快速链接:

.

👉👉👉 ARMv8/ARMv9架构入门到精通-[目录] 👈👈👈

- 付费专栏-付费课程 **【购买须知】:**
- 联系方式-加入交流群 ----**联系方式-加入交流群**
- 个人博客笔记导读目录(全部)

引流关键词:armv8, armv9, gic,gicv2,gicv3,异常, 中断, irq,fiq,serror,sync, 同步异常, 异步异常, 向量表, 向量表基地址, VBAR, vbar_el3, 中断嵌套, 中断级联, Linux Kernel, optee,ATF,TF-A,optee,hypervisor, SPM

目录

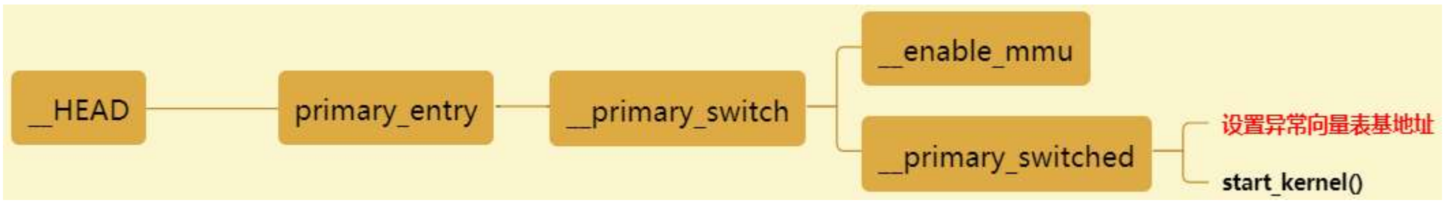
1 Linux Kernel arm64中断向量表的定义

```
(linux/arch/arm64/kernel/entry.S)

/*
 * Exception vectors.
 */
    .pushsection ".entry.text", "ax"

    .align  11
SYM_CODE_START(vectors)
    kernel_ventry   1, sync_invalid         // Synchronous EL1t
    kernel_ventry   1, irq_invalid          // IRQ EL1t
    kernel_ventry   1, fiq_invalid          // FIQ EL1t
    kernel_ventry   1, error_invalid        // Error EL1t

    kernel_ventry   1, sync             // Synchronous EL1h
    kernel_ventry   1, irq              // IRQ EL1h
    kernel_ventry   1, fiq              // FIQ EL1h
    kernel_ventry   1, error            // Error EL1h

    kernel_ventry   0, sync             // Synchronous 64-bit EL0
    kernel_ventry   0, irq              // IRQ 64-bit EL0
    kernel_ventry   0, fiq              // FIQ 64-bit EL0
    kernel_ventry   0, error            // Error 64-bit EL0

#ifdef CONFIG_COMPAT
    kernel_ventry   0, sync_compat, 32      // Synchronous 32-bit EL0
    kernel_ventry   0, irq_compat, 32       // IRQ 32-bit EL0
    kernel_ventry   0, fiq_compat, 32       // FIQ 32-bit EL0
    kernel_ventry   0, error_compat, 32     // Error 32-bit EL0
#else
    kernel_ventry   0, sync_invalid, 32     // Synchronous 32-bit EL0
    kernel_ventry   0, irq_invalid, 32      // IRQ 32-bit EL0
    kernel_ventry   0, fiq_invalid, 32      // FIQ 32-bit EL0
    kernel_ventry   0, error_invalid, 32        // Error 32-bit EL0
#endif
SYM_CODE_END(vectors)
```

思考：

1、这里有没有按照armv8定义的异常向量表排列？不是每一个offset只有128bytes地址空间吗，如何做到的？
2、Linux Kernel arm64体系中不是没有实现FIQ吗，这里为何实现了？
3、第一组异常向量为何没有实现？

2 Linux Kernel arm64设置中断向量表的基地址



```
(linux/arch/arm64/kernel/head.S)

SYM_FUNC_START_LOCAL(__primary_switched)
    adrp    x4, init_thread_union
    add sp, x4, #THREAD_SIZE
    adr_l   x5, init_task
    msr sp_el0, x5          // Save thread_info

    adr_l   x8, vectors         // load VBAR_EL1 with virtual
    msr vbar_el1, x8            // vector table address
    isb

......
    b   start_kernel
SYM_FUNC_END(__primary_switched)
```

> **思考:**
>
> 1、设置VBAR_EL1，如果系统系统里有8个ARM Core，那么8个Core都需要设置吗，分别如何设置的?

3 kernel_ventry宏的介绍

```
(linux/arch/arm64/kernel/entry.S)

    .macro kernel_ventry, el, label, regsize = 64
    .align 7
#ifdef CONFIG_UNMAP_KERNEL_AT_EL0
    .if \el == 0
alternative_if ARM64_UNMAP_KERNEL_AT_EL0
    .if \regsize == 64
    mrs x30, tpidrro_el0
    msr tpidrro_el0, xzr
    .else
    mov x30, xzr
    .endif
alternative_else_nop_endif
    .endif
#endif

    sub sp, sp, #PT_REGS_SIZE
#ifdef CONFIG_VMAP_STACK
    /*
     * Test whether the SP has overflowed, without corrupting a GPR.
     * Task and IRQ stacks are aligned so that SP & (1 << THREAD_SHIFT)
     * should always be zero.
     */
    add sp, sp, x0          // sp' = sp + x0
    sub x0, sp, x0          // x0' = sp' - x0 = (sp + x0) - x0 = sp
    tbnz    x0, #THREAD_SHIFT, 0f
    sub x0, sp, x0          // x0'' = sp' - x0' = (sp + x0) - sp = x0
    sub sp, sp, x0          // sp'' = sp' - x0 = (sp + x0) - x0 = sp
    b   el\()\el\()_\label

```

```
0:
    /*
     * Either we've just detected an overflow, or we've taken an exception
     * while on the overflow stack. Either way, we won't return to
     * userspace, and can clobber EL0 registers to free up GPRs.
     */

    /* Stash the original SP (minus PT_REGS_SIZE) in tpidr_el0. */
    msr tpidr_el0, x0

    /* Recover the original x0 value and stash it in tpidrro_el0 */
    sub x0, sp, x0
    msr tpidrro_el0, x0

    /* Switch to the overflow stack */
    adr_this_cpu sp, overflow_stack + OVERFLOW_STACK_SIZE, x0

    /*
     * Check whether we were already on the overflow stack. This may happen
     * after panic() re-enables interrupts.
     */
    mrs x0, tpidr_el0           // sp of interrupted context
    sub x0, sp, x0          // delta with top of overflow stack
    tst x0, #~(OVERFLOW_STACK_SIZE - 1) // within range?
    b.ne    __bad_stack         // no? -> bad stack pointer

    /* We were already on the overflow stack. Restore sp/x0 and carry on. */
    sub sp, sp, x0
    mrs x0, tpidrro_el0
#endif
    b   el\()\el\()_\label
    .endm
```

注意.align=7，说明该段代码是以2^7=128字节对其的，这和向量表中每一个offset的大小是一致的
代码看似非常复杂，其实最终跳转到了 `b el\()\el\()_\label`，翻译一下，其实就是跳转到了如下这样的函数中

```
el1_sync_invalid
el1_irq_invalid
el1_fiq_invalid
el1_error_invalid

el1_sync
el1_irq
el1_fiq
el1_error
```

```
el0_sync
el0_irq
el0_fiq
el0_error
```

4 未实现的异常向量: elx_yyy_invalid

未实现的向量定义为了elx_yyy_invalid函数, 该invalid函数其实也是一种实现，它最终调用了panic函数
例如el1_irq_invalid的Flow： el1_irq_invalid --> `bl bad_mode` --> panic(“bad mode”)

```
SYM_CODE_START_LOCAL(el1_irq_invalid)
    inv_entry 1, BAD_IRQ
SYM_CODE_END(el1_irq_invalid)


/*
 * Bad Abort numbers
 *-----------------
 */
#define BAD_SYNC    0
#define BAD_IRQ     1
#define BAD_FIQ     2
#define BAD_ERROR   3

/*
 * Invalid mode handlers
 */
    .macro  inv_entry, el, reason, regsize = 64
    kernel_entry \el, \regsize
    mov x0, sp
    mov x1, #\reason
    mrs x2, esr_el1
    bl  bad_mode
    ASM_BUG()
    .endm
```

```
 /*
  * bad_mode handles the impossible case in the exception vector. This is always
  * fatal.
  */
 asmlinkage void notrace bad_mode(struct pt_regs *regs, int reason, unsigned int esr
 {
    arm64_enter_nmi(regs);

    console_verbose();

    pr_crit("Bad mode in %s handler detected on CPU%d, code 0x%08x -- %s\n",
        handler[reason], smp_processor_id(), esr,
        esr_get_class_string(esr));

    __show_regs(regs);
    local_daif_mask();
    panic("bad mode");
}
```

5 el1_irq的介绍 - 跳转到注册的handler函数

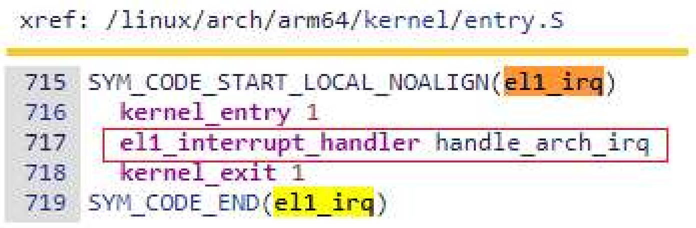
抛开事务看本质， `el1_interrupt_handler handle_arch_irq` 其实就是调用**handle_arch_irq**， 而handle_arch_irq指向**irq-gic-v3.c**中定义的handler函数

```
    .align  6
SYM_CODE_START_LOCAL_NOALIGN(el1_irq)
    kernel_entry 1
    el1_interrupt_handler handle_arch_irq
    kernel_exit 1
SYM_CODE_END(el1_irq)
```

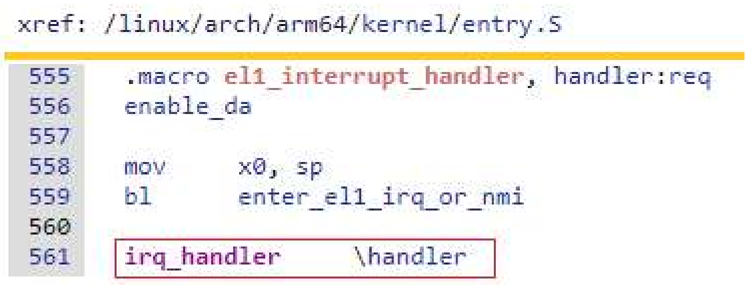
这里我们就不再深究kernel_entry和kernel_exit，它俩里面干得事情非常多。当前我们需要了解，一个是保存general purpose寄存器，一个是恢复就可以了。

```
.macro  kernel_entry, el, regsize = 64
.if \regsize == 32
mov w0, w0              // zero upper 32 bits of x0
.endif
stp x0, x1, [sp, #16 * 0]
stp x2, x3, [sp, #16 * 1]
stp x4, x5, [sp, #16 * 2]
stp x6, x7, [sp, #16 * 3]
stp x8, x9, [sp, #16 * 4]
stp x10, x11, [sp, #16 * 5]
stp x12, x13, [sp, #16 * 6]
stp x14, x15, [sp, #16 * 7]
stp x16, x17, [sp, #16 * 8]
stp x18, x19, [sp, #16 * 9]
stp x20, x21, [sp, #16 * 10]
stp x22, x23, [sp, #16 * 11]
stp x24, x25, [sp, #16 * 12]
stp x26, x27, [sp, #16 * 13]
stp x28, x29, [sp, #16 * 14]
......
```

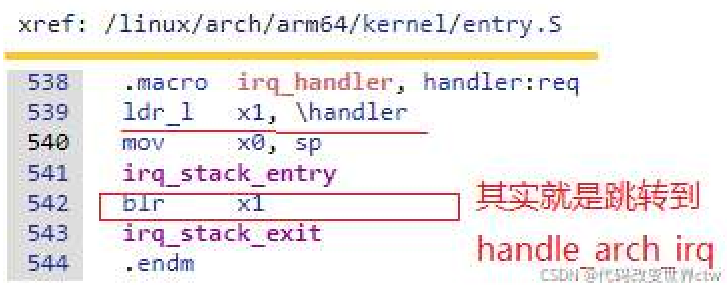
```
.macro  kernel_exit, el
......
msr elr_el1, x21            // set up the return data
msr spsr_el1, x22
ldp x0, x1, [sp, #16 * 0]
ldp x2, x3, [sp, #16 * 1]
ldp x4, x5, [sp, #16 * 2]
ldp x6, x7, [sp, #16 * 3]
ldp x8, x9, [sp, #16 * 4]
ldp x10, x11, [sp, #16 * 5]
ldp x12, x13, [sp, #16 * 6]
ldp x14, x15, [sp, #16 * 7]
ldp x16, x17, [sp, #16 * 8]
ldp x18, x19, [sp, #16 * 9]
ldp x20, x21, [sp, #16 * 10]
ldp x22, x23, [sp, #16 * 11]
ldp x24, x25, [sp, #16 * 12]
ldp x26, x27, [sp, #16 * 13]
ldp x28, x29, [sp, #16 * 14]
ldr lr, [sp, #S_LR]
add sp, sp, #PT_REGS_SIZE       // restore sp
......
```



```
xref: /linux/arch/arm64/kernel/entry.S

SYM_CODE_START_LOCAL_NOALIGN(el1_irq)
	kernel_entry 1
	el1_interrupt_handler handle_arch_irq
	kernel_exit 1
SYM_CODE_END(el1_irq)
```



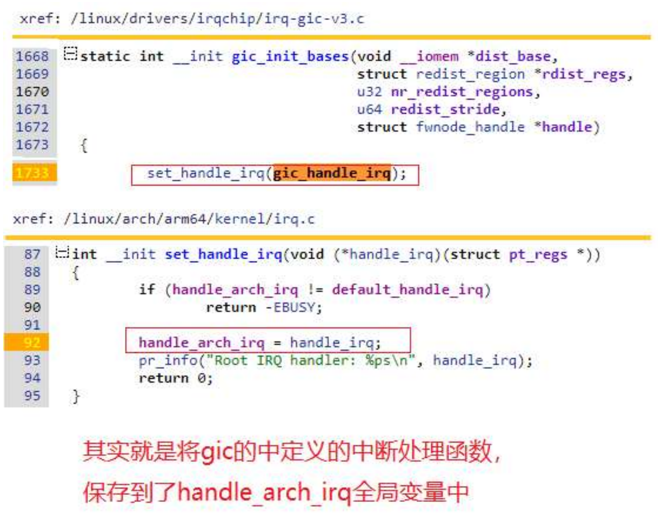
```
xref: /linux/arch/arm64/kernel/entry.S

	.macro el1_interrupt_handler, handler:req
	enable_da

	mov	x0, sp
	bl	enter_el1_irq_or_nmi

	irq_handler	\handler
```



```
xref: /linux/arch/arm64/kernel/entry.S

	.macro	irq_handler, handler:req
	ldr_l	x1, \handler
	mov	x0, sp
	irq_stack_entry
	blr	x1
	irq_stack_exit
	.endm
```

其实就是跳转到

handle arch irq

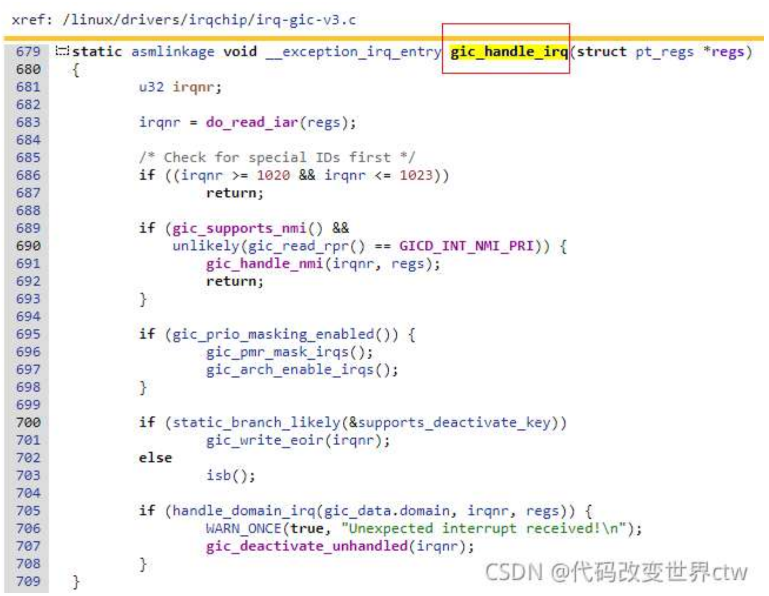


```
xref: /linux/drivers/irqchip/irq-gic-v3.c

static int __init gic_init_bases(void __iomem *dist_base,
				 struct redist_region *rdist_regs,
				 u32 nr_redist_regions,
				 u64 redist_stride,
				 struct fwnode_handle *handle)
{

	set_handle_irq(gic_handle_irq);
```

```
xref: /linux/arch/arm64/kernel/irq.c

int __init set_handle_irq(void (*handle_irq)(struct pt_regs *))
{
	if (handle_arch_irq != default_handle_irq)
		return -EBUSY;

	handle_arch_irq = handle_irq;
	pr_info("Root IRQ handler: %ps\n", handle_irq);
	return 0;
}
```

其实就是将gic的中定义的中断处理函数，

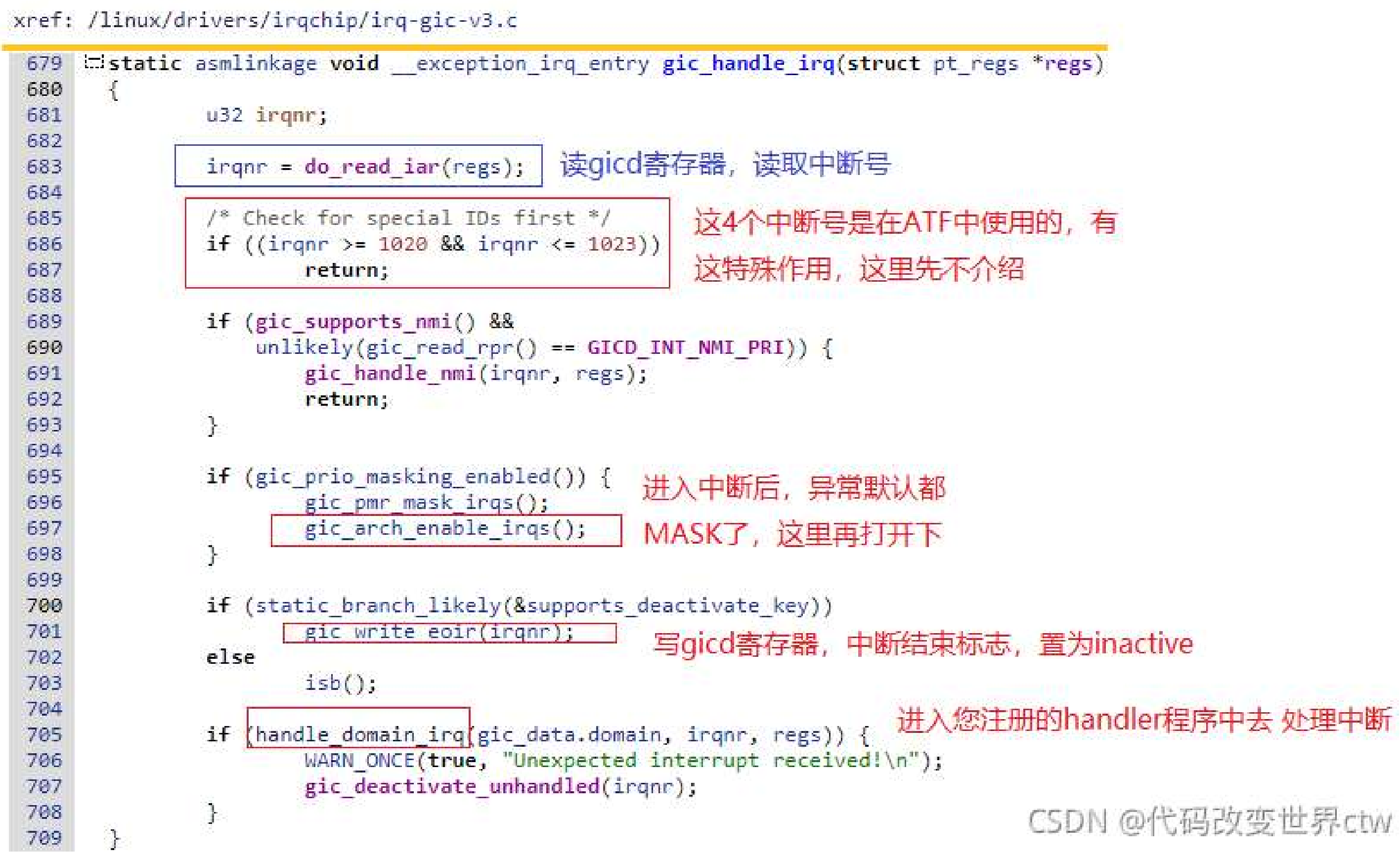
保存到了handle_arch_irq全局变量中



```
xref: /linux/drivers/irqchip/irq-gic-v3.c

static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
	u32 irqnr;

	irqnr = do_read_iar(regs);

	/* Check for special IDs first */
	if ((irqnr >= 1020 && irqnr <= 1023))
		return;

	if (gic_supports_nmi() &&
	    unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
		gic_handle_nmi(irqnr, regs);
		return;
	}

	if (gic_prio_masking_enabled()) {
		gic_pmr_mask_irqs();
		gic_arch_enable_irqs();
	}

	if (static_branch_likely(&supports_deactivate_key))
		gic_write_eoir(irqnr);
	else
		isb();

	if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
		WARN_ONCE(true, "Unexpected interrupt received!\n");
		gic_deactivate_unhandled(irqnr);
	}
}
```

我们再来剖析gic_handle_irq()函数，其实就是涉及gic的读写了，从gic中读取硬件中断号，然后调用handle_domain_irq函数，找到相匹配的中断hander函数，然后回调。

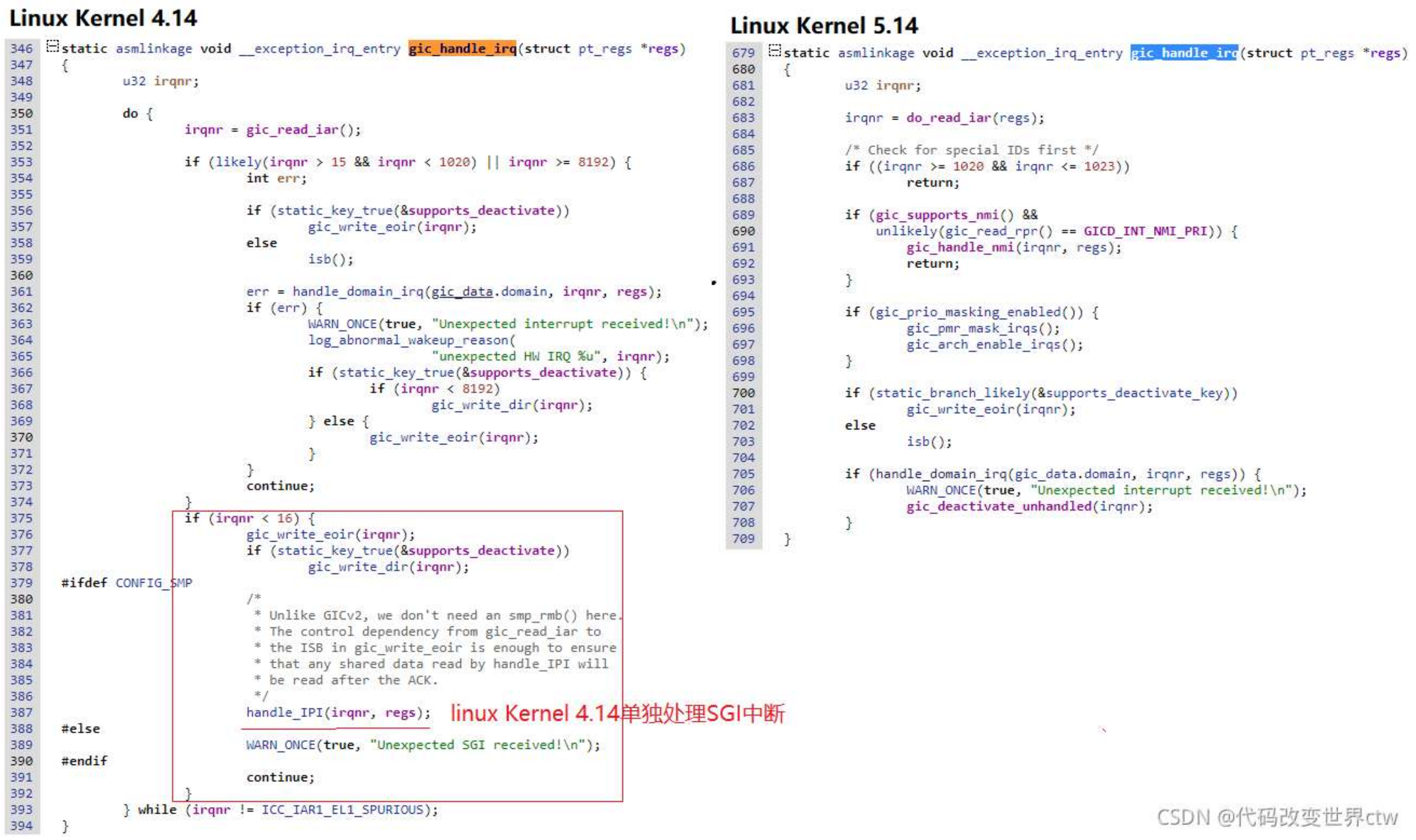
```
(linux/drivers/irqchip/irq-gic-v3.c)

static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
    u32 irqnr;

    irqnr = do_read_iar(regs);

    /* Check for special IDs first */
    if ((irqnr >= 1020 && irqnr <= 1023))
        return;
    if (gic_supports_nmi() &&
        unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
        gic_handle_nmi(irqnr, regs);
        return;
    }

    if (gic_prio_masking_enabled()) {
        gic_pmr_mask_irqs();
        gic_arch_enable_irqs();
    }

    if (static_branch_likely(&supports_deactivate_key))
        gic_write_eoir(irqnr);
    else
        isb();

    if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
        WARN_ONCE(true, "Unexpected interrupt received!\n");
        gic_deactivate_unhandled(irqnr);
    }
}
```


读gicd寄存器，读取中断号
这4个中断号是在ATF中使用的，有这特殊作用，这里先不介绍
进入中断后，异常默认都MASK了，这里再打开下
写gicd寄存器，中断结束标志，置为inactive
进入您注册的handler程序中去 处理中断
CSDN @代码改变世界ctw

```
xref: /linux/drivers/irqchip/irq-gic-v3.c
static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
        u32 irqnr;

        irqnr = do_read_iar(regs);

        /* Check for special IDs first */
        if ((irqnr >= 1020 && irqnr <= 1023))
                return;

        if (gic_supports_nmi() &&
            unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
                gic_handle_nmi(irqnr, regs);
                return;
        }

        if (gic_prio_masking_enabled()) {
                gic_pmr_mask_irqs();
                gic_arch_enable_irqs();
        }

        if (static_branch_likely(&supports_deactivate_key))
                gic_write_eoir(irqnr);
        else
                isb();

        if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
                WARN_ONCE(true, "Unexpected interrupt received!\n");
                gic_deactivate_unhandled(irqnr);
        }
}
```

另外注意一点，在Linux Kernel5.0之后，gic中的handler处理函数，发生了一些细微的变化，如下所

示：



Linux Kernel 4.14

```c
static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
        u32 irqnr;

        do {
                irqnr = gic_read_iar();

                if (likely(irqnr > 15 && irqnr < 1020) || irqnr >= 8192) {
                        int err;

                        if (static_key_true(&supports_deactivate))
                                gic_write_eoir(irqnr);
                        else
                                isb();

                        err = handle_domain_irq(gic_data.domain, irqnr, regs);
                        if (err) {
                                WARN_ONCE(true, "Unexpected interrupt received!\n");
                                log_abnormal_wakeup_reason(
                                                "unexpected HW IRQ %u", irqnr);
                                if (static_key_true(&supports_deactivate)) {
                                        if (irqnr < 8192)
                                                gic_write_dir(irqnr);
                                } else {
                                        gic_write_eoir(irqnr);
                                }
                        }
                        continue;
                }
                if (irqnr < 16) {
                        gic_write_eoir(irqnr);
                        if (static_key_true(&supports_deactivate))
                                gic_write_dir(irqnr);
#ifdef CONFIG_SMP
                        /*
                         * Unlike GICv2, we don't need an smp_rmb() here.
                         * The control dependency from gic_read_iar to
                         * the ISB in gic_write_eoir is enough to ensure
                         * that any shared data read by handle_IPI will
                         * be read after the ACK.
                         */
                        handle_IPI(irqnr, regs);
#else
                        WARN_ONCE(true, "Unexpected SGI received!\n");
#endif
                        continue;
                }
        } while (irqnr != ICC_IAR1_EL1_SPURIOUS);
}
```

Linux Kernel 5.14

```c
static asmlinkage void __exception_irq_entry gic_handle_irq(struct pt_regs *regs)
{
        u32 irqnr;

        irqnr = do_read_iar(regs);

        /* Check for special IDs first */
        if ((irqnr >= 1020 && irqnr <= 1023))
                return;

        if (gic_supports_nmi() &&
            unlikely(gic_read_rpr() == GICD_INT_NMI_PRI)) {
                gic_handle_nmi(irqnr, regs);
                return;
        }

        if (gic_prio_masking_enabled()) {
                gic_pmr_mask_irqs();
                gic_arch_enable_irqs();
        }

        if (static_branch_likely(&supports_deactivate_key))
                gic_write_eoir(irqnr);
        else
                isb();

        if (handle_domain_irq(gic_data.domain, irqnr, regs)) {
                WARN_ONCE(true, "Unexpected interrupt received!\n");
                gic_deactivate_unhandled(irqnr);
        }
}
```

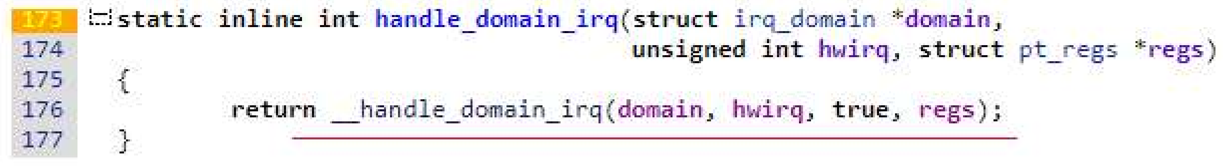
6 handle_domain_irq

> **补充IRQ Domain介绍**
> 在linux kernel中，我们使用下面两个ID来标识一个来自外设的中断：
>
> 1、IRQ number。CPU需要为每一个外设中断编号，我们称之IRQ Number。这个IRQ number是一个虚拟的interrupt ID，和硬件无关，仅仅是被CPU用来标识一个外设中断。
>
> 2、HW interrupt ID。对于interrupt controller而言，它收集了多个外设的interrupt request line并向上传递，因此，interrupt controller需要对外设中断进行编码。Interrupt controller用HW interrupt ID来标识外设的中断。在interrupt controller级联的情况下，仅仅用HW interrupt ID已经不能唯一标识一个外设中断，还需要知道该HW interrupt ID所属的interrupt controller（HW interrupt ID在不同的Interrupt controller上是会重复编码的）。
>
> 这样，CPU和interrupt controller在标识中断上就有了一些不同的概念，但是，对于驱动工程师而言，我们和CPU视角是一样的，我们只希望得到一个IRQ number，而不关系具体是那个interrupt controller上的那个HW interrupt ID。这样一个好处是在中断相关的硬件发生变化的时候，驱动软件不需要修改。因此，linux kernel中的中断子系统需要提供一个将HW interrupt ID映射到IRQ number上来的机制…
>
> （本段转载自:http://www.wowotech.net/linux_kenrel/irq-domain.html）

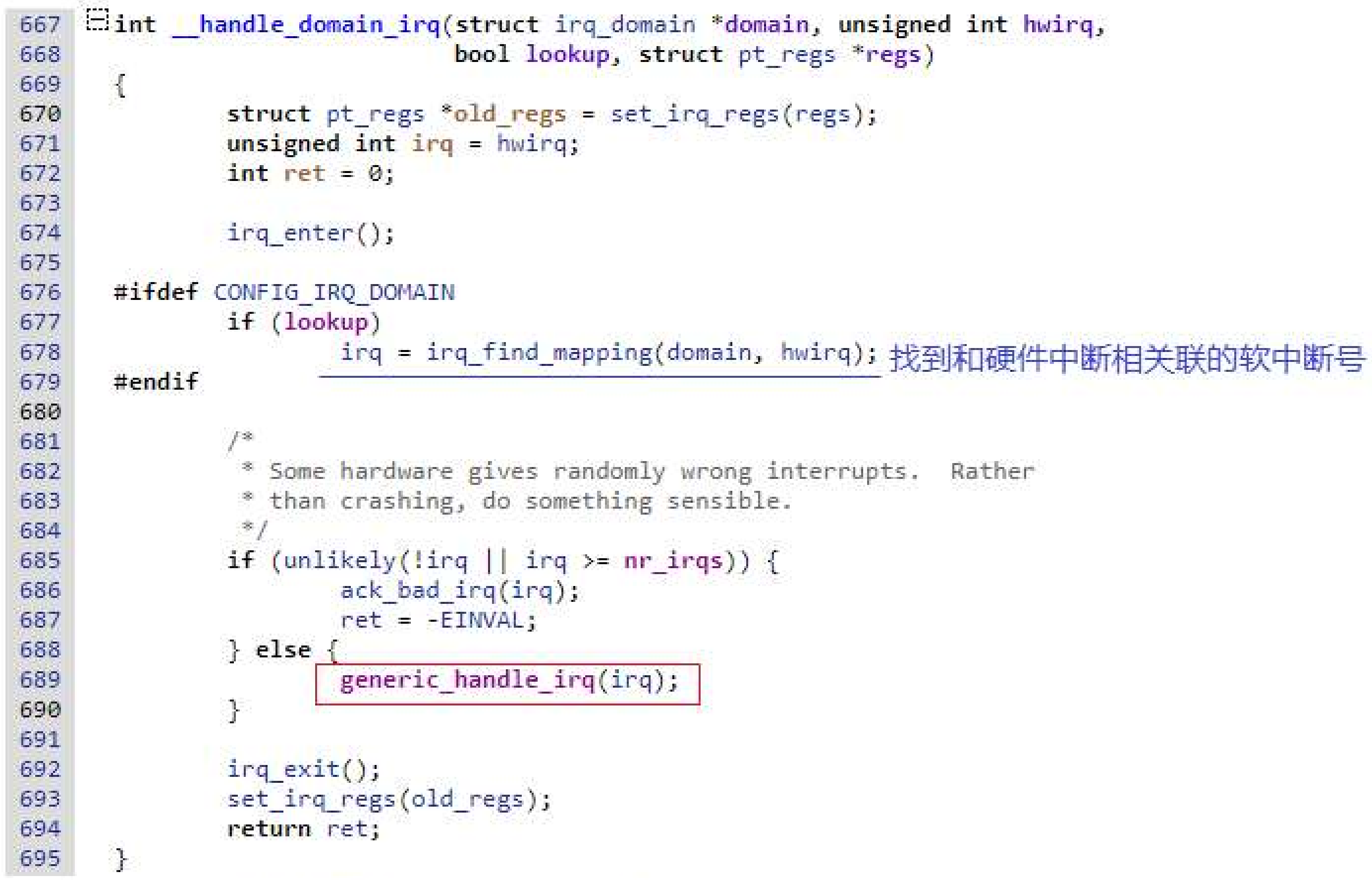
思考:

1、上文提到"在interrupt controller级联的情况下", 为什么会有中断级联，一个gic控制器可以连接好几千个中断难道还不够吗?

handle_domain_irq的处理流程如下所示，最终是调用到了我们request_irq注册的中断处理函数.

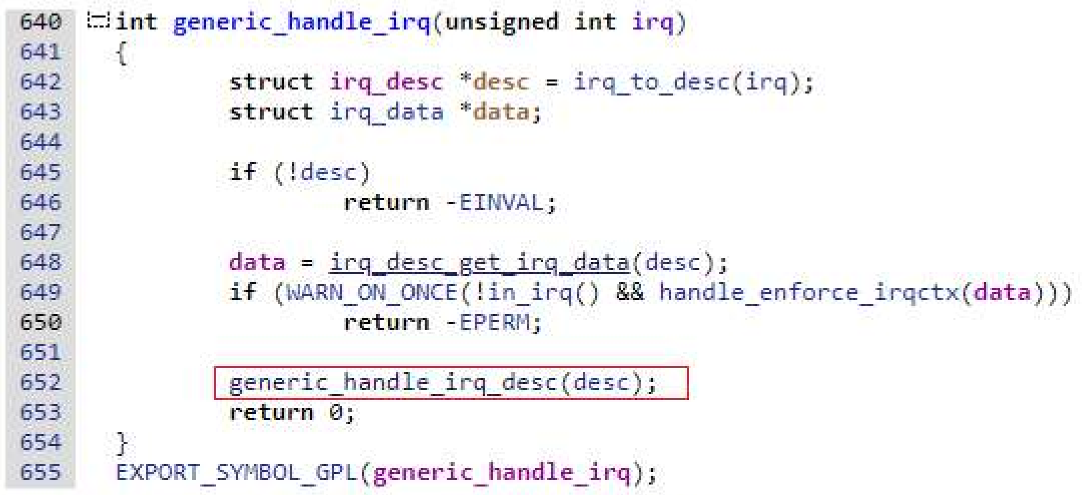


```c
static inline int handle_domain_irq(struct irq_domain *domain,
                                    unsigned int hwirq, struct pt_regs *regs)
{
        return __handle_domain_irq(domain, hwirq, true, regs);
}
```

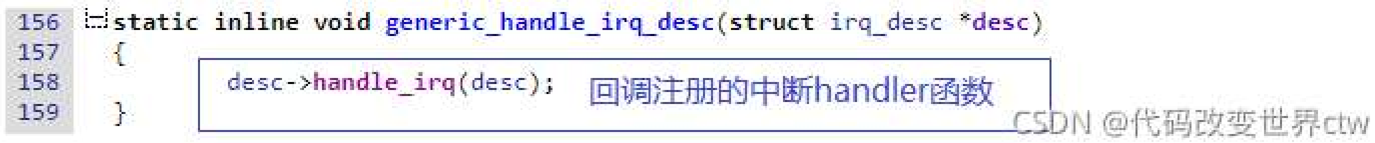


```c
int __handle_domain_irq(struct irq_domain *domain, unsigned int hwirq,
                        bool lookup, struct pt_regs *regs)
{
        struct pt_regs *old_regs = set_irq_regs(regs);
        unsigned int irq = hwirq;
        int ret = 0;

        irq_enter();

#ifdef CONFIG_IRQ_DOMAIN
        if (lookup)
                irq = irq_find_mapping(domain, hwirq);
#endif

        /*
         * Some hardware gives randomly wrong interrupts.  Rather
         * than crashing, do something sensible.
         */
        if (unlikely(!irq || irq >= nr_irqs)) {
                ack_bad_irq(irq);
                ret = -EINVAL;
        } else {
                generic_handle_irq(irq);
        }

        irq_exit();
        set_irq_regs(old_regs);
        return ret;
}
```

找到和硬件中断相关联的软中断号



```c
int generic_handle_irq(unsigned int irq)
{
        struct irq_desc *desc = irq_to_desc(irq);
        struct irq_data *data;

        if (!desc)
                return -EINVAL;

        data = irq_desc_get_irq_data(desc);
        if (WARN_ON_ONCE(!in_irq() && handle_enforce_irqctx(data)))
                return -EPERM;

        generic_handle_irq_desc(desc);
        return 0;
}
EXPORT_SYMBOL_GPL(generic_handle_irq);
```

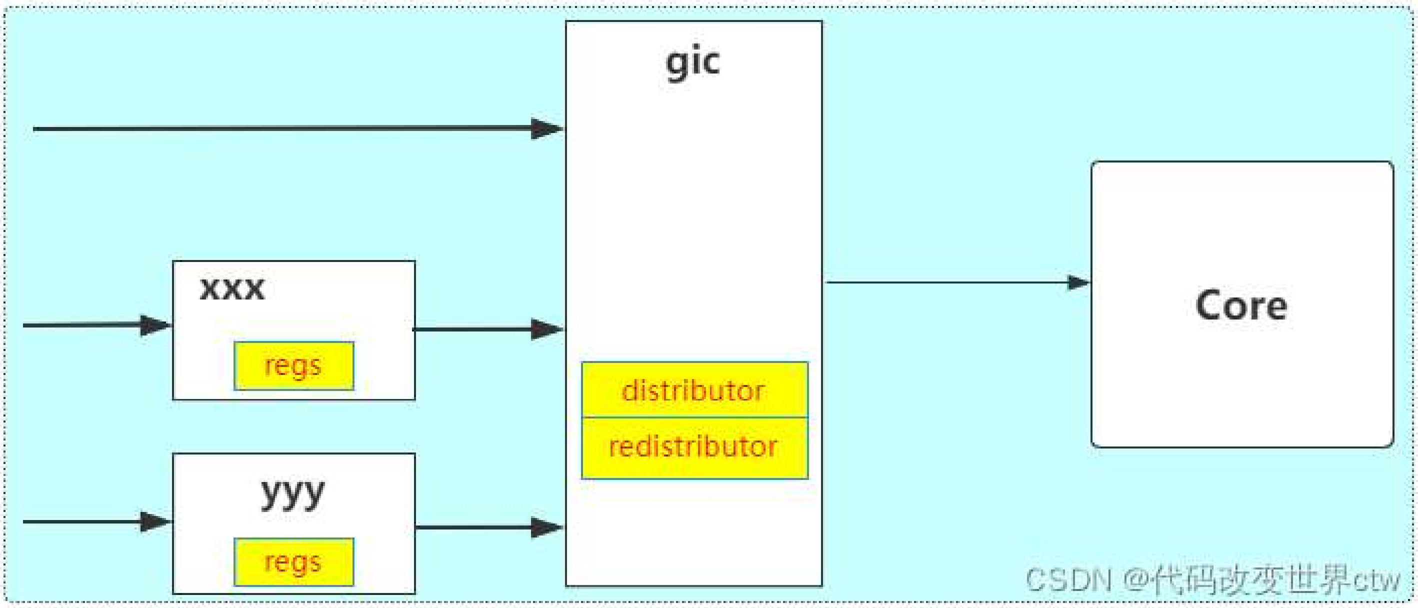


```c
static inline void generic_handle_irq_desc(struct irq_desc *desc)
{
        desc->handle_irq(desc);
}
```

回调注册的中断handler函数

7 关于中断级联的介绍

这也是我想不通的地方，一个gic控制器可以连接好几千个中断难道还不够吗？ 也许是为了SOC方便设计。例如某平台就使用到了级联的方式



（具体代码，就不能做介绍了）

Non-cacheable,Cacheable, non-shareable,inner-shareable,outer-shareable, optee、ATF、TF-A、Trustzone、optee3.14、MMU、VMSA、cache、TLB、arm、armv8、armv9、TEE、安全、内存管理、页表…