amp linux中断

一、发生中断时的执行流程

> W(b)　vector_irq + stubs_offset -> vector_irq -> __irq_usr或者__irq_svc -> irq_handler -> gic_handle_irq -> handle_IRQ -> generic_handle_irq -> generic_handle_irq_desc -> desc->handle_irq

1、汇编阶段的处理

　　发生中断时，cpu进入irq异常模式，会跑到异常向量表运行。linux在之前已经初始化了异常向量表和异常处理代码，异常向量表在虚拟地址0xffff0000处。异常向量表和处理代码在arch/arm/kernel/entry_armv.S中定义。

```
..........................................................................................
/* 异常向量表 */
     .equ    stubs_offset, __vectors_start + 0x200 - __stubs_start
    .globl   __vectors_start
__vectors_start:     // 异常向量表起始
 ARM(    swi SYS_ERROR0   )
...
    W(b)     vector_und + stubs_offset
    W(ldr)   pc, .LCvswi + stubs_offset
    W(b)     vector_pabt + stubs_offset
    W(b)     vector_dabt + stubs_offset
    W(b)     vector_addrexcptn + stubs_offset
    W(b)     vector_irq + stubs_offset
    W(b)     vector_fiq + stubs_offset

    .globl   __vectors_end
__vectors_end:   // 异常向量表结束
...........................................................................
/* 异常处理函数 */
    .globl   __stubs_start
__stubs_start:   // 异常处理函数起始

    // irq、data abort、prefetch abort和udef异常处理函数都用vector_stub宏定义
    vector_stub irq, IRQ_MODE, 4
```

```
    .long   __irq_usr               @ 如果进入中断前是用户态则跳到__irq_usr
    .long   __irq_invalid           @  1
    .long   __irq_invalid           @  2
    .long   __irq_svc               @  如果进入中断前是内核态则跳到__irq_svc
    .long   __irq_invalid           @  4
    .long   __irq_invalid           @  5
    .long   __irq_invalid           @  6
    .long   __irq_invalid           @  7
    .long   __irq_invalid           @  8
    .long   __irq_invalid           @  9
    .long   __irq_invalid           @  a
    .long   __irq_invalid           @  b
    .long   __irq_invalid           @  c
    .long   __irq_invalid           @  d
    .long   __irq_invalid           @  e
    .long   __irq_invalid           @  f


    vector_stub dabt, ABT_MODE, 8

    .long   __dabt_usr              @  0  (USR_26 / USR_32)
    .long   __dabt_invalid          @  1  (FIQ_26 / FIQ_32)
    .long   __dabt_invalid          @  2  (IRQ_26 / IRQ_32)
    .long   __dabt_svc              @  3  (SVC_26 / SVC_32)
    .long   __dabt_invalid          @  4
...

// fiq、swi和address异常单独使用特定编写的处理函数
vector_fiq:
    subs    pc, lr, #4

vector_addrexcptn:
    b   vector_addrexcptn

    .align  5

.LCvswi:
    .word   vector_swi

    .globl  __stubs_end
__stubs_end:    /* 异常处理函数结束 */
..................................................................................
/* 除了fiq、swi和address异常之外的异常处理函数宏 */
    .macro  vector_stub, name, mode, correction=0
    .align  5
vector_\name:
    .if \correction
    sub lr, lr, #\correction        //修正返回地址lr_<exception>
    .endif

    @
    @ Save r0, lr_<exception> (parent PC) and spsr_<exception> (parent CPSR)
    @
```

```
    /* 将r0，修正后的lr_<exception>，spsr_<exception>保存到sp_<exception>中，
     * spsr_<exception>是进入中断前的cpsr，sp_<exception>空间很小
     */
    stmia   sp, {r0, lr}    @ save r0, lr
    mrs lr, spsr            @ lr = spsr_<exception>
    str lr, [sp, #8]        @ save spsr

    @
    @ Prepare for SVC32 mode.  IRQs remain disabled.
    @
    /* 在进入irq异常时处理器自动将cpsr的最低5位mode位设置为irq，同时设置I和F
     * 将cpsr读到r0中，然后将模式域切换为SVC模式，
     * 再将r0写到spsr_<exception>，因此现在还是<exception>不是svc，
     * 同时未更改前的spsr_<exception>也保存着副本在sp_<exception>中
     */
    mrs r0, cpsr
    eor r0, r0, #(\mode ^ SVC_MODE | PSR_ISETSTATE)
    msr spsr_cxsf, r0

    @
    @ the branch table must immediately follow this code
    @
    /*
     * 保存sp_<exception>到r0，
     * 根据发生中断前的模式进入对应的子处理程序，如irq模式下的__irq_usr或者__irq_svc，
     * 同时将以修改的spsr_<exception>恢复到cpsr中，因此等于将中断前的cpsr的模式域切换为svc再恢复
     */
    and lr, lr, #0x0f   //获取修改前的spsr_<exception>即中断前的cpsr的模式
    mov r0, sp        //将sp_<exception>保存到r0，然后可以在子程序中使用
    ldr lr, [pc, lr, lsl #2]        //lr = pc + lr << 2
    movs    pc, lr      /* 根据lr跳转到适合的子处理程序如irq模式下的__irq_usr或者__irq_svc，
                        因为movs，所以同时cpsr = 修改过的spsr_<exception> */
ENDPROC(vector_\name)

    .align  2
    @ handler addresses follow this label
1:
    .endm
.............................................................................
/* 用户态的irq处理入口 */
        .align  5
__irq_usr:
    usr_entry       // 保存usr中断现场
    kuser_cmpxchg_check
    irq_handler     // c预言irq处理入口，调用gic_handle_irq
    get_thread_info tsk     //
    mov why, #0             // why = 0
    b   ret_to_user_from_irq    // 从中断返回usr
...
ENDPROC(__irq_usr)
....................................................................................
/* 内核态的irq处理入口 */
      .align    5
```

```
__irq_svc:
    svc_entry       // 保存svc中断现场
    irq_handler     // c预言irq处理入口，调用gic_handle_irq

#ifdef CONFIG_PREEMPT
    get_thread_info tsk     //
    ldr r8, [tsk, #TI_PREEMPT]      @ get preempt count
    ldr r0, [tsk, #TI_FLAGS]        @ get flags
    teq r8, #0              @ if preempt count != 0
    movne   r0, #0              @ force flags to 0
    tst r0, #_TIF_NEED_RESCHED
    blne    svc_preempt     //
#endif


...
    svc_exit r5             // 内核态恢复中断现场
...
ENDPROC(__irq_svc)
.................................................................................
/* c语言的irq处理入口 */
     .macro irq_handler
#ifdef CONFIG_MULTI_IRQ_HANDLER
    ldr r1, =handle_arch_irq    //对于socfpga会设置为gic_handle_irq
    mov r0, sp              //传给gic_handle_irq的参数r0 = sp_svc，即pt_regs
    adr lr, BSYM(9997f)
    ldr pc, [r1]            //调用gic_handle_irq
#else
    arch_irq_handler_default
#endif
9997:
    .endm
```

位于__vectors_start和__vectors_end之间的是真正的异常向量表，位于__stubs_start和__stubs_end之间的是处理代码。对于irq中断先进入W(b)　vector_irq + stubs_offset然后进入vector_irq，根据进入irq前的模式进入__irq_usr或者__irq_svc，无论是__irq_usr还是__irq_svc都会进入irq_handler，然后调用handle_arch_irq即gic_handle_irq。其中__irq_usr和__irq_svc分别是保存从用户和内核态进入中断的保存中断现场。

```
/* 内核态的保存中断现场  */
    .macro  svc_entry, stack_hole=0
...
     /* sp_svc = sp_svc - (S_FRAME_SIZE  - 4), S_FRAME_SIZE是struct pt_regs的大小
      * p_svc指向pt_regs中的ARM_r1
      */
    sub sp, sp, #(S_FRAME_SIZE + \stack_hole - 4)
...
     /* 将r1到r12保存到sp_svc中，因为r1到r12在vector_\name中没有改变，因此
       这里的r1到r12是发生中断异常前的r1到r12 */
    stmia   sp, {r1 - r12}
    /* r0保存着sp_<exception>，因此这里把vector_\name中保存到sp_<exception>
       中的r0, lr_<exception>, spsr_<exception>恢复到r3,r4,r5 */
    ldmia   r0, {r3 - r5}
    /* S_SP是offsetof(struct pt_regs, ARM_sp)
```

```
      将pt_regs中ARM_sp处的地址给r7 */
    add r7, sp, #S_SP - 4    @ here for interlock avoidance
    mov r6, #-1              @  ""  ""      ""        ""
    add r2, sp, #(S_FRAME_SIZE + \stack_hole - 4)    //r2保存原来未发生异常前的sp_svc
...
    // 将发生中断异常前的r0保存到sp_svc中的pt_regs中
    str r3, [sp, #-4]!      @ save the "real" r0 copied
                        @ from the exception stack
    mov r3, lr  //将lr_svc保存在r3中

    @
    @ We are now ready to fill in the remaining blanks on the stack:
    @
    @  r2 - sp_svc
    @  r3 - lr_svc
    @  r4 - lr_<exception>, already fixed up for correct return/restart
    @  r5 - spsr_<exception>
    @  r6 - orig_r0 (see pt_regs definition in ptrace.h)
    @
    stmia   r7, {r2 - r6}   //将最后几个保存到pt_regs中

...
    .endm
...........................................................
/* 用户态的保存中断现场 */
    .macro  usr_entry
...
    sub sp, sp, #S_FRAME_SIZE   //sp_svc指向pt_regs中的ARM_r0
    stmib   sp, {r1 - r12}      //将发生中断前的r1到r12保存到sp_svc的pt_regs中
    //将中断前的r0，pc（lr_<exception>）和cpsr(spsr_<exception>)恢复到r3,r4,r5
    ldmia   r0, {r3 - r5}
    //r0指向pt_regs中的ARM_pc
    add r0, sp, #S_PC       @ here for interlock avoidance
    mov r6, #-1         @  ""  ""      ""        ""
    /* 将发生中断异常前的r0保存到pt_regs中 */
    str r3, [sp]        @ save the "real" r0 copied from the exception stack

    @
    @ We are now ready to fill in the remaining blanks on the stack:
    @
    @  r4 - lr_<exception>, already fixed up for correct return/restart
    @  r5 - spsr_<exception>
    @  r6 - orig_r0 (see pt_regs definition in ptrace.h)
    @
    @ Also, separately save sp_usr and lr_usr
    @
    stmia   r0, {r4 - r6}   /* 将这几个保存到pt_regs中，和svc不同的是没有保存sp_usr和lr_usr,放
到下一步 */
    stmdb   r0, {sp, lr}^    //将sp_usr和lr_usr保存到pt_regs中，因为^，所以不是sp_svc和lr_svc
...
    .endm
```

看了保存中断现场，现在来看看恢复中断现场。

```
....................................................................
/* 内核态恢复中断现场 */
    .macro  svc_exit, rpsr
    msr spsr_cxsf, \rpsr    //将spsr_<exception>恢复，因为传参是r5
    ...
    /* 恢复所有的pt_regs，同时spsr_<exception>写到cpsr,
     * 此时中断从新打开，但在这之前的irq_handler中处理软中断时会打开，
     * 处理完软中断再关闭，然后到这一步再恢复
     */
    ldmia   sp, {r0 - pc}^          @ load r0 - pc, cpsr
    .endm
....................................................................
/* 用户态恢复中断现场 */
    ENTRY(ret_to_user_from_irq)
    ldr r1, [tsk, #TI_FLAGS]    /* 从svc栈的thread_info获取发生中断前的进程的task_struct中的
flags */
    tst r1, #_TIF_WORK_MASK     //获取flags中的work部分
    bne work_pending    //如果不为0说明有工作要做(有信号或者要被调度)
no_work_pending:
...
    //如果没有工作就正常恢复用户中断现场
    arch_ret_to_user r1, lr //什么都不做
    restore_user_regs fast = 0, offset = 0  //将保存的用户寄存器恢复
ENDPROC(ret_to_user_from_irq)


    /* 将保存的用户寄存器恢复 */
    .macro  restore_user_regs, fast = 0, offset = 0
    ldr r1, [sp, #\offset + S_PSR]  //获取发生异常前的cpsr到r1
    ldr lr, [sp, #\offset + S_PC]!  //获取异常后返回的地址到Lr_svc，同时更新sp_svc指向
    msr spsr_cxsf, r1           //将发生异常前的cpsr写到spsr_svc中
    ...
    .if \fast
    ldmdb   sp, {r1 - lr}^          @ get calling r1 - lr
    .else
    ldmdb   sp, {r0 - lr}^          //将sp_svc指向的pt_regs恢复到r0到r12以及sp_usr和lr_usr
    .endif
    mov r0, r0              @ ARMv5T and earlier require a nop
                        @ after ldm {}^
    add sp, sp, #S_FRAME_SIZE - S_PC    //将sp_svc指回没发生异常前的地方
    movs    pc, lr              //返回发生异常前的地址，同时恢复异常前的cpsr
    .endm

/* 处理pending事物,调度或者处理信号 */
work_pending:
    mov r0, sp              @ 'regs'
    mov r2, why             @ 'syscall'
    bl  do_work_pending     //调度或者处理信号（没打开中断，为什么能调度？因为中断已经处理完）
    cmp r0, #0  //如果do_work_pending返回值为0则跳到no_work_pending，否则要restar还是strace什么的
    beq no_work_pending
    movlt   scno, #(__NR_restart_syscall - __NR_SYSCALL_BASE)
    ldmia   sp, {r0 - r6}           @ have to reload r0 - r6
    b   local_restart           @ ... and off we go
```

```
/* 调度或处理信号 */
asmlinkage int do_work_pending(struct pt_regs *regs, unsigned int thread_flags, int
syscall)
{
    do {
        if (likely(thread_flags & _TIF_NEED_RESCHED)) { //调度
            schedule();
        } else {        //处理信号
            if (unlikely(!user_mode(regs)))
                return 0;
            local_irq_enable();
            if (thread_flags & _TIF_SIGPENDING) {
                int restart = do_signal(regs, syscall);
                if (unlikely(restart)) {
                    /*
                     * Restart without handlers.
                     * Deal with it without leaving
                     * the kernel space.
                     */
                    return restart;
                }
                syscall = 0;
            } else {
                clear_thread_flag(TIF_NOTIFY_RESUME);
                tracehook_notify_resume(regs);
            }
        }
        local_irq_disable();
        thread_flags = current_thread_info()->flags;
    } while (thread_flags & _TIF_WORK_MASK);
    return 0;
}
```

2、c语言阶段的处理

```
/* c语言的irq处理入口 */
     .macro irq_handler
#ifdef CONFIG_MULTI_IRQ_HANDLER
    ldr r1, =handle_arch_irq    //对于socfpga会设置为gic_handle_irq
    mov r0, sp              //传给gic_handle_irq的参数r0 = sp_svc，即pt_regs
    adr lr, BSYM(9997f)
    ldr pc, [r1]            //调用gic_handle_irq
#else
    arch_irq_handler_default
#endif
9997:
    .endm
```

c语言阶段的irq处理入口irq_handler，这是一个宏，在系统初始化时会设置handle_arch_irq为gic_handle_irq，因此gic_handle_irq是真正的c语言阶段的irq处理入口。

```
/* 真正的c语言阶段的irq处理入口 */
asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
    u32 irqstat, irqnr;
    struct gic_chip_data *gic = &gic_data[0];
    void __iomem *cpu_base = gic_data_cpu_base(gic);

    do {
        /* 如果没有中断挂起则返回一个1023 */
        irqstat = readl_relaxed(cpu_base + GIC_CPU_INTACK);
        irqnr = irqstat & ~0x1c00;   // 读出硬件中断号
/*
***************************************************************************
*                                     Embest Tech co., ltd
*                                     www.embest-tech.com
***************************************************************************
*
*chage the IPI interrupt handler, cause the another cpu not secheule the linux thread any
more.
*/
        /* 在smp中是likely(irqnr > 15 && irqnr < 1021)，
         * 并且有单独的irqnr < 16的handle_IPI而不用handle_IRQ，
         * 在amp中将sgi当普通的irq来操作，去掉原来的handle_IPI，
         * 因为这是内核用来进行多核调度负载均衡专用的中断
        */
        if (likely(irqnr < 1021)) {
            if(irqnr < 16) {
                /* 如果是sgi中断则写EOI，在smp中就是这样子 */
                writel_relaxed(irqstat, cpu_base + GIC_CPU_EOI);
            }
            irqnr = irq_find_mapping(gic->domain, irqnr);   // 硬件中断号转化为虚拟中断号
            handle_IRQ(irqnr, regs);                        // 进入irqnr的通用层处理
            continue;
        }
        break;  //没有中断挂起则退出循环
    } while (1);
}
```

在读取寄存器得到硬件中断号后传给handle_IRQ进入注册的irq处理，在原来的smp中sgi是通过handle_IPI来处理的，每个sgi对应特定的行为，不需要像spi中断一样注册中断处理函数，但在amp方案中sgi需要自定义行为，因此按照spi的方式注册使用，因此都会调用handle_IRQ处理事先注册的irq。

```
/* 处理注册的irq，此时irq已经是虚拟中断号 */
void handle_IRQ(unsigned int irq, struct pt_regs *regs)
{
    struct pt_regs *old_regs = set_irq_regs(regs);

    irq_enter();    /* 增加preempt count HARDOFF，禁止抢占 */
```

```
    if (unlikely(irq >= nr_irqs)) {
        if (printk_ratelimit())
            printk(KERN_WARNING "Bad IRQ%u\n", irq);
        ack_bad_irq(irq);
    } else {
        generic_handle_irq(irq);    /* 进入硬中断通用层 */
    }

    irq_exit();     /* 减少preempt count HARDOFF，同时检查是否可以和需要执行软中断 */
    set_irq_regs(old_regs);
}
```

在irq_enter()中通过增加当前进程的thread info的preempt count中的hardirq域的值表示进入硬件中断上下文，在irq_exit()中减少preempt count，接着判断如果preempt count为0表示可以触发软中断处理。

```
/* irq处理通用层 */
int generic_handle_irq(unsigned int irq)
{
    struct irq_desc *desc = irq_to_desc(irq); //取出irq对应的irq_desc

    if (!desc)
        return -EINVAL;
    generic_handle_irq_desc(irq, desc);     //调用流控层
    return 0;
}


  static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
     /* 调用流控层函数，
      * 对于amp来说sgi和spi调用的是handle_fasteoi_irq，ppi调用的是handle_percpu_devid_irq,
      * 对于smp来说spi调用的是handle_fasteoi_irq，ppi调用的是handle_percpu_devid_irq，
      * 而sgi不会跑到这里，在上文说过。
      */
    desc->handle_irq(irq, desc);
}
```

通用层接口generic_handle_irq通过desc->handle_irq调用进入中断映射设置的流控层函数，对于amp的sgi和spi调用的是handle_fasteoi_irq。

```
void handle_fasteoi_irq(unsigned int irq, struct irq_desc *desc)
{
    raw_spin_lock(&desc->lock);     //锁住

    ...

    /* 如果该中断没有action或者中断被关闭了则将该中断状态设置为IRQS_PENDING，
     * 然后调用irq_chip中的mask屏蔽该中断
    */
    if (unlikely(!desc->action || irqd_irq_disabled(&desc->irq_data))) {
        desc->istate |= IRQS_PENDING;
```

```
        mask_irq(desc);
        goto out;
    }

    /* 如果irq状态包含IRQS_ONESHOT则屏蔽该中断 */
    if (desc->istate & IRQS_ONESHOT)
        mask_irq(desc);

    preflow_handler(desc);
    handle_irq_event(desc);     /* 真正处理硬中断 */

    /* 如果irq状态包含IRQS_ONESHOT则打开该中断，因为前面屏蔽了 */
    if (desc->istate & IRQS_ONESHOT)
        cond_unmask_irq(desc);

out_eoi:
    desc->irq_data.chip->irq_eoi(&desc->irq_data);  /* 发送eoi */
out_unlock:
    raw_spin_unlock(&desc->lock);   //开锁
    return;
...
}
```

在handle_fasteoi_irq中真正处理irq的是handle_irq_event(desc)，

```
irqreturn_t handle_irq_event(struct irq_desc *desc)
{
    struct irqaction *action = desc->action;
    irqreturn_t ret;

    desc->istate &= ~IRQS_PENDING;  //清除IRQS_PENDING
    /* 设置中断为IRQD_IRQ_INPROGRESS状态，表示正在处理硬件中断 */
    irqd_set(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    raw_spin_unlock(&desc->lock);   //开锁

    /* 处理action并做相应的动作 */
    ret = handle_irq_event_percpu(desc, action);

    raw_spin_lock(&desc->lock); //锁
    /* 清除IRQD_IRQ_INPROGRESS状态，表示硬件中断处理完成 */
    irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);
    return ret;
}
```

handle_irq_event的核心是handle_irq_event_percpu，它真正调用我们注册的中断处理函数，并根据情况作进一步处理。

```
/* 处理action */
irqreturn_t handle_irq_event_percpu(struct irq_desc *desc, struct irqaction *action)
{
    irqreturn_t retval = IRQ_NONE;
```

```
    unsigned int flags = 0, irq = desc->irq_data.irq;

    do {
        irqreturn_t res;

        trace_irq_handler_entry(irq, action);
        res = action->handler(irq, action->dev_id); //运行注册的中断处理函数
        trace_irq_handler_exit(irq, action, res);

        if (WARN_ONCE(!irqs_disabled(),"irq %u handler %pF enabled interrupts\n",
                  irq, action->handler))
            local_irq_disable();
        /* 根据action->handler的返回值来作相应处理 */
        switch (res) {
        case IRQ_WAKE_THREAD:   /* 如果返回IRQ_WAKE_THREAD说明要唤醒中断线程 */
            /*
             * Catch drivers which return WAKE_THREAD but
             * did not set up a thread function
             */
            if (unlikely(!action->thread_fn)) {
                warn_no_thread(irq, action);
                break;
            }

            irq_wake_thread(desc, action);      /* 唤醒中断线程 */

            /* Fall through to add to randomness */
        case IRQ_HANDLED:       /* 如果返回IRQ_HANDLED说明中断处理完 */
            flags |= action->flags;
            break;

        default:
            break;
        }

        retval |= res;
        action = action->next;      /* 取下一个action，如果是共享中断才有 */
    } while (action);

    add_interrupt_randomness(irq, flags);

    if (!noirqdebug)
        note_interrupt(irq, desc, retval);
    return retval;
}
```

二、系统启动时的中断初始化

1、定义机器描述符

首先机器描述符结构体在编译链接被连接进一个.arch.info.init的段中，接下来内核初始化会在这里找到他调用里面对应阶段的初始化函数。

```
/* socfpga.c */
DT_MACHINE_START(SOCFPGA, "Altera SOCFPGA")
    .smp            = smp_ops(socfpga_smp_ops),
    .map_io     = socfpga_map_io,
    .init_irq   = gic_init_irq,     /* 在start_kernel中的init_IRQ()中调用 */
    .handle_irq     = gic_handle_irq,   /* 在setup_arch中赋值给handle_arch_irq */
    .timer      = &dw_apb_timer,
    .nr_irqs        = SOCFPGA_NR_IRQS,  //自定义512
    .init_machine   = socfpga_cyclone5_init,
    .restart    = socfpga_cyclone5_restart,
    .reserve        = socfpga_ucosii_reserve,
    .dt_compat  = altera_dt_match,
MACHINE_END

#define DT_MACHINE_START(_name, _namestr)       \
static const struct machine_desc __mach_desc_##_name    \
 __used                         \
 __attribute__((__section__(".arch.info.init"))) = {    \
    .nr     = ~0,               \
    .name       = _namestr,
```

2、中断子系统初始化

```
asmlinkage void __init start_kernel(void)
{
    ...
    setup_arch(&command_line);  /* 设置irq_handler宏中的handle_arch_irq为gic_handle_irq */
    ...
    early_irq_init();   /* 预先分配保留512个desc,同时设置位图，将这些desc插入基数树或者数组 */
    init_IRQ();         /* 调用机器描述符定义的gic_init_irq */
    ...
}
```

```
void __init setup_arch(char **cmdline_p)
{
    struct machine_desc *mdesc;
    ...
    mdesc = setup_machine_fdt(__atags_pointer);     /* 获取机器描述符 */
    ...
#ifdef CONFIG_MULTI_IRQ_HANDLER
    /* 将机器描述符的handle_irq即gic_handle_irq赋给handle_arch_irq变量
     * 因此irq_handler宏中的handle_arch_irq就是gic_handle_irq
    */
    handle_arch_irq = mdesc->handle_irq;
#endif
```

```
    ...
}
```

```
int __init early_irq_init(void)
{
    int i, initcnt, node = first_online_node;
    struct irq_desc *desc;

    ...

    /* Let arch update nr_irqs and return the nr of preallocated irqs */
    initcnt = arch_probe_nr_irqs();
    printk(KERN_INFO "NR_IRQS:%d nr_irqs:%d %d\n", NR_IRQS, nr_irqs, initcnt);

    if (WARN_ON(nr_irqs > IRQ_BITMAP_BITS))
        nr_irqs = IRQ_BITMAP_BITS;

    if (WARN_ON(initcnt > IRQ_BITMAP_BITS))
        initcnt = IRQ_BITMAP_BITS;

    if (initcnt > nr_irqs)
        nr_irqs = initcnt;

    /* 预先分配512个desc,同时设置位图，将这些desc插入基数树或者数组 */
    for (i = 0; i < initcnt; i++) {
        desc = alloc_desc(i, node, NULL);
        set_bit(i, allocated_irqs);
        irq_insert_desc(i, desc);
    }
    return arch_early_irq_init();
}
```

```
void __init init_IRQ(void)
{
    /* DT_MACHINE_START(SOCFPGA, "Altera SOCFPGA") */
    machine_desc->init_irq();   /* 调用gic_init_irq */
}

static void __init gic_init_irq(void)
{
    of_irq_init(irq_match);     /* 匹配设备树成功后调用irq_match中的gic_of_init */
    ...
}

/* 被gic_init_irq使用 */
const static struct of_device_id irq_match[] = {
    { .compatible = "arm,cortex-a9-gic", .data = gic_of_init, },
    {}
};

/* 设备树 */
/include/ "skeleton.dtsi"
```

```
/ {
    #address-cells = <1>;
    #size-cells = <1>;

    ...

    intc: intc@fffed000 {
        compatible = "arm,cortex-a9-gic";
        #interrupt-cells = <3>;
        interrupt-controller;
        reg = <0xfffed000 0x1000>,
              <0xfffec100 0x100>;
    };
    ...
}
```

gic_init_irq又会调用of_irq_init。

```
void __init of_irq_init(const struct of_device_id *matches)
{
    struct device_node *np, *parent = NULL;
    struct intc_desc *desc, *temp_desc;
    struct list_head intc_desc_list, intc_parent_list;

    INIT_LIST_HEAD(&intc_desc_list);    /* 初始化中断控制器链表 */
    INIT_LIST_HEAD(&intc_parent_list);

    /* 对于和irq_match中的compatible属性"arm,cortex-a9-gic"匹配的每一个设备树节点，
     * 在socfpga设备树中只有一个节点符合,如上所示，该节点就是描述gic控制器的节点，
     * 创建中断控制器描述符，设置里面的设备树节点和父中断控制器节点然后加入链表
    */
    for_each_matching_node(np, matches) {
        if (!of_find_property(np, "interrupt-controller", NULL))
            continue;
        /*
         * Here, we allocate and populate an intc_desc with the node
         * pointer, interrupt-parent device_node etc.
         */
        desc = kzalloc(sizeof(*desc), GFP_KERNEL);
        if (WARN_ON(!desc))
            goto err;

        desc->dev = np;     /* 保存该中断控制器设备树节点 */
        desc->interrupt_parent = of_irq_find_parent(np); // 保存该中断控制器的父中断控制器节点
        if (desc->interrupt_parent == np)   //根中断控制器的interrupt_parent为NULL
            desc->interrupt_parent = NULL;
        list_add_tail(&desc->list, &intc_desc_list);    /* 将该中断控制器加入中断控制器链表 */
    }

    /*
     * The root irq controller is the one without an interrupt-parent.
     * That one goes first, followed by the controllers that reference it,
```

```
     * followed by the ones that reference the 2nd level controllers, etc.
     */
    /* 从intc_desc_list中取出中断控制器出来处理，首先取出的是根中断控制器 */
    while (!list_empty(&intc_desc_list)) {
        /*
         * Process all controllers with the current 'parent'.
         * First pass will be looking for NULL as the parent.
         * The assumption is that NULL parent means a root controller.
         */
        list_for_each_entry_safe(desc, temp_desc, &intc_desc_list, list) {
            const struct of_device_id *match;
            int ret;
            of_irq_init_cb_t irq_init_cb;

            if (desc->interrupt_parent != parent)
                continue;

            list_del(&desc->list);  //从intc_desc_list中删除当前控制器
            /* 匹配设备树节点和matches的compatile,
             * 匹配成功后match = matches
            */
            match = of_match_node(matches, desc->dev);
            if (WARN(!match->data,
                "of_irq_init: no init function for %s\n",
                match->compatible)) {
                kfree(desc);
                continue;
            }

            pr_debug("of_irq_init: init %s @ %p, parent %p\n",
                 match->compatible,
                 desc->dev, desc->interrupt_parent);
            irq_init_cb = (of_irq_init_cb_t)match->data; // 将match中的data即gic_of_init
            ret = irq_init_cb(desc->dev, desc->interrupt_parent);   /* 调用gic_of_init */
            if (ret) {
                kfree(desc);
                continue;
            }

            /*
             * This one is now set up; add it to the parent list so
             * its children can get processed in a subsequent pass.
             */
            list_add_tail(&desc->list, &intc_parent_list);
        }

        /* Get the next pending parent that might have children */
        desc = list_first_entry(&intc_parent_list, typeof(*desc), list);
        if (list_empty(&intc_parent_list) || !desc) {
            pr_err("of_irq_init: children remain, but no parents\n");
            break;
        }
        list_del(&desc->list);
```

```
        parent = desc->dev;
        kfree(desc);
    }

    list_for_each_entry_safe(desc, temp_desc, &intc_parent_list, list) {
        list_del(&desc->list);
        kfree(desc);
    }
err:
    list_for_each_entry_safe(desc, temp_desc, &intc_desc_list, list) {
        list_del(&desc->list);
        kfree(desc);
    }
}
```

of_irq_init最终会通过调用gic_of_init进行终端子系统的核心初始化。

```
int __init gic_of_init(struct device_node *node, struct device_node *parent)
{
    void __iomem *cpu_base;
    void __iomem *dist_base;
    u32 percpu_offset;
    int irq;

    if (WARN_ON(!node))
        return -ENODEV;

    /* 获取分发器的基地址 */
    dist_base = of_iomap(node, 0);
    WARN(!dist_base, "unable to map gic dist registers\n");

    /* 获取cpu接口基地址 */
    cpu_base = of_iomap(node, 1);
    WARN(!cpu_base, "unable to map gic cpu registers\n");

    if (of_property_read_u32(node, "cpu-offset", &percpu_offset))
        percpu_offset = 0;

    /* 核心初始化 */
    gic_init_bases(gic_cnt, -1, dist_base, cpu_base, percpu_offset, node);

    /* 不太可能运行 */
    if (parent) {
        irq = irq_of_parse_and_map(node, 0);
        gic_cascade_irq(gic_cnt, irq);
    }
    gic_cnt++;
    return 0;
}
```

gic_of_init的核心是gic_init_bases。

```
void __init gic_init_bases(unsigned int gic_nr, int irq_start,
               void __iomem *dist_base, void __iomem *cpu_base,
               u32 percpu_offset, struct device_node *node)
{
    irq_hw_number_t hwirq_base;
    struct gic_chip_data *gic;
    int gic_irqs, irq_base;

    BUG_ON(gic_nr >= MAX_GIC_NR);
    printk("gic_nr=%d\n", gic_nr);

    /* 取出gic_chip_data,并根据参数设置里面的值 */
    gic = &gic_data[gic_nr];
    ...
    {           /* Normal, sane GIC... */
        WARN(percpu_offset,
             "GIC_NON_BANKED not enabled, ignoring %08x offset!",
             percpu_offset);
        gic->dist_base.common_base = dist_base;
        gic->cpu_base.common_base = cpu_base;
        gic_set_base_accessor(gic, gic_get_common_base);
    }

    /*
     * For primary GICs, skip over SGIs.
     * For secondary GICs, skip over PPIs, too.
     */
    if (gic_nr == 0 && (irq_start & 31) > 0) {
        hwirq_base = 16;
        if (irq_start != -1)
            irq_start = (irq_start & ~31) + 16;
    } else {
        hwirq_base = 32;
    }
/*
***************************************************************************
*                                     Embest Tech co., ltd
*                                    www.embest-tech.com
***************************************************************************
*
*chage the IPI interrupt property, cause the another cpu not secheule the linux thread any
more.
*/
    /* 在amp中将hwirq_base从新设为0，因为sgi不再使用hand_ipi，而当做普通中断一样注册使用 */
    hwirq_base = 0;

    /*
     * Find out how many interrupts are supported.
     * The GIC only supports up to 1020 interrupt sources.
     */
    /* 读取硬件得到支持的中断源数目 */
    gic_irqs = readl_relaxed(gic_data_dist_base(gic) + GIC_DIST_CTR) & 0x1f;
    gic_irqs = (gic_irqs + 1) * 32;
```

```
    if (gic_irqs > 1020)
        gic_irqs = 1020;
    gic->gic_irqs = gic_irqs;

    /* 修正gic_irqs为可以注册的中断个数，
     * 在smp中需要减去16个sgi，因为它们不是注册使用而是handipi处理，
     * 而在amp中，不需要该，因为sgi当普通的一样使用
    */
    gic_irqs -= hwirq_base; /* calculate # of irqs to allocate */
    /* 申请gic_irqs个desc并设置位图，这些是直接连接gic的中断，
     * 返回第一个虚拟中断号irq,
     * 个人感觉会返回512，因为在early_irq_init中预先分配了512个
    */
    irq_base = irq_alloc_descs(irq_start, 0, gic_irqs, numa_node_id());
    if (IS_ERR_VALUE(irq_base)) {
        WARN(1, "Cannot allocate irq_descs @ IRQ%d, assuming pre-allocated\n",
             irq_start);
        irq_base = irq_start;
    }
    printk("gic_irqs=%d irq_base=%d hwirq_base=%d\n", gic_irqs, irq_base, (u32)hwirq_base);

    /* 创建并注册gic的domain，并将gic_irq_domain_ops绑定该domain，
     * 同时调用gic_irq_domain_ops中的gic_irq_domain_map将irq_base开始的
     * gic_irqs个desc设置流控函数和irq_chip,同时映射硬件中断号和虚拟中断号
    */
    gic->domain = irq_domain_add_legacy(node, gic_irqs, irq_base,
                hwirq_base, &gic_irq_domain_ops, gic);
    if (WARN_ON(!gic->domain))
        return;

    gic_chip.flags |= gic_arch_extn.flags;
    /* 初始化分发器，在smp中由启动核初始化，但在amp中bm已经初始化，
     * 所以这里基本不做什么，只设置了一个分发器锁
    */
    gic_dist_init(gic);
    gic_cpu_init(gic);  //初始化cpu接口
    gic_pm_init(gic);
}
```

```
struct irq_domain *irq_domain_add_legacy(struct device_node *of_node,
                     unsigned int size,
                     unsigned int first_irq,
                     irq_hw_number_t first_hwirq,
                     const struct irq_domain_ops *ops,
                     void *host_data)
{
    struct irq_domain *domain;
    unsigned int i;

    /* 分配一个domain并设置revmap_type为IRQ_DOMAIN_MAP_LEGACY */
    domain = irq_domain_alloc(of_node, IRQ_DOMAIN_MAP_LEGACY, ops, host_data);
    if (!domain)
```

```
        return NULL;

    /* 硬件中断号和虚拟中断号相关联(映射) */
    domain->revmap_data.legacy.first_irq = first_irq;
    domain->revmap_data.legacy.first_hwirq = first_hwirq;
    domain->revmap_data.legacy.size = size;

    ...

    /*
     * 为first_irq(first_hwirq)开始的size个中断的desc设置流控层和irq_chip
    */
    for (i = 0; i < size; i++) {
        int irq = first_irq + i;
        int hwirq = first_hwirq + i;

        /* IRQ0 gets ignored */
        if (!irq)
            continue;

        /* Legacy flags are left to default at this point,
         * one can then use irq_create_mapping() to
         * explicitly change them
         */
        if (ops->map)
            ops->map(domain, irq, hwirq);   /* 调用gic_irq_domain_map设置desc的流控层和
irq_chip， 这里没有映射硬件中断号和虚拟中断号，                                     在前面
legacy方法映射，在4.0版本内核是
gic_irq_domain_map中映射     */

        /* Clear norequest flags */
        irq_clear_status_flags(irq, IRQ_NOREQUEST);
    }

    irq_domain_add(domain);
    return domain;
}
```

```
static int gic_irq_domain_map(struct irq_domain *d, unsigned int irq,
                irq_hw_number_t hw)
{
/*
***************************************************************************
*                                       Embest Tech co., ltd
*                                      www.embest-tech.com
***************************************************************************
*
* chage the IPI interrupt property, because the another cpu not secheule the linux thread
any more.
*
*/
    if ((hw < 32)&& (hw > 15)) {    //本来是(hw < 32)
        irq_set_percpu_devid(irq);
```

```
        irq_set_chip_and_handler(irq, &gic_chip,
                    handle_percpu_devid_irq);  /* 设置ppi的流控层函数 */
        set_irq_flags(irq, IRQF_VALID | IRQF_NOAUTOEN);
    } else {
        irq_set_chip_and_handler(irq, &gic_chip,
                    handle_fasteoi_irq);   /* 设置spi和sgi的流控层函数 */
        set_irq_flags(irq, IRQF_VALID | IRQF_PROBE);
    }
    irq_set_chip_data(irq, d->host_data);
    return 0;
}
```

三、中断解析映射

中断解析映射是通过irq_of_parse_and_map来实现的，它先通过of_irq_map_one解析设备结点的中断相关属性，然后通过解析设备结点的中断相关属性来分配desc(gic的映射不需要分配，因为前面分配了)并映射虚拟中断号和硬件中断号。

中断解析映射是注册中断的前一个手工步骤，但是在amp中注册sgi15不需要手工映射，因为irq_domain_add_legacy中已经为包括sgi在内的所有直接gic中断映射了。当然我们还是可以为了规范在注册前手工映射一遍，但是在手工映射过程中irq_find_mapping会发现已经映射，所以会直接返回虚拟中断号。但是我们并没有在设备树中找到sgi15，因此我们就无法满足这种规范，根据代码分析中直接知道了sgi15对应的虚拟中断号为512+15(对于所有的直接gic中断都是512+hirq)，所以直接request_irq(512+15,...)。

```
/**
 * irq_of_parse_and_map - Parse and map an interrupt into linux virq space
 * @device: Device node of the device whose interrupt is to be mapped
 * @index: Index of the interrupt to map
 *
 * This function is a wrapper that chains of_irq_map_one() and
 * irq_create_of_mapping() to make things easier to callers
 */
unsigned int irq_of_parse_and_map(struct device_node *dev, int index)
{
    struct of_irq oirq;

    /* 解析设备结点的中断相关属性 */
    if (of_irq_map_one(dev, index, &oirq))
        return 0;

    /* 分配desc，但gic的不用，因为前面事先分配了
     * 映射虚拟中断号和硬件中断号
    */
    return irq_create_of_mapping(oirq.controller, oirq.specifier,
                    oirq.size);
}
```

of_irq_map_one就不分析了，直接分析irq_create_of_mapping。

```
unsigned int irq_create_of_mapping(struct device_node *controller,
                   const u32 *intspec, unsigned int intsize)
{
    struct irq_domain *domain;
    irq_hw_number_t hwirq;
    unsigned int type = IRQ_TYPE_NONE;
    unsigned int virq;

    domain = controller ? irq_find_host(controller) : irq_default_domain;
    ...

    /* If domain has no translation, then we assume interrupt line */
    if (domain->ops->xlate == NULL)
        hwirq = intspec[0];
    else {
        /* 调用gic_irq_domain_xlate解析设备树中的硬件中断号和类型
         * 放在out_hwirq和out_type中
        */
        if (domain->ops->xlate(domain, controller, intspec, intsize,
                     &hwirq, &type))
            return 0;
    }

    /* Create mapping */
    /* 先查找映射，如果找到返回虚拟中断号，
     * 如果没有则创建映射并返回虚拟中断号,
     * 创建映射的过程有可能分配desc，取决于
     * domian类型是否为leacy，如果为leacy则
     * 不需要分配，因为事先分配了，如果不是leacy
     * 则需分配
    */
    virq = irq_create_mapping(domain, hwirq);
    if (!virq)
        return virq;

    /* Set type if specified and different than the current one */
    if (type != IRQ_TYPE_NONE &&
        type != (irqd_get_trigger_type(irq_get_irq_data(virq))))
        irq_set_irq_type(virq, type);   /* 如果情况符合就设置中断类型 */
    return virq;
}
```

irq_create_of_mapping首先通过gic_irq_domain_xlate解析出硬件中断号和类型，然后通过irq_create_mapping进行实际的映射工作。

```
unsigned int irq_create_mapping(struct irq_domain *domain,
                irq_hw_number_t hwirq)
{
    unsigned int hint;
    int virq;

    ...
```

```
    /* Check if mapping already exists */
    /* 先查找映射是否事先存在，
     * 对于直接gic的irq，在irq_domain_add_legacy中已经映射
    */
    virq = irq_find_mapping(domain, hwirq);
    if (virq) {
        pr_debug("-> existing mapping on virq %d\n", virq);
        return virq;
    }

    /* Get a virtual interrupt number */
    /* 本工程的gic domain revmap_type为IRQ_DOMAIN_MAP_LEGACY，
     * 但是已经在上一步返回了，这里的目的是留给那些别的revmap_type
     * 同样为IRQ_DOMAIN_MAP_LEGACY的domain的irq
     */
    if (domain->revmap_type == IRQ_DOMAIN_MAP_LEGACY)
        return irq_domain_legacy_revmap(domain, hwirq);

    /* Allocate a virtual interrupt number */
    /* 对于没有事先映射且不是IRQ_DOMAIN_MAP_LEGACY的才会进行下面的分配desc */
    hint = hwirq % nr_irqs;
    if (hint == 0)
        hint++;
    virq = irq_alloc_desc_from(hint, of_node_to_nid(domain->of_node));
    ...

    return virq;
}
```

四、中断注册

中断注册通常有两种接口，一种是request_irq，表示非线程化的接口，另一种是request_threaded_irq，表示可线程化的接口，其中request_irq也是调用了request_threaded_irq，只不过在第三个参数中传入NULL表示非线程化。

```
request_irq(unsigned int irq, irq_handler_t handler, unsigned long flags,
        const char *name, void *dev)
{
    return request_threaded_irq(irq, handler, NULL, flags, name, dev);
}
```

```
int request_threaded_irq(unsigned int irq, irq_handler_t handler,
             irq_handler_t thread_fn, unsigned long irqflags,
             const char *devname, void *dev_id)
{
    struct irqaction *action;
    struct irq_desc *desc;
    int retval;
```

```
    /* 如果是共享中断则dev_id不能为null */
    if ((irqflags & IRQF_SHARED) && !dev_id)
        return -EINVAL;

    desc = irq_to_desc(irq);    /* 获取irq的desc */
    if (!desc)
        return -EINVAL;

/*
**************************************************************************
*                                    Embest Tech co., ltd
*                                   www.embest-tech.com
**************************************************************************
*
*make IPI can request ugly, shoud be modify to check desc->hwirq
*/
    /* 在amp中加了这一句，强制使得这个irq为可以request */
    irq_settings_clr_norequest(desc);

    /* 如果是norequest或者为私有cpu的irq则返回错误码，
     * 在smp中sgi和ppi都设置_IRQ_PER_CPU_DEVID标记，
     * 但amp中只设置ppi不设置sgi，因此sgi在这里会通过
    */
    if (!irq_settings_can_request(desc) ||
        WARN_ON(irq_settings_is_per_cpu_devid(desc)))
        return -EINVAL;

    /* handler和thread_fn不能同时为NULL，
     * handler为NULL，thread_fn不为空时会设置默认handler，
     * 会简单返回IRQ_WAKE_THREAD表示唤醒注册的中断线程
    */
    if (!handler) {
        if (!thread_fn)
            return -EINVAL;
        handler = irq_default_primary_handler;
    }

    /* 分配和设置action */
    action = kzalloc(sizeof(struct irqaction), GFP_KERNEL);
    if (!action)
        return -ENOMEM;
    action->handler = handler;
    action->thread_fn = thread_fn;
    action->flags = irqflags;
    action->name = devname;
    action->dev_id = dev_id;

    chip_bus_lock(desc);
    retval = __setup_irq(irq, desc, action);    /* 大部分工作 */
    chip_bus_sync_unlock(desc);

    if (retval)
        kfree(action);
```

```
    ...
    return retval;
}
```

在]__setup_irq中继续做剩下的大部分注册工作。

```
static int
__setup_irq(unsigned int irq, struct irq_desc *desc, struct irqaction *new)
{
    struct irqaction *old, **old_ptr;
    unsigned long flags, thread_mask = 0;
    int ret, nested, shared = 0;
    cpumask_var_t mask;

    if (!desc)
        return -EINVAL;

    /* 在gic_irq_domain_map中设置为其他的，其他控制器的应该也类似
     * 在创建注册domain时调用自身的map，在里面设置irq_data.chip
    */
    if (desc->irq_data.chip == &no_irq_chip)
        return -ENOSYS;
    if (!try_module_get(desc->owner))
        return -ENODEV;

    /*
     * Check whether the interrupt nests into another interrupt
     * thread.
     */
    /* 检测中断是否可以嵌套，这里的嵌套是在执行中断线程的过程中被同类型的中断抢占，
     * 如果可以嵌套，则handler被覆盖为
     * irq_nested_primary_handler，这个函数只打印一句话并返回IRQ_NONE，
     *  所以如果发生嵌套不会重新执行thread_fn，
     * 如果不可以嵌套，先通过irq_settings_can_thread初步判断是否可以线程化，
     * 如果可以再通过irq_setup_forced_threading进一步判断和设置中断线程化
    */
    nested = irq_settings_is_nested_thread(desc);
    if (nested) {   //嵌套的中断线程化
        if (!new->thread_fn) {
            ret = -EINVAL;
            goto out_mput;
        }
        /*
         * Replace the primary handler which was provided from
         * the driver for non nested interrupt handling by the
         * dummy function which warns when called.
         */
        new->handler = irq_nested_primary_handler;
    } else {    //非嵌套的
        if (irq_settings_can_thread(desc))
            irq_setup_forced_threading(new);
    }
```

```
/*
 * Create a handler thread when a thread function is supplied
 * and the interrupt does not nest into another interrupt
 * thread.
 */
/* 对于非嵌套的中断创建一个内核线程task_struct，并赋值给new */
if (new->thread_fn && !nested) {
    struct task_struct *t;
    /* 创建一个内核线程，线程函数为irq_thread，irq_thread会取出new的thread来执行 */
    t = kthread_create(irq_thread, new, "irq/%d-%s", irq,
               new->name);
    if (IS_ERR(t)) {
        ret = PTR_ERR(t);
        goto out_mput;
    }
    /*
     * We keep the reference to the task struct even if
     * the thread dies to avoid that the interrupt code
     * references an already freed task_struct.
     */
    get_task_struct(t);
    new->thread = t;
}

if (!alloc_cpumask_var(&mask, GFP_KERNEL)) {
    ret = -ENOMEM;
    goto out_thread;
}

/*
 * Drivers are often written to work w/o knowledge about the
 * underlying irq chip implementation, so a request for a
 * threaded irq without a primary hard irq context handler
 * requires the ONESHOT flag to be set. Some irq chips like
 * MSI based interrupts are per se one shot safe. Check the
 * chip flags, so we can avoid the unmask dance at the end of
 * the threaded handler for those.
 */
/* 如果中断控制器不支持嵌套，根据我以前的测试gic就不支持，只支持不同的中断的抢占，
 * 那就将new->flags的IRQF_ONESHOT关掉，不需要了
*/
if (desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)
    new->flags &= ~IRQF_ONESHOT;

/*
 * The following block of code has to be executed atomically
 */
raw_spin_lock_irqsave(&desc->lock, flags);
old_ptr = &desc->action;    //old_ptr存放desc->action的地址
old = *old_ptr;             //old存放desc->action
/* 如果old不为null说明本次注册之前就有别的注册，说明是个共享中断 */
if (old) {
```

```
    /*
     * Can't share interrupts unless both agree to and are
     * the same type (level, edge, polarity). So both flag
     * fields must have IRQF_SHARED set and the bits which
     * set the trigger type must match. Also all must
     * agree on ONESHOT.
     */
    if (!((old->flags & new->flags) & IRQF_SHARED) ||
        ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK) ||
        ((old->flags ^ new->flags) & IRQF_ONESHOT))
        goto mismatch;

    /* All handlers must agree on per-cpuness */
    if ((old->flags & IRQF_PERCPU) !=
        (new->flags & IRQF_PERCPU))
        goto mismatch;

    /* add new interrupt at end of irq queue */
    /* 循环处理所有该desc的action */
    do {
        /*
         * Or all existing action->thread_mask bits,
         * so we can find the next zero bit for this
         * new action.
         */
        //thread_mask累积所有action的thread_mask
        thread_mask |= old->thread_mask;
        old_ptr = &old->next;
        old = *old_ptr;
    } while (old);
    shared = 1;
}

/*
 * Setup the thread mask for this irqaction for ONESHOT. For
 * !ONESHOT irqs the thread mask is 0 so we can avoid a
 * conditional in irq_wake_thread().
 */
if (new->flags & IRQF_ONESHOT) {
    /*
     * Unlikely to have 32 resp 64 irqs sharing one line,
     * but who knows.
     */
    if (thread_mask == ~0UL) {
        ret = -EBUSY;
        goto out_mask;
    }
    /*
     * The thread_mask for the action is or'ed to
     * desc->thread_active to indicate that the
     * IRQF_ONESHOT thread handler has been woken, but not
     * yet finished. The bit is cleared when a thread
     * completes. When all threads of a shared interrupt
```

```
         * line have completed desc->threads_active becomes
         * zero and the interrupt line is unmasked. See
         * handle.c:irq_wake_thread() for further information.
         *
         * If no thread is woken by primary (hard irq context)
         * interrupt handlers, then desc->threads_active is
         * also checked for zero to unmask the irq line in the
         * affected hard irq flow handlers
         * (handle_[fasteoi|level]_irq).
         *
         * The new action gets the first zero bit of
         * thread_mask assigned. See the loop above which or's
         * all existing action->thread_mask bits.
         */
        new->thread_mask = 1 << ffz(thread_mask);

    } else if (new->handler == irq_default_primary_handler &&
           !(desc->irq_data.chip->flags & IRQCHIP_ONESHOT_SAFE)) {
        /*
         * The interrupt was requested with handler = NULL, so
         * we use the default primary handler for it. But it
         * does not have the oneshot flag set. In combination
         * with level interrupts this is deadly, because the
         * default primary handler just wakes the thread, then
         * the irq lines is reenabled, but the device still
         * has the level irq asserted. Rinse and repeat....
         *
         * While this works for edge type interrupts, we play
         * it safe and reject unconditionally because we can't
         * say for sure which type this interrupt really
         * has. The type flags are unreliable as the
         * underlying chip implementation can override them.
         */
        pr_err("Threaded irq requested with handler=NULL and !ONESHOT for irq %d\n",
               irq);
        ret = -EINVAL;
        goto out_mask;
    }


    if (!shared) {
        init_waitqueue_head(&desc->wait_for_threads);

        /* Setup the type (level, edge polarity) if configured: */
        if (new->flags & IRQF_TRIGGER_MASK) {
            ret = __irq_set_trigger(desc, irq,
                    new->flags & IRQF_TRIGGER_MASK);

            if (ret)
                goto out_mask;
        }

        desc->istate &= ~(IRQS_AUTODETECT | IRQS_SPURIOUS_DISABLED | \
```

```
                IRQS_ONESHOT | IRQS_WAITING);
        irqd_clear(&desc->irq_data, IRQD_IRQ_INPROGRESS);

        if (new->flags & IRQF_PERCPU) {
            irqd_set(&desc->irq_data, IRQD_PER_CPU);
            irq_settings_set_per_cpu(desc);
        }

        if (new->flags & IRQF_ONESHOT)
            desc->istate |= IRQS_ONESHOT;

        if (irq_settings_can_autoenable(desc))
            irq_startup(desc, true);
        else
            /* Undo nested disables: */
            desc->depth = 1;

        /* Exclude IRQ from balancing if requested */
        if (new->flags & IRQF_NOBALANCING) {
            irq_settings_set_no_balancing(desc);
            irqd_set(&desc->irq_data, IRQD_NO_BALANCING);
        }

        /* Set default affinity mask once everything is setup */
        setup_affinity(irq, desc, mask);

    } else if (new->flags & IRQF_TRIGGER_MASK) {
        unsigned int nmsk = new->flags & IRQF_TRIGGER_MASK;
        unsigned int omsk = irq_settings_get_trigger_mask(desc);

        if (nmsk != omsk)
            /* hope the handler works with current  trigger mode */
            pr_warning("irq %d uses trigger mode %u; requested %u\n",
                   irq, nmsk, omsk);
    }

    /* 将new插入desc的action链表 */
    new->irq = irq;
    *old_ptr = new;

    /* Reset broken irq detection when installing new handler */
    desc->irq_count = 0;
    desc->irqs_unhandled = 0;

    /*
     * Check whether we disabled the irq via the spurious handler
     * before. Reenable it and give it another chance.
     */
    if (shared && (desc->istate & IRQS_SPURIOUS_DISABLED)) {
        desc->istate &= ~IRQS_SPURIOUS_DISABLED;
        __enable_irq(desc, irq, false);
    }
```

```
    raw_spin_unlock_irqrestore(&desc->lock, flags);

    /*
     * Strictly no need to wake it up, but hung_task complains
     * when no hard interrupt wakes the thread up.
     */
    if (new->thread)
        wake_up_process(new->thread);   /* 唤醒中断线程 */

    register_irq_proc(irq, desc);
    new->dir = NULL;
    register_handler_proc(irq, new);
    free_cpumask_var(mask);

    return 0;

mismatch:
    if (!(new->flags & IRQF_PROBE_SHARED)) {
        pr_err("Flags mismatch irq %d. %08x (%s) vs. %08x (%s)\n",
               irq, new->flags, new->name, old->flags, old->name);
#ifdef CONFIG_DEBUG_SHIRQ
        dump_stack();
#endif
    }
    ret = -EBUSY;

out_mask:
    raw_spin_unlock_irqrestore(&desc->lock, flags);
    free_cpumask_var(mask);

out_thread:
    if (new->thread) {
        struct task_struct *t = new->thread;

        new->thread = NULL;
        kthread_stop(t);
        put_task_struct(t);
    }
out_mput:
    module_put(desc->owner);
    return ret;
}
```

```
static void irq_setup_forced_threading(struct irqaction *new)
{
    /* 进一步判断是否可以线程化 */
    if (!force_irqthreads)
        return;
    if (new->flags & (IRQF_NO_THREAD | IRQF_PERCPU | IRQF_ONESHOT))
        return;

    /*  */
    new->flags |= IRQF_ONESHOT;
```

```
    /* 如果可以并要强制线程化，并且注册时不提供thread_fn，
     * 则将handler移到thread_fn，将handler设置为irq_default_primary_handler
    */
    if (!new->thread_fn) {
        set_bit(IRQTF_FORCED_THREAD, &new->thread_flags);
        new->thread_fn = new->handler;
        new->handler = irq_default_primary_handler;
    }
}
```

五、各环节整合