IRQ is triggered by hardware, then ...

part one

==

arch/arm/kernel/entry-armv.S

```
/*
 * Interrupt dispatcher
 */
	vector_stub	irq, IRQ_MODE, 4

	.long	__irq_usr		@  0  (USR_26 / USR_32)
	.long	__irq_invalid		@  1  (FIQ_26 / FIQ_32)
	.long	__irq_invalid		@  2  (IRQ_26 / IRQ_32)
	.long	__irq_svc		@  3  (SVC_26 / SVC_32)
	.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
```

当CPU处于user mode时，trjgger IRQ,进入__irq_usr

```
	.align	5
__irq_usr:
	usr_entry
	kuser_cmpxchg_check
	irq_handler
	get_thread_info tsk
	mov	why, #0
	b	ret_to_user_from_irq
 UNWIND(.fnend		)
ENDPROC(__irq_usr)

	.align	5
__irq_svc:
	svc_entry
	irq_handler
```

当CPU处于kernel mode(SVC mode)时，trjgger IRQ,进入__irq_svc

```
#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, irq = 1                    @ return from exception
 UNWIND(.fnend          )
ENDPROC(__irq_svc)
```

在SVC中当从irq service返回时，要判断当前是否可以preemptable（通过判断当前task的preempt count是否为zero）；如果可以则马上发生context switch。

最终这两者都会进入　　irq_handler macro。

```
        .macro  irq_handler
#ifdef CONFIG_MULTI_IRQ_HANDLER
        ldr     r1, =handle_arch_irq
        mov r0, sp
        adr     lr, BSYM(9997f)
        ldr     pc, [r1]
#else
```

```
arch_irq_handler_default

#endif

9997:

        .endm
```

CONFIG_MULTI_IRQ_HANDLER=y （in Gr2 / Gs2 LSP）

```
#ifdef CONFIG_MULTI_IRQ_HANDLER

        .globl      handle_arch_irq

handle_arch_irq:

        .space     4

#endif
```

由于CONFIG_MULTI_IRQ_HANDLER is set,执行的code是

```
        ldr    r1, =handle_arch_irq
        mov r0, sp
        adr   lr, BSYM(9997f)
        ldr    pc, [r1]
```

也就是跳转到handle_arch_irq 所存放的函数指针处去执行。

从静态code看，handle_arch_irq处只是留了4个bytes的空间，里面什么都没有。

Gr2 / Gs2用的主interrupt controller是GIC-400，所以handle_arch_irq处应该fill它的handler。

in drivers/irqchip/irq-gic.c

```
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, i;
        int nr_routable_irqs;

        BUG_ON(gic_nr >= MAX_GIC_NR);

        ......

        if (gic_nr == 0) {
#ifdef CONFIG_SMP
                set_smp_cross_call(gic_raise_softirq);
                register_cpu_notifier(&gic_cpu_notifier);
#endif
                set_handle_irq(gic_handle_irq);
        }
        gic_chip.flags |= gic_arch_extn.flags;
        gic_dist_init(gic);
        gic_cpu_init(gic);
        gic_pm_init(gic);
```

```
}
```

in arch/arm/kernel/irq.c

```
#ifdef CONFIG_MULTI_IRQ_HANDLER
void __init set_handle_irq(void (*handle_irq)(struct pt_regs *))
{
	if (handle_arch_irq)
		return;

	handle_arch_irq = handle_irq;
}
#endif
```

也就是

```
	ldr	r1, =handle_arch_irq
	mov r0, sp
	adr	lr, BSYM(9997f)
	ldr	pc, [r1]
```

其实是

```
	ldr	r1, =handle_arch_irq
	mov r0, sp
	adr	lr, BSYM(9997f)
	call gic_handle_irq
```

===

part two

handle_arch_irq的初始化。

in in drivers/irqchip/irq-gic.c

```
IRQCHIP_DECLARE(gic_400, "arm,gic-400", gic_of_init);
IRQCHIP_DECLARE(cortex_a15_gic, "arm,cortex-a15-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a9_gic, "arm,cortex-a9-gic", gic_of_init);
IRQCHIP_DECLARE(cortex_a7_gic, "arm,cortex-a7-gic", gic_of_init);
IRQCHIP_DECLARE(msm_8660_qgic, "qcom,msm-8660-qgic", gic_of_init);
IRQCHIP_DECLARE(msm_qgic2, "qcom,msm-qgic2", gic_of_init);
```

in drivers/irqchip/irqchip.h

```
/*
 * This macro must be used by the different irqchip drivers to declare
 * the association between their DT compatible string and their
 * initialization function.
 *
 * @name: name that must be unique accross all IRQCHIP_DECLARE of the
 * same file.
 * @compstr: compatible string of the irqchip driver
 * @fn: initialization function
```

```
 */
#define IRQCHIP_DECLARE(name, compat, fn) OF_DECLARE_2(irqchip, name, compat, fn)

#define OF_DECLARE_2(table, name, compat, fn) \
		_OF_DECLARE(table, name, compat, fn, of_init_fn_2)

#define _OF_DECLARE(table, name, compat, fn, fn_type)			\
	static const struct of_device_id __of_table_##name		\
		__used __section(__##table##_of_table)			\
		 = { .compatible = compat,				\
		     .data = (fn == (fn_type)NULL) ? fn : fn  }
```

又是Linux kernel code惯常用的老套路，

在特定section中填写如下structure

```
struct of_device_id
{
	char name[32];
	char type[32];
	char compatible[128];
	const void *data;
};

IRQCHIP_DECLARE(gic_400, "arm,gic-400", gic_of_init);
==>
```

static const struct of_device_id __of_table_gic_400 __used __section(__irqchip_of_table)

= { .compatible = "arm,gic-400",

.data = gic_of_init };

IRQCHIP_DECLARE(cortex_a15_gic, "arm,cortex-a15-gic", gic_of_init);

==>

static const struct of_device_id __of_table_cortex_a15_gic __used __section(__irqchip_of_table)

= { .compatible = "arm,cortex-a15-gic",

.data = gic_of_init };

IRQCHIP_DECLARE(cortex_a9_gic, "arm,cortex-a9-gic", gic_of_init);

==>

static const struct of_device_id __of_table_cortex_a9_gic __used __section(__irqchip_of_table)

= { .compatible = "arm,cortex-a9-gic",

.data = gic_of_init };

etc.

这些structure都被link editor放入“__irqchip_of_table” section,组成struct of_device_id array.

in vmlinux.lds

*(.init.data) *(.meminit.data) *(.init.rodata) *(.meminit.rodata) . = ALIGN(8); __clk_of_table = .; *(__clk_of_table) *(__clk_of_table_end) . = ALIGN(8); __reservedmem_of_table = .; *(__reservedmem_of_table) *(__reservedmem_of_table_end) . = ALIGN(8); __clksrc_of_table = .; *(__clksrc_of_table) *(__clksrc_of_table_end) . = ALIGN(8); __cpu_method_of_table = .; *(__cpu_method_of_table) *(__cpu_method_of_table_end) . = ALIGN(32); __dtb_start = .; *(.dtb.init.rodata) __dtb_end = .; **. = ALIGN(8); __irqchip_of_table = .; *(__irqchip_of_table) *(__irqchip_of_table_end)**

. = ALIGN(16); __setup_start = .; *(.init.setup) __setup_end = .;

该数组的head由__irqchip_of_table指向。

in arch/arm/mach-pegmatite/pegmatite.c

```
DT_MACHINE_START(PEGMATITE_DT, "Marvell Pegmatite (Device Tree)")
#ifdef CONFIG_SMP
	.smp		= smp_ops(pegmatite_smp_ops),
#endif
	.init_machine	= pegmatite_dt_init,
	.map_io		= pegmatite_map_io,
	.init_early = pegmatite_init_early,
	.init_irq	= pegmatite_init_irq,
	.init_time	= pegmatite_timer_and_clk_init,
	.restart	= pegmatite_restart,
	.dt_compat	= pegmatite_dt_compat,
#ifdef CONFIG_ZONE_DMA
	.dma_zone_size	= SZ_256M,
#endif
MACHINE_END

static void __init pegmatite_init_irq(void)
{
	irqchip_init();
```

```
}

in drivers/irqchip/irqchip.c

/*
 * This special of_device_id is the sentinel at the end of the
 * of_device_id[] array of all irqchips. It is automatically placed at
 * the end of the array by the linker, thanks to being part of a
 * special section.
 */
static const struct of_device_id
irqchip_of_match_end __used __section(__irqchip_of_table_end);

extern struct of_device_id __irqchip_of_table[];

void __init irqchip_init(void)
{
        of_irq_init(__irqchip_of_table);
}

in drivers/of/irq.c

void __init of_irq_init(const struct of_device_id *matches)
```

该函数完成interrupt controller initialize,会调用各个interrupt controller的初始化函数，比如gic-400的gic_of_init（）。

```
asmlinkage __visible void __init start_kernel(void)

{

    ......



        /* init some links before init_ISA_irqs() */

        early_irq_init();

        init_IRQ();        <--- 这之后才可以trigger IRQ

        tick_init();



    ......



}
```

in arch/arm/kernel/irq.c

```
void __init init_IRQ(void)

{

        int ret;



        if (IS_ENABLED(CONFIG_OF) && !machine_desc->init_irq)

                irqchip_init();

        else

                machine_desc->init_irq();        会调用machine descriptor中的.init_irq callback function.
```

```
        if (IS_ENABLhandle_arch_irqED(CONFIG_OF) && IS_ENABLED(CONFIG_CACHE_L2X0) &&
            (machine_desc->l2c_aux_mask || machine_desc->l2c_aux_val)) {
                outer_cache.write_sec = machine_desc->l2c_write_sec;
                ret = l2x0_of_init(machine_desc->l2c_aux_val,
                                   machine_desc->l2c_aux_mask);
                if (ret)
                        pr_err("L2C: failed to init: %d\n", ret);
        }
}
```

其实在pegmatite.c中machine descruptor完全没必要给.init_irq field赋值。在default情况下，irqchip_init()也会被调用。

总结一下，start_kernel（ ）

```
                |
                |
               \|/
            init_IRQ();
                |
                |
               \|/
           irqchip_init()
                |
                |
               \|/
     of_irq_init(__irqchip_of_table);
```

如果有多个interrupt controller，且有父子关系，则该函数要保证

```
        |                        先初始化parent，然后是child。这个关系由dts决定。
        |
       \|/
    gic_of_init()
        |
        |
       \|/
    gic_init_bases()
        |
        |
       \|/
    set_handle_irq(gic_handle_irq)      设置handle_arch_irq ariable。
```

由此hardware interrupt可以产生了！

```
__irq_usr / __irq_svc     ( arch/arm/kernel/entry-armv.S )
        |
        |
       \|/
  gic_handle_irq()          (drivers/irqchip/irq-gic.c)
```

==

从interrupt controller的ISR到device的ISR

```
static 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 {
                irqstat = readl_relaxed(cpu_base + GIC_CPU_INTACK);
                irqnr = irqstat & GICC_IAR_INT_ID_MASK;

                if (likely(irqnr > 15 && irqnr < 1021)) {
                        handle_domain_irq(gic->domain, irqnr, regs);
                        continue;
                }
                if (irqnr < 16) {
                        writel_relaxed(irqstat, cpu_base + GIC_CPU_EOI);
#ifdef CONFIG_SMP
                        handle_IPI(irqnr, regs);
#endif
                        continue;
                }
                break;
        } while (1);
}
```

核心函数是

```
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);
}


/**
 * __handle_domain_irq - Invoke the handler for a HW irq belonging to a domain
 * @domain:	The domain where to perform the lookup
 * @hwirq:	The HW irq number to convert to a logical one
 * @lookup:	Whether to perform the domain lookup or not
 * @regs:	Register file coming from the low-level handling code
 *
 * Returns:	0 on success, or -EINVAL if conversion has failed
 */
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);	①
#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);                    ②
        }

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

该函数主要做了两件事

① irq = irq_find_mapping(domain, hwirq)

convert physical hardware interrupt number into virtual interrupt number by irq domain.

每个inqchip都可以实现自己的from physical hardware interrupt number to virtual interrupt number的mapping。

比如在dts中

```
rtc@d0626800 {
        compatible = "marvell,orion-rtc";
        reg = <0x0 0xd0626800 0x0 0x100 0x0 0xd0627000 0x0 0x30>;
        reg-names = "rtcregs", "mpmumiscregs";
        interrupts = <0x0 0x5 0x4>;
};
```

该RTC device的interrupt <0x0 0x5 0x4> 是physical hardware interrupt number，而kernel则是用virtual interrupt number来管理IRQ service的，所以必须由irq domain来完成转换。这个工作是由这里的irq-gic来做的。

②generic_handle_irq(irq); 这里的irq是virtual interrupt number.

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

    if (!desc)
        return -EINVAL;
    generic_handle_irq_desc(irq, desc);
    return 0;
}

struct irq_desc *irq_to_desc(unsigned int irq)
{
    return (irq < NR_IRQS) ? irq_desc + irq : NULL;
```

}

virtual irq number实际上是irq_desc array的index。

struct irq_desc irq_desc[NR_IRQS];

而struct irq_desc则记录了hardware interrupt handler的信息。

```
/**
 * struct irq_desc - interrupt descriptor
 * @irq_data:        per irq and chip data passed down to chip functions
 * @kstat_irqs:          irq stats per cpu
 * @handle_irq:          highlevel irq-events handler
 * @preflow_handler:     handler called before the flow handler (currently used by sparc)
 * @action:          the irq action chain
 * @status:          status information
 * @core_internal_state__do_not_mess_with_it: core internal status information
 * @depth:           disable-depth, for nested irq_disable() calls
 * @wake_depth:          enable depth, for multiple irq_set_irq_wake() callers
 * @irq_count:       stats field to detect stalled irqs
 * @last_unhandled:      aging timer for unhandled count
 * @irqs_unhandled:      stats field for spurious unhandled interrupts
 * @threads_handled:     stats field for deferred spurious detection of threaded handlers
 * @threads_handled_last: comparator field for deferred spurious detection of theraded handlers
 * @lock:        locking for SMP
```

```
 * @affinity_hint:      hint to user space for preferred irq affinity
 * @affinity_notify:   context for notification of affinity changes
 * @pending_mask:       pending rebalanced interrupts
 * @threads_oneshot:    bitfield to handle shared oneshot threads
 * @threads_active:     number of irqaction threads currently running
 * @wait_for_threads:   wait queue for sync_irq to wait for threaded handlers
 * @nr_actions:         number of installed actions on this descriptor
 * @no_suspend_depth: number of irqactions on a irq descriptor with
 *                  IRQF_NO_SUSPEND set
 * @force_resume_depth:      number of irqactions on a irq descriptor with
 *                  IRQF_FORCE_RESUME set
 * @dir:          /proc/irq/ procfs entry
 * @name:             flow handler name for /proc/interrupts output
 */
struct irq_desc {
	struct irq_data		irq_data;
	unsigned int __percpu	*kstat_irqs;
	irq_flow_handler_t	handle_irq;
#ifdef CONFIG_IRQ_PREFLOW_FASTEOI
	irq_preflow_handler_t	preflow_handler;
#endif
	struct irqaction	*action;	/* IRQ action list */
	unsigned int		status_use_accessors;
	unsigned int		core_internal_state__do_not_mess_with_it;
	unsigned int		depth;		/* nested irq disables */
```

```
	unsigned int		wake_depth;	/* nested wake enables */
	unsigned int		irq_count;	/* For detecting broken IRQs */
	unsigned long		last_unhandled;	/* Aging timer for unhandled count */
	unsigned int		irqs_unhandled;
	atomic_t		threads_handled;
	int			threads_handled_last;
	raw_spinlock_t		lock;
	struct cpumask		*percpu_enabled;
#ifdef CONFIG_SMP
	const struct cpumask	*affinity_hint;
	struct irq_affinity_notify *affinity_notify;
#ifdef CONFIG_GENERIC_PENDING_IRQ
	cpumask_var_t		pending_mask;
#endif
#endif
	unsigned long		threads_oneshot;
	atomic_t		threads_active;
	wait_queue_head_t	wait_for_threads;
#ifdef CONFIG_PM_SLEEP
	unsigned int		nr_actions;
	unsigned int		no_suspend_depth;
	unsigned int		force_resume_depth;
#endif
#ifdef CONFIG_PROC_FS
	struct proc_dir_entry	*dir;
```

```
#endif
        int                     parent_irq;
        struct module           *owner;
        const char              *name;
} ____cacheline_internodealigned_in_smp;
```

每个hardware interrupt handler都要在这个array中登记，以便当handware interrupt发生时，kernel能从该array中找到，并调用中断处理程序。

```
static inline void generic_handle_irq_desc(unsigned int irq, struct irq_desc *desc)
{
        desc->handle_irq(irq, desc);          调用device driver在irq_desc[]中注册的ISR
}
```

以RTC interrupt handler为例，

drivers/rtc/rtc-mv62xx.c是RTC "marvell,orion-rtc"的driver.

在其probe() function中与interrupt相关的如下

```
static int __init mv_rtc_probe(struct platform_device *pdev)
{
        ......
```

```
    pdata->irq = platform_get_irq(pdev, 0);        （I）

    ......

    if (pdata->irq >= 0) {
        writel(0, pdata->ioaddr + RTC_ALARM_INTERRUPT1_MASK_REG_OFFS);
        if (devm_request_irq(&pdev->dev, pdata->irq, mv_rtc_interrupt,        （II）
                IRQF_SHARED,
                pdev->name, pdata) < 0) {
            dev_warn(&pdev->dev, "interrupt not available.\n");
            pdata->irq = -1;
        }
    }

    ......

}
```

（I）

pdata->irq = platform_get_irq(pdev, 0);

从RTC的device node中得到physical interrupt number，并通过该irq所在的interrupt controller的irq domain完成从 <0x0 0x5 0x4> 到virtual interrupt number的mapping。这里的返回值pdata->irq即是virtual irq。

```
/**
```

```
 * platform_get_irq - get an IRQ for a device
 * @dev: platform device
 * @num: IRQ number index
 */
int platform_get_irq(struct platform_device *dev, unsigned int num)
{
#ifdef CONFIG_SPARC
	/* sparc does not have irqs represented as IORESOURCE_IRQ resources */
	if (!dev || num >= dev->archdata.num_irqs)
		return -ENXIO;
	return dev->archdata.irqs[num];
#else
	struct resource *r;
	if (IS_ENABLED(CONFIG_OF_IRQ) && dev->dev.of_node) {
		int ret;

		ret = of_irq_get(dev->dev.of_node, num);
		if (ret >= 0 || ret == -EPROBE_DEFER)
			return ret;
	}

	r = platform_get_resource(dev, IORESOURCE_IRQ, num);

	return r ? r->start : -ENXIO;
#endif
```

```
}

/**
 * of_irq_get - Decode a node's IRQ and return it as a Linux irq number
 * @dev: pointer to device tree node
 * @index: zero-based index of the irq
 *
 * Returns Linux irq number on success, or -EPROBE_DEFER if the irq domain
 * is not yet created.
 *
 */
int of_irq_get(struct device_node *dev, int index)
{
	int rc;
	struct of_phandle_args oirq;
	struct irq_domain *domain;

	rc = of_irq_parse_one(dev, index, &oirq);		( 1 )
	if (rc)
		return rc;

	domain = irq_find_host(oirq.np);				( 2 )
	if (!domain)
		return -EPROBE_DEFER;
```

return irq_create_of_mapping(&oirq); **（3）**

}

（1）从RTC的device node中得到physical irq，即这里的<0x0 0x5 0x4>

（2）找到RTC所对应的irqchip(interrupt controller)的irq domain (系统中可以由多个不同类型的irqchip级联，所以要找)

（3）把<0x0 0x5 0x4> mapping成virtual irq，也就是array struct irq_desc irq_desc[NR_IRQS]中的某个index。

（II）

devm_request_irq(&pdev->dev, pdata->irq, mv_rtc_interrupt, IRQF_SHARED, pdev->name, pdata)

该函数实质性的工作就是把mv_rtc_interrupt()这个RTC的interrupt handler的信息填到irq_desc[pdata->irq]所对应的structure中。这样当中断产生时，kernel可以找到该handler，以便调用。