRIPTION("INTERRUPT LED driver");
272. MODULE_VERSION("v1.0");
273. MODULE_LICENSE("GPL");
```

197 行在驱动入口函数中，初始化 gpio 之后，使用 gpio_to_irq 函数通过 gpio 端口号来获取中断号。

199 行通过中断号向内核申请中断。上升沿或下降沿触发，命名为”alinxkey”，中断服务函数为 key_handler。

对照前面说的中断步骤，现在我们只要实现 key_handler 这个函数就可以了。

71 行实现了 key_handler，内容很简单先是开启了一个 50ms 的 timer，之后返回 IRQ_RETVAL(IRQ_HANDLED)就行了。

252 行驱动出口函数中把注册的中断号释放。

关于自旋锁保护的对象，实际上就是 alinx_char.key_sts 这个值，因为这个值在读函数中操作了，在中断开启定时器回掉函数中也操作了，这两个操作是有可能同时发生的，因此需要保护。

8.2.4 测试程序

和第六章的测试程序相同。

8.2.5 运行测试

测试目标是用板子上的 ps key1 去控制 ps led1，测试步骤如下：

```
mount -t nfs -o nolock 192.168.1.107:/home/alinx/work /mnt
cd /mnt
mkdir /tmp/qt
mount qt_lib.img /tmp/qt
cd /tmp/qt
source ./qt_env_set.sh
cd /mnt
insmod ./ax-concled-dev.ko
insmod ./ax_irq_dev.ko
cd ./build-ax-key-test-ZYNQ-Debug
./ax-key-test /dev/interrupt_led
```

IP 和路径根据实际情况调整。测试现象也与第 6 章相同。