第三十一章 Linux 中断实验

不管是单片机裸机实验还是 Linux 下的驱动实验，中断都是频繁使用的功能，在裸机中使用中断我们需要做一大堆的工作，比如配置寄存器，使能 IRQ 等等。但是 Linux 内核提供了完善的中断框架，我们只需要申请中断，然后注册中断处理函数即可，使用非常方便，不需要一系列复杂的寄存器配置。本章我们就来学习一下如何在 Linux 下使用中断。

31.1 Linux 中断简介

31.1.1 Linux 中断 API 函数

先来回顾一下裸机实验里面中断的处理方法：

①、使能中断，初始化相应的寄存器。

②、注册中断服务函数，也就是向 irqTable 数组的指定标号处写入中断服务函数

②、中断发生以后进入 IRQ 中断服务函数，在 IRQ 中断服务函数在数组 irqTable 里面查找具体的中断处理函数，找到以后执行相应的中断处理函数。

在 Linux 内核中也提供了大量的中断相关的 API 函数，我们来看一下这些跟中断有关的 API 函数：

1、中断号

每个中断都有一个中断号，通过中断号即可区分不同的中断，有的资料也把中断号叫做中断线。在 Linux 内核中使用一个 int 变量表示中断号，关于中断号我们已经在第十七章讲解过了。

2、request_irq 函数

在 Linux 内核中要想使用某个中断是需要申请的，request_irq 函数用于申请中断，request_irq 函数可能会导致睡眠，因此不能在中断上下文或者其他禁止睡眠的代码段中使用 request_irq 函数。request_irq 函数会激活(使能)中断，所以不需要我们手动去使能中断，request_irq 函数原型如下：

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

函数参数和返回值含义如下：

irq：要申请中断的中断号。

handler：中断处理函数，当中断发生以后就会执行此中断处理函数。

flags：中断标志，可以在文件 include/linux/interrupt.h 里面查看所有的中断标志，这里我们介绍几个常用的中断标志，如表 31.1.1.1 所示：

标志	描述
IRQF_SHARED	多个设备共享一个中断线，共享的所有中断都必须指定此标志。如果使用共享中断的话，request_irq 函数的 dev 参数就是唯一区分他们的标志。
IRQF_ONESHOT	单次中断，中断执行一次就结束。
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。

表 31.1.1.1 常用的中断标志

比如 STM32MP157 开发板上的 KEY0 使用 PG3，按下 KEY0 以后为低电平，因此可以设置为下降沿触发，也就是将 flags 设置为 IRQF_TRIGGER_FALLING。表 31.1.1.1 中的这些标志可以通过“|”来实现多种组合。

name：中断名字，设置以后可以在/proc/interrupts 文件中看到对应的中断名字。

dev：如果将 flags 设置为 IRQF_SHARED 的话，dev 用来区分不同的中断，一般情况下将 dev 设置为设备结构体，dev 会传递给中断处理函数 irq_handler_t 的第二个参数。

返回值：0 中断申请成功，其他负值 中断申请失败，如果返回-EBUSY 的话表示中断已经被申请了。

3、free_irq 函数

使用中断的时候需要通过 request_irq 函数申请，使用完成以后就要通过 free_irq 函数释放掉相应的中断。如果中断不是共享的，那么 free_irq 会删除中断处理函数并且禁止中断。free_irq 函数原型如下所示：

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

函数参数和返回值含义如下：

irq：要释放的中断。

dev：如果中断设置为共享(IRQF_SHARED)的话，此参数用来区分具体的中断。共享中断只有在释放最后中断处理函数的时候才会被禁止掉。

返回值：无。

4、中断处理函数

使用 request_irq 函数申请中断的时候需要设置中断处理函数，中断处理函数格式如下所示：

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

第一个参数是要中断处理函数要相应的中断号。第二个参数是一个指向 void 的指针，也就是个通用指针，需要与 request_irq 函数的 dev 参数保持一致。用于区分共享中断的不同设备，dev 也可以指向设备数据结构。中断处理函数的返回值为 irqreturn_t 类型，irqreturn_t 类型定义如下所示：

示例代码 31.1.1.1 irqreturn_t 结构

```
11 enum irqreturn {
12     IRQ_NONE          = (0 << 0),
13     IRQ_HANDLED       = (1 << 0),
14     IRQ_WAKE_THREAD   = (1 << 1),
15 };
16
17 typedef enum irqreturn irqreturn_t;
```

可以看出 irqreturn_t 是个枚举类型，一共有三种返回值。一般中断服务函数返回值使用如下形式：

```
return IRQ_RETVAL(IRQ_HANDLED)
```

5、中断使能与禁止函数

常用的中断使用和禁止函数如下所示：

```
void enable_irq(unsigned int irq)
void disable_irq(unsigned int irq)
```

enable_irq 和 disable_irq 用于使能和禁止指定的中断，irq 就是要禁止的中断号。

disable_irq 函数要等到当前正在执行的中断处理函数执行完才返回，因此使用者需要保证不会产生新的中断，并且确保所有已经开始执行的中断处理程序已经全部退出。在这种情况下，可以使用另外一个中断禁止函数：

```
void disable_irq_nosync(unsigned int irq)
```

disable_irq_nosync 函数调用以后立即返回，不会等待当前中断处理程序执行完毕。上面三个函数都是使能或者禁止某一个中断，有时候我们需要关闭当前处理器的整个中断系统，也就是在学习 STM32 的时候常说的关闭全局中断，这个时候可以使用如下两个函数：

```
local_irq_enable()
local_irq_disable()
```

local_irq_enable 用于使能当前处理器中断系统，local_irq_disable 用于禁止当前处理器中断系统。假如 A 任务调用 local_irq_disable 关闭全局中断 10S，当关闭了 2S 的时候 B 任务开始运行，B 任务也调用 local_irq_disable 关闭全局中断 3S，3 秒以后 B 任务调用 local_irq_enable 函数将全局中断打开了。此时才过去 2+3=5 秒的时间，然后全局中断就被打开了，此时 A 任务要关闭 10S 全局中断的愿望就破灭了，然后 A 任务就“生气了”，结果很严重，可能系统都要被 A 任务整崩溃。为了解决这个问题，B 任务不能直接简单粗暴的通过 local_irq_enable 函数来打开全局中断，而是将中断状态恢复到以前的状态，要考虑到别的任务的感受，此时就要用到下面两个函数：

```
local_irq_save(flags)
local_irq_restore(flags)
```

这两个函数是一对，local_irq_save 函数用于禁止中断，并且将中断状态保存在 flags 中。local_irq_restore 用于恢复中断，将中断到 flags 状态。

31.1.2 上半部与下半部

在有些资料中也将上半部和下半部称为顶半部和底半部，都是一个意思。我们在使用 request_irq 申请中断的时候注册的中断服务函数属于中断处理的上半部，只要中断触发，那么中断处理函数就会执行。我们都知道中断处理函数一定要快点执行完毕，越短越好，但是现实往往是残酷的，有些中断处理过程就是比较费时间，我们必须要对其进行处理，缩小中断处理函数的执行时间。比如电容触摸屏通过中断通知 SOC 有触摸事件发生，SOC 响应中断，然后通过 IIC 接口读取触摸坐标值并将其上报给系统。但是我们都知道 IIC 的速度最高也只有 400Kbit/S，所以在中断中通过 IIC 读取数据就会浪费时间。我们可以将通过 IIC 读取触摸数据的操作暂后执行，中断处理函数仅仅相应中断，然后清除中断标志位即可。这个时候中断处理过程就分为了两部分：

上半部：上半部就是中断处理函数，那些处理过程比较快，不会占用很长时间的处理就可以放在上半部完成。

下半部：如果中断处理过程比较耗时，那么就将这些比较耗时的代码提出来，交给下半部去执行，这样中断处理函数就会快进快出。

因此，Linux 内核将中断分为上半部和下半部的主要目的就是实现中断处理函数的快进快出，那些对时间敏感、执行速度快的操作可以放到中断处理函数中，也就是上半部。剩下的所有工作都可以放到下半部去执行，比如在上半部将数据拷贝到内存中，关于数据的具体处理就可以放到下半部去执行。至于哪些代码属于上半部，哪些代码属于下半部并没有明确的规定，一切根据实际使用情况去判断，这个就很考验驱动编写人员的功底了。这里有一些可以借鉴的参考点：

①、如果要处理的内容不希望被其他中断打断，那么可以放到上半部。

②、如果要处理的任务对时间敏感，可以放到上半部。

③、如果要处理的任务与硬件有关，可以放到上半部

④、除了上述三点以外的其他任务，优先考虑放到下半部。

上半部处理很简单，直接编写中断处理函数就行了，关键是下半部该怎么做呢？Linux 内核提供了多种下半部机制，接下来我们来学习一下这些下半部机制。

1、软中断

一开始 Linux 内核提供了“bottom half”机制来实现下半部，简称“BH”。后面引入了软中断和 tasklet 来替代“BH”机制，完全可以使用软中断和 tasklet 来替代 BH，从 2.5 版本的 Linux 内核开始 BH 已经被抛弃了。Linux 内核使用结构体 softirq_action 表示软中断，softirq_action 结构体定义在文件 include/linux/interrupt.h 中，内容如下：

示例代码 31.1.2.1 softirq_action 结构体

```
541 struct softirq_action
542 {
543     void    (*action)(struct softirq_action *);
544 };
```

在 kernel/softirq.c 文件中一共定义了 10 个软中断，如下所示：

示例代码 31.1.2.2 softirq_vec 数组

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

NR_SOFTIRQS 是枚举类型，定义在文件 include/linux/interrupt.h 中，定义如下：

示例代码 31.1.2.3 softirq_vec 数组

```
enum
{
    HI_SOFTIRQ=0,               /* 高优先级软中断        */
    TIMER_SOFTIRQ,              /* 定时器软中断          */
    NET_TX_SOFTIRQ,             /* 网络数据发送软中断    */
    NET_RX_SOFTIRQ,             /* 网络数据接收软中断    */
    BLOCK_SOFTIRQ,
    IRQ_POLL_SOFTIRQ,
    TASKLET_SOFTIRQ,            /* tasklet 软中断        */
    SCHED_SOFTIRQ,              /* 调度软中断            */
    HRTIMER_SOFTIRQ,            /* 高精度定时器软中断    */
    RCU_SOFTIRQ,                /* RCU 软中断            */

    NR_SOFTIRQS
};
```

可以看出，一共有 10 个软中断，因此 NR_SOFTIRQS 为 10，因此数组 softirq_vec 有 10 个元素。softirq_action 结构体中的 action 成员变量就是软中断的服务函数，数组 softirq_vec 是个全局数组，因此所有的 CPU(对于 SMP 系统而言)都可以访问到，每个 CPU 都有自己的触发和控制机制，并且只执行自己所触发的软中断。但是各个 CPU 所执行的软中断服务函数确是相同的，都是数组 softirq_vec 中定义的 action 函数。要使用软中断，必须先使用 open_softirq 函数注册对应的软中断处理函数，open_softirq 函数原型如下：

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

函数参数和返回值含义如下：

nr：要开启的软中断，在示例代码 31.1.2.3 中选择要开启的软中断。

action：软中断对应的处理函数。

返回值：没有返回值。

注册好软中断以后需要通过 raise_softirq 函数触发，raise_softirq 函数原型如下：

```
void raise_softirq(unsigned int nr)
```

函数参数和返回值含义如下：

nr：要触发的软中断，在示例代码 31.1.2.3 中选择要注册的软中断。

返回值：没有返回值。

软中断必须在编译的时候静态注册！Linux 内核使用 softirq_init 函数初始化软中断，softirq_init 函数定义在 kernel/softirq.c 文件里面，函数内容如下：

示例代码 31.1.2.4 softirq_init 函数内容

```
575 void __init softirq_init(void)
576 {
577     int cpu;
578 
579     for_each_possible_cpu(cpu) {
580         per_cpu(tasklet_vec, cpu).tail =
581             &per_cpu(tasklet_vec, cpu).head;
582         per_cpu(tasklet_hi_vec, cpu).tail =
583             &per_cpu(tasklet_hi_vec, cpu).head;
584     }
585 
586     open_softirq(TASKLET_SOFTIRQ, tasklet_action);
587     open_softirq(HI_SOFTIRQ, tasklet_hi_action);
588 }
```

从示例代码 31.1.2.4 可以看出，softirq_init 函数默认会打开 TASKLET_SOFTIRQ 和 HI_SOFTIRQ。

2、tasklet

tasklet 是利用软中断来实现的另外一种下半部机制，在软中断和 tasklet 之间，建议大家使用 tasklet。tasklet_struct 结构体如下所示：

示例代码 31.1.2.5 tasklet_struct 结构体

```
592 struct tasklet_struct
593 {
594     struct tasklet_struct *next;    /* 下一个 tasklet          */
595     unsigned long state;            /* tasklet 状态            */
596     atomic_t count;                 /* 计数器,记录对 tasklet 的引用数 */
597     void (*func)(unsigned long);    /* tasklet 执行的函数      */
598     unsigned long data;             /* 函数 func 的参数        */
599 };
```

第 597 行的 func 函数就是 tasklet 要执行的处理函数，用户实现具体的函数内容，相当于中断处理函数。如果要使用 tasklet，必须先定义一个 tasklet_struct 变量，然后使用 tasklet_init 函数对其进行初始化，taskled_init 函数原型如下：

```
void tasklet_init(struct tasklet_struct     *t,
                  void (*func)(unsigned long),
                  unsigned long  data);
```

函数参数和返回值含义如下：

t：要初始化的 tasklet

func：tasklet 的处理函数。

data：要传递给 func 函数的参数

返回值：没有返回值。

也可以使用宏 DECLARE_TASKLET 来一次性完成 tasklet 的定义和初始化，DECLARE_TASKLET 定义在 include/linux/interrupt.h 文件中，定义如下:

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

其中 name 为要定义的 tasklet 名字，其实就是 tasklet_struct 类型的变量名，func 就是 tasklet 的处理函数，data 是传递给 func 函数的参数。

在上半部，也就是中断处理函数中调用 tasklet_schedule 函数就能使 tasklet 在合适的时间运行，tasklet_schedule 函数原型如下：

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

函数参数和返回值含义如下：

t：要调度的 tasklet，也就是 DECLARE_TASKLET 宏里面的 name。

返回值：没有返回值。

关于 tasklet 的参考使用示例如下所示：

示例代码 31.1.2.7 tasklet 使用示例

```
/* 定义 taselet                  */
struct tasklet_struct testtasklet;

/* 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(&testtasklet);
    ......
}

/* 驱动入口函数                  */
static int __init xxxx_init(void)
{
    ......
    /* 初始化 tasklet            */
    tasklet_init(&testtasklet, testtasklet_func, data);
    /* 注册中断处理函数          */
    request_irq(xxx_irq, test_handler, 0,       , &xxx_dev);
    ......
```

```
}
```

2、工作队列

工作队列是另外一种下半部执行方式，工作队列在进程上下文执行，工作队列将要推后的工作交给一个内核线程去执行，因为工作队列工作在进程上下文，因此工作队列允许睡眠或重新调度。因此如果你要推后的工作可以睡眠那么就可以选择工作队列，否则的话就只能选择软中断或 tasklet。

Linux 内核使用 work_struct 结构体表示一个工作，内容如下(省略掉条件编译)：

示例代码 31.1.2.8 work_struct 结构体

```
struct work_struct {
    atomic_long_t data;
    struct list_head entry;
    work_func_t func;        /* 工作队列处理函数 */
};
```

这些工作组织成工作队列，工作队列使用 workqueue_struct 结构体表示，内容如下(省略掉条件编译)：

示例代码 31.1.2.9 workqueue_struct 结构体

```
struct workqueue_struct {
    struct list_head     pwqs;
    struct list_head     list;
    struct mutex         mutex;
    int                  work_color;
    int                  flush_color;
    atomic_t             nr_pwqs_to_flush;
    struct wq_flusher    *first_flusher;
    struct list_head     flusher_queue;
    struct list_head     flusher_overflow;
    struct list_head     maydays;
    struct worker        *rescuer;
    int                  nr_drainers;
    int                  saved_max_active;
    struct workqueue_attrs  *unbound_attrs;
    struct pool_workqueue   *dfl_pwq;
    char                 name[WQ_NAME_LEN];
    struct rcu_head      rcu;
    unsigned int         flags ____cacheline_aligned;
    struct pool_workqueue __percpu *cpu_pwqs;
    struct pool_workqueue __rcu *numa_pwq_tbl[];
};
```

Linux 内核使用工作者线程(worker thread)来处理工作队列中的各个工作，Linux 内核使用 worker 结构体表示工作者线程，worker 结构体内容如下：

示例代码 31.1.2.10 worker 结构体

```
struct worker {
    union {
        struct list_head     entry;
```

```
        struct hlist_node     hentry;
    };

    struct work_struct        *current_work;
    work_func_t               current_func;
    struct pool_workqueue     *current_pwq;
    struct list_head          scheduled;
    struct task_struct        *task;
    struct worker_pool        *pool;
    struct list_head          node;
    unsigned long             last_active;
    unsigned int              flags;
    int                       id;
    int                       sleeping;
    char                      desc[WORKER_DESC_LEN];
    struct workqueue_struct   *rescue_wq;
    work_func_t               last_func;
};
```

从示例代码 31.1.2.10 可以看出，每个 worker 都有一个工作队列，工作者线程处理自己工作队列中的所有工作。在实际的驱动开发中，我们只需要定义工作(work_struct)即可，关于工作队列和工作者线程我们基本不用去管。简单创建工作很简单，直接定义一个 work_struct 结构体变量即可，然后使用 INIT_WORK 宏来初始化工作，INIT_WORK 宏定义如下：

```
#define INIT_WORK(_work, _func)
```

_work 表示要初始化的工作，_func 是工作对应的处理函数。

也可以使用 DECLARE_WORK 宏一次性完成工作的创建和初始化，宏定义如下：

```
#define DECLARE_WORK(n, f)
```

n 表示定义的工作(work_struct)，f 表示工作对应的处理函数。

和 tasklet 一样，工作也是需要调度才能运行的，工作的调度函数为 schedule_work，函数原型如下所示：

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

函数参数和返回值含义如下：

work：要调度的工作。

返回值：0 成功，其他值 失败。

关于工作队列的参考使用示例如下所示：

示例代码 31.1.2.11 工作队列使用示例

```
/* 定义工作(work)          */
struct  work_struct testwork;

/* work 处理函数           */
void testwork_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(&testwork);
    ......
}

/* 驱动入口函数                */
static int __init xxxx_init(void)
{
    ......
    /* 初始化 work             */
    INIT_WORK(&testwork, testwork_func_t);
    /* 注册中断处理函数        */
    request_irq(xxx_irq, test_handler, 0,       , &xxx_dev);
    ......
}
```

31.1.3 设备树中断信息节点

1、GIC 中断控制器

STM32MP1 有三个与中断有关的控制器：GIC、EXTI 和 NVIC，其中 NVIC 是 Cortex-M4 内核的中断控制器，本教程只讲解 Cortex-A7 内核，因此就只剩下了 GIC 和 EXTI。首先是 GIC，全称为：Generic Interrupt Controller，关于 GIC 的详细内容可以查看文档《ARM Generic Interrupt Controller(ARM GIC 控制器)V2.0》，此文档已经放到了开发板光盘中，路径为：开发板光盘→ 4、参考资料→ ARM Generic Interrupt Controller(ARM GIC 控制器)V2.0.pdf。

GIC 是 ARM 公司给 Cortex-A/R 内核提供的一个中断控制器，类似 Cortex-M 内核中的 NVIC。目前 GIC 有 4 个版本:V1~V4，V1 是最老的版本，已经被废弃了。V2~V4 目前正在大量的使用。GIC V2 是给 ARMv7-A 架构使用的，比如 Cortex-A7、Cortex-A9、Cortex-A15 等，V3 和 V4 是给 ARMv8-A/R 架构使用的，也就是 64 位芯片使用的。STM32MP1 是 Cortex-A7 内核，因此我们主要讲解 GIC V2。GIC V2 最多支持 8 个核。ARM 会根据 GIC 版本的不同研发出不同的 IP 核，那些半导体厂商直接购买对应的 IP 核即可，比如 ARM 针对 GIC V2 就开发出了 GIC400 这个中断控制器 IP 核。当 GIC 接收到外部中断信号以后就会报给 ARM 内核，但是 ARM 内核只提供了四个信号给 GIC 来汇报中断情况：VFIQ、VIRQ、FIQ 和 IRQ，他们之间的关系如图 31.1.3.1 所示：

图 31.1.3.1 中断示意图

在图 31.1.3.1 中，GIC 接收众多的外部中断，然后对其进行处理，最终就只通过四个信号报给 ARM 内核，这四个信号的含义如下：

VFIQ:虚拟快速 FIQ。

VIRQ:虚拟快速 IRQ。

FIQ:快速中断 IRQ。

IRQ:外部中断 IRQ。

VFIQ 和 VIRQ 是针对虚拟化的，我们讨论不虚拟化，剩下的就是 FIQ 和 IRQ 了，本教程我们只使用 IRQ。所以相当于 GIC 最终向 ARM 内核就上报一个 IRQ 信号。那么 GIC 是如何完成这个工作的呢？GIC V2 的逻辑图如图 31.1.3.2 所示：

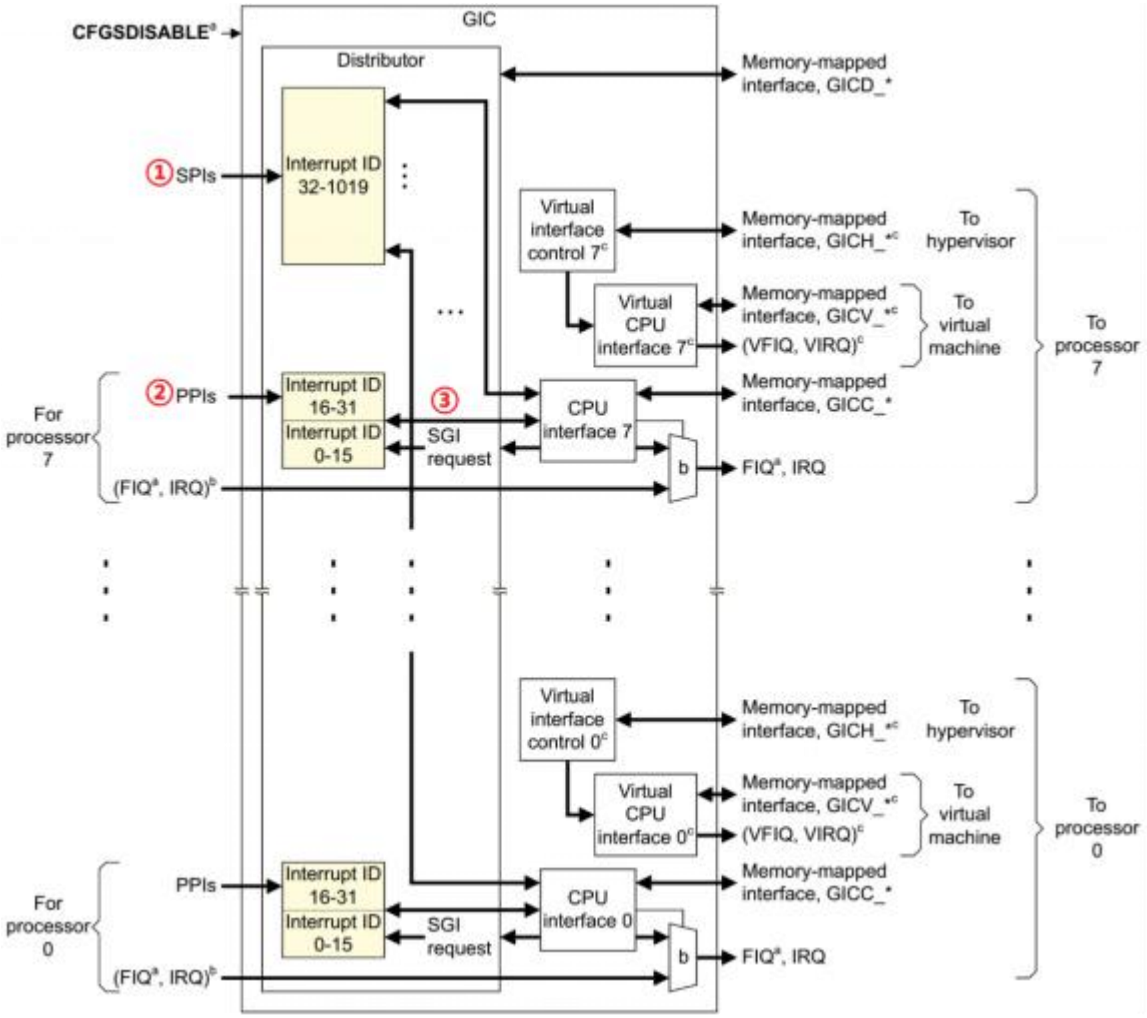

图 31.1.3.2 GIC V2 总体框图

图 31.1.3.2 中左侧部分就是中断源，中间部分就是 GIC 控制器，最右侧就是中断控制器向处理器内核发送中断信息。我们重点要看的肯定是中间的 GIC 部分，GIC 将众多的中断源分为分为三类：

①、SPI(Shared Peripheral Interrupt),共享中断，顾名思义，所有 Core 共享的中断，这个是最常见的，那些外部中断都属于 SPI 中断(注意！不是 SPI 总线那个中断) 。比如 GPIO 中断、串口中断等等，这些中断所有的 Core 都可以处理，不限定特定 Core。

②、PPI(Private Peripheral Interrupt)，私有中断，我们说了 GIC 是支持多核的，每个核肯定有自己独有的中断。这些独有的中断肯定是要指定的核心处理，因此这些中断就叫做私有中断。

③、SGI(Software-generated Interrupt)，软件中断，由软件触发引起的中断，通过向寄存器 GICD_SGIR 写入数据来触发，系统会使用 SGI 中断来完成多核之间的通信。

2、中断 ID

中断源有很多，为了区分这些不同的中断源肯定要给他们分配一个唯一 ID，这些 ID 就是中断 ID。每一个 CPU 最多支持 1020 个中断 ID，中断 ID 号为 ID0~ID1019。这 1020 个 ID 包含了 PPI、SPI 和 SGI，那么这三类中断是如何分配这 1020 个中断 ID 的呢？这 1020 个 ID 分配如下：

ID0~ID15：这 16 个 ID 分配给 SGI。

ID16~ID31：这 16 个 ID 分配给 PPI。

ID32~ID1019：这 988 个 ID 分配给 SPI，像 GPIO 中断、串口中断等这些外部中断 ，至于具体到某个 ID 对应哪个中断那就由半导体厂商根据实际情况去定义了。比如 STM32MP1 系列总共分配了 265 个中断 ID(有很多并未使用，只是保留着)，加上前面属于 PPI 和 SGI 的 32 个 ID，STM32MP1 的中断源共有 256+32=288 个，这 288 个中断 ID 对应的中断源可以在《STM32MP157 参考手册》中找到详细的解释。找到“21.2　GIC Interrupts”小节的“Table 117. STM32MP157 interrupt mapping for Cortex®-A7 GIC (continued)”，我们重点关注的是从 ID32 开始的 SPI 中断，因为这些才是 STM32MP1 的外设中断，如图 31.1.3.3 所示(由于表太大，这里只是截取其中一部分)：

0	32	WWDG1_IT	Window watchdog 1 early wakeup interrupt	-
1	33	PVD_AVD	PVD & AVD detector through EXTI	16
2	34	TAMP	Tamper interrupt (include LSECSS interrupts)	(18)
3	35	RTC_WKUP_ALARM	RTC wakeup timer and alarms (A and B) interrupt	(19)
4	36	TZC_IT	TrustZone DDR address space controller	-
5	37	RCC	RCC global interrupt	-
6	38	EXTI0	EXTI line 0 interrupt	0
7	39	EXTI1	EXTI line 1 interrupt	1
8	40	EXTI2	EXTI line 2 interrupt	2
9	41	EXTI3	EXTI line 3 interrupt	3
10	42	EXTI4	EXTI line 4 interrupt	4
11	43	DMA1_STR0	DMA1 stream0 global interrupt	-
12	44	DMA1_STR1	DMA1 stream1 global interrupt	-
13	45	DMA1_STR2	DMA1 stream2 global interrupt	-
---	---	---	---	---
215	247	-	Reserved	-
216	248	nCTIIRQ0	Cortex®-A7 core#0 CTI interrupt	-
217	249	nCTIIRQ1	Cortex®-A7 core#1 CTI interrupt	-
210	242	-	Reserved	-
211	243	-	Reserved	-
212 to 255	244 to 287	-	Reserved	-

图 31.1.3.3 STM32MP1 中断源

关于 GIC 就先讲到这里，我们接下来讲解一下 EXTI。

3、EXTI- 外部中断和事件控制器

EXTI 是什么东西呢？全称是：Extended interrupt and event controller，中文一般叫做外部中断和事件控制器。EXTI 是 ST 自己设计的，用来辅助 GIC 管理 STM32MP1 相应中断的。EXTI 通过可配置的事件输入和直接事件输入来管理唤醒。它可以针对电源控制提供唤醒请求、针对 CPU 事件输入生成事件。EXTI 唤醒请求可让系统从停止模式唤醒，以及让 CPU 从 CSTOP 和 CSTANDBY 模式唤醒。此外，EXTI 还可以在运行模式下生成中断请求和事件请求，这个非常重要，因为在实际使用中 EXTI 主要是为 STM32 的 GPIO 中断服务的。

EXTI 主要特性如下：

• 支持 76 个输入事件

• 两个 CPU 内核都支持。

• 所有事件输入均可让 CPU 唤醒。

EXTI 的异步输入事件可以分为 2 组：

①、可配置事件（来自能够生成脉冲的 I/O 或外设的信号），这类事件具有以下特性：

- 可选择的有效触发边沿。
- 中断挂起状态寄存器位。
- 单独的中断和事件生成屏蔽。
- 支持软件触发。

②、直接事件（来自其他外设的中断和唤醒源，需要在外设中清除），这类事件具有以下

特性：

- 固定上升沿有效触发。
- EXTI 中无中断挂起状态寄存器位（中断挂起状态由生成事件的外设提供）。
- 单独的中断和事件生成屏蔽。
- 不支持软件触发。

对于 GPIO 中断来说，就是可配置事件，EXTI 和 GIC 的关系如图 31.1.3.4 所示：

图 31.1.3.4 EXTI 框架

从图 31.1.3.4 可以看出 STM32MP1 的中断处理方式有 5 种：

①、外设直接产生中断到 GIC，然后由 GIC 通知 CPU 内核。

②、GPIO 或外设产生中断到 EXTI，EXTI 将信号提交给 GIC，最终再将中断信号提交给 CPU。

③、GPIO 或外设产生中断到 EXTI，EXTI 直接将中断信号提交给 CPU。

Linux 系统会用到这三种中断方式，一个外设最多可以有两种中断方式，为啥是两种而不是三种后面会说。

STM32MP1 的所有 GPIO 都有中断功能，而 GPIO 中断是我们最常用的功能。STM32 每一组 GPIO 最多有 16 个 IO，比如 PA0~PA15，因此每组 GPIO 就有 16 个中断，这 16 个 GPIO 事件输入对应 EXTI0~15，其中 PA0、PB0 等都对应 EXTI0，如图 31.1.3.5 所示：

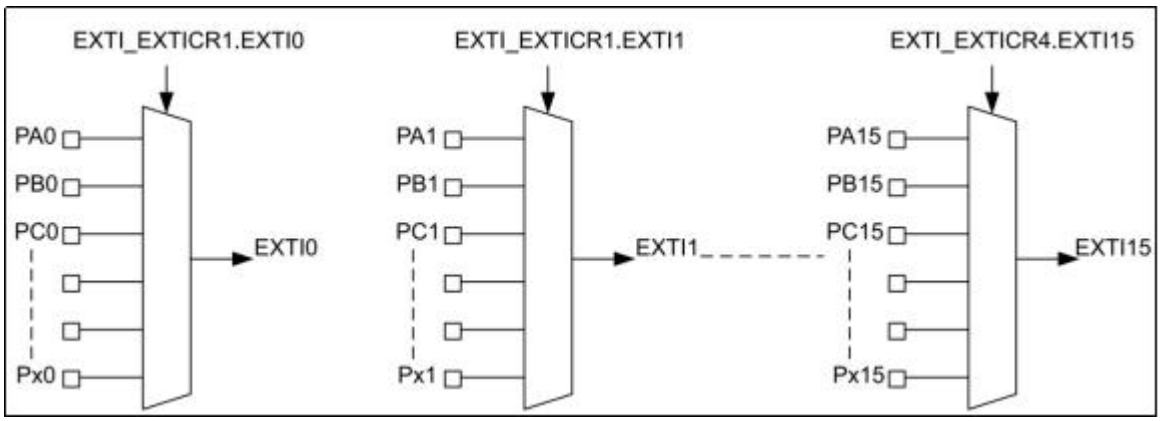

图 31.1.3.5 EXTI0~15 对应的 GPIO

如果要在 Linux 系统中使用中断，那么就需要在设备树中设置好中断属性信息，Linux 内核通过读取设备树中的中断属性信息来配置中断，GIC 控制器的设备树绑定信息参考文档 Documentation/devicetree/bindings/ interrupt-controller/arm,gic.yaml，EXTI 控制器的设备树绑定信息参考文档 Documentation/devicetree/bindings/interrupt-controller/st,stm32-exti.txt。

4、GIC 控制器节点

打开 stm32mp151.dtsi 文件，其中的 intc 节点就是 GIC 的中断控制器节点，节点内容如下所示：

示例代码 31.1.3.1 中断控制器 intc 节点

```
1  intc: interrupt-controller@a0021000 {
2      compatible = "arm,cortex-a7-gic";
3      #interrupt-cells = <3>;
4      interrupt-controller;
5      reg =  <0xa0021000 0x1000>,
6             <0xa0022000 0x2000>;
7  };
```

第 2 行，compatible 属性值为“arm,cortex-a7-gic”在 Linux 内核源码中搜索“arm,cortex-a7-gic”即可找到 GIC 中断控制器驱动文件。

第 3 行，#interrupt-cells 和#address-cells、#size-cells 一样。表示此中断控制器下设备的 cells 大小，对于设备而言，会使用 interrupts 属性描述中断信息，#interrupt-cells 描述了 interrupts 属性的 cells 大小，也就是一条信息有几个 cells。每个 cells 都是 32 位整形值，对于 ARM 处理的 GIC 来说，一共有 3 个 cells，这三个 cells 的含义如下：

第一个 cells：中断类型，0 表示 SPI 中断，1 表示 PPI 中断。

第二个 cells：中断号，对于 SPI 中断来说中断号的范围为 32~287(256 个)，对于 PPI 中断来说中断号的范围为 16~31，但是该 cell 描述的中断号是从 0 开始。

第三个 cells：标志，bit[3:0]表示中断触发类型，为 1 的时候表示上升沿触发，为 2 的时候表示下降沿触发，为 4 的时候表示高电平触发，为 8 的时候表示低电平触发。bit[15:8]为 PPI 中断的 CPU 掩码。

第 4 行，interrupt-controller 节点为空，表示当前节点是中断控制器。

我们来看一下 STM32MP1 的 SPI6 是如何在设备树节点中描述中断信息的，首先是查阅《STM32MP157 参考手册》第“21.2 GIC Interrupts”小节中的表 117，也就是图 31.1.3.3。找到 SPI6 对应的中断号，如图 31.1.3.6 所示：

Num	ID	Acronym	Description	EXTI event (1)
85	117	SPI5	SPI5 global interrupt	(40)
86	118	SPI6	SPI6 global interrupt	(41)

图 31.1.3.6 SPI6 中断

从图 31.1.3.6 可以看出，第一列的“Num”就是 SPI6 的中断号：86，注意这里并没有算前面 32 个中断号，如果加上前面 32 个中断号的话就是第二列“ID”，为：86+32=118。

打开 stm32mp151.dtsi，找到 SPI6 节点内容，如下所示：

示例代码 31.1.3.2 SPI6 节点

```
1   spi6: spi@5c001000 {
2       #address-cells = <1>;
3       #size-cells = <0>;
4       compatible = "st,stm32h7-spi";
5       reg = <0x5c001000 0x400>;
6       interrupts = <GIC_SPI 86 IRQ_TYPE_LEVEL_HIGH>;
7       clocks = <&scmi0_clk CK_SCMI0_SPI6>;
8       resets = <&scmi0_reset RST_SCMI0_SPI6>;
9       dmas = <&mdma1 34 0x0 0x40008 0x0 0x0 0x0>,
10            <&mdma1 35 0x0 0x40002 0x0 0x0 0x0>;
11      dma-names = "rx", "tx";
12      power-domains = <&pd_core>;
13      status = "disabled";
14  };
```

第 6 行，interrupts 描述中断中断源的信息，第一个表示中断类型，为 GIC_SPI，也就是共享中断。第二个表示中断号为 86，来源就是图 31.1.3.6。第三个表示中断触发类型是高电平触发。

2、EXTI 控制器节点

对于 GPIO 中断而言，要用到 EXTI，所以我们接下来看一下 EXTI 设备节点。打开 stm32mp151.dtsi 文件，其中的 exti 节点就是 EXTI 的中断控制器节点，节点内容如下所示：

示例代码 51.1.3.3 中断控制器 exti 节点

```
1   exti: interrupt-controller@5000d000 {
2       compatible = "st,stm32mp1-exti", "syscon";
3       interrupt-controller;
```

```
4       #interrupt-cells = <2>;
5       reg = <0x5000d000 0x400>;
6       hwlocks = <&hsem 1 1>;
7   };
```

第 3 行，表明 exti 节点是个中断控制器。

第 4 行，interrupt-cells=2，表明 exti 的子节点里面第一个 cell 表示为中断号，也可以叫 EXTI 事件编号，第二个 cell 表示中断标志位。其它的设备树属性和 GIC 控制器是一样的。

前面说了，GPIO 用到了 EXTI，因此 GPIO 节点信息里面的 EXTI 相关内容，在 stm32mp151.dtsi 文件中找到如下所示内容：

示例代码 31.1.3.4　GPIO 中断控制器节点

```
1   pinctrl: pin-controller@50002000 {
2       #address-cells = <1>;
3       #size-cells = <1>;
4       compatible =                        ;
5       ranges = <0 0x50002000 0xa400>;
6       interrupt-parent = <&exti>;
7       st,syscfg = <&exti 0x60 0xff>;
8       hwlocks = <&hsem 0 1>;
9       pins-are-numbered;
10
11      gpioa: gpio@50002000 {
12          gpio-controller;
13          #gpio-cells = <2>;
14          interrupt-controller;
15          #interrupt-cells = <2>;
16          reg = <0x0 0x400>;
17          clocks = <&rcc GPIOA>;
18          st,bank-name =          ;
19          status =            ;
20      };
......
120
121     gpiok: gpio@5000c000 {
122         gpio-controller;
123         #gpio-cells = <2>;
124         interrupt-controller;
125         #interrupt-cells = <2>;
126         reg = <0xa000 0x400>;
127         clocks = <&rcc GPIOK>;
128         st,bank-name =          ;
129         status =            ;
130     };
131 };
132
```

```
133 pinctrl_z: pin-controller-z@54004000 {
134     #address-cells = <1>;
135     #size-cells = <1>;
136     compatible =                          ;
137     ranges = <0 0x54004000 0x400>;
138     pins-are-numbered;
139     interrupt-parent = <&exti>;
140     st,syscfg = <&exti 0x60 0xff>;
141     hwlocks = <&hsem 0 1>;
142
143     gpioz: gpio@54004000 {
144         gpio-controller;
145         #gpio-cells = <2>;
146         interrupt-controller;
147         #interrupt-cells = <2>;
148         reg = <0 0x400>;
149         clocks = <&scmi0_clk CK_SCMI0_GPIOZ>;
150         st,bank-name =        ;
151         st,bank-ioport = <11>;
152         status =           ;
153     };
154 };
```

第 1~131 行，pinctrl 节点，此节点有 11 个子节点，gpioa~gpiok，分别对应 GPIOA~GPIOK 这 11 组 IO。第 8 行，通过 interrupt-parent 属性指定 pinctrl 所有子节点的中断父节点为 exti，这样 GPIO 的中断就和 EXTI 联系起来了。第 11~20 行为 gpioa 节点，第 14 行表明 gpioa 节点也是个中断控制器，第 15 行设置 interrupt-cells 为 2，那么对于具体的 GPIO 而言，interrupts 属性第一个 cell 为某个 IO 在所处组的编号，第二个 cell 表示中断触发方式。比如现在要设置 PA1 这个引脚为下降沿触发，那么就要 interrupts = <1 IRQ_TYPE_EDGE_FALLING>。

第 133~154 行，pinctrl_z 节点，由于 GPIOZ 这一组对应的寄存器地址和 GPIOA~GPIOK 不是连续的，所以单独使用 pinctrl_z 来描述 GPIOZ，含义和 pinctrl 一样。

我们来看一个具体的应用，打开 stm32mp15xx-dkx.dtsi 文件，找到如下所示内容：

示例代码 31.1.3.6 hdmi 节点信息

```
1   hdmi-transmitter@39 {
2       compatible = "sil,sii9022";
3       reg = <0x39>;
4       iovcc-supply = <&v3v3_hdmi>;
5       cvcc12-supply = <&v1v2_hdmi>;
6       reset-gpios = <&gpioa 10 GPIO_ACTIVE_LOW>;
7       interrupts = <1 IRQ_TYPE_EDGE_FALLING>;
8       interrupt-parent = <&gpiog>;
9       #sound-dai-cells = <0>;
10      status = "okay";
11  };
```

sii9022 是 ST 官方在开发板上的一个 HDMI 芯片，上述代码就是 sii9022 的节点信息，sii9022a 芯片有一个中断，此引脚链接到了 STM32MP1 的 PG1 上，此中断是下降沿触发。

第 7 行，interrupts 设置中断信息，1 表示本组内第一个 IO，在这里就是 PG1。IRQ_TYPE_EDGE_FALLING 表示下降沿触发。

第 8 行，interrupt-parent 属性设置中断控制器，这里是有 gpiog 作为中断控制器。结合上面的 interrupts 属性，这两行的目的就是设置 PG1 为下降沿出触发。可以看出使用起来是非常的简单，在我们实际编写代码的时候，只需要通过 interrupt-parent 和 interrupts 这两个属性即可设置某个 GPIO 的中断功能。

简单总结一下与中断有关的设备树属性信息：

①、#interrupt-cells，指定中断源的信息 cells 个数。

②、interrupt-controller，表示当前节点为中断控制器。

③、interrupts，指定中断号，触发方式等。

④、interrupt-parent，指定父中断，也就是中断控制器。

⑤、interrupts-extended，指定中断控制器、中断号、中断类型和触发方式，这个属性比较特殊，是新加入的。前面说了，要通过 interrupts 和 interrupt-parent 一起设置某个 IO 的中断方式。这里我们也可以只使用 interrupts-extended 属性一次性指定中断父节点，IO 编号，中断方式等。打开 stm32mp157f-ev1-a7-examples.dts 文件，里面有如下所示代码：

示例代码 31.1.3.7 test_keys 节点

```
1  test_keys {
2      compatible =            ;
3      #address-cells = <1>;
4      #size-cells = <0>;
5      autorepeat;
6      status =        ;
7      /* gpio needs vdd core in retention for wakeup */
8      power-domains = <&pd_core_ret>;
9
10     button@1 {
11         label =        ;
12         linux,code = <BTN_1>;
13         interrupts-extended = <&gpioa 13 IRQ_TYPE_EDGE_FALLING>;
14         status =        ;
15         wakeup-source;
16     };
17 };
```

很明显，上述代码用于描述一个按键，此按键采用中断方式，这个按键使用到了 PA1 这个引脚。第 13 行直接通过 interrupts-extended 属性描述了所有的中断信息，第一个参数为 gpioa，第二个参数为 13，第三个参数表示下降沿触发。如果使用 interrupts 和 interrupt-parent 来描述的话就是：

```
interrupt-parent = <&gpioa>;
interrupts = <13 IRQ_TYPE_EDGE_FALLING>;
```

大家根据自己的喜好来选择使用哪种形式描述即可。

31.1.4 获取中断号

编写驱动的时候需要用到中断号，我们用到的中断号，中断信息已经写到了设备树里面，因此可以通过 irq_of_parse_and_map 函数从 interupts 属性中提取到对应的设备号，函数原型如下：

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

函数参数和返回值含义如下：

dev：设备节点。

index：索引号，interrupts 属性可能包含多条中断信息，通过 index 指定要获取的信息。

返回值：中断号。

如果使用 GPIO 的话，可以使用 gpio_to_irq 函数来获取 gpio 对应的中断号，函数原型如下：

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

函数参数和返回值含义如下：

gpio：要获取的 GPIO 编号。

返回值：GPIO 对应的中断号。

31.2 硬件原理图分析

本章实验硬件原理图参考 29.2 小节即可。

31.3 实验程序编写

本实验对应的例程路径为：开发板光盘→1、程序源码→2、Linux 驱动例程→13_irq。

本章实验我们驱动正点原子的 STM32MP157 开发板上的 KEY0 按键，不过我们采用中断的方式，并且采用定时器来实现按键消抖，应用程序读取按键值并且通过终端打印出来。通过本章我们可以学习到 Linux 内核中断的使用方法，以及对 Linux 内核定时器的回顾。

31.3.1 修改设备树文件

本章实验使用到了按键 KEY0，按键 KEY0 使用中断模式，因此需要在“key”节点下添加中断相关属性，添加完成以后的“key”节点内容如下所示：

示例代码 31.3.1.1 key 节点信息

```
1   key {
2       compatible = "alientek,key";
3       status = "okay";
4       key-gpio = <&gpiog 3 GPIO_ACTIVE_LOW>;
5       interrupt-parent = <&gpiog>;
6       interrupts = <3 IRQ_TYPE_EDGE_BOTH>;
7   };
```

第 5 行，设置 interrupt-parent 属性值为“gpiog”，因为 KEY0 所使用的 GPIO 为 PG3，所以要设置 KEY0 的中断控制器为 gpiog。

第 6 行，设置 interrupts 属性，也就是设置中断源，第一个 cells 的 3 表示 GPIOG 组的 3 号 IO。IRQ_TYPE_EDGE_BOTH 定义在文件 include/linux/irq.h 中，定义如下：

示例代码 31.3.1.2 中断状态

```
75  enum {
76      IRQ_TYPE_NONE           = 0x00000000,
```

```
77      IRQ_TYPE_EDGE_RISING        = 0x00000001,
78      IRQ_TYPE_EDGE_FALLING    = 0x00000002,
79      IRQ_TYPE_EDGE_BOTH       = (IRQ_TYPE_EDGE_FALLING |
                                    IRQ_TYPE_EDGE_RISING),
80      IRQ_TYPE_LEVEL_HIGH      = 0x00000004,
81      IRQ_TYPE_LEVEL_LOW       = 0x00000008,
82      IRQ_TYPE_LEVEL_MASK      = (IRQ_TYPE_LEVEL_LOW |
                                    IRQ_TYPE_LEVEL_HIGH),
......
100 };
```

从示例代码 31.3.1.2 中可以看出，IRQ_TYPE_EDGE_BOTH 表示上升沿和下降沿同时有效，相当于 KEY0 按下和释放都会触发中断。

设备树编写完成以后使用“make dtbs”命令重新编译设备树，然后使用新编译出来的 stm32mp157d-atk.dtb 文件启动 Linux 系统。

31.3.2 按键中断驱动程序编写

新建名为“13_irq”的文件夹，然后在 13_irq 文件夹里面创建 vscode 工程，工作区命名为“keyirq”。工程创建好以后新建 keyirq.c 文件，在 keyirq.c 里面输入如下内容：

示例代码 31.3.2.1 keyirq.c 文件代码

```
11  #include <linux/types.h>
12  #include <linux/kernel.h>
13  #include <linux/delay.h>
14  #include <linux/ide.h>
15  #include <linux/init.h>
16  #include <linux/module.h>
17  #include <linux/errno.h>
18  #include <linux/gpio.h>
19  #include <linux/cdev.h>
20  #include <linux/device.h>
21  #include <linux/of.h>
22  #include <linux/of_address.h>
23  #include <linux/of_gpio.h>
24  #include <linux/semaphore.h>
25  #include <linux/of_irq.h>
26  #include <linux/irq.h>
27  #include <asm/mach/map.h>
28  #include <asm/uaccess.h>
29  #include <asm/io.h>
30
31  #define KEY_CNT         1         /* 设备号个数    */
32  #define KEY_NAME        "key"     /* 名字          */
33
34  /* 定义按键状态 */
35  enum key_status {
```

```
36      KEY_PRESS = 0,          /*按键按下      */
37      KEY_RELEASE,            /*按键松开      */
38      KEY_KEEP,               /* 按键状态保持 */
39  };
40
41  /* key 设备结构体 */
42  struct key_dev{
43      dev_t devid;              /* 设备号       */
44      struct cdev cdev;         /* cdev         */
45      struct class *class;      /* 类           */
46      struct device *device;    /* 设备         */
47      struct device_node  *nd; /* 设备节点     */
48      int key_gpio;             /* key 所使用的 GPIO 编号*/
49      struct timer_list timer; /* 按键值       */
50      int irq_num;              /* 中断号       */
51      spinlock_t spinlock;      /* 自旋锁       */
52  };
53
54  static struct key_dev key;          /* 按键设备 */
55  static int status = KEY_KEEP;       /* 按键状态 */
56
57  static irqreturn_t key_interrupt(int irq, void *dev_id)
58  {
59      /* 按键防抖处理，开启定时器延时 15ms */
60      mod_timer(&key.timer, jiffies + msecs_to_jiffies(15));
61      return IRQ_HANDLED;
62  }
63
64  /*
65   * @description   : 初始化按键 IO，open 函数打开驱动的时候
66   *                  初始化按键所使用的 GPIO 引脚。
67   * @param         : 无
68   * @return        : 无
69   */
70  static int key_parse_dt(void)
71  {
72      int ret;
73      const char *str;
74
75      /* 设置 LED 所使用的 GPIO    */
76      /* 1、获取设备节点：key *   /
77      key.nd = of_find_node_by_path("/key");
78      if(key.nd == NULL) {
79          printk("key node not find!\r\n");
```

```
80          return -EINVAL;
81      }
82
83      /* 2.读取 status 属性 */
84      ret = of_property_read_string(key.nd, "status", &str);
85      if(ret < 0)
86          return -EINVAL;
87
88      if (strcmp(str, "okay"))
89          return -EINVAL;
90
91      /* 3、获取 compatible 属性值并进行匹配 */
92      ret = of_property_read_string(key.nd, "compatible", &str);
93      if(ret < 0) {
94          printk("key: Failed to get compatible property\n");
95          return -EINVAL;
96      }
97
98      if (strcmp(str, "alientek,key")) {
99          printk("key: Compatible match failed\n");
100         return -EINVAL;
101     }
102
103     /* 4、 获取设备树中的 gpio 属性，得到 KEY0 所使用的 KYE 编号 */
104     key.key_gpio = of_get_named_gpio(key.nd, "key-gpio", 0);
105     if(key.key_gpio < 0) {
106         printk("can't get key-gpio");
107         return -EINVAL;
108     }
109
110     /* 5 、获取 GPIO 对应的中断号 */
111     key.irq_num = irq_of_parse_and_map(key.nd, 0);
112     if(!key.irq_num){
113         return -EINVAL;
114     }
115
116     printk("key-gpio num = %d\r\n", key.key_gpio);
117     return 0;
118 }
119
120 static int key_gpio_init(void)
121 {
122     int ret;
123     unsigned long irq_flags;
```

```
124
125     ret = gpio_request(key.key_gpio, "KEY0");
126     if (ret) {
127         printk(KERN_ERR "key: Failed to request key-gpio\n");
128         return ret;
129     }
130
131     /* 将 GPIO 设置为输入模式            */
132     gpio_direction_input(key.key_gpio);
133
134     /* 获取设备树中指定的中断触发类型    */
135     irq_flags = irq_get_trigger_type(key.irq_num);
136     if (IRQF_TRIGGER_NONE == irq_flags)
137         irq_flags = IRQF_TRIGGER_FALLING | IRQF_TRIGGER_RISING;
138
139     /* 申请中断 */
140     ret = request_irq(key.irq_num, key_interrupt, irq_flags,
                          "Key0_IRQ", NULL);
141     if (ret) {
142         gpio_free(key.key_gpio);
143         return ret;
144     }
145
146     return 0;
147 }
148
149 static void key_timer_function(struct timer_list *arg)
150 {
151     static int last_val = 1;
152     unsigned long flags;
153     int current_val;
154
155     /* 自旋锁上锁 */
156     spin_lock_irqsave(&key.spinlock, flags);
157
158     /* 读取按键值并判断按键当前状态 */
159     current_val = gpio_get_value(key.key_gpio);
160     if (0 == current_val && last_val)  /* 按下     */
161         status = KEY_PRESS;
162     else if (1 == current_val && !last_val)
163         status = KEY_RELEASE;          /* 松开     */
164     else
165         status = KEY_KEEP;             /*状态保持  */
166
```

```
167     last_val = current_val;
168
169     /* 自旋锁解锁 */
170     spin_unlock_irqrestore(&key.spinlock, flags);
171 }
172
173 /*
174  * @description   : 打开设备
175  * @param - inode : 传递给驱动的 inode
176  * @param - filp  : 设备文件，file 结构体有个叫做 private_data 的成员变量
177  *                  一般在 open 的时候将 private_data 指向设备结构体。
178  * @return        : 0 成功;其他 失败
179  */
180 static int key_open(struct inode *inode, struct file *filp)
181 {
182     return 0;
183 }
184
185 /*
186  * @description   : 从设备读取数据
187  * @param - filp  : 要打开的设备文件(文件描述符)
188  * @param - buf   : 返回给用户空间的数据缓冲区
189  * @param - cnt   : 要读取的数据长度
190  * @param - offt  : 相对于文件首地址的偏移
191  * @return        : 读取的字节数，如果为负值，表示读取失败
192  */
193 static ssize_t key_read(struct file *filp, char __user *buf,
194             size_t cnt, loff_t *offt)
195 {
196     unsigned long flags;
197     int ret;
198
199     /* 自旋锁上锁   */
200     spin_lock_irqsave(&key.spinlock, flags);
201
202     /* 将按键状态信息发送给应用程序 */
203     ret = copy_to_user(buf, &status, sizeof(int));
204
205     /* 状态重置     */
206     status = KEY_KEEP;
207
208     /* 自旋锁解锁   */
209     spin_unlock_irqrestore(&key.spinlock, flags);
210
```

```
211     return ret;
212 }
213
214 /*
215  * @description   : 向设备写数据
216  * @param - filp  : 设备文件，表示打开的文件描述符
217  * @param - buf   : 要写给设备写入的数据
218  * @param - cnt   : 要写入的数据长度
219  * @param - offt  : 相对于文件首地址的偏移
220  * @return        : 写入的字节数，如果为负值，表示写入失败
221  */
222 static ssize_t key_write(struct file *filp, const char __user *buf,
                             size_t cnt, loff_t *offt)
223 {
224     return 0;
225 }
226
227 /*
228  * @description   : 关闭/释放设备
229  * @param - filp  : 要关闭的设备文件(文件描述符)
230  * @return        : 0 成功;其他 失败
231  */
232 static int key_release(struct inode *inode, struct file *filp)
233 {
234     return 0;
235 }
236
237 /* 设备操作函数 */
238 static struct file_operations key_fops = {
239     .owner = THIS_MODULE,
240     .open = key_open,
241     .read = key_read,
242     .write = key_write,
243     .release =  key_release,
244 };
245
246 /*
247  * @description   : 驱动入口函数
248  * @param         : 无
249  * @return        : 无
250  */
251 static int __init mykey_init(void)
252 {
253     int ret;
```

```
254
255     /* 初始化自旋锁     */
256     spin_lock_init(&key.spinlock);
257
258     /* 设备树解析       */
259     ret = key_parse_dt();
260     if(ret)
261         return ret;
262
263     /* GPIO 中断初始化 */
264     ret = key_gpio_init();
265     if(ret)
266         return ret;
267
268     /* 注册字符设备驱动 */
269     /* 1、创建设备号    */
270     ret = alloc_chrdev_region(&key.devid, 0, KEY_CNT, KEY_NAME);
271     if(ret < 0) {
272         pr_err("%s Couldn't alloc_chrdev_region, ret=%d\r\n",
                    KEY_NAME, ret);
273         goto free_gpio;
274     }
275
276     /* 2、初始化 cdev   */
277     key.cdev.owner = THIS_MODULE;
278     cdev_init(&key.cdev, &key_fops);
279
280     /* 3、添加一个 cdev */
281     cdev_add(&key.cdev, key.devid, KEY_CNT);
282     if(ret < 0)
283         goto del_unregister;
284
285     /* 4、创建类        */
286     key.class = class_create(THIS_MODULE, KEY_NAME);
287     if (IS_ERR(key.class)) {
288         goto del_cdev;
289     }
290
291     /* 5、创建设备      */
292     key.device = device_create(key.class, NULL, key.devid, NULL,
                                   KEY_NAME);
293     if (IS_ERR(key.device)) {
294         goto destroy_class;
295     }
```

```
296
297     /* 6、初始化 timer，设置定时器处理函数,还未设置周期，所有不会激活定时器 */
298     timer_setup(&key.timer, key_timer_function, 0);
299
300     return 0;
301
302 destroy_class:
303     device_destroy(key.class, key.devid);
304 del_cdev:
305     cdev_del(&key.cdev);
306 del_unregister:
307     unregister_chrdev_region(key.devid, KEY_CNT);
308 free_gpio:
309     free_irq(key.irq_num, NULL);
310     gpio_free(key.key_gpio);
311     return -EIO;
312 }
313
314 /*
315  * @description   : 驱动出口函数
316  * @param         : 无
317  * @return        : 无
318  */
319 static void __exit mykey_exit(void)
320 {
321     /* 注销字符设备驱动 */
322     cdev_del(&key.cdev);                    /* 删除 cdev    */
323     unregister_chrdev_region(key.devid, KEY_CNT); /* 注销设备号 */
324     del_timer_sync(&key.timer);             /* 删除 timer   */
325     device_destroy(key.class, key.devid); /*注销设备       */
326     class_destroy(key.class);               /* 注销类       */
327     free_irq(key.irq_num, NULL);            /* 释放中断     */
328     gpio_free(key.key_gpio);                /* 释放 IO      */
329 }
330
331 module_init(mykey_init);
332 module_exit(mykey_exit);
333 MODULE_LICENSE("GPL");
334 MODULE_AUTHOR("ALIENTEK");
335 MODULE_INFO(intree, "Y");
```

第 35~39 行，定义了一个枚举类型，包含 3 个常量 KEY_PRESS、KEY_RELEASE、KEY_KEEP，分别用来表示按键的 3 种不同的状态，即按键按下、按键松开以及按键状态保持。

第 42~52 行，结构体 key_dev 为按键设备所对应的结构体，key_gpio 为按键 GPIO 编号，

irq_num 为按键 IO 对应的中断号；除此之外，结构体当中还定义了一个定时器用于实现按键的去抖操作，还定义了一个自旋锁用于实现对关键代码的保护操作。

第 54 行，定义一个按键设备 key。

第 55 行，定义一个 int 类型的静态全局变量 status 用来表示按键的状态。

第 57~62 行，key_interrupt 函数是按键 KEY0 中断处理函数，参数 dev_id 是一个 void 类型的指针，本驱动程序并没使用到这个参数；这个中断处理函数很简单直接开启定时器，延时 15 毫秒，用于实现按键的软件防抖。

第 70~118 行，key_parse_dt 函数中主要是对设备树中的属性进行了解析，获取设备树中的 key 节点，通过 of_get_named_gpio 函数得到按键的 GPIO 编号，通过 irq_of_parse_and_map 函数获取按键的中断号，irq_of_parse_and_map 函数会解析 key 节点中的 interrupt-parent 和 interrupts 属性然后得到一个中断号，后面就可以使用这个中断号去申请以及释放中断了。

第 120~147 行，key_gpio_init 函数中主要对 GPIO 以及中断进行了相关的初始化。使用 gpio_request 函数申请 GPIO 使用权，通过 gpio_direction_input 将 GPIO 设置为输出模式；irq_get_trigger_type 函数可以获取到 key 节点中定义的中断触发类型，最后使用 request_irq 申请中断，并设置 key_interrupt 函数作为我们的按键中断处理函数，当按键中断发生之后便会跳转到该函数执行；request_irq 函数会默认使能中断，所以不需要 enable_irq 来使能中断，当然我们也可以在申请成功之后先使用 disable_irq 函数禁用中断，等所有工作完成之后再来使能中断，这样会比较安全，建议大家这样使用。

第 149~171 行，key_timer_function 函数为定时器定时处理函数，它的参数 arg 在本驱动程序中我们并没有使用到；该函数中定义了一个静态局部变量 last_val 用来保存按键上一次读取到的值，变量 current_val 用来存放当前按键读取到的值；第 159~165 行，通过读取到的按键值以及上一次读取到的值来判断按键当前所属的状态，如果本次读取的值为 0，而上一次读取的值 1，则表示按键按下；如果本次读取的值为 1，而上一次读取的值 0，则表示按键松开；如果本次读取的值为 0，而上一次读取的值也是 0，则表示按键一直被按着；如果本次读取的值为 1，而上一次读取的值也是 1，则表示没有触碰按键。第 167 行，当状态判断完成之后，会将 current_val 的值赋值给 last_val。本函数中也使用自旋锁对全局变量 status 进行加锁保护！

第 193~212 行，key_read 函数，对应应用程序的 read 函数。此函数向应用程序返回按键状态信息数据；这个函数其实很简单，使用 copy_to_user 函数直接将 statue 数据发送给应用程序，status 变量保存了按键当前的状态，发送完成之后再将按键状态重置即可！需要注意的是，该函数中使用了自旋锁进行保护。

第 238~244 行，按键设备的 file_operations 结构体。

第 251~312 行，mykey_init 是驱动入口函数，第 256 行调用 spin_lock_init 初始化自旋锁变量，298 行对定时器进行初始化并将 key_timer_function 函数绑定为定时器定时处理函数，当定时时间到了之后便会跳转到该函数执行。

第 319~329 行，mykey_exit 驱动出口函数，第 324 行调用 del_timer_sync 函数删除定时器，代码中已经注释得非常详细了，这里便不再多说！

31.3.2 编写测试 APP

测试 APP 要实现的内容很简单，通过不断的读取/dev/key 设备文件来获取按键值来判断当前按键的状态，从按键驱动上传到应用程序的数据可以有 3 个值，分别为 0、1、2；0 表示按键按下时的这个状态，1 表示按键松开时对应的状态，而 2 表示按键一直被按住或者松开；搞懂数据代表的意思之后，我们开始编写测试程序，在 13_irq 目录下新建名为 keyirqApp.c

的文件，然后输入如下所示内容：

示例代码 31.3.3.1 keyrqApp.c 文件代码

```
1  /***************************************************************
2  Copyright © ALIENTEK Co., Ltd. 1998-2029. All rights reserved.
3  文件名       : keyApp.c
4  作者         : 正点原子 Linux 团队
5  版本         : V1.0
6  描述         : Linux 中断驱动实验
7  其他         : 无
8  使用方法     : ./keyirqApp /dev/key
9  论坛         : www.openedv.com
10 日志         : 初版 V1.0 2021/1/30 正点原子 Linux 团队创建
11 ***************************************************************/
12
13 #include <stdio.h>
14 #include <unistd.h>
15 #include <sys/types.h>
16 #include <sys/stat.h>
17 #include <fcntl.h>
18 #include <stdlib.h>
19 #include <string.h>
20
21 /*
22  * @description   : main 主程序
23  * @param - argc  : argv 数组元素个数
24  * @param - argv  : 具体参数
25  * @return        : 0 成功;其他 失败
26  */
27 int main(int argc, char *argv[])
28 {
29     int fd, ret;
30     int key_val;
31
32     /* 判断传参个数是否正确 */
33     if(2 != argc) {
34         printf("Usage:\n"
35             "\t./keyApp /dev/key\n"
36            );
37         return -1;
38     }
39
40     /* 打开设备 */
41     fd = open(argv[1], O_RDONLY);
42     if(0 > fd) {
```

```
43          printf("ERROR: %s file open failed!\n", argv[1]);
44          return -1;
45      }
46
47      /* 循环读取按键数据 */
48      for ( ; ; ) {
49
50          read(fd, &key_val, sizeof(int));
51          if (0 == key_val)
52              printf("Key Press\n");
53          else if (1 == key_val)
54              printf("Key Release\n");
55      }
56
57      /* 关闭设备 */
58      close(fd);
59      return 0;
60  }
```

第 48~55 行使用 for 循环不断的读取按键值，如果读取到的值是 0 则打印“Key Press”字符串，而过读取到的值是 1 则打印“Key Release”字符串。

31.4 运行测试

31.4.1 编译驱动程序和测试 APP

1、编译驱动程序

编写 Makefile 文件，本章实验的 Makefile 文件和第四十章实验基本一样，只是将 obj-m 变量的值改为 keyirq.o，Makefile 内容如下所示：

示例代码 31.4.1.1 Makefile 文件

```
1  KERNELDIR := /home/zuozhongkai/linux/my_linux/linux-5.4.31
......
4  obj-m := keyirq.o
......
11 clean:
12  $(MAKE) -C $(KERNELDIR) M=$(CURRENT_PATH) clean
```

第 4 行，设置 obj-m 变量的值为 keyirq.o。

输入如下命令编译出驱动模块文件：

```
make -j32
```

编译成功以后就会生成一个名为“keyirq.ko”的驱动模块文件。

2、编译测试 APP

输入如下命令编译测试 keyirqApp.c 这个测试程序：

```
arm-none-linux-gnueabihf-gcc keyirqApp.c -o keyirqApp
```

编译成功以后就会生成 keyirqApp 这个应用程序。

31.4.2 运行测试

将上一小节编译出来 keyirq.ko 和 keyirqApp 这两个文件拷贝到 rootfs/lib/modules/5.4.31 目录中，重启开发板，进入到目录 lib/modules/5.4.31 中，输入如下命令加载 keyirq.ko 驱动模块：

```
depmod                  //第一次加载驱动的时候需要运行此命令
modprobe keyirq.ko      //加载驱动
```

驱动加载成功以后可以通过查看/proc/interrupts 文件来检查一下对应的中断有没有被注册上，输入如下命令：

```
cat /proc/interrupts
```

结果如图 51.4.2.1 所示：

```
 47:        803          0  stm32-exti-h-direct  30 Level     40010000.serial
 48:          0          0  stm32-exti-h-direct  70 Level     eth0
 49:          0          0  stm32-exti-h-direct  19 Level     5c004000.rtc
 50:          0          0  stm32-exti-h-direct  61 Edge      4c001000.mailbox
 51:          0          0     GIC-0 133 Level     4c001000.mailbox
 52:          0          0  stm32gpio   3 Edge      Key0_IRQ
IPI0:         0          0  CPU wakeup interrupts
IPI1:         0          0  Timer broadcast interrupts
```

图 31.4.2.1 proc/interrupts 文件内容

从图 31.4.2.1 可以看出 keyirq.c 驱动文件里面的 KEY0 中断已经存在了，触发方式为跳边沿(Edge)。

接下来使用如下命令来测试中断：

```
./keyirqApp /dev/key
```

按下开发板上的 KEY0 键，终端就会输出按键值，如图 31.4.2.2 所示：

```
[root@ATK-stm32mp1]:/lib/modules/5.4.31$ ./keyirqApp /dev/key
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
Key Press
Key Release
```

图 31.4.2.2 读取到的按键值

从图 31.4.2.2 可以看出，按键值获取成功，并且不会有按键抖动导致的误判发生，说明按键消抖工作正常。如果要卸载驱动的话输入如下命令即可：

```
rmmod keyirq.ko
```