# GIC驱动代码分析（废弃）

作者：linuxer 发布于：2014-7-4 14:34 分类：中断子系统

这份文档状态是：废弃，新的文档请访问http://www.wowotech.net/linux_kenrel/gic_driver.html

一、前言

GIC（Generic Interrupt Controller）是ARM公司提供的一个通用的中断控制器。GIC通过AMBA（Advanced Microcontroller Bus Architecture）这样的片上总线连接到一个或者多个ARM processor上。本文主要分析了linux kernel中GIC中断控制器的驱动代码。

具体的分析方法是按照source code为索引，逐段分析。对于每一段分析的代码，力求做到每个细节都清清楚楚。这不可避免要引入很多对GIC的硬件描述，此外，具体GIC中断控制器的驱动代码和linux kernel中断子系统的交互也会描述，但本文不会描述linux kernel的generic interrupt subsystem。

本文以OMAP4460这款SOC为例，OMAP4460内部集成了GIC的功能。具体的linux kernel的版本是linux3.14.。

二、GIC DTS描述

1、中断系统概述

对于中断系统，主要有三个角色：

（1）processor。主要用于处理中断

（2）Interrupt Generating Device。通过硬件的interrupt line表明自身需要处理器的进一步处理（例如有数据到来、异常状态等）

（3）interrupt controller。负责收集各个外设的异步事件，用有序、可控的方式通知一个或者多个processor。

2、DTS如何描述Interrupt Generating Device

对于Interrupt Generating Device，我们需要定义下面两个属性：

（1） Interrupt属性。该属性主要描述了中断的HW interrupt ID以及类型。

（2）interrupt-parent 属性。该属性主要描述了该设备的interrupt request line连接到哪一个interrupt controller。

在OMAP4460系统中，我们以一个简单的串口为例子，具体的描述在linux-3.14\arch\arm\boot\dts\omap4.dtsi文件中：

```
uart3: serial@48020000 {
        compatible = "ti,omap4-uart";
        reg = <0x48020000 0x100="">;
        interrupts = <GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>;
        ti,hwmods = "uart3";
        clock-frequency = <48000000>;
    };
```

对于uart3，interrupts属性用3个cell（对于device tree，cell是指由32bit组成的一个信息单位）表示。GIC_SPI 描述了interrupt type。对于GIC，它可以管理4种类型的中断：

站内搜索

请输入关键词 搜索

功能

留言板
评论列表
支持者列表

最新评论

ora
@victor：有同样的困惑，通过/proc/pagetyp...
eshin
@David：您提问题的时间是2018年,这时候发布的规格书...
orangeboyye
@linuxer：非常感谢，之前加过你的微信，微信聊。
明天
@太空的树懒：讲的很清楚了 （4）很多异常处理的代码返回的...
linuxer
@orangeboyye：这位同学，有没有兴趣来OPPO搞内...
orangeboyye
@luke：两个写者不能同时执行，因为一个写者要基于另一个写...

文章分类

Linux内核分析(23)
统一设备模型(15)
电源管理子系统(43)
中断子系统(15)
进程管理(31)
内核同步机制(22)
GPIO子系统(5)
时间子系统(14)
通信类协议(7)
内存管理(31)
图形子系统(2)
文件系统(5)
TTY子系统(6)
u-boot分析(3)
Linux应用技巧(13)
软件开发(6)
基础技术(13)
蓝牙(16)
ARMv8A Arch(15)
显示(3)
USB(1)
基础学科(10)
技术漫谈(12)
项目专区(0)
X Project(28)

（1）外设中断（Peripheral interrupt）。根据目标CPU的不同，外设的中断可以分成PPI（Private Peripheral Interrupt）和SPI（Shared Peripheral Interrupt）。PPI只能分配给一个确定的processor，而SPI可以由Distributor将中断分配给一组Processor中的一个进行处理。外设类型的中断一般通过一个interrupt request line的硬件信号线连接到中断控制器，可能是电平触发的（Level-sensitive），也可能是边缘触发的（Edge-triggered）。

（2）软件触发的中断（SGI，Software-generated interrupt）。软件可以通过写GICD_SGIR寄存器来触发一个中断事件，这样的中断，可以用于processor之间的通信。

（3）虚拟中断（Virtual interrupt）和Maintenance interrupt。这两种中断和本文无关，不再赘述。

在DTS中，外设的interrupt type有两种，一种是SPI，另外一种是PPI。SGI用于processor之间的通信，和外设无关。

uart3的interrupt属性中的74表示该外设使用的GIC interrupt ID号。GIC最大支持1020个HW interrupt ID，具体的ID分配情况如下：

（1）ID0~ID31是用于分发到一个特定的process的interrupt。标识这些interrupt不能仅仅依靠ID，因为各个interrupt source都用同样的ID0~ID31来标识，因此识别这些interrupt需要interrupt ID + CPU interface number。ID0~ID15用于SGI，ID16~ID31用于PPI。PPI类型的中断会送到指定的process上，和其他的process无关。SGI是通过写GICD_SGIR寄存器而触发的中断。Distributor通过processor source ID、中断ID和target processor ID来唯一识别一个SGI。

（2）ID32~ID1019用于SPI。

uart3的interrupt属性中的IRQ_TYPE_LEVEL_HIGH用来描述触发类型。

很奇怪，uart3并没有定义interrupt-parent属性，这里定义interrupt-parent属性的是root node，具体的描述在linux-3.14\arch\arm\boot\dts\omap4.dtsi文件中：

```
/ {
    compatible = "ti,omap4430", "ti,omap4";
    interrupt-parent = <&gic>;

 略去无关内容

 }
```

难道root node会产生中断到interrupt controller吗？当然不会，只不过如果一个能够产生中断的device node没有定义interrupt-parent的话，其interrupt-parent属性就是跟随parent node。因此，与其在所有的下游设备中定义interrupt-parent，不如统一在root node中定义了。

3、DTS如何描述GIC

linux-3.14\arch\arm\boot\dts\omap4.dtsi文件中，

```
    gic: interrupt-controller@48241000 {
        compatible = "arm,cortex-a9-gic";
        interrupt-controller;
        #interrupt-cells = <3>;
        reg = <0x48241000 0x1000="">,
              <0x48240100 0x0100="">;
    };
```

compatible属性用来描述GIC的programming model。该属性的值是string list，定义了一系列的modle（每个string是一个model）。这些字符串列表被操作系统用来选择用哪一个driver来驱动该设备。假设定义该属性：compatible = “a厂商，p产品”, “标准bbb类型设备”。那么linux kernel可能首先使用“a厂商，p产品”来匹配适合的driver，如果没有匹配到，那么使用字符串“标准bbb类型设备”来继续寻找适合的driver。compatible属性有两个应用场景：

（1）对于root node，compatible属性是用来匹配machine type的（参考Device Tree相关文档）

（2）对于普通的HW block的节点，例如interrupt-controller，compatible属性是用来匹配适合的driver的。

interrupt-controller这个没有定义value的属性用来表明本设备节点就是一个interrupt controller。理解#interrupt-cells这个属性需要理解interrupt specifier和interrupt domain这两个概念。interrupt specifier其实就是外设interrupt的属性值，对于uart3而言，其interrupt specifier就是<GIC_SPI 74 IRQ_TYPE_LEVEL_HIGH>，也就是说，interrupt specifier定义了一个外设产生中断的规格（HW interrupt ID + interrupt type）。具体如何解析interrupt specifier？这个需要限定在一定的上下文中，不同的interrupt controller会有不同的解释。因此，对于一个包含多个interrupt controller的系统，每个interrupt controller及其相连的外设组成一个interrupt domain，各个外设的interrupt specifier只能在属于它的那个


**随机文章**

“极致”神话和产品观念
X-003-UBOOT-基于Bubblegum-96平台的u-boot移植说明
perfbook memory barrier（14.2章节）中文翻译（下）
Linux DMA Engine framework(1)_概述
X-012-KERNEL-serial early console的移植

**文章存档**

2022年4月(2)
2022年2月(2)
2021年12月(1)
2021年11月(5)
2021年7月(1)
2021年6月(1)
2021年5月(3)
2020年3月(3)
2020年2月(2)
2020年1月(3)
2019年12月(3)
2019年5月(4)
2019年3月(1)
2019年1月(3)
2018年12月(2)
2018年11月(1)
2018年10月(2)
2018年8月(1)
2018年6月(1)
2018年5月(1)
2018年4月(7)
2018年2月(4)
2018年1月(5)
2017年12月(2)
2017年11月(2)
2017年10月(1)
2017年9月(5)
2017年8月(4)
2017年7月(4)
2017年6月(3)
2017年5月(3)
2017年4月(1)
2017年3月(8)
2017年2月(6)
2017年1月(5)
2016年12月(6)
2016年11月(11)
2016年10月(9)
2016年9月(6)
2016年8月(9)
2016年7月(5)
2016年6月(8)
2016年5月(8)
2016年4月(7)
2016年3月(5)
2016年2月(5)
2016年1月(6)
2015年12月(6)
2015年11月(9)
2015年10月(9)
2015年9月(4)
2015年8月(3)
2015年7月(7)
2015年6月(3)
2015年5月(6)
2015年4月(9)
2015年3月(9)
2015年2月(6)
2015年1月(6)
2014年12月(17)
2014年11月(8)
2014年10月(9)
2014年9月(7)

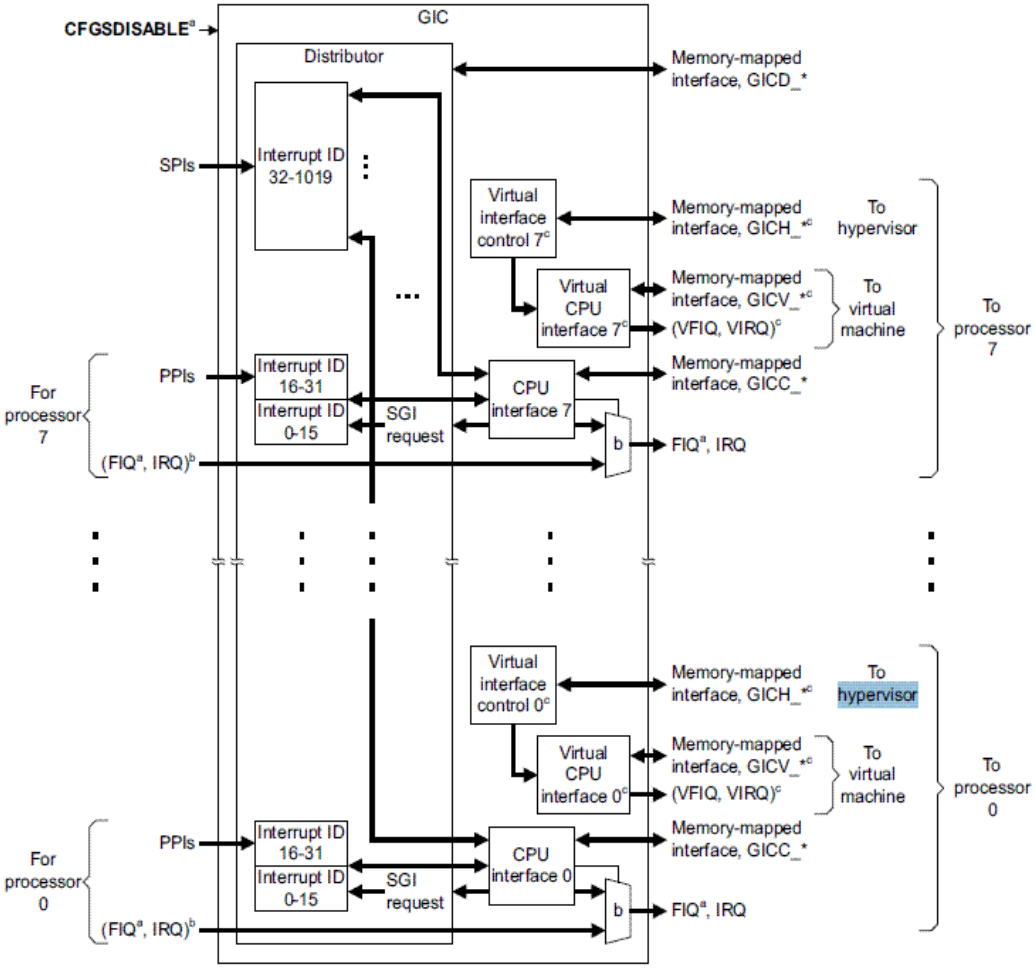
interrupt domain中得到解析。#interrupt-cells定义了在该interrupt domain中，用多少个cell来描述一个外设的interrupt specifier。

reg属性定义了GIC的memory map的地址，有两段，分别描述了CPU interface和Distributor的地址段。理解CPU interface和Distributor这两个术语需要GIC的一些基本的硬件知识，参考下节描述。

三、GIC的HW block diagram描述

1、GIC HW概述

GIC的block diagram如下图所示：




GIC可以清晰的划分成两个block，一个block是Distributor（上图的左边的block），一个是CPU interface。CPU interface有两种，一种就是和普通processor接口，另外一种是和虚拟机接口的。Virtual CPU interface在本文中不会详细描述。

2、Distributor

Distributor的主要的作用是检测各个interrupt source的状态，控制各个interrupt source的行为，分发各个interrupt source产生的中断事件到各个processor。Distributor对中断的控制包括：

（1）中断enable或者disable的控制。Distributor对中断的控制分成两个级别。一个是全局中断的控制。一旦disable了全局的中断，那么任何的interrupt source产生的interrupt event都不会被传递到CPU interface。另外一个级别是对针对各个interrupt source进行控制，disable某一个interrupt source会导致该interrupt event不会分发到CPU interface，但不影响其他interrupt source产生interrupt event的分发。

（2）控制中断事件分发到process。一个interrupt事件可以分发给一个process，也可以分发给若干个process。

（3）优先级控制。

（3）interrupt属性设定。例如是level-sensitive还是edge-triggered，是属于group 0还是group 1。

Distributor可以管理若干个interrupt source，这些interrupt source用ID来标识，我们称之interrupt ID。

2、CPU interface

CPU interface这个block主要用于和process进行接口。该block的主要功能包括：

（1）enable或者disable。对于ARM，CPU interface block和process之间的中断信号线是nIRQ和nFIQ这两个signal。如果disable了中断，那么即便是Distributor分发了一个中断事件到CPU interface，但是也不会assert指定的nIRQ或者nFIQ通知processor。

（2）ackowledging中断。processor会向CPU interface block应答中断，中断一旦被应答，Distributor就会把该中断的状态从pending状态修改成active。如果没有后续pending的中断，那么CPU interface就会deassert nIRQ或者nFIQ的signal。如果在这个过程中又产生了新的中断，那么Distributor就会把该中断的状态从pending状态修改成pending and active。这时候，CPU interface仍然会保持nIRQ或者nFIQ信号的asserted状态，也就是向processor signal下一个中断。

（3）中断处理完毕的通知。当interrupt handler处理完了一个中断的时候，会向写CPU interface的寄存器从而通知GIC CPU已经处理完该中断。做这个动作一方面是通知Distributor将中断状态修改为deactive，另外一方面，如果一个中断没有完成处理，那么后续比该中断优先级低的中断不会assert到processor。一旦标记中断处理完成，被block掉的那些比当前优先级低的中断就会递交给processor。

（4）设定priority mask。通过priority mask，可以mask掉一些优先级比较低的中断，这些中断不会通知到CPU。

（5）设定preemption的策略

（6）在多个中断事件同时到来的时候，选择一个优先级最高的通知processor

四、系统启动过程中，如何调用GIC driver的初始化函数

在linux-3.14\drivers\irqchip目录下保存在各种不同的中断控制器的驱动代码，irq-gic.c就是GIC的驱动代码。

1、IRQCHIP_DECLARE宏定义

在linux-3.14\drivers\irqchip目录下的irqchip.h文件中定义了IRQCHIP_DECLARE宏如下：

```
#define IRQCHIP_DECLARE(name,compstr,fn)                \
    static const struct of_device_id irqchip_of_match_##name    \
    __used __section(__irqchip_of_table)                \
    = { .compatible = compstr, .data = fn }
```

这个宏就是被各个irq chip driver用来声明其DT compatible string和初始化函数的对应关系的。IRQCHIP_DECLARE中的compstr就是DT compatible string，fun就是初始化函数。irq-gic.c文件中声明的对应关系包括：

```
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(msm_8660_qgic, "qcom,msm-8660-qgic", gic_of_init);
IRQCHIP_DECLARE(msm_qgic2, "qcom,msm-qgic2", gic_of_init);
```

从上面的定义可以看出来，A9和A15的gic初始化函数都是gic_of_init。另外两个定义是和高通的CPU相关，我猜测是高通使用了GIC，但是自己又做了一些简单的修改，但无论如何，初始化函数都是一个，那就是gic_of_init。

在linux kernel编译的时候，你可以配置多个irq chip进入内核，编译系统会把所有的IRQCHIP_DECLARE宏定义的数据放入到一个特殊的section中（section name是__irqchip_of_table），我们称这个特殊的section叫做irq chip table。这个table也就保存了kernel支持的所有的中断控制器的驱动代码初始化函数和DT compatible string的对应关系。具体执行哪一个初始化函数是由bootloader传递给kernel的DTB决定的。

2、OMAP4460的machine定义

在linux-3.14/arch/arm/mach-omap2目录下的board-generic.c的文件中定义了OMAP4460的machine如下：

```
DT_MACHINE_START(OMAP4_DT, "Generic OMAP4 (Flattened Device Tree)")
 。。。。。。删除无关代码
   .init_irq    = omap_gic_of_init,
。。。。。。删除无关代码
MACHINE_END
```

在系统初始化的时候，会调用start_kernel->init_IRQ->machine_desc->init_irq()函数。

3、omap_gic_of_init过程分析

omap_gic_of_init的代码如下（删除了无关代码）：

```
void __init omap_gic_of_init(void)
{
    irqchip_init();
}
```

在driver/irqchip/irqchip.c文件中定义了irqchip_init函数，如下：

```
void __init irqchip_init(void)
{
    of_irq_init(__irqchip_begin);
}
```

__irqchip_begin就是内核irq chip table的首地址，这个table也就保存了kernel支持的所有的中断控制器的驱动代码初始化函数和DT compatible string的对应关系。of_irq_init函数scan bootloader传递来的DTB，找到所有的中断控制器节点，并形成树状结构（系统可以有多个interrupt controller）。之后，从root interrupt controller开始，对于每一个interrupt controller，scan irq chip table，匹配DT compatible string，一旦匹配到，就调用该interrupt controller的初始化函数。具体的代码分析可以参考Device Tree代码分析文档。

五、GIC driver初始化代码分析

gic_of_init的代码如下：

```
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;

    dist_base = of_iomap(node, 0);----------------映射GIC Distributor的寄存器地址空间
    WARN(!dist_base, "unable to map gic dist registers\n");

    cpu_base = of_iomap(node, 1);----------------映射GIC CPU interface的寄存器地址空间
    WARN(!cpu_base, "unable to map gic cpu registers\n");

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

    gic_init_bases(gic_cnt, -1, dist_base, cpu_base, percpu_offset, node);))-----主处理过程，后面详述
    if (!gic_cnt)
        gic_init_physaddr(node); -----对于不支持big.LITTLE switcher（CONFIG_BL_SWITCHER）的系统，该函数为空。

    if (parent) {--------处理interrupt级联
        irq = irq_of_parse_and_map(node, 0);
        gic_cascade_irq(gic_cnt, irq);
    }
    gic_cnt++;
    return 0;
}
```

我们首先看看这个函数的参数，node参数代表需要初始化的那个interrupt controller的device node，parent参数指向其parent。对于cpu-offset binding，可以参考linux-3.14/Documentation/devicetree/bindings/arm/gic.txt文件中关于cpu-offset 的描述。对于OMAP4460，其GIC支持banked register，因此percpu_offset等于0。

gic_init_bases的代码如下：

```
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;

    BUG_ON(gic_nr >= MAX_GIC_NR);

    gic = &gic_data[gic_nr];
     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);

    /*
     * Initialize the CPU interface map to all CPUs.
     * It will be refined as each CPU probes its ID.
     */
    for (i = 0; i < NR_GIC_CPU_IF; i++)
        gic_cpu_map[i] = 0xff;

    /*
     * 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;
    }

    /*
     * Find out how many interrupts are supported.
     * The GIC only supports up to 1020 interrupt sources.
     */
    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;

    gic_irqs -= hwirq_base; /* calculate # of irqs to allocate */
    irq_base = irq_alloc_descs(irq_start, 16, 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;
    }
    gic->domain = irq_domain_add_legacy(node, gic_irqs, irq_base,
                    hwirq_base, &gic_irq_domain_ops, gic);
    if (WARN_ON(!gic->domain))
        return;

    if (gic_nr == 0) {
#ifdef CONFIG_SMP
        set_smp_cross_call(gic_raise_softirq);---------设定raise SGI的方法
        register_cpu_notifier(&gic_cpu_notifier);---------在multi processor环境下，当其他processor online
```

```
的时候，需要调用回调函数来初始化GIC的cpu interface。
#endif
        set_handle_irq(gic_handle_irq);
    }

    gic_chip.flags |= gic_arch_extn.flags;
    gic_dist_init(gic);---------具体的硬件初始代码，不再赘述，可以参考GIC reference manual
    gic_cpu_init(gic);
    gic_pm_init(gic);
}
```

这个函数的前半部分都是为了向系统中注册一个irq domain的数据结构。为何需要struct irq_domain这样一个数据结构呢？从linux kernel的角度来看，任何外部的设备的中断都是一个异步事件，kernel都需要识别这个事件。在内核中，用IRQ number来标识某一个设备的某个interrupt request。有了IRQ number就可以定位到该中断的描述符（struct irq_desc）。但是，对于中断控制器而言，它不并知道IRQ number，它只是知道HW interrupt number（中断控制器会为其支持的interrupt source进行编码，这个编码被称为Hardware interrupt number ）。不同的软件模块用不同的ID来识别interrupt source，这样就需要映射了。如何将Hardware interrupt number 映射到IRQ number呢？这需要一个translation object，内核定义为struct irq_domain。

每个interrupt controller都会形成一个irq domain，负责解析其下游的interrut source。如果interrupt controller有级联的情况，那么一个非root interrupt controller的中断控制器也是其parent irq domain的一个普通的interrupt source。struct irq_domain定义如下：

```
struct irq_domain {
……
    const struct irq_domain_ops *ops;
    void *host_data;

……
};
```

这个数据结构是属于linux kernel通用中断子系统的一部分，我们这里只是描述相关的数据成员。host_data成员是底层interrupt controller的私有数据，linux kernel通用中断子系统不应该修改它。对于GIC而言，host_data成员指向一个struct gic_chip_data的数据结构，定义如下：

```
struct gic_chip_data {
    union gic_base dist_base;---------GIC Distributor的地址空间
    union gic_base cpu_base;---------GIC CPU interface的地址空间
#ifdef CONFIG_CPU_PM---------GIC 电源管理相关的成员
    u32 saved_spi_enable[DIV_ROUND_UP(1020, 32)];
    u32 saved_spi_conf[DIV_ROUND_UP(1020, 16)];
    u32 saved_spi_target[DIV_ROUND_UP(1020, 4)];
    u32 __percpu *saved_ppi_enable;
    u32 __percpu *saved_ppi_conf;
#endif
    struct irq_domain *domain;---------该GIC对应的irq domain数据结构
    unsigned int gic_irqs;---------GIC支持的IRQ的数目
#ifdef CONFIG_GIC_NON_BANKED
    void __iomem *(*get_base)(union gic_base *);
#endif
};
```

对于GIC支持的IRQ的数目，这里还要赘述几句。实际上并非GIC支持多少个HW interrupt ID，其就支持多少个IRQ。对于SGI，其处理比较特别，并不归入IRQ number中。因此，对于GIC而言，其SGI（从0到15的那些HW interrupt ID）不需要irq domain进行映射处理，也就是说SGI没有对应的IRQ number。如果系统越来越复杂，一个GIC不能支持所有的interrupt source（目前GIC支持1020个中断源，这个数目已经非常的大了），那么系统还需要引入secondary GIC，这个GIC主要负责扩展外设相关的interrupt source，也就是说，secondary GIC的SGI和PPI都变得冗余了（这些功能，primary GIC已经提供了）。这些信息可以协助理解代码中的hwirq_base的设定。

一个GIC支持的IRQ的数目可以通过其支持的HW interrupt的数目减去hwirq_base得到，那么如何获取GIC支持的HW interrupt的数目呢？GIC有一个寄存器叫做Interrupt Controller Type Register，通过这个寄存器可以获得下列信息：

（1）whether the GIC implements the Security Extensions
（2）the maximum number of interrupt IDs that the GIC supports
（3）the number of CPU interfaces implemented
（4）if the GIC implements the Security Extensions, the maximum number of implemented Lockable Shared Peripheral Interrupts (LSPIs).

获得了GIC支持的IRQ数目后，就可以调用irq_alloc_descs函数分配IRQ描述符资源了。之后，通过调用irq_domain_add_legacy函数注册GIC的irq domain数据结构。这里有一个重要的数据结构gic_irq_domain_ops，其类型是struct irq_domain_ops ，定义如下：

```
struct irq_domain_ops {
    int (*match)(struct irq_domain *d, struct device_node *node);
    int (*map)(struct irq_domain *d, unsigned int virq, irq_hw_number_t hw);
    void (*unmap)(struct irq_domain *d, unsigned int virq);
    int (*xlate)(struct irq_domain *d, struct device_node *node,
            const u32 *intspec, unsigned int intsize,
            unsigned long *out_hwirq, unsigned int *out_type);
};
```

对于GIC，其irq domain的操作函数是gic_irq_domain_ops，定义如下：

```
const struct irq_domain_ops gic_irq_domain_ops = {
    .map = gic_irq_domain_map,
    .xlate = gic_irq_domain_xlate,
};
```

irq domain的概念是一个通用中断子系统的概念，在irq chip这个层次，我们需要和通用中断子系统中的irq domain core code进行接口，gic_irq_domain_ops就是这个接口的定义。

（1）map成员函数：用来建立irq number和hw interrupt ID之间的联系
（2）Xlate成员函数：给定某个外设的device tree node 和interrupt specifier，该函数可以解码出该设备使用的hw interrupt ID和linux irq type value

具体向系统注册irq domain是通过irq_domain_add_legacy函数进行的：

```
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;

    domain = __irq_domain_add(of_node, first_hwirq + size,
                first_hwirq + size, 0, ops, host_data);
    if (!domain)
        return NULL;

    irq_domain_associate_many(domain, first_irq, first_hwirq, size);

    return domain;
}
```

这个函数主要进行两个步骤，一个是通过__irq_domain_add将GIC对应的irq domain加入到系统，另外一个就是调用irq_domain_associate_many来建立IRQ number和hw interrupt ID之间的关系。而这个关系的建立实际上是通过irq domain的操作函数中的map回调函数进行的，对于GIC而言就是gic_irq_domain_map，代码如下：

```
static int gic_irq_domain_map(struct irq_domain *d, unsigned int irq,
                irq_hw_number_t hw)
{
```

```
    if (hw < 32) {
        irq_set_percpu_devid(irq);
        irq_set_chip_and_handler(irq, &gic_chip,
                 handle_percpu_devid_irq);
        set_irq_flags(irq, IRQF_VALID | IRQF_NOAUTOEN);
    } else {
        irq_set_chip_and_handler(irq, &gic_chip,
                 handle_fasteoi_irq);
        set_irq_flags(irq, IRQF_VALID | IRQF_PROBE);
    }
    irq_set_chip_data(irq, d->host_data);
    return 0;
}
```

这个函数的主要功能就是把一个IRQ号（参数中的unsigned int irq）和一个hw interrupt ID（参数中的irq_hw_number_t hw）联系起来。每个IRQ都有其对应的描述符，用struct irq_desc表示：

```
struct irq_desc {
    struct irq_data        irq_data;

    irq_flow_handler_t    handle_irq;
……
} __
```

函数irq_set_chip_and_handler其实主要就是设定struct irq_desc中的irq_data和handle_irq成员。handle_irq就是发生了一个IRQ后需要调用的中断处理函数。irq_data数据结构中的chip成员就是指向具体的硬件信息，也就是GIC的描述符。我们定义如下：

```
static struct irq_chip gic_chip = {
    .name            = "GIC",
    .irq_mask        = gic_mask_irq,
    .irq_unmask        = gic_unmask_irq,
    .irq_eoi        = gic_eoi_irq,
    .irq_set_type        = gic_set_type,
    .irq_retrigger        = gic_retrigger,
#ifdef CONFIG_SMP
    .irq_set_affinity    = gic_set_affinity,
#endif
    .irq_set_wake        = gic_set_wake,
};
```

struct irq_chip数据结构中有各种各样的操作函数，例如enable或者disable一个中断，ack一个中断等。

综上所述，经过了初始化过程之后，对于linux的内存管理系统而言，增加了GIC的地址空间映射。对于linux kernel的generic interrupt subsystem，增加了若干个IRQ的描述符，并且设定了这些IRQ描述符的处理函数以及相关的irq_data数据结构（和具体HW相关，该数据结构的成员包括抽象具体硬件的interrupt controller的描述符以及irq domain数据结构）。

六、GIC硬件操作

1、gic_mask_irq函数

这个函数用来mask一个interrupt source。代码如下：

```
static void gic_mask_irq(struct irq_data *d)
{
    u32 mask = 1 << (gic_irq(d) % 32);

    raw_spin_lock(&irq_controller_lock);
    writel_relaxed(mask, gic_dist_base(d) + GIC_DIST_ENABLE_CLEAR + (gic_irq(d) / 32) * 4);
    if (gic_arch_extn.irq_mask)
        gic_arch_extn.irq_mask(d);
```

```
    raw_spin_unlock(&irq_controller_lock);
}
```

GIC有若干个叫做Interrupt Clear-Enable Registers（具体数目是和GIC支持的hw interrupt数目相关，我们前面说过的，GIC是一个高度可配置的interrupt controller）。这些Interrupt Clear-Enable Registers寄存器的每个bit可以控制一个interrupt source是否forward到CPU interface，写入1表示Distributor不再forward该interrupt，因此CPU也就感知不到该中断，也就是mask了该中断。特别需要注意的是：写入0无效，而不是unmask的操作。

由于不同的SOC厂商在集成GIC的时候可能会修改，也就是说，也有可能mask的代码要微调，这是通过gic_arch_extn这个全局变量实现的。在gic-irq.c中这个变量的全部成员都设定为NULL，各个厂商在初始中断控制器的时候可以设定其特定的操作函数。

2、gic_unmask_irq函数

这个函数用来unmask一个interrupt source。代码如下：

```
static void gic_unmask_irq(struct irq_data *d)
{
    u32 mask = 1 << (gic_irq(d) % 32);

    raw_spin_lock(&irq_controller_lock);
    if (gic_arch_extn.irq_unmask)
        gic_arch_extn.irq_unmask(d);
    writel_relaxed(mask, gic_dist_base(d) + GIC_DIST_ENABLE_SET + (gic_irq(d) / 32) * 4);
    raw_spin_unlock(&irq_controller_lock);
}
```

GIC有若干个叫做Interrupt Set-Enable Registers的寄存器。这些寄存器的每个bit可以控制一个interrupt source。当写入1的时候，表示Distributor会forward该interrupt到CPU interface，也就是意味这unmask了该中断。特别需要注意的是：写入0无效，而不是mask的操作。

3、gic_eoi_irq函数

当processor处理中断的时候就会调用这个函数用来结束中断处理。代码如下：

```
static void gic_eoi_irq(struct irq_data *d)
{
    if (gic_arch_extn.irq_eoi) {
        raw_spin_lock(&irq_controller_lock);
        gic_arch_extn.irq_eoi(d);
        raw_spin_unlock(&irq_controller_lock);
    }

    writel_relaxed(gic_irq(d), gic_cpu_base(d) + GIC_CPU_EOI);
}
```

对于GIC而言，其中断状态有四种：

| 中断状态 | 描述 |
| --- | --- |
| Inactive | 中断未触发状态，该中断即没有Pending也没有Active |
| Pending | 由于外设硬件产生了中断事件（或者软件触发）该中断事件已经通过硬件信号通知到GIC，等待GIC分配的那个CPU进行处理 |
| Active | CPU已经应答（acknowledge）了该interrupt请求，并且正在处理中 |
| Active and Pending | 当一个中断源处于Active状态的时候，同一中断源又触发了中断，进入pending状态 |

processor ack了一个中断后，该中断会被设定为active。当处理完成后，仍然要通知GIC，中断已经处理完毕了。这时候，如果没有pending的中断，GIC就会将该interrupt设定为inactive状态。操作GIC中的End of Interrupt Register可以完成end of interrupt事件通知。

4、gic_set_type函数

这个函数用来设定一个interrupt source的type，例如是level sensitive还是edge triggered。代码如下：

```
static int gic_set_type(struct irq_data *d, unsigned int type)
{
    void __iomem *base = gic_dist_base(d);
    unsigned int gicirq = gic_irq(d);
    u32 enablemask = 1 << (gicirq % 32);
    u32 enableoff = (gicirq / 32) * 4;
    u32 confmask = 0x2 << ((gicirq % 16) * 2);
    u32 confoff = (gicirq / 16) * 4;
    bool enabled = false;
    u32 val;

    /* Interrupt configuration for SGIs can't be changed */
    if (gicirq < 16)
        return -EINVAL;

    if (type != IRQ_TYPE_LEVEL_HIGH && type != IRQ_TYPE_EDGE_RISING)
        return -EINVAL;

    raw_spin_lock(&irq_controller_lock);

    if (gic_arch_extn.irq_set_type)
        gic_arch_extn.irq_set_type(d, type);

    val = readl_relaxed(base + GIC_DIST_CONFIG + confoff);
    if (type == IRQ_TYPE_LEVEL_HIGH)
        val &= ~confmask;
    else if (type == IRQ_TYPE_EDGE_RISING)
        val |= confmask;

    /*
     * As recommended by the spec, disable the interrupt before changing
     * the configuration
     */
    if (readl_relaxed(base + GIC_DIST_ENABLE_SET + enableoff) & enablemask) {
        writel_relaxed(enablemask, base + GIC_DIST_ENABLE_CLEAR + enableoff);
        enabled = true;
    }

    writel_relaxed(val, base + GIC_DIST_CONFIG + confoff);

    if (enabled)
        writel_relaxed(enablemask, base + GIC_DIST_ENABLE_SET + enableoff);

    raw_spin_unlock(&irq_controller_lock);

    return 0;
}
```

对于SGI类型的interrupt，是不能修改其type的，因为GIC中SGI固定就是edge-triggered。对于GIC，其type只支持高电平触发（IRQ_TYPE_LEVEL_HIGH）和上升沿触发（IRQ_TYPE_EDGE_RISING）的中断。另外需要注意的是，在更改其type的时候，先disable，然后修改type，然后再enable。

5、gic_retrigger

这个接口用来resend一个IRQ到CPU。

```
static int gic_retrigger(struct irq_data *d)
{
    if (gic_arch_extn.irq_retrigger)
        return gic_arch_extn.irq_retrigger(d);

    /* the genirq layer expects 0 if we can't retrigger in hardware */
    return 0;
}
```

看起来这是功能不是通用GIC拥有的功能，各个厂家在集成GIC的时候，有可能进行功能扩展。

6、gic_set_affinity

在多处理器的环境下，外部设备产生了一个中断就需要送到一个或者多个处理器去，这个设定是通过设定处理器的affinity进行的。具体代码如下：

```
static int gic_set_affinity(struct irq_data *d, const struct cpumask *mask_val,
            bool force)
{
    void __iomem *reg = gic_dist_base(d) + GIC_DIST_TARGET + (gic_irq(d) & ~3);
    unsigned int shift = (gic_irq(d) % 4) * 8;
    unsigned int cpu = cpumask_any_and(mask_val, cpu_online_mask);
    u32 val, mask, bit;

    if (cpu >= NR_GIC_CPU_IF || cpu >= nr_cpu_ids)
        return -EINVAL;

    raw_spin_lock(&irq_controller_lock);
    mask = 0xff << shift;
    bit = gic_cpu_map[cpu] << shift;
    val = readl_relaxed(reg) & ~mask;
    writel_relaxed(val | bit, reg);
    raw_spin_unlock(&irq_controller_lock);

    return IRQ_SET_MASK_OK;
}
```

GIC Distributor中有一个寄存器叫做Interrupt Processor Targets Registers，这个寄存器用来设定制定的中断送到哪个process去。由于GIC最大支持8个process，因此每个hw interrupt ID需要8个bit来表示送达的process。每一个Interrupt Processor Targets Registers由32个bit组成，因此每个Interrupt Processor Targets Registers可以表示4个HW interrupt ID的affinity。

7、gic_set_wake

这个接口用来设定唤醒CPU的interrupt source。对于GIC，代码如下：

```
static int gic_set_wake(struct irq_data *d, unsigned int on)
{
    int ret = -ENXIO;

    if (gic_arch_extn.irq_set_wake)
        ret = gic_arch_extn.irq_set_wake(d, on);

    return ret;
}
```

设定唤醒的interrupt和具体的厂商相关，这里不再赘述。

*原创文章，转发请注明出处。蜗窝科技，www.wowotech.net。*

**标签: GIC 代码分析**

« 智慧家庭之我见_多媒体篇 | 在PowerShell中使用Vim»

**评论：**

**maotou**
2014-09-02 15:19

hi linuxer,看了好几遍您的文章,我刚接触linux内核,从arm中的中断处理开始,看了一个月还是挺迷茫,虽然感觉自己对原理理解了,可还是不知道怎么从代码中去看,因为各个代码都是散的,没有一个很清楚的调理.linuxer前辈有些什么好的建议么? 还有对于GIC有个不能理解的是,一个GIC能提供1024个中断号,那么他是有1024个引脚去接到设备中么?

回复