Interrupt in Linux（硬件篇）

—— 细节、实现，与疑问

Author: ZX_WING (xing5820@163.com)

Contributor: BLUESKY_JXC

（此文基于 **2.6.20** 内核版本）

写在前面的话

前段时间在 ChinaUnix 和很多朋友一起讨论了 Linux 中断的问题，发现不少地方都搞不清楚。于是静下心来，收集了一些资料，配合源代码进行了学习。经过一个月的时间，已经大体了解了关于中断的细节和实现，但仍然有很多疑问，我把它们总结了一下，写下这篇文章，和大家一起讨论。个人水平有限，难免有错，希望大家在发现错误后能反馈到我的邮箱，我好及时更正。

这篇文章不是学习 Linux 中断的入门文章，如果对中断没有概念，笔者建议先阅读《Understanding Linux Kernel(3thd Edition)》。此文是对 ULK3 的补充，讲了一些它没讲的内容。文中的硬件知识，适用于 Intel x86 平台、x86_64 平台，但不适用于 IA64 的 SAPIC 系统。

此外，本文对于 Linux 源码的讲解采用的是 ULK3 的模式，笔者比较反感源码注释。对于开源项目来说，“告诉我你要做什么，但不要告诉我你是怎么做的”、“代码就是最好的注释”，这是我的信条。

内容提要

第一章：介绍了 PIC 和 APIC 系统的基本架构，提供了了解现代中断系统构成的基本知识。

第二章：论述了 Linux 如何探测中断硬件，以及如何初始化它们。

第三章：补充了一些中断系统的硬件知识，没有它们你也应该能读懂前两章的内容。

文中用大量“题外话”介绍了中断相关的知识和原理，它们大部分是笔者感兴趣的，例如“Remote IRR 的作用”、“Edge 中断的共享与丢失”、“伪中断产生的原因”等。以“笔者”开头的文字，是作者自己对一些问题的看法，其中有很多不能解决的疑问，如果你知道答案，希望能通过 xing5820@163.com 告诉我，让我及时更新相关内容。同时，非常欢迎指出文中的错误之处。

此文虽取名为“硬件篇”，但不代表就有一个“软件篇”存在。虽然目前内核使用了 Generic Interrupt Layer，但这只是对原__do_IRQ()路径的封装，ULK3 的内容完全适用于当前内核中断系统，软件相关内容可以参考此书。

Revision History

日期	版本	描述
2008.5.3	1.1	• 根据 albcamus 同学的建议，修改了文中多处错误 • 根据 Bluesky_jxc 同学的补充，修改了 edge 中断共享与丢失的内容 • Bluesky_jxc 同学补充了 PIRQ Table 章节
2008.4.27	1.0	最初发表版本

目录

从某种意义上来说，现代 PC 架构是由中断驱动的。这种大量发生的异步事件，让顺序的二进制代码在不同的执行路径上跳转，勾画出操作系统的整个脉络。

中断是硬件和软件交互的机制，一个中断的产生经历了 设备 ---- 中断控制器 ---- CPU（噢，让我们先不考虑 MSI）3 个流程。要了解 Linux 中断系统的构成，硬件知识是少不了的。我们就先从硬件说起。

第一章 中断控制器概述

中断控制器是连接设备和 CPU 的桥梁，一个设备产生中断后，需要经过中断控制器的转发，才能最终到达 CPU。时代发展至今，中断控制器经历了 PIC(Programmable Interrupt Controller，可编程中断控制器)和 APIC（Advanced Programmable Interrupt Controller，高级可编程中断控制器）两个阶段。前者在 UP（Uni-processor，单处理器）上叱诧风云，并威风至今，我这台 T42 的记笔记本上用的就是它。随着 SMP(Symmetric Multiple Processor，对称多处理器)的流行，APIC 已广为流行并将最终取代 PIC。

1.1 史前的 PIC

8259A 是即我们通常说的 PIC，如图 1-1 所示：

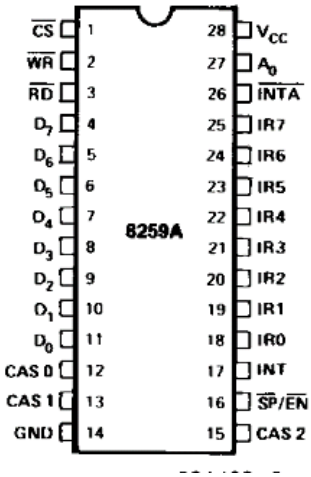


图 1-1 8259A

其中最重要的管脚是 IR0~IR7（Interrupt Request0~7，用于连接设备）、INT（连接 CPU，当有中断请求时，拉高该管脚以通知 CPU 中断的到来）、INTA（连接 CPU，CPU 通过该管脚应答中断请求，并通知 PIC 提交中断的 vector 到数据线）。此外，由于一片 8259A 只能连接 8 个设备，对于现代 PC 架构来说显得过少，通常会通过 CS（片选）将两个 8259A 连在一起构成一个可以连接 15 个设备（有一个管脚用于串联另一片 8259A）的 PIC。

既然是可编程的芯片，8259A 当然少不了寄存器。除了 ICW（Initialization Command

Word，初始化命令寄存器，用于初始化 8259A）和 OCW（Operation Command Word，操作命令字，用于控制 8259A）外，最重要的有 3 个寄存器：

- IRR：Interrupt Request Register，中断请求寄存器，共 8bit，对应 IR0~IR7 八个中断管脚。当某个管脚的中断请求到来后，若该管脚没有被屏蔽，IRR 中对应的 bit 被置一。表示 PIC 已经收到设备的中断请求，但还未提交给 CPU。
- ISR：In Service Register，服务中寄存器，共 8bit，每 bit 意义同上。当 IRR 中的某个中断请求被发送给 CPU 后，ISR 中对应的 bit 被置一。表示中断已发送给 CPU，但 CPU 还未处理完。
- IMR：Interrupt Mask Register，中断屏蔽寄存器，共 8bit，每 bit 意义同上。用于屏蔽中断。当某 bit 置一时，对应的中断管脚被屏蔽。

与 APIC 不同，PIC 的每个管脚具有优先级，以 0 号管脚最高。也就说，连接号码较小的设备具有较高的中断优先级。通过对 PIC 的 ICW 寄存器编程，可以设定起始 vector 号，以计算当前中断的 vector。例如，起始 vector 号设为 16，IR3 管脚产生了中断请求，则 IR3 对应的 vector = 16+3 =19。通过 PIC 发起中断的典型流程如下：

1、 一个或多个 IR 管脚上产生电平信号，若对应的中断没有被屏蔽，IRR 中相应的 bit 被置一。
2、 PIC 拉高 INT 管脚通知 CPU 中断发生。
3、 CPU 通过 INTA 管脚应答 PIC，表示中断请求收到。
4、 PIC 收到 INTA 应答后，将 IRR 中具有最高优先级的 bit 清零，并设置 ISR 中对应的 bit。
5、 CPU 通过 INTA 管脚第二次发出脉冲，PIC 收到后计算最高优先级中断的 vector，并将它提交到数据线上。
6、 等待 CPU 写 EOI。收到 EOI 后，ISR 中最高优先级的 bit 被清零。如果 PIC 处于 AEOI 模式，当第二个 INTA 脉冲收到后，ISR 中最高优先级的 bit 自动清零。

上述流程中，EOI（End Of Interrupt）是 CPU 通知 PIC 中断处理结束的方式，由中断处理程序发起。对于 PIC 来说，EOI 由 OCW 寄存器中的一个 bit 表示，可以通过写该寄存器发起 EOI。AEOI（Auto EOI）是 PIC 的一种工作模式，由 OCW 寄存器配置。PIC 有多种工作模式，我们最常使用的是 PIC 默认的模式，被称为 Full Nested Mode。有兴趣的朋友可以参考 8259A 的 spec 了解详细信息。

前面提到，PIC 的各个管脚是具有优先级的，当一个中断正被 CPU 处理时，优先级等于或低于该中断的中断被自动屏蔽。一个比当前中断优先级更高的中断会被马上发送给 CPU，而不管 CPU 是否为当前中断写 EOI。

PIC 产生于那个动态配置还很少见的时代，大多中断管脚都被一些 legacy 设备（leagcy 可以理解为老式设备）占用了，一个典型的 PIC 中断分配见图 1-2：

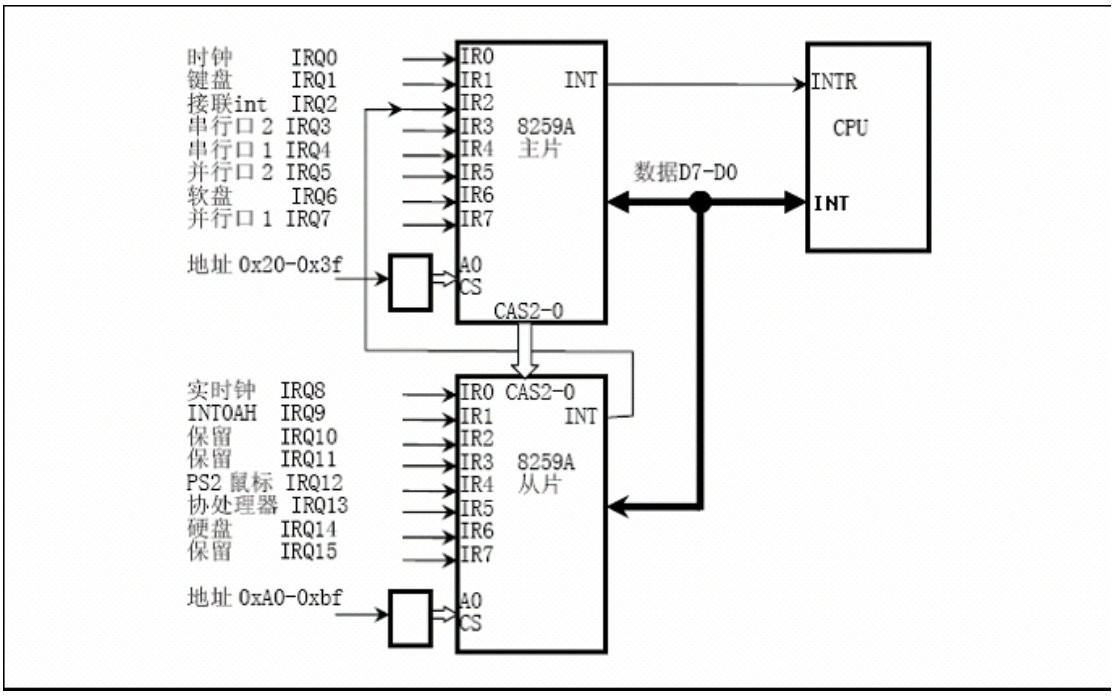


图 1-2 典型的 PIC 中断分配（摘自《Linux 内核完全注释》）

当然，现代 PC 通常没有图中那么多并口、串口，也没有协处理器，一般来说 5、7、9、10、11 管脚是可以被其它设备使用的。但万恶的是，出于兼容的目的，即使某个管脚没有接 legacy 设备，BIOS 通常也会把它预留下来，例如 IRQ3、4、13 就经常是这种情况。由此可见，PIC 能接的设备实在太少了，更致命的是，它无法适用于 MP（Multiple Processor，多处理器）平台。随着 SMP 越来越普及，PIC 注定要退出历史舞台，让位给来者。

笔者：上面对于 PIC 工作模式的论述，特别是优先级的论述，是基于 PIC 默认的 Full Nested 模式说的。这是 PIC 最常用的模式。实际上 PIC 还有优先级轮转(rotating)、特殊优先级轮转模式。优先级轮转是指 PIC 在服务完一个管脚后将其优先级降低，并升高未服务管脚的优先级，以实现一种类似轮询的模式。这和后面讲到的 LAPIC Arb 机制类似。

1.2 现代的 APIC

APIC 虽号称现代，但也出现 10 几年了，PC 机市场总是很晚才能接触到新的技术，前面说了，我的 T42 用的还是 PIC 呢。APIC 相较于 PIC 来说，最大的优点是能适用于 MP 平台，当然，管脚多是它另一个优点。APIC 由两部分组成，一个称为 LAPIC（Local APIC，本地高级中断控制器），一个称为 IOAPIC（I/O APCI，I/O 高级中断控制器）。前者位于 CPU 中，在 MP 平台，每个 CPU 都有一个自己的 LAPIC。后者通常位于南桥上，像 PIC 一样，连接各个产生中断的设备。在一个典型的具有多个处理器的 PC 平台，通常有一个 IOAPIC 和多个 LAPIC，它们相互配合，形成一个中断的分发网络，图 1-3 显示了这个典型的情况：

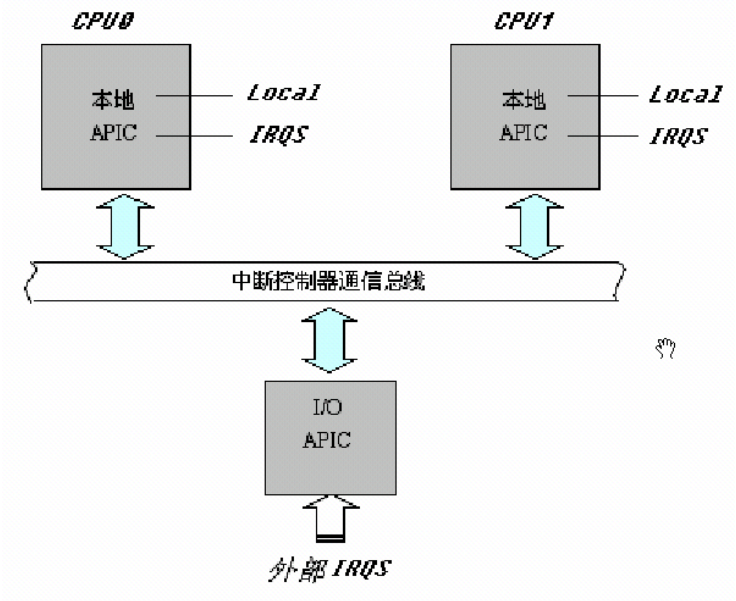


图 1-3 APIC 模式（摘自《深入理解 Linux 内核》）

图中的中断控制器通信总线，是 IOAPIC 和 LAPIC 通信的桥梁，在 Intel 的 P6 架构和 Pentium 系列 CPU 中，它是一条单独的 APIC 总线。时代在进步，Pentium4 和 Xeon 系列 CPU 出现后，APIC Bus 已经不存在了， 取而代之的是， LAPIC 之间、LAPIC 和 IOAPIC 之间的通信，通过前端总线来传递。

1.2.1 IOAPIC

话分两头，让我们先来看看 IOAPIC。和 PIC 对比，IOAPIC 最大的作用在于中断分发。根据其内部的 PRT（Programmable Redirection Table）表，IOAPIC 可以格式化出一条中断消息，发送给某个 CPU 的 LAPIC，由 LAPIC 通知 CPU 进行处理。目前典型的 IOAPIC 具有 24 个中断管脚，每个管脚对应一个 RTE（Redirection Table Entry，PRT 表项）。与 PIC 不同的是，IOAPIC 的管脚没有优先级，也就是说，连接在管脚上的设备是平等的。但这并不意味着 APIC 系统中没有硬件优先级。设备的中断优先级由它对应的 vector 决定，APIC 将优先级控制的功能放到了 LAPIC 中，我们在后面会看到。

要搞清楚 IOAPIC 是怎么工作的，PRT 表是关键，下表列出了 RTE 的格式：

Bit	描述
63:56	Destination Field，目的字段，R/W（可读写）。根据 Destination Filed（见下）值的不同，该字段值的意义不同，它有两个意义： Physical Mode（Destination Mode 为 0 时）： 其值为 APIC ID，用于标识一个唯一的 APIC。 Logical Mode（Destination Mode 为 1 时）：其值根据 LAPIC 的不同配置，代表一组 CPU（具体见 LAPIC 相关内容）
55:17	Reserved，预留未用。
16	Interrupt Mask，中断屏蔽位，R/W。置一时，对应的中断管脚被屏蔽，这时产生的

	中断将被忽略。清零时，对应管脚产生的中断被发送至 LAPIC。
15	Trigger Mode，触发模式，R/W。指明该管脚的的中断由什么方式触发。 1：Level，电平触发 2：Edge，边沿触发
14	Remote IRR，远程 IRR，RO（只读）。只对 level 触发的中断有效，当该中断是 edge 触发时，该值代表的意义未定义。 当中断是 level 触发时，LAPIC 接收了该中断，该位置一，LAPIC 写 EOI 时，该位清零。
13	Interrupt Input Pin Polarity（INTPOL），中断管脚的极性，R/W。指定该管脚的有效电平是高电平还是低电平。 0：高电平 1：低电平
12	Delivery Status，传送状态，RO。 0：IDEL，当前没有中断 1：Send Pending，IOAPIC 已经收到该中断，但由于某种原因该中断还未发送给 LAPIC 笔者：某种原因，例如 IOAPIC 没有竞争到总线
11	Destination Mode，目的地模式，R/W。 0：Physical Mode，解释见 Destination Field 1：Logical Mode，同上
10:8	Delivery Mode，传送模式，R/W。用于指定该中断以何种方式发送给目的 APIC，各种模式需要和相应的触发方式配合。可选的模式如下，字段相应的值以二进制表示： Fixed： 000b，发送给 Destination Filed 列出的所有 CPU，level、edge 触发均可。 Lowest Priority：001b，发送给 Destination Filed 列出的 CPU 中，优先级最低的 CPU（CPU 的优先级见 LAPIC 相关内容）。Level、edge 均可 SMI：010b，System Management Interrupt，系统管理中断。只能为 edge 触发，并且 vector 字段写 0 NMI：100b，None Mask Interrupt，不可屏蔽中断。发送给 Destination Field 列出的所有 CPU，Vector 字段值被忽略。NMI 是 edge 触发，Trigger Mode 字段中的值对 NMI 无影响，但建议配置成 edge。 INIT：101b，发送给 Destination Filed 列出的所有 CPU，LAPIC 收到后执行 INIT 中断（详细信息参考相关 CPU spec 中 INIT 中断一节）。触发模式同 NMI。 ExtINT：111b，发送给 Destination Filed 列出的所有 CPU。CPU 收到该中断后，认为这是一个 PIC 发送的中断请求，并回应 INTA 信号（该 INTA 脚连接到的是与该管脚相连的 PIC 上，而非 IOAPIC 上） 笔者：ExtINT 用于 PIC 接在 APIC 上的情况，见后面的 Virtual Wire Mode
7:0	Interrupt Vector，中断向量，R/W。指定该中断对应的 vector，范围从 10h 到 FEh（x86 架构前 16 个 vector 被系统预留，见后面相关内容）

表 1-1 RTE 格式

当 IOAPIC 某个管脚接收到中断信号后，会根据该管脚对应的 RTE，格式化出一条中断消息，发送给某个（或多个）CPU 的 LAPIC。从上表我们可以看出，该消息包含了一个中断的所有信息。

题外话 —— Remote IRR 有什么用？

Bluesky_jxc 同学曾经问我为啥 Remote IRR 对 edge 触发无用？我说可能是实现相关吧。嗯，通常解释不清楚的问题都可以推到实现相关的头上。这个问题可以画 3 个等号："Remote IRR 有何用？" == "为什么 edge 触发不用 Remote IRR？" == "为什么 level 触发的 EOI 要广播到所有 IOAPIC？"

这确实是个实现问题，Intel 公司的 P.K.Nizar 等人在《Multi-Processor Computer System With Interrupt Controllers Providing Remote Reading》中论述了这个问题。这是一篇讲早期 APIC 系统电路设计的文章，那时还不叫 APIC，而是 MPIC（Multi-PIC），扯远了。

Remote IRR 实际应该叫"Monitor Remote IRR"，用于监控对应中断管脚的状态。它与中断管脚 INTIN#以异或的逻辑驱动 IOAPIC 的消息单元。异或结果为 1 时，发送消息。消息分两种：level-assert 和 level-deassert。当 Remote IRR 为 0，INTIN#为 1，发送 level-assert 消息，LAPIC 收到后将 IRR 对应 bit 置一。Remote IRR 为 1，INTIN#为 0，发送 level-deassert 消息，LAPIC 收到后将 IRR 对应 bit 清零。

Remote IRR 还可以保证 level 触发中断共享情况下，CPU 服务完所有中断。

举个例子，有两个 PCI 设备（Dev A、Dev B）共享一个中断管脚 INTIN1，Dev A 先把管脚拉至有效电平（INTIN1 为 1），Dev B 在一段时间后也同样动作（此时 Dev A 仍然在有效电平）。当 INTIN1 由 0 跳变到 1 时，其对应的 Remote IRR bit 为 0。IOAPIC 发送 Level Assert 消息通知中断发生，Remote IRR 置 1。当 CPU 处理完 Dev A 的中断后，发送 EOI 到 IOAPIC。此时 Remote IRR 清 0，由于 Dev B 的中断还没处理，INTIN1 仍然为 1，故又一条中断消息产生。在所有中断处理完后，INTIN1 清 0，Remote IRR 为 1，发送 level-deassert 消息，LAPIC 清零 IRR 对应 bit。此时 INTIN1 对应的中断全部处理完。在 LAPIC 为 Dev B 写 EOI 时，Remote IRR 清零。当 CPU 正在处理中断时，INT1 为 1，Remote IRR 为 1，没有中断消息。

对于 edge 触发中断，由于中断管脚不会一直处于有效电平，故不需要 Remote IRR。EOI 广播的原因，上面的论述已有暗示，不废话了。

1.2.2 LAPIC

收到来自 IOAPIC 的中断消息后，LAPIC 会将该中断交由 CPU 处理。和 IOAPIC 比较，LAPIC 具有更多的寄存器以及更复杂的机制。但对于处理来自 IOAPIC 的中断消息，最重要的寄存器还是 IRR、ISR 以及 EOI。

图 1-4 显示了 x86 平台上，IRR 和 ISR 的格式：

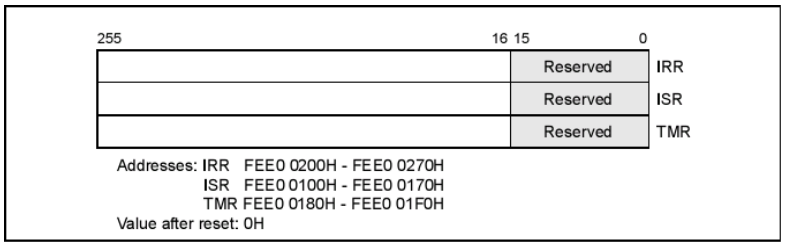


图 1-4 IRR、ISR 构成

与 PIC 中的 IRR、ISR 不同的是，LAPIC 的 ISR、IRR 均为 256bit 寄存器，对应 x86 平台上的 256 个中断 vector，其中 0~15 为架构预留。

- IRR：功能和 PIC 的类似，代表 LAPIC 已接收中断，但还未交 CPU 处理。
- ISR：功能和 PIC 类似，代表 CPU 已开始处理中断，但还未完成。与 PIC 有所不同的是，当 CPU 正在处理某中断时，同类型中断如果发生，相应的 IRR bit 会再次置一（PIC 模式下，同类型的中断被屏蔽）；如果某中断被 pending 在 IRR 中，同类型的中断发生，则 ISR 中相应的 bit 被置一。这说明在 APIC 系统中，同一类型中断最多可以被计数两次（想不通什么意思？想不通就联想一下 Linux 可信信号）。超过两次时，不同架构处理不一样。对于 Pentium 系列 CPU 和 P6 架构，中断消息被 LAPIC 拒绝；对于 Pentium4 和 Xeon 系列，新来的中断和 IRR 中对应的 bit 重叠。

笔者：上图还有个 TMR 寄存器，即 Trigger Mode Register，用于表示当前正在处理中断的触发模式。1 为 level，0 为 edge。对于 level 触发的中断，当软件写 EOI 时，会被广播到所有 IOAPIC，消息中含有中断的 vector，IOAPIC 收到后检查自己的 PRT 表，把相应 RTE 的 Remote IRR 位清零。

对于 IRR，x86 spec 说："当 CPU 准备处理中断时，IRR 中最高优先级的 bit 被清零，ISR 中对应 bit 被置一"。这样说不能算错，但至少不准确。根据《Multi-Processor Computer System With Interrupt Controllers Providing Remote Reading》一文中 APIC 的实现，只有对于 edge 触发的中断，ISR 对应 bit 置一时 IRR 相应 bit 才清零。对于 level 触发，IRR 中的 bit 保留到软件写 EOI，LAPIC 收到 IOAPIC 发出的 Level-deassert 消息后才清零。

如果你仔细看了前面关于 Remote IRR 的介绍，此时一定会产生一个疑问。这里说同类型中断发生两次，会同时 Pending 在 ISR 和 IRR 的对应 bit 中。问题是，前面 Remote IRR 的异或逻辑保证了在写 EOI 前，新的中断不会发过来啊？呵呵，或许你想到答案了，这种 pending 两次的机制，只对 edge 触发中断有效。

与 PIC 一样，LAPIC 同样需要软件写 EOI 来知会中断处理的完成，不同的是，LAPIC 中的 EOI 是一个 32bit 寄存器，如下图：

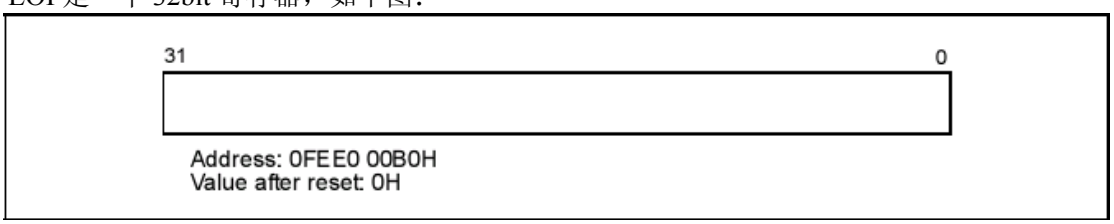


图 1-5 EOI 寄存器格式

X86 架构中，对 EOI 写 0 表示中断处理完成。由上述几个寄存器相互配合，一个典型的 LAPIC 中断处理流程是：

对于 Pentium4、Xeon 系列：

1. 通过中断消息的 destination field 字段，确定该中断是否是发送给自己的。
2. 如果该中断的 delivery mode 为 NMI、SMI、INIT、ExtINT、SIPI，直接交由 CPU 处理。
3. 如果不为 2）中所列的中断，置一 IRR 中相应的 bit。
4. 当中断被 pending 到 IRR 或 ISR 中后，根据 TPR 和 PPR 寄存器，判断当前最高优先级的中断是否能发送给 CPU 处理。
5. 软件写 EOI 通知中断处理完成。如果中断为 level 触发，该 EOI 广播到所有 IOAPIC（见前）。NMI、SMI、INIT、ExtINT、SIPI 类型中断无需写 EOI。

对于 Pentium 系列和 P6 架构：

1. 确定该中断是否由自己接收。如果是一个 IPI，且 delivery mode 为 lowest priority，LAPIC 与其它 LAPIC 一起仲裁该 IPI 由谁接收。
2. 若该中断由自己接收，且类型为 NMI、SMI、INIT、ExtINIT、INIT-deassert、或 MP 协议中的 IPI 中断（BIPI、FIPI、SIPI），直接交由 CPU 处理。
3. 将中断 pending 到 IRR 或 ISR，若该已有相同的的中断 pending 到 IRR 和 ISR 上，拒绝该中断消息，并通知 IOAPIC “retry”。
4. 同 Pentium4、Xeon 系列流程。
5. 同 Pentium4、Xeon 系列流程。

1.2.3 IOAPIC 发出的中断消息是如何找到 LAPIC 的？

上面两个流程的第一步都是确定是否由自己接收中断。前面我们提到，RTE 中的 Destination Field 用于指定由哪个 APIC 接收，并且分为 Physical 和 Logical 两种模式。对于 LAPIC，两种模式有着不同的意义。

Physical 模式：在该模式下，RTE 中的 Destination Field 表示的是 LAPIC ID。对于 LAPIC 来说，系统在 RESET 后，都会分配一个唯一的 ID 用作标识。在 X86 平台下，我们可以通过 LAPIC ID 寄存器得到它。图 1-6 为其格式：

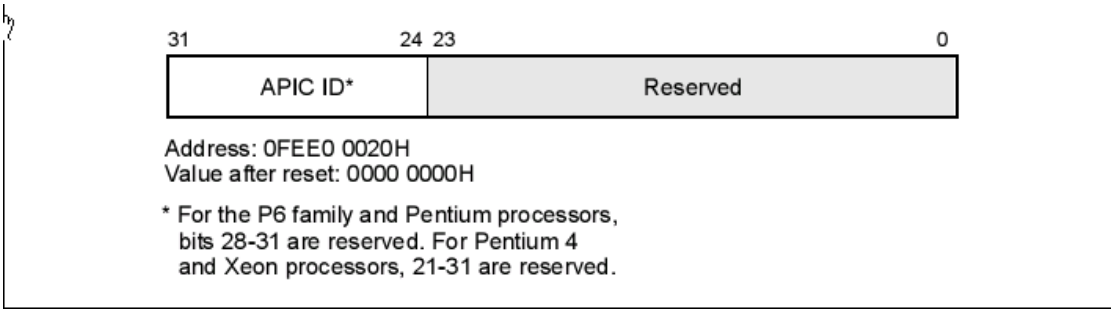


图 1-6 Local APIC ID

操作系统或 BIOS，通常会使用 LAPIC ID 唯一的标识一个 CPU。在 Pentium 系列和 P6 架构中，由于 APIC BUS 最多只支持 15 个 LAPIC ID，即一个 MP 平台最多只能有 15 个 CPU，RTE 中的 destination field 表示 LAPIC ID 时只用了 4bit，LAPIC ID 寄存器也只有 4bit 可用。对于 Pentium4 和 Xeon 系列，APIC ID 被扩展至 8bit，最多支持 255 个 LAPIC。系统 RESET 后，可以用 CPUID 指令(EAX 写参数 1，EBX 的 24~31 即为返回的 ID)获得默认的 LAPIC ID。某些 CPU 允许软件更改默认的 ID 号，但通常来说，软件应该避免这样的行为。无论何时，CPUID 指令返回的都是系统 RESET 后默认分配的 LAPIC ID，即使当前的 LAPIC ID 寄存器已经被软件更改过。

题外话——关于 APIC ID

APIC ID 分为 LAPIC ID 和 IOAPIC ID。前者唯一的标识系统中某个 LAPIC，后者唯一标识某个 IOAPIC。LAPIC ID 前面已经介绍了，根据 MP spec（Multiple Processor Specification，多处理器规范）规定，LAPIC ID 必须唯一，但可以不连续。与 LAPIC ID 不同，IOAPIC ID 在系统 RESET 后统一清零，操作系统或 BIOS 负责验证 IOAPIC ID 是否唯一，如果有冲突检测到，由操作系统或 BIOS 重新分配。重分配的原则是从系统中所有 LAPIC ID 后最小的数字开始分配。例如当前系统有两个 LAPIC，ID 为 0、1，则分配 IOAPIC ID 应该从 2 开始。

需要注意的是，MP spec 对 APIC ID 的分配规则只适用于系统用 APIC BUS 的情况。X86 spec 说 4bit 的 LAPIC ID 可以表示 15 个 CPU，还有一个 0x0f 预留给了广播。当 RTE 的 destination field 字段代表 APIC ID，且值为 0x0f 时，该中断消息广播给所有 CPU。

笔者：LAPIC ID 是从 0 开始分配的，IOAPIC ID 在系统 RESET 后自动清 0。MP spec 说“操作系统在探测到 IOAPIC ID 冲突时，有义务为它分配一个新的 ID”。那相同的 LAPIC ID 和 IOAPIC ID 算不算冲突呢？算，但是对于 Pentium4 和 Xeon 系列使用前端总线通讯的系统，”It’s not an issue”。在此种系统中，LAPIC 是要用 ID 参与前端总线竞争的，IOAPIC 却不用，因为是北桥代理它竞争总线。IOAPIC ID 只用于区分多个 IOAPIC，它和 LAPIC ID 不在一个上下文。

APIC BUS 下，IOAPIC ID 要用于竞争总线，不能和 LAPIC ID 冲突，分配规则“题外话”中已说明。

当中断消息通过 Physical 模式发送时，LAPIC 通过 LAPIC ID 来判断该中断是否由自己接收。

Logical 模式：在该模式下，中断消息中的 Destination Field 包含的不是 LAPIC ID，而是被称为 MDA（Message Destination Address，消息目的地地址）的信息。此时，LAPIC 需要两个额外的寄存器来判断自己是否为中断消息的目的地。它们是 LDR 和 DFR。

LDR 的格式如图 1-7 所示：

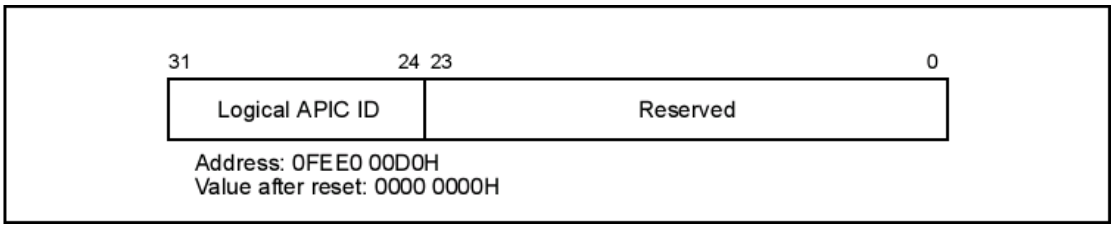


图 1-7 LDR 格式

LDR 全称是 Logical Destination Register，逻辑目的地寄存器。该寄存器包含一个 8bit 的逻辑 APIC ID（注意区分，它和 LAPIC ID 不是一个东西），在 Logical 模式下用于和 MDA 匹配。LDR 的格式由 DFR 指定，DFR 如图 8 所示：

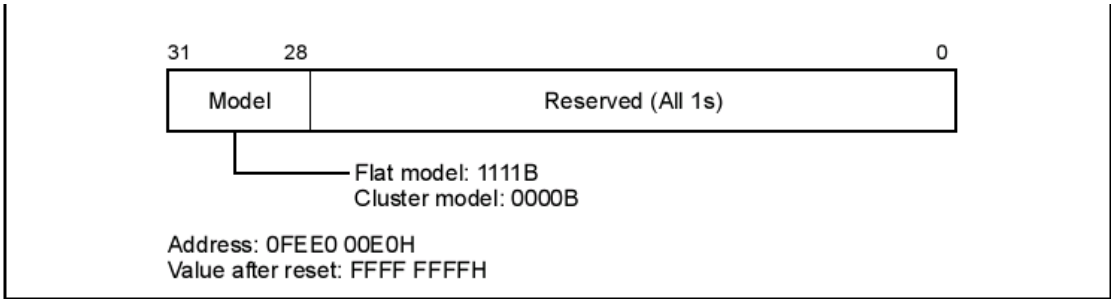


图 1-8 DFR 格式

DFR，Destination Format Register，目的地格式寄存器。该寄存器包含一个 4bit 的 mode 字段，用于指定 LDR 中的 Logical APIC ID 用何种方式与 MDA 匹配。通过这两个寄存器的配合，Logical 模式又被分为了 Flat 与 Cluster 两种模式。

- Flat 模式：DFR 的 model 值为 1111b，此时，LAPIC 将 MDA 与 LDR 的 logical APIC ID 做位与，如结果不为 0 则接收中断。Logical APIC ID 中每个 bit 代表一个 LAPIC，故 8bit 最多代表 8 个 CPU。
- Cluster 模式：DFR 的 model 值为 0000b。我不得不说我可能会把你搞晕了，但实际上确实如此，x86 太 TMD 复杂了。Cluster 模式又分为两种模式：Flat Cluster 模式和 Hierarchical Cluster 模式。
  - Flat Cluster 模式：该模式只支持 P6 架构和 Pentium 系列 CPU，并假定所有 APIC 通过 APIC BUS 通讯。该模式将 MDA 编码为两个部分，高 4bit 为簇号，低 4bit 标识LAPIC在该簇内的ID（每个bit代表一个LAPIC，故一个簇最多有4个LAPIC）。与之对应，LDR 的 logical APIC ID 也被编码成同样两个部分。

    工作在该模式时，LAPIC 先将 MDA 的高 4bit 和 Logical APIC ID 的高 4bit 比较，以确定自己是否是中断的目的簇。若是，将 MDA 的低 4bit 与 Logical APIC ID

的低 4bit 位与，若值不为 0 则接收中断。否则拒绝。

通过这种方法，高 4bit 的簇号可以表示 15 个簇，低 4bit 的 ID 可以代表簇内的 4 个 CPU，最多可以支持 60 个 CPU。但由于 APIC BUS 的限制，具体的说是 APIC Arb ID（APIC 仲裁 ID）的限制，该模式最多只支持 15 个 CPU。

- Hierarchical Cluster 模式：支持 P6 架构和 Pentium 系列，以及 Xeon、Pentium4 系列。该模式通过为每个簇引入一个“簇管理器”，将 Flat Cluster 模式中平等的簇构成一个具有等级结构的分级网络，并最多支持 60 个 CPU。这个话题太远了，相关 spec 没有更多资料，就不多做介绍了。

> 笔者：不要问我 Hierarchical Cluster 模式下，P6 架构和 Pentium 系列如何突破 APIC BUS 对于 15 个 CPU 的限制的，我真的不知道。
>
> “通过这种方法，高 4bit 的簇号可以表示 15 个簇”，实际上应该是 0~15 共 16 个簇，但 MDA 全 1 时为广播到所有 LAPIC，笔者认为簇号 1111b 被预留给广播模式了。

对于 Flat Cluster 模式，我们举个例子吧（此例中，中断为 Fix delivery mode。关于 Lowest Prority 的例子见后面内容）。假设有三个 CPU 的 logical 模式配置为（此时 DFR 的 model 值为 0000b）：CPU1 的 LDR 值为 0000 0001b，CPU2 的 LDR 值为 0001 0010b，CPU3 的 LDR 值为 0000 0100b，则 CPU1、CPU3 为一簇，CPU2 为另一簇。IOAPIC 发出一条中断消息，其 Destination Mode 为 1，destination field 值为 0000 00001b。三个 LAPIC 收到该消息后，CPU1、CPU3 通过 destination field 的高 4bit 判断出该消息目的地为本簇，再将自身 Logical APIC ID 的低 4bit 与 destination field 低 4bit 位与，最终 CPU1 接收该中断消息，CPU2、CPU3 丢弃。

图 1-9 总结了中断消息的 Destination Field 字段用于寻址 LAPIC 的几种模式：

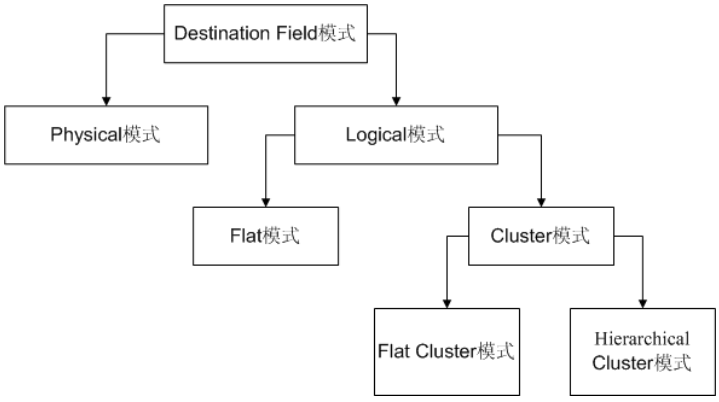


图 1-9 Destination Field 模式

1.2.4 TPR、PPR、APR 和 Lowest priority —— 中断发给 Whom？

RTE 的 delivery mode 有一中模式为 lowest priority，即最低优先级，它是 Linux 配置 RTE

时使用的模式。这里的最低优先级不是指中断的优先级，而是指将中断发送给 destination field 列出的 CPU 中，优先级最低的一个。如何决定一个 CPU 的优先级呢？x86 平台依靠 TPR 寄存器和 PPR 寄存器。

TPR，task priority register，任务优先级寄存器，它确定当前 CPU 可处理什么优先级别范围内的中断。具有如下的格式：

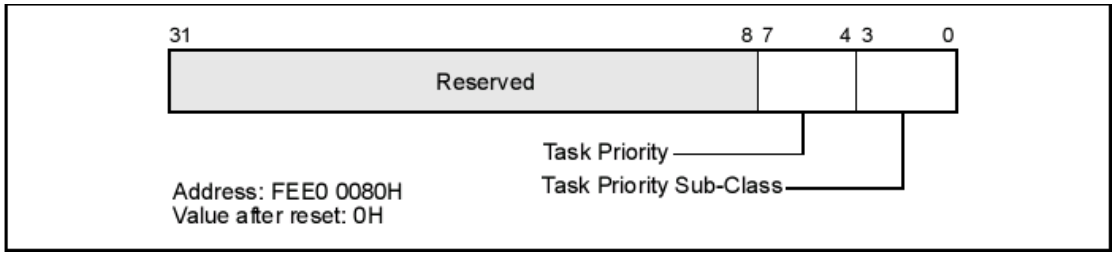


图 1-10 TPR 寄存器

TPR 寄存器接收 0~15 共 16 个值，对应 16 个 CPU 规定的中断优先级级别，值越大优先级越高。CPU 只处理比 TPR 中值优先级别更高的中断。例如 TPR 中值为 8，则级别小于等于 8 的中断被屏蔽（注意，屏蔽不代表拒绝，LAPIC 接收它们，把它们 pending 到 IRR 中，但不交 CPU 处理。见前面 LAPIC 中断处理流程）。值 15 表示屏蔽所有中断；值 0 表示接收所有中断，噢，这也是 Linux 为 TPR 设置的默认值。注意，TPR 是由软件读/写的，硬件不更改它。

题外话 —— x86 的中断优先级级别

我们知道每个中断都有一个 vector 与之对应，x86 平台共有 256 个 vector，除去架构预留和被异常等占去的 0~31 号 vector，可供外部中断使用的还有 224 个（噢，实际上只有 223 个，还要除去一个 INT 80）。中断的优先级别由下列公式：

优先级别 = vector / 16

这里“/”是 c 语言的除，不是数学里的除号，所以我们是没有小数的。16~255 号 vector 构成了 1~15 共 15 个优先级别，中断拥有 2~15 级别。对于同一个级别的中断，vector 号越大的优先级越高。例如 vector33、34 都属于级别 2，34 的优先级就比 33 高。所以，对于 8bit 的 vector，又可以划分成两部分，高 4bit 表示中断优先级别，低 4bit 表示该中断在这一级别中的位置。

噢，应该讲清楚了，记住，TPR 的值增加 1，将会屏蔽 16 个 vector 对应的中断。NMI、SMI、ExtINT、INIT、start-up delivery 的中断不受 TPR 约束。

笔者：TPR，任务优先级寄存器 ，很容易的就让人和进程的优先级联想到一起。实际上，x86 的 spec 也强调这里的“Task”代表操作系统中的进程、线程、任务、程序 ……不过，Linux 的进程优先级和它没有关系。如果 Linux 的中断处理例程线程化了，如果 Linux 具有更强的实时性，如果 Linux 跟着 Windows、Solaris 学习，如果 …… Linux 或许会把它和进程优先级挂上勾。现在嘛，Linux 没有那样做。不过，APIC 设计 TPR 的目的，真是给 Task 用的，期望进程切换时也会更新 TPR 值，这是 APIC 的历史文档告诉我们的。

TPR 为 8bit 组成，从图中我们可以看到，0~15 这 16 个值应该写到高 4bit 去。如果我们同时也写了低 4bit 会怎么样？从关于 TPR 的论述来看，CPU 会忽略低 4bit 的值。

PPR，Processor priority register，处理器优先级寄存器。该寄存器决定当前 CPU 正在处理的中断的优先级级别，以确定一个 Pending 在 IRR 上的中断是否发送给 CPU。与 TPR 不同，它的值由 CPU 写而不是软件写。PPR 取值范围为[0,15]，计算方式由下列伪代码描述：

```
If TPR[7:4] >= ISR[7:4] THEN
    PPR[7:0] = TPR[7:0]
ELSE
    PPR[7:4] = ISRV[7:4]
    PPR[3:0] = 0
```

这里，ISRV[7:4]标识当前 ISR 中，最高优先级中断对应 vector 的高 4bit，如前面所说，这代表了该中断的优先级级别。简而言之，取 TPR 和正在服务的最高优先级中断中，优先级级别高的。好了，都知道了，IRR 中 pending 的中断，优先级级别必须高于 PPR 中值才会被发送给 CPU 处理，否则，继续等 ……

谁是 Lowest Priority？

有了前面两个寄存器的论述，现在可以看看 Lowest Priority 是怎么决定的了。先来从**Pentium4 和 Xeon 系列**说吧，它们要简单点（实际上更复杂，但因为资料少，我可以说的简单点。具体信息？去看 mindshare 那本讲前端总线的书吧，2000 多页的英文读本）

对于这个系列，LAPIC 和 IOAPIC 通过前端总线通讯。X86 spec 对这个描述是：“芯片组负责从 destination filed 列出的 CPU 中，选出一个优先级最低的接收中断。如果是 Pentium4，CPU 会用一个特殊的总线周期（bus cycle）将每个 CPU 当前的任务优先级提交到总线，芯片组记录它们，并作为挑选最低优先级 CPU 的依据”。完了，够简单吧^_^

笔者：从 spec 来看，无论是 Xeon 还是 Pentium4，都是由芯片组选最低优先级 CPU，依据是什么？spec 讲的不清楚。但对于 pentium4，提到了“用一个特殊的 bus cycle 将当前各 CPU 的 task priority 提交到总线”。这个 task priority 具体指什么？spec 也没说。但在 TPR 相关章节中有这么一句话：“注意，task priority 用于决定 CPU 的仲裁优先级（见 8.6.2.4，Lowest Priority Delivery Mode）”。虽然没有明说，但我们有理由认为，这里的 task priority 就是 TPR 的值。

P6 架构的系统，依靠一个 APR 寄存器(Arbitration Priority Register，优先级仲裁寄存器)决定 CPU 优先级，其格式如图 1-11：

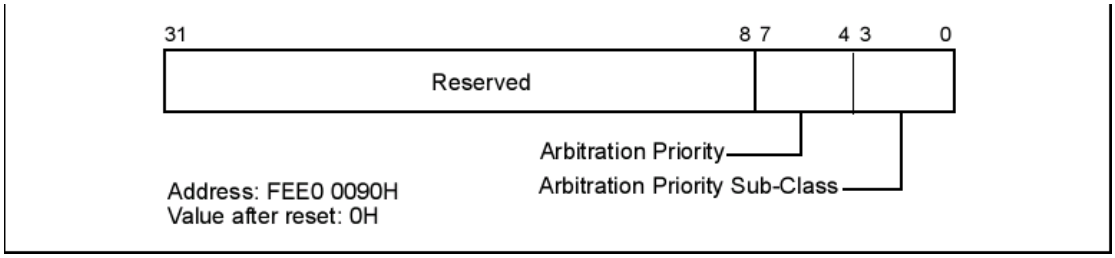


图 1-11 APR 寄存器

APR 值可由下列伪代码计算：

```
If ( (TPR[7:4] >= IRRV[7:4]) && (TPR[7:4] > ISRV[7:4]) )
    APR[7:0] = TPR[7:0]
Else
{
    APR[7:4] = max(TPR[7:4],IRRV[7:4],ISRV[7:4])
    APR[3:0] = 0
}
```

上述代码表述了这么一个意思：当 TRP 的值大于 IRR 中最高优先级中断的优先级级别，并大于 ISR 中最高优先级中断的优先级级别时，APR 等于 TPR。否则，APR 高 4bit 取 TPR、IRRV、ISRV 三者中优先级级别最高的；低 4bit 取 0

题外话 —— Focus Processor

对于 P6 架构和 Pentium 系列 CPU，spec 提到了一个 Focus Processor 的概念。如果一个中断 A 正在被某 CPU1 处理，或者 pending 在 CPU1 的 IRR 上，则称 CPU1 为中断 A 的 Focus Processor。当系统中有 Focus Processor 存在时，Focus Processor 可能（may）接收该中断，不管自身的优先级是多少。Xeon 系列没有 Focus Processor 的概念。

笔者：关于 Focus Processor 是尽量按 spec 的原话翻译，当然，E 文水平有限，翻译的不好。不过大概就是这个意思。Focus Processor 在哪儿设定？伪中断寄存器（见后面的伪中断相关内容）。spec 说：“If a focus processor exists, it may accept the interrupt, regardless of its priority”，spec 用了一个 may。这种不清不楚的用词太万恶了，让人完全搞不清楚它什么时候会 may，什么时候又 may not。不管怎样，Focus Processor 给了我们这样一种暗示，即当系统中大量产生同一种中断时，该中断会被发送给同一个 CPU 处理。这样的好处是可以避免 cache 颠簸，缺点也十分明显：运行在该 CPU 上的进程可能饿死、中断处理过慢、系统中断负载不平衡 …… 我们又注意到，Xeon 系列取消了这个功能，应该是顺应民心吧。既然已经有了 PRT 表，CPU 就不应该再提供这种重定向中断的功能。

发了一堆牢骚，其实情况没那么糟糕。Focus Processor 只对 edge 触发中断有效。而目前平台上大量使用的 PCI 是 level 触发的。为什么？想想 Focus Processor 的定义，再去看看前面关于同类型中断 pending 两次的说明吧。

我们来举一个 Pentium4 和 Xeon 系列优先级仲裁的例子，说明谁是 Lowest Priority。假

设有 CPU1、CPU2、CPU3 三个 CPU，相应的 TPR 值为：TPR1=5、TPR2=6、TPR3=10，IOAPIC 以 lowest priority 模式发送一条中断消息，该中断对应的优先级级别为 3。则 CPU1 具有最低优先级，接收该中断。此时，该中断被 pending 到 IRR 中，但不会交给 CPU 处理，因为其优先级级别低于 TPR 值。

至此，IOAPIC 和 LAPIC 相互配合的中断机制已经大体介绍完了，其中提到了一部分寄存器和它们的功用。APIC 的其它内容在后面分析 Linux 实现时补充，一次把硬件内容写完，你看着烦，我写着也累，而且逻辑容易乱。

第二章 Linux 的中断子系统

从这章开始，我们来看看 Linux 中断子系统的实现。

题外话 —— IRQ、Pin、GSI、Vector

这几个概念经常把人搅晕掉，下面的内容经常要用到它们，还是先说清楚为妙。

IRQ 是 PIC 时代的产物，由于 ISA 设备通常是连接到固定的 PIC 管脚，所以说一个设备的 IRQ 实际是指它连接的 PIC 管脚号。IRQ 暗示着中断优先级，例如 IRQ0 比 IRQ3 有着更高的优先级。当前进到 APIC 时代后，或许是出于习惯，人们仍习惯用 IRQ 表示一个设备的中断号，但对于 16 以下的 IRQ，它们可能不再与 IOAPIC 的管脚对应，例如 PIT 此时接的是 2 号管脚。

Pin 是管脚号，通常它表示 IOAPIC 的管脚（前面说了，PIC 时代我们用 IRQ）。Pin 的最大值受 IOAPIC 管脚数限制，目前取值范围是[0,23]。

GSI 是 ACPI 引入的概念，全称是 Global System Interrupt。它为系统中每个中断源指定一个唯一的中断号。下图展示了 GSI 的思想：

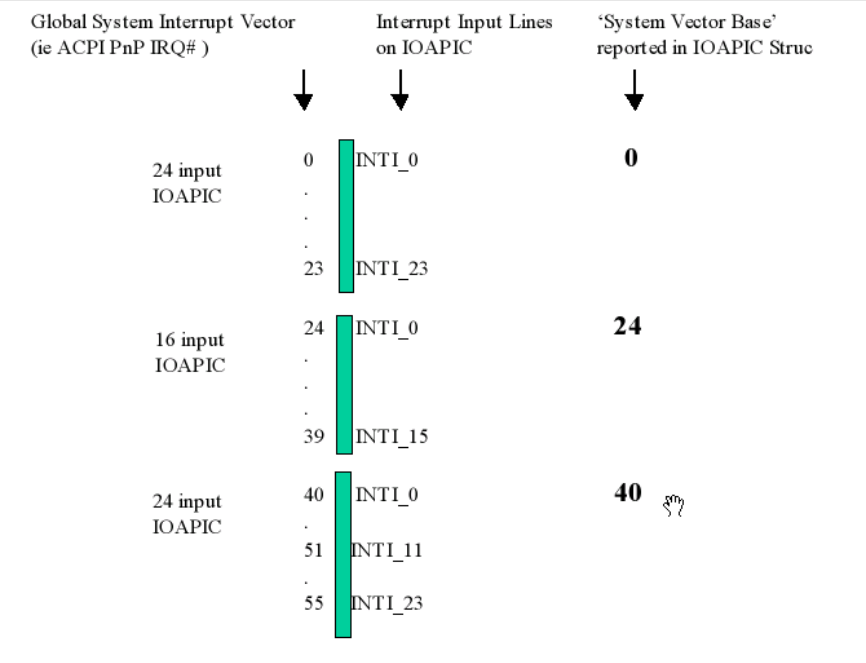


图 2-1

图 2-1 中有 3 个 IOAPIC：IOAPIC0~2。IOAPIC0 有 24 个管脚，其 GSI base 为 0，每个管脚的 GSI=GSI base + pin，故 IOAPIC0 的 GSI 范围为[0~23]。IOAPIC1 有 16 个管脚，GSI base 为 24，GSI 范围为[24,39]，依次类推。ACPI 要求 ISA 的 16 个 IRQ 应该被 identify map 到 GSI 的[0,15]。

IRQ 和 GSI 在 APIC 系统中常常被混用，实际上对 15 以上的 IRQ，它和 GSI 相等。我们在谈到 IRQ 时，一定要注意它所处的语境。

Vector 是 CPU 的概念，是中断在 IDT 表中的索引。每个 IRQ（或 GSI）都对应一个 Vector。在 PIC 模式下，IRQ 对应的 vector=start vector + IRQ；在 APIC 模式下，IRQ/GSI 的 vector 由操作系统分配。

2.1 中断探测

操作系统如何知道平台上硬件中断系统的情况？BIOS 告诉它的。表，是 BIOS 报告系统资源的通常方式。MP spec 定义了 MP table，ACPI 规范定义了 MADT（Multiple APIC Description Table）。虽然构建方式不一样，但从 OS 看来两者大同小异，都支持 LAPIC entry、IOAPIC entry，以及其它几个与它们相关的 entry。这些 entry 描述了平台 APIC 硬件的情况。下面根据 Linux 两条不同的中断探测路径，介绍如何根据 MP table 或 MADT 构建中断子系统的主要数据结构。

2.1.1 关键数据结构

Linux 最初实现的是 MP spec 相关规范，故定义一组符合 MP spec 的数据结构来管理平台硬件。ACPI 被引入后，虽然汇报硬件资源的方式变了，但仍可以使用 MP spec 的数据结构来管理。Linux 的 ACPI 解析路径，实际上是用 ACPI 的方式读表，填充 MP spec 的数据结构。由此看来，我们只要理解 MP spec 数据结构就可以了，和中断相关的有如下几个：

> 笔者：这些结构体和 MP spec 规定的各个 entry 结构是相同的，下面表格将两者列在一起说明，以后就不单独画 MP spec 各 entry 的结构表了。

struct mpc_config_ioapic：对应 MP spec 的 IOAPIC entry ，各字段如下：

结构体成员	MP spec 字段	长度（byte）	描述
mpc_type	TYPE	1	类型，2(IOAPIC ENTRY)
mpc_apicid	IOAPIC ID	1	IOAPIC ID
mpc_apicver	IOAPIC Version	1	IOAPIC 版本
mpc_flags	IOAPIC Flags：EN	1	最低 bit 有效，其余 bit 预留 0：IOAPIC disabled 1：IOAPIC enabled
mpc_apicaddr	IOAPIC Address	4	该 IOAPIC 基地址

表 2-1 IOAPIC entry 和 struct mpc_config_ioapic

内核用 mp_ioapics 全局数组存放系统中所有 IOAPIC 对应的 struct mpc_config_ioapic，nr_ioapics 为 IOAPIC 数量。目前 Linux 最多支持 64 个 IOAPIC，当数量超过时可以配置 MAX_IO_APICS 宏扩大限制。

struct mpc_config_intsrc：对应 MP spec 的 IO interrupt entry，代表各个中断源（一个 IOAPIC 管脚连接一个中断源）各字段如下：

结构体成员	MP spec 字段	长度（byte）	描述
mpc_type	TYPE	1	3（IOAPIC interrupt）
mpc_irqtype	INTERRUPT TYPE	1	INT：APIC 模式中断 NMI：不可屏蔽中断 SMI：系统管理中断 ExtINT: vector 由 PIC 提供。如果 8259 用作外部 PIC,即 PIC INTR 接 APIC 的一个管脚，vector 由 8259 提供
mpc_irqflag	PO：位于 2byte 0bit EL：位于 2byte 1bit	2	PO：中断管脚极性 EL：触发模式
mpc_srcbus	SOURCE BUS ID	1	产生中断的总线
mpc_srcbusirq	SOURCE BUS IRQ	1	相对于产生中断的总线，该中断管脚代表的中断号。例如 0，相对于 ISA 总线即 IRQ0
mpc_dstapic	DESTINATION IOAPIC ID	1	该中断连接的 IOAPIC。0xff 标识连接到所有的 IOAPIC
mpc_dstirq	DESTINATION IOAPIC INTN#	1	连接到 IOAPIC 的管脚号

表 2-2 IO interrupt entry 和 mpc_config_intsrc

不同总线中断共享情况：如果两个 IO interrupt entry 的 destination（IOAPIC ID、INTN#）相同，则共享。例如 IPCI-device1/INTA#和 ISA-IRQ2 如对应相同的 destination，则两者共享，接同样的 IOAPIC 和 INTN#。

题外话 —— 中断共享和中断丢失

我们不讨论 Level 触发的情况，因为意义不大，它可以共享且不会丢失中断。

Edge 触发，问题就比较多了。它通常对应 ISA 设备，一个常见的说法是“ISA 设备不能共享中断”，但也有人说这是个“superstition”（内核邮件列表曾有关于它的争论）。

从能找到的资料来看，标准的 ISA 设备是不支持中断共享的，但问题是有很多非标准实现的存在。从操作系统的角度看，支持 ISA 中断共享并非难事，我们完全可以使用 level 触发同样的方式去处理 edge 触发（例如 Linux 的实现）。Edge 共享的一个难题是，当一个设备在另一个设备的中断正在被处理时发起中断，该中断会丢失（或者说要等到另一个设备再发起中断时才会处理），因为此时该中断管脚通常是被屏蔽的。为了解决这个问题，一些非标准的实现会用“re-trigger while in service”、“sequencing of interrupt-generating”加以保证。前者会在一个中断服务完后再触发一次中断；后者会保证中断事件顺序产生，即处理完一个再产生一个。无论怎样，我们尽量在操作系统中支持 ISA 中断共享，但应该认识到 edge 中断是不应该共享的。

从理论上分析，edge 中断会丢失，但我没找到足够的资料论述其丢失的具体情况，以及丢失后如何处理。Edge 中断丢失出现在中断管脚被 mask，设备又发起中断时。一些非官方资料说硬件会有一些 re-trigger 的机制，以及 ISA 驱动应实现 timer poll 来确保丢失的中断

能被处理。没有更多的信息，留个疑问在这里吧。

Bluesky_jxc 同学补充了一些观点：

这个问题和塑料袋争论过，edge 触发的中断是可以共享的，但是需要考虑几个问题：

1. 设备需要中断状态位
2. 电路需要引入三态，而且空闲时需要上拉电阻。
3. ISR 链表。

ISA 设计的时候没有考虑到这个需求，因此可以这么说“ISA 中断可以共享，但是不要假设其它设备能支持共享功能”。还是那句话，能不能是一个问题，而用不用是另一个问题。

内核用 mp_irqs 全局数组记录系统中所有的 mpc_config_intsrc。目前支持的最大数——MAX_IRQ_SOURCES 为 256。

Struct mpc_config_lintsrc：定义和 mpc_config_intsrc 一样。只是 mpc_dstapic 代表 LAPIC，且 mpc_dstirq 只能取值 0 或 1。

2.1.2 中断探测内核路径

2.1.2.1 MP Spec 路径

MP spec 的相关代码位于 arch/i386/kernel/mpparse.c 中，主函数是 get_smp_config()，它被 setup_arch()调用，用 MP table 的信息配置系统。我们从 get_smp_config()开始，看看 Linux 是怎么做的：

笔者：在此之前，我建议先跳文章末尾，阅读 MP spec 对硬件中断系统的规定

1、首先检查系统支持 PIC 模式还是 Virtual Wire 模式。如果支持 PIC 模式，全局变量 pic_mode=1，否则 pic_mode=0。接着获得 LAPIC 地址，存入 mp_lapic_addr 全局变量。最后检查 mp table 是否提供了默认配置。这些信息都可以通过 MP Floating Pointer entry 得到。

题外话 ——MP Floating Pointer entry 和 Default Configuration

MP Floating Pointer entry 有两个字段用于描述一些特殊的属性，如下表：

字段	偏移（bytes:bits）	长度（bit）	描述
MP FEATURE INFORMATION BYTE1	11	8	全 0：使用 MP configuration table 不为 0：使用 Default configuration
MP FEATURE INFORMATION BYTE2	12:0 12:7	7 1	Bit0-6:预留 Bit7：IMCRP，该位置一，PIC mode 被实现；否则 Virtual Wire Mode 被实现

表 2-3 MP Floating Pointer Structure（只摘录的部分字段）

Default Configuration 是 MP spec 为了简化 BIOS 设计，给特定的平台提供的默认配置。它对平台有如下假设：

- 系统支持两个 CPU
- CPU 为 Intel 兼容指令集
- LAPIC 位于默认地址 FEE0_0000H
- LAPIC ID 从 0 开始连续分配
- 系统有一个 IOAPIC 位于 FEC0_0000H
- 系统实现了 PIC mode 或 Virtual Wire Mode
- 根据平台总线类型、APIC 类型的不同，默认配置又分 7 种情况，这里我们只关心一种，即平台总线为 PCI+ISA、APIC 是集成型时的情况。

笔者：MP spec 给了两种 APIC 类型，一种是 82489DX，一种是 Integrated 类型。个人认为目前流行的属于 Integrated。

来看一下默认配置表，图中画框的是我们讨论的情况：

First I/O APIC INTINx	Config 1	Config 2	Config 3	Config 4	Config 5	Config 6	Config 7	Comments
INTIN0	8259A INTR	8259A INTR	8259A INTR	8259A INTR	8259A INTR	8259A INTR	N/C	INTR output from master 8259A or equivalent
INTIN1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	IRQ1	Keyboard controller buffer full
INTIN2	IRQ0	N/C	IRQ0	IRQ0	IRQ0	IRQ0	IRQ0	8254 Timer
INTIN3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	IRQ3	
INTIN4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	IRQ4	
INTIN5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	IRQ5	
INTIN6	IRQ6	IRQ6	IRQ6	IRQ6	IRQ6	IRQ6	IRQ6	
INTIN7	IRQ7	IRQ7	IRQ7	IRQ7	IRQ7	IRQ7	IRQ7	
INTIN8	IRQ8	IRQ8	IRQ8	IRQ8	IRQ8	IRQ8	IRQ8	Real time clock
INTIN9	IRQ9	IRQ9	IRQ9	IRQ9	IRQ9	IRQ9	IRQ9	
INTIN10	IRQ10	IRQ10	IRQ10	IRQ10	IRQ10	IRQ10	IRQ10	
INTIN11	IRQ11	IRQ11	IRQ11	IRQ11	IRQ11	IRQ11	IRQ11	
INTIN12	IRQ12	IRQ12	IRQ12	IRQ12	IRQ12	IRQ12	IRQ12	
INTIN13	IRQ13	N/C	IRQ13	IRQ13	IRQ13	IRQ13	IRQ13	Floating point exception and DMA chaining
INTIN14	IRQ14	IRQ14	IRQ14	IRQ14	IRQ14	IRQ14	IRQ14	
INTIN15	IRQ15	IRQ15	IRQ15	IRQ15	IRQ15	IRQ15	IRQ15	

图 2-2 Default Configuration

该配置有几个需要注意的地方，INTIN0（IOAPIC 的 0 管脚）接的是 PIC，INTIN2 接的是 IRQ0，即 PIT；IRQ2 不存在，因为 PIC 为两片 8259a，其中第二片 8259a 接第一片的 IRQ2 管脚。

2、 如果平台使用 Default Configuration，调用 construct_default_ISA_mptable()。该函数会对 CPU、总线、中断等使用默认配置，我们只关心和中断相关的部分，流程如下：

a、 注册 IOAPIC。由于 Default Configuration 中只有一个 IOAPIC，故其配置如下：

```
ioapic.mpc_type = MP_IOAPIC;
ioapic.mpc_apicid = 2;
ioapic.mpc_apicver = mpc_default_type > 4 ? 0x10 : 0x01;
```

```
ioapic.mpc_flags = MPC_APIC_USABLE;
ioapic.mpc_apicaddr = 0xFEC00000;
MP_ioapic_info(&ioapic);
```

注意这里 IOAPIC ID 为 2，是从 LAPIC ID 后最小的数字分配的（Default Configuration 有两个 CPU）。MP_ioapic_info() 用于把该 struct mpc_config_ioapic 存入 mp_ioapics 数组。

b、 调用 construct_default_ioirq_mptable()配置 ISA 中断。该函数首先配置 struct mpc_config_intsrc 中各字段，除 dstirq。如下：

```
intsrc.mpc_type = MP_INTSRC;
intsrc.mpc_irqflag = 0;                 /* conforming */
intsrc.mpc_srcbus = 0;
intsrc.mpc_dstapic = mp_ioapics[0].mpc_apicid;
intsrc.mpc_irqtype = mp_INT;
```

其中 mpc_irqflag=0 表示中断的触发模式、管脚极性由中断所在的总线决定。接着配置各个 ISA 中断，即设置 dstirq。内核为了正确的配置 PCI 中断，使用了 ELCR（Edge/Level Controller Register，用于判断某 IRQ 的触发方式，详情见 ICH9 spec），在使用前，检查 ELCR 是否可用。

题外话 —— IRQ0、1、2、13 是什么？

Kernel 检查 ELCR 是否可用的方式，是判断 IRQ0、1、2、13 是否有 level 触发，因为这 4 个 IRQ 只能为 edge 触发。那么，它们是什么？IRQ0 是 PIT timer；IRQ1 是键盘；IRQ2 是连接的 slave 8259a；IRQ13 是协处理器，也就是老式的浮点运算单元。实际上还有一个可用于检查，即 IRQ8——RTC，它也是 edge 触发。

设置 dstirq 代码如下（简化后）：

```
for (i = 0; i < 16; i++) {
        if (i == 2)
                continue;
        if (ELCR_fallback) {
                if (ELCR_trigger(i))
                        intsrc.mpc_irqflag = 13;
                else
                        intsrc.mpc_irqflag = 0;
        }
        intsrc.mpc_srcbusirq = i;
        intsrc.mpc_dstirq = i ? i : 2;          /* IRQ0 to INTIN2 */
        MP_intsrc_info(&intsrc);
 }
```

IRQ2 不设置，IRQ0 连接到 IOAPIC 管脚 2。其它 IRQx 被 identify map 到 0 号 IOAPIC 的 1~15 脚。此外，如果 ELCR 检查到该 IRQ 接的是 PCI 中断，

则重新设置 mpc_irqflag 配置触发方式。最后调用 MP_intsrc_info()把各 ISA 中断配置存入 mp_irqs 数组。

> 笔者：这里 ELCR_trigger(i)返回 1 表示该 IRQ 是 level 触发。从内核的注释看，此时为 PCI 中断。令人费解的是 intsrc.mpc_irqflag = 13 代表了高电平有效、电平触发。而 PCI 应该是低电平有效、电平触发。内核错了？

最后，还要描述一下 0 号管脚接 PIC 的情况，如下：

```
intsrc.mpc_irqtype = mp_ExtINT;      /*注意和前面的 MP_INT 类型区分*/
 intsrc.mpc_srcbusirq = 0;
 intsrc.mpc_dstirq = 0;                    /* 8259A to INTIN0 */
 MP_intsrc_info(&intsrc);
```

c、 注册 mpc_config_lintsrc 。Default configuration 中 LINT0 接 ExtINT，LINT1 接 NMI。调用 MP_lintsrc_info 注册（仅打印信息，无实用）。

3、 如果系统未实用默认配置，就需要解析 MP table。该任务在 smp_read_mpc()中完成，它将 MP table 的各个 entry 分别存入 mp_ioapics、mp_irqs 等数组。

4、 至此，平台上各 IOAPIC、中断源的信息应已存储到各相关数组里。Kernel 再做最后一次检查，如果 mp_irq_entries == 0，说明 MP table 没有报告中断源，construct_default_ioirq_mptable()将系统设置到 ISA 模式。

2.1.2.2 MADT 路径

2.1.2.2.1 MADT 表结构

MADT 表的 entry 和 MP table 大同小异。

MADT 表头结构：该结构拥有很多字段，例如版本号、OEM 信息等，其中和本文内容相关的字段有 3 个，见下表：

字段	长度（bytes）	偏移	描述
LAPIC Address	4	36	32bit 物理地址，用于 CPU 访问自身 LAPIC
Flags	4	40	Bit1 表示系统是否有 PIC 存在。 1：有，当 APIC 使用时，PIC 的各管脚必须被 mask； 0：无。 其它 bit 预留
APIC structures	-	44	各种中断的描述条目

表 2-4 MADT 表头

题外话 ——APIC 的地址

Spec 规定，LAPIC 和 IOAPIC 的寄存器必须被实现成 MMIO 方式。对于 LAPIC 来说，每个 CPU 使用同样的地址访问自己的 LAPIC，默认地址是 FEE0_0000H。对于 IOAPIC，默

认基地址是 FEC0_0000H，每个 IOAPIC 占 4k 地址，从基地址开始连续。

LAPIC 地址对齐到 4K 边界，IOAPIC 地址对齐到 1K 边界。

笔者：每个 CPU 都用同样的物理地址访问自己的 LAPIC。这说明除了 x86 平台的 port I/O 具有 64K 独立的物理地址空间外，LAPIC 也拥有独立的物理地址。我能想到的理由是防止 CPU 访问不属于自己的 LAPIC。

LAPIC 相关的 MADT 条目共有三个，分别是：LAPIC entry、LAPIC NMI entry、LAPIC address override entry。

字段	长度（bytes）	偏移	描述
Type	1	0	0（LAPIC）
Length	1	1	8
ACPI Processor ID	1	2	在 ACPI processor operator 中列出的 processorID
APIC ID	1	3	LAPIC ID
Flags	4	4	Bit0：enable 位， 0：disable； 1：enable。 其它 bit 预留

表 2-5　LAPIC entry

字段	**长度（byte）**	**偏移**	**描述**
Type	**1**	**0**	**4（LAPIC NMI）**
Length	**1**	**1**	**6**
APCI Processor ID	**1**	**2**	**Processor object 中的 CPU 的 ID。0xff 代表所有 CPU**
Flags	**2**	**3**	**MPS INIT flags**
LAPIC LINT#	**1**	**5**	**LINT 管脚号（0 or 1）**

表 2-6 LAPIC NMI entry

该表描述 NMI 中断接 LAPIC LINTx 管脚的情况，其中 MPS INI flags 为和 MP spec 兼容的属性，见下面的 MPS INI flags 表。

字段	长度（byte）	偏移	描述
Type	1	0	5（LAPIC Address Override）
Length	1	1	12
Reserved	2	2	全 0
LAPIC address	8	4	LAPIC 的物理地址

表 2-7 LAPIC address override entry

该表用于重载 MADT 表头中的 LAPIC 地址，整个 MADT 表中只应该包含一个 LAPIC address override entry。

LAPIC Flags	长度（bit）	偏移	描述
Polarity	2	0	IOAPIC 管脚有效电平极性 00 由总线决定极性（例如 EISA 总线为低电平有效，电平触发）

			01 高电平有效 10 预留 11 低电平有效
Trigger Mode	2	2	管脚触发模式 00 总线决定（例如 ISA 总线为 edge 触发） 01 edge 10 预留 11 level
预留	12	4	全 0

表 2-8 MPS INIT flags

IOAPIC 相关 MADT 条目：IOAPIC entry、Interrupt Source Overrides entry、NMI entry。

字段	长度（bytes）	偏移	描述
Type	1	0	1（IOAPIC）
Length	1	1	12
IOAPIC ID	1	2	IOAPIC ID
Reserved	1	3	0
IOAPIC address	4	4	32bit 物理地址，CPU 通过该地址访问 IOAPIC。该地址对于每个 IOAPIC 都是唯一的
GSI Base	4	8	该 IOAPIC 在 GSI 空间中的起始号。GSI＝GSI base＋管脚号

表 2-9 IOAPIC entry

字段	长度(bytes)	偏移	描述
Type	1	0	2（interrupt source override）
Length	1	1	10
Bus	1	2	0（ISA）
Source	1	3	在 ISA 上的 IRQ 号
GSI	4	4	Map 到 GSI 空间后的 GSI 号
Flags	2	8	见 MP INIT flags 表

表 2-10 Interrupt Source Overrides(ISO)

ISA 中断接 PIC 的 0~15 脚，通常需要 identify mapping 到 GSI 空间。具体的说，ISA 中断应该按接 PIC 的顺序接 0 号 IOAPIC 的 0~15 脚。若平台实现有差异，某个 ISA 中断没有被 identify mapping 的时候，需要一个 ISO 结构来描述。特别需要注意的是，管脚极性不同时，也需要一个 ISO 结构描述。

举两个例子：

例 1: PIT 接 PIC 的 0 号脚，即 IRQ0。当接 IOAPIC 时，它通常接在 2 号管脚上，即 INTN2。此时需要一个 ISO 来描述此差异，source 字段为 0，GSI 字段为 2（0 号 IOAPIC 的 GSI base 为 0）。

例 2: SCI，System Control Interrupt，通常接 PIC 的管脚 9，即 IRQ9，FADT 会在 SCI_INT

字段里报告该值。如接到 IOAPIC 的 11 号脚，则需要一个 ISO 描述。Source 字段为 9，GSI 字段为 11。

字段	长度（bytes）	偏移	描述
Type	1	0	3（NMI）
Length	1	1	8
Flags	2	2	见 MPS INIT Flags
GSI	4	4	该 NMI 的 GSI 号

表 2-11 NMI entry

描述 IOAPIC 某个管脚被配置成 NMI 的情况。NMI 不应该被设备使用。

2.1.2.2.2 ACPI 中断探测过程

X86 的 ACPI 相关代码位于 arch/i386/kernel/acpi 目录，其中对各表的解析位于 boot.c 文件。MADT 表的解析经过下列路径：

1、 acpi_boot_initv()调用 acpi_process_madt()，后者是解析 MADT 表的主函数。

2、 acpi_process_madt()首先检查 MADT 表头，确定 MADT 表包含多少个描述 APIC 部件的 entry，以及获得 LAPIC 的默认地址并放入 acpi_lapic_addr 全局变量中。这在 acpi_parse_madt()函数中完成。

3、 紧接着，acpi_parse_madt_lapic_entries()被调用以获得和 LAPIC 相关的信息。该函数做的事情比较多,依次执行下列步骤：

a、 调用 acpi_parse_lapic_addr_ovr()检查是否有 LAPIC address override entry，如有，用其中的 LAPIC 地址覆盖 acpi_lapic_addr 全局变量。

b、 至此，系统的 LAPIC 地址就已确定了。调用 mp_register_lapic_address()注册，如下：

```
mp_lapic_addr = (unsigned long) acpi_lapic_addr;
set_fixmap_nocache(FIX_APIC_BASE, mp_lapic_addr);
```

set_fixmap_nocache()建立一个 un-cacheable 的 identify mapping 映射。

c、 调用 acpi_parse_lapic()解析 LAPIC entry。如果 LAPIC entry 的 flag 字段为 enable，将 LAPIC ID 和 ProcessorID 的对应关系维护在 x86_acpiid_to_apicid 全局数组中。并调用 mp_register_lapic()注册 LAPIC 信息。

> 笔者： ACPI 的 LAPIC entry 和 MP spec 的 processor entry 对应。mp_register_lapic()最终调用 MP_processor_info()注册 CPU 信息。为了不把话题铺的太开，我们不介绍这个过程。

d、 最后调用 acpi_parse_lapic_nmi()，该函数除打印一些信息，以及在 NMI 没有接到 LINT1 脚时打印一条警告信息，并没有实际用处。

4、 如果解析到了 LAPIC 信息，并且没有错误发生，acpi_parse_madt_lapic_entries()函

数返回 0 表示执行成功，此时会继续解析 IOAPIC entry。acpi_parse_madt_ioapic_entries()进行如下工作：

a、 调用 acpi_parse_ioapic()解析 IOAPIC entry，并调用 mp_register_ioapic()注册 IOAPIC 信息，等同 mp_ioapic_info()。此外，它还注册 IOAPIC 对应的 struct mp_ioapic_routing 结构。

b、 调用 acpi_parse_int_src_ovr()解析 ISO 条目，如发现 override，调用 acpi_sci_ioapic_setup()重载 SCI 中断，以及调用 mp_override_legacy_irq()重载 ISA 中断。

c、 至此，ISA 中断的信息已经全部获得了，调用 mp_config_acpi_legacy_irqs()配置 ISA 中断。该函数功能等同 construct_default_ioirq_mptable()。

d、 调用 acpi_parse_nmi_src()解析 NMI entry，仅打印相关信息。从 kernel 的注释来看，Linux 还不支持 IOAPIC 管脚配置成 NMI。

到此为止，无论是从 MP spec 路径，还是 ACPI 路径。Linux 完成了平台 APIC 系统的探测。我们来看看目前得到了些什么东西：

- LAPIC 地址
- IOAPIC 相关信息，在 mp_ioapics 数组中
- 中断源相关信息，在 mp_irqs 数组中

嗯，看上去够了。

2.1.3 APIC 的初始化

smp_boot_cpus()将调用 LAPIC 和 IOAPIC 的初始化函数。

2.1.3.1 LAPIC 初始化：

1、 调用 verify_local_APIC()，通过 LAPIC version 寄存器、LAPIC ID 寄存器验证 LAPIC。

2、 此时系统可能还处于 PIC 模式或 Virtual Wire 模式，调用 connect_bsp_APIC()通过 IMCR 切换入 APIC 模式

3、 调用 setup_local_APIC()设置 LAPIC。该函数流程如下：

a、 在一系列检验后，首先调用 init_apic_ldr()设置 DFR 和 LDR。目前 Linux 采用的是 Flat 模式。LDR 的 logical APIC ID = 1UL << smp_processor_id()。

> 笔者：Logical APIC ID 只有 8bit，当平台 CPU 超过 8 个后岂非要重复？不过问题似乎不大，因为是用 Lowest Prioirty 模式。但可能会对 IPI 有所影响，有机会验证一下

b、 设置 TPR 为 0。

笔者：从 Linux 的注释来看，TPR 将一直为 0。X86 spec 有这么一段话："对于使用 lowest priority delivery mode 发送中断，却又不更新 TPR 的 OS，芯片组将记忆 TPR 并将同一中断发送给同样的 CPU 处理。这将引起性能损失"。这应该是指 Xeon 和 Pentium4 系列，因为 P6 和 Pentium 系列是由 APR 决定 CPU 优先级的。这是否意味着，Linux 在 Intel 新的平台上，中断性能受到了影响？

c、 设置伪中断寄存器，enable LAPIC、打开 focus processor、设置伪中断的 vector 为 0xff。

题外话 —— 什么是伪中断

X86 spec 对它的描述是"当 INTR 被拉高，CPU 发出了 INTA cycle，软件屏蔽了中断，视之为伪中断"。软件屏蔽了中断？从前面的内容我们可以看到，LAPIC 情况下，CPU 没有提供屏蔽中断的方法（TPR 只阻止中断交给 CPU 处理，不影响 pending 到 IRR），cli？

Mp spec 对伪中断有描述："当一个中断在第一个 INTA 周期后，第二个 INTA 周期前变为无效，则为伪中断"。

8259A 的 spec 虽然没有明讲伪中断，但有这样的描述："PIC 收到一个中断时，拉高 INT 脚，此时 CPU 的应答周期随之开始（指 CPU 第一次 INTA 应答）。如果一个优先级更高的中断发生在第一次 INTA 和第二次 INTA 之间，INT 在第二次 INTA 后立刻拉低。在一段没规定长短的时间后，再次拉高 INT 脚通知高优先级中断的到来 ……"（此段话不是描述伪中断，而是值 INT 脚应该保持到第二次 INTA 的到来）

综合来看，伪中断就是 INT 脚没有维持足够长的有效电平，它应该维持到第二次 INTA 结束。

那么这种情况是如何产生的呢？笔者认为这是在第二个 INTA 到来前，CPU 通过 PIC 的 IMR 寄存器屏蔽中断，导致 INT 脚变低所致。由于没有找到 8259 的电路图，这仅是我个人猜测。

以上是关于 PIC 的，现在来看看 APIC。MP spec 提到："对于 APIC 系统，更容易产生伪中断， because the device interrupt may be latched and recognized without the INTA cycle"。（最后一句怎么翻译我都觉得不恰当，大家自己理解吧）。在不知道 APIC 的构造前，我完全无法理解这句话。起初认为，mask 中断发生在 IOAPIC 送出消息前，不会产生伪中断；mask 发生在 IOAPIC 送出消息后，由于 LAPIC 会接收，也不会有伪中断。但真相并非如此。LAPIC 在 CPU 中是怎样连接的？如果我告诉你 LAPIC 也有一个 INT 脚和 INTA 脚连 CPU，就像 PIC 一样，你会不会大吃一惊？真的是这样，虽然这两个脚在 CPU 内部只是两条硬连线而已。LAPIC 在决定提交中断给 CPU 时，同样拉高 INT 脚，

CPU 用两次 INTA 应答，就和 PIC 一样。记得我们前面说的 Remote IRR 异或机制吗？如果在第二次 INTA 之前，mask 了 IOAPIC 上对应的中断，会产生一条 level-deassert 消息，导致 IRR 对应 bit 清零。当第二个 INTA 到达时，LAPIC 中的 Proritier（一个部件）在 IRR 中找不到对应的 bit，无法提交 vector，最终产生一个伪中断。这暗示了只有 level 触发会产生伪中断。

此外，还有一种情况是 EOI 和设备驱动应答顺序出错而引起的。如果一个设备是 level 触发，设备必须保证在 EOI 到达 IOAPIC 前将中断线拉低，否则产生一个伪中断。这同样是 Remote IRR 引起的。该情形是：EOI 先到达，Remote IRR 清 0，此时中断线仍然为 1，产生 level-assert 消息。LAPIC 收到后设置 IRR，并向 CPU 提交中断，CPU 以第一个 INTA 应答。在第二次 INTA 发起前，驱动应答了设备，中断线拉低，产生 level-deassert 消息，LAPIC 清除 IRR。怎样，是不是和前面产生伪中断的过程一样？这个例子给了我们两个暗示：1）操作系统在设计时，必须调用了设备的中断处理程序后才写 EOI。2）设备的中断应答寄存器一定不能被 cache，否则延迟可能导致 EOI 比驱动的应答先到。

对于伪中断，硬件会提交一个预定好的伪中断 vector，操作系统需要给伪中断 vector 注册一个 handler，通常是打印一条信息或加个统计值什么的。注意，处理完伪中断不写 EOI。

X86 的伪中断寄存器除了可以设置伪中断 vector 外，还有一些其它功能，其格式如图：

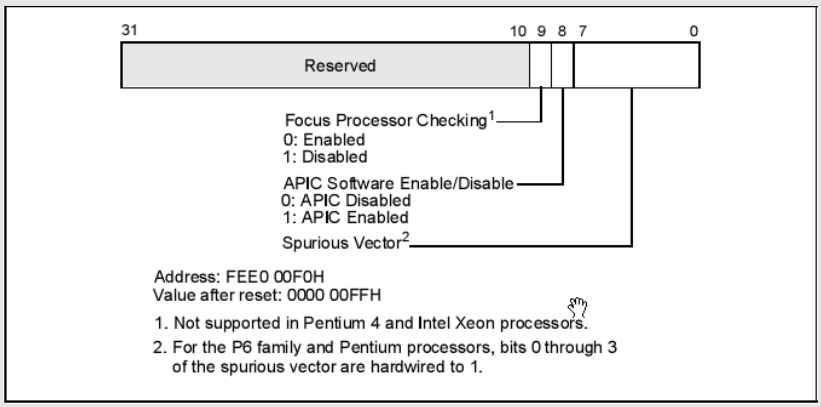


图 2-4 伪中断寄存器

其中 0~7bit 设置 vector；8bit 用于 enable/disable LAPIC，Linux 就是使用该方式 enable LAPIC 的。9bit 是 enable/disable focus processor。

笔者：对于 PIC，没有伪中断寄存器设置 vector。通常是使用和 IRQ7 相同的 vector 为伪中断 vector。

d、 设置 LINT0、LINT1 管脚。对于 BSP，如果系统有 PIC 模式，且 LINT0 没有被屏蔽，对应的 LVT 被设置成 ExtINT 模式。LINT1 管脚设置成 NMI 模式；如果是 AP，则将 LINT0 设置成 ExtINT 模式并屏蔽掉。LINT1 管脚设置成 NMI

模式并屏蔽掉。

题外话 —— 什么是 LVT?

LVT，Local Vector Table，是 LAPIC 自身中断源的配置表，它和 IOAPIC 的 PRT 表有类似结构。到目前为止，LAPIC 最多有 6 个中断源，根据不同系列的 CPU，中断源可能更少一些。6 个中断源如下：

- APIC Timer 中断：APIC 时钟，对应 LVT Timer Register
- 热量探测中断：就是探测 CPU 温度那个小探头，对应 LVT Thermal Monitor Register
- 性能计数溢出中断：对应 LVT Performance Counter Register
- LINT0 脚中断：对应 LVT LINT0 Register
- LINT1 脚中断：对应 LVT LINT1 Register
- APIC 错误中断：当 APIC 探测到一个内部错误时产生，对应 LVT Error Register

各个寄存器的格式如图：

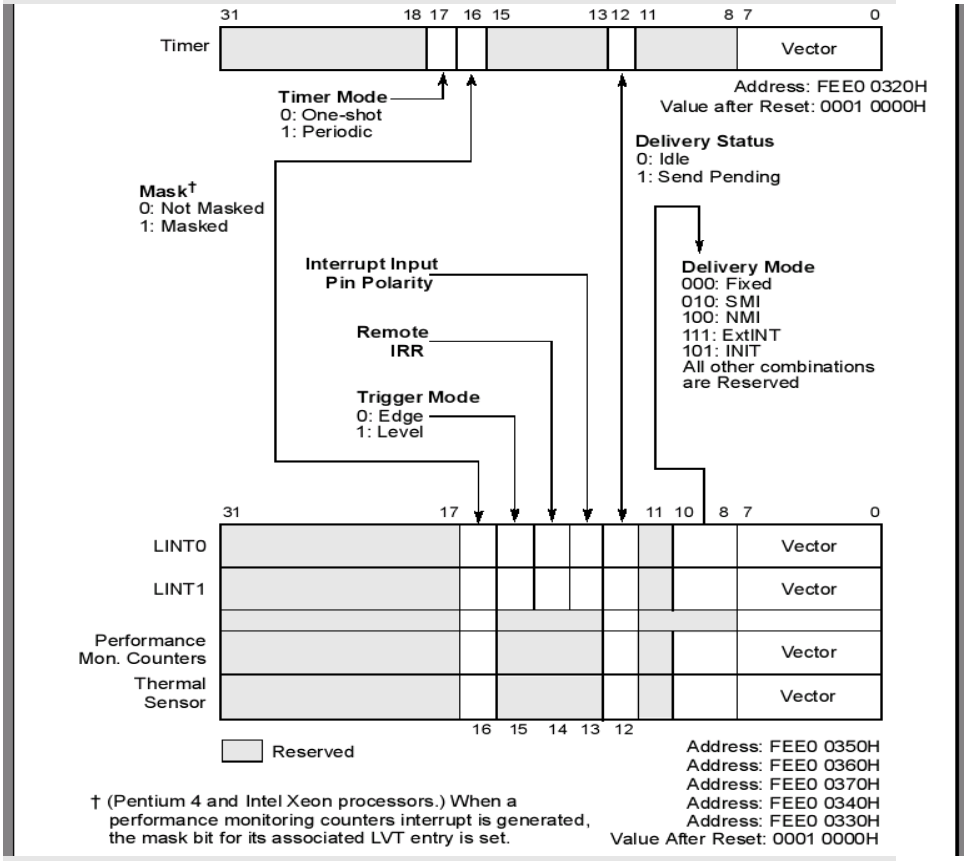


图 2-6 LVT 格式

e、 设置 APIC Error 的 vector 为 0xfe，清 0 并 enable 该中断。

至此，LAPIC 相关设置就已完成。当然，还有一些我们没提到，比如 APCI timer 的设置。这些以后再说吧。

2.1.3.1 IOAPIC 的初始化：

smpboot_setup_io_apic()函数会调用 setup_IO_APIC()完成所有的工作。setup_IO_APIC()流程如下：

1、 调用 enable_IO_APIC()对初始化一些基本数据结构，这些结构是：

- irq_2_pin 全局数组：用于记录 IRQ 和 pin 的对应关系，以 IRQ 为索引，可以得到对应 pin 的信息。Pin 用结构体 struct irq_pin_list 表示，它有三个字段：

字段	描述
Apic	该 pin 从属的 IOAPIC，该值用于所以 mp_ioapics 数组
Pin	管脚号
Next	下一个 pin 结构体

表 2-12 struct irq_pin_list 结构体

目前 Linux 中 irq_2_pin 元素个数为 2 * NR_IRQS。NR_IRQS 是平台最大 IRQ 数量，对于 IOAPIC，该值为 224。对于 PIC，该值为 16。

- pirq_entries 全局数组：若在内核启动参数中配置了 PIRQ 参数，其值记录在 pirq_entries 数组中。目前内核允许最多配置 8 个 PIRQ。
- nr_ioapic_registers 全局数组：用于记录每个 IOAPIC 的管脚数，目前内核最多支持 64 个 IOAPIC，故该数组长度为 64。
- ioapic_i8259 全局变量：记录 PIC 所在的 IOAPIC 与管脚。有两个字段，apic 表示 IOAPIC（同 struct irq_pin_list 的 apic 字段），pin 代表管脚号。

> 笔者：关于 PIRQ 内核参数的详细信息，请参考内核文档 Documentation/i386/IO-apic.txt。呵呵，不要被名字迷惑以为内容很多，它除了讲 PIRQ 什么都没有。此外，如果看了该文档仍不明白 PCI 中断线和 PIRQ 的换算关系，可以参考 PCI spec 的 2.2.6 节，里面有一个 INTA~D 到 PIRQ 的换算公式。这里就不多做介绍了。PIRQ 内核参数在 ioapic_pirq_setup()函数中解析。

enable_IO_APIC()首先将 irq_2_pin 和 pirq_entires 初始化到无效状态，并读取每个 IOAPIC 的管脚数存入 nr_ioapic_registers 数组。接着探测是否有 PIC 接到 IOAPIC 上，这通过读取每个 IOAPIC 的 PRT 表，寻找是否有 RTE 被设置成了 ExtINT 模式实现。如果找到，把 apic 和 pin 信息记录在 ioapic_i8259 中。最后通过 mp_irqs 数组，验证 MP table 报告的 PIC 信息和自己读取到的是否相同，如果不同，信任 MP table 并更改 ioapic_i8259 相应字段。最后将所有 IOAPIC 的 PRT 表 mask 掉，RTE 为 SMI 类型除外。

2、 设置一个 bit map ——unsigned long io_apic_irqs。该 bit map 每一 bit 对应一个 IRQ，该 bit 为 1，表示该 IRQ 接 IOAPIC，否则接 PIC。如果系统处于 ACPI 模式，io_apic_irqs = ~0UL；否则，io_apic_irqs = ~(1<<2)。

> 笔者：如果系统遵循 MP spec，IRQ2 用于接第二片 i8259，故系统中不会有 IRQ2

存在。对于 ACPI 模式，没找资料说 PIC 的接法。

3、 如果系统不处于 ACPI 模式，调用 setup_ioapic_ids_from_mpc()。该函数根据从 MP table 得到的 IOAPIC ID，设置各 IOAPIC 的 APIC ID 寄存器。并查找是否有冲突，如有冲突则重新分配，并更新 mp_irqs 中对应中断的 IOAPIC ID 字段。

笔者: 还记得我们前面有一个关于 APIC ID 的题外话吗？我们说对于 Pentium4 和 Xeon 系列，IOAPIC ID 只用于区分多个 IOAPIC，即使 LAPIC ID 冲突也无所谓。在 setup_ioapic_ids_from_mpc()中，如果系统的 APIC 是 xAPIC，不用检查 IOAPIC ID 直接返回，因为在没有 APIC BUS 的情况下，它们毫无意义。什么是 xAPIC？Pentium4 和 Xeon 系列用的 APIC 就叫 xAPIC。内核佐证了我们前面的说法。

4、 调用 sync_Arb_IDs()，如果当前系统 APIC 使用 APIC BUS 通讯，该函数将广播一个 Level 触发、类型为 INIT 的 IPI（Inter Processor Interrupt，处理器间中断），将各 LAPIC 的 Arb 同步为其自身 LAPIC ID。

题外话 ——Arb 和 LAPIC 总线竞争

Arb，Arbitration Register，仲裁寄存器。该寄存器用 4 个 bit 表示 0~15 共 16 个优先级（15 为最高优先级），用于确定 LAPIC 竞争 APIC BUS 的优先级。系统 RESET 后，各 LAPIC 的 Arb 被初始化为其 LAPIC ID。总线竞争时，Arb 值最大的 LAPIC 赢得总线，同时将自身的 Arb 清零，并将其它 LAPIC 的 Arb 加一。由此可见，Arb 仲裁是一个轮询机制。Level 触发的 INIT IPI 可以将各 LAPIC 的 Arb 同步回当前的 LAPIC ID。

对于 Pentium4 和 Xeon 系列，总线竞争由前端总线协议决定。

笔者：IOAPIC 是否有 Arb 用于总线竞争？我猜想是有的。但 x86 spec 没说，在看 APIC 系统实现时，关于总线竞争的部分又看的太模糊了，没大看懂。想着以后不用这东西，看着就没动力了，有兴趣的朋友可以深究一下。

关于 IPI 的内容，《Interrupt in Linux（软件篇）》中会介绍（如果我写了^_^）

5、 调用 setup_IO_APIC_irqs()。该函数是 IOAPIC 初始化中的重中之重，几乎所有工作都是由它完成的，下面我们看看其流程：

a、 初始化各 IOAPIC 的 PRT 表。内核用 struct IO_APIC_route_entry 结构表示 RTE，下表描述了内核初始化 RTE 的格式：

字段	对应 RTE 字段	初始值
Vector	Vector	见后面内容
Delivery_mode	Delivery Mode	Lowest Priority（001）
Dest_mode	Destination Mode	Logical（1）
Polarity	Polarity	irq_polarity()函数根据 mp_irqs 中的信息，或总线类型决定
Trigger	Trigger Mode	irq_trigger()函数根据 mp_irqs 中的信息，或总线类型决定

Mask	Mask	Enable(0)
logical_dest	Destination Field	所有可用的 CPU

表 2-13 内核初始 RTE 结构值

从表中可以看出，内核将 RTE 配置成了 logical 模式，并且中断消息的目的地是所有 CPU，Delivery mode 为 lowest priority。这样 IOAPIC 的中断消息由优先级最低的 CPU 接收。

b、 调用 pin_2_irq()将 pin 转换成 IRQ。对于 ISA 中断，pin 对应的 IRQ 由 mp_irqs 数组得到。对于 PCI 中断，使用 GSI 方式把 pin 转换成唯一的 IRQ，其代码如下：

```
i = irq = 0;
while (i < apic)
    irq += nr_ioapic_registers[i++];
irq += pin;
```

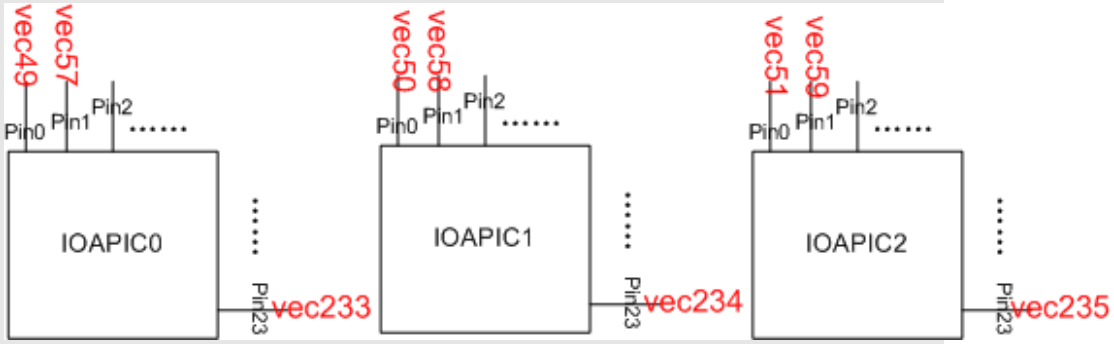
笔者：可以看出，内核使用了 GSI 的思想，同时我们也可以看出 IRQ 和 GSI 是个可以互换的概念。

c、 调用 add_pin_to_irq()将 IRQ 和 pin 的对应关系存入 irq_2_pin 数组。

d、 如果是该 IRQ 接 IOAPIC 管脚（用前面提到的 io_apic_irqs bit map 判断），调用 assign_irq_vector()为该 IRQ 分配一个 vector，然后调用 ioapic_register_intr()给该 IRQ 注册一个 handler。分配的 vector 会存入一个 irq_vector 全局数组，用 IRQ 号为索引，可得到对应的 vector。

题外话 —— Linux 的 Vector 分配策略

__ assign_irq_vector()写的比较晦涩，较为难懂。我算法烂，就不画流程图论述过程了。画了一副图，展示 vector 分配的策略：



图中有 3 个 IOAPIC。根据 Linux 分配 IRQ 的策略，IOAPIC0 24 个管脚对应 IRQ0~23，pin0 对应 IRQ0。IOAPIC1 的 pin0 对应 IRQ24 …… 依此类推。由于 vector 本身是代表优先级的，为了公平，Linux 将所有 vector 平均的分配给 3 个 IOAPIC。IRQ0 分配到 vector49，IRQ24 分配到 vector50，IRQ48 分配到 vector51 …… 依次类推。Linux 中设备可用的第一个 vector 是 0x31，也就

是 vector49，最大可用 vector 为 0xef。但实际上__ assign_irq_vector()的实现将最大可用 vector 限制到了 238，最多支持 8 个 IOAPIC（每个 IOAPIC 24 个管脚，且最后一个 IOAPIC 有 3 个管脚分配不到 vector）。在分配的过程中，避开了 vector 0x80。

这个函数看的我有点头疼，为什么限制到 238？为什么从 49 开始分配？没用的 vector 留给谁的？暂时不去想了，大家可以深究一下。

笔者：上面例子中的假设，IOAPIC0 的 pin0 对应 IRQ0，实际上不是这样的。前面已经讲了 ISA 中断的 IRQ 从 MP table 得到，IRQ0 实际对应 pin2。这样假设只是为了论述方便。

ioapic_register_intr()注册的 handler，只是通用的处理函数（如处理 level 中断的，处理 edge 中断的），它会具体再调用设备的中断处理函数。ULK3 上的中断处理路径__do_IRQ()已经过时了，内核已引入 generic IRQ layer。

如果我写了《软件篇》，会介绍该处理机制。内核文档 Documentation/DocBook/genericirq.tmpl 详细讲解了 generic IRQ layer，很简单的，大家可以自己看看。

Vector 是值越大优先级越高，PIT 对应 IRQ0，按 Linux 的分配策略，它的优先级最低。是我错了？还是 PIT 在 Linux 中本身就不重要？

e、 对于 ISA 中断，调用 disable_8259A_irq()将 PIC 上对应的管脚 mask 掉（进入 APIC 模式后，PIC 要被 mask 掉）。

f、 至此，我们已经配置好一个 RTE 的全部信息，调用__ioapic_write_entry()将该 RTE 写到 IOAPIC 中。

6、 调用 init_IO_APIC_traps()为 irq_vector 数组中未分配到 vector 的 IRQ 设置默认的处理函数（通常这些 IRQ 不会发生，但 Linux 尽可能的保证安全。这个属于 generic IRQ layer 的内容，就不多讲了）。

好了，搞定。IOAPIC 已经设置好了，有些内容我们没提到，例如 check_timer()，有机会在说吧。

第三章 你应该知道的硬件知识

3.1 APIC 和 PIC 共存下的系统

PIC 彻底走进历史的日子已经不远，Windows 已经开始叫嚣要放弃对 PIC 系统的支持，新的架构也彻底和 PIC 说 byebye（例如我们伟大的 IA64^_^）。当然，PIC 还会继续在单片机、嵌入式领域发挥余热，这就不是我们所能理会的了。但目前 PIC 和 APIC 共存的情况还普遍存在，MP spec 为 PIC 和 APIC 共存的平台规定了三种模式：PIC mode、Virtual Wire Mode、Symmetric I/O Mode。

> 笔者：Symmetric 这个单词打着甚为麻烦，下文把 Symmetric I/O Mode 称为 APIC mode。

三种模式中，PIC mode 和 Virtual Wire Mode 互斥存在，所谓有你没它。APIC mode 是所有 MP 平台最终进入的模式。Spec 规定，为了 PC/AT compatibility，系统在 RESET 后首先进入 PIC mode 或者 Virtual Wire mode，操作系统（或 BIOS）在适当时候切换入 APIC mode。

3.1.1 PIC mode

IMCR，Interrupt Mode Configuration Register，中断模式配置寄存器，控制当前系统的中断模式——PIC？还是 APIC？当系统 RESET 后，该寄存器清 0，系统默认进入 PIC 模式。此时 BSP（Boot Startup Processor，多处理器系统中第一个启动的 CPU）的 NMI 和 INTR 脚为硬连线，直接从外部接入，不经过 APIC。下图显示了这个结构：

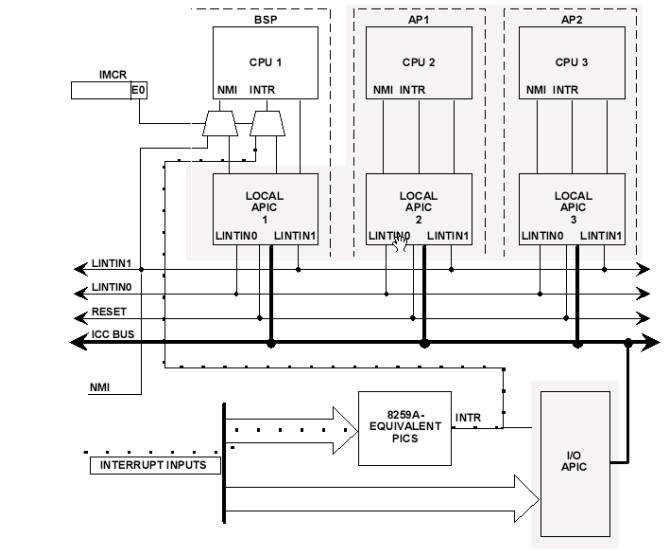


图 3-1 PIC 模式中断连接图

注意图中的虚线，PIC 模式下，外部中断通过 PIC 直接到达 BSP 的 INTR 脚，而 NMI

直接连接 BSP 的 NMI 脚。

对 IMCR 写 1，可将系统切换至 APIC 模式。此时外部中断直接通过 APIC 到达 CPU，NMI 则连接 LAPIC 的 LINT1 脚。

题外话 —— IMCR 的访问

MP spec 规定，I/O 端口 22h 和 23h 用于支持 IMCR 寄存器。对 22h 端口写 70h 可选中 IMCR，此时对 23h 端口读/写即可。此外，如果 PIC mode 没有实现，IMCR 则可能没有实现。MP table 的 MP feature information 字节的 IMCRP bit 报告平台是否有 ICMR。

笔者：IMCR 和 PIC mode 可能已被埋入了历史的黄土。笔者查阅了 ICH9（ICH 即南桥）的 spec，没有找到该寄存器的描述。Google 了一下，除了 mp spec 其它地方都没提到它。窃以为，此物已死。

3.1.2 Virtual Wire Mode

顾名思义，该模式有一条“虚导线”。这条“虚导线”就是 APIC——LAPIC 或 IOAPIC。除此之外，该模式和 PIC 模式没有大的区别。

IOAPIC 用作“虚导线”情况如下：

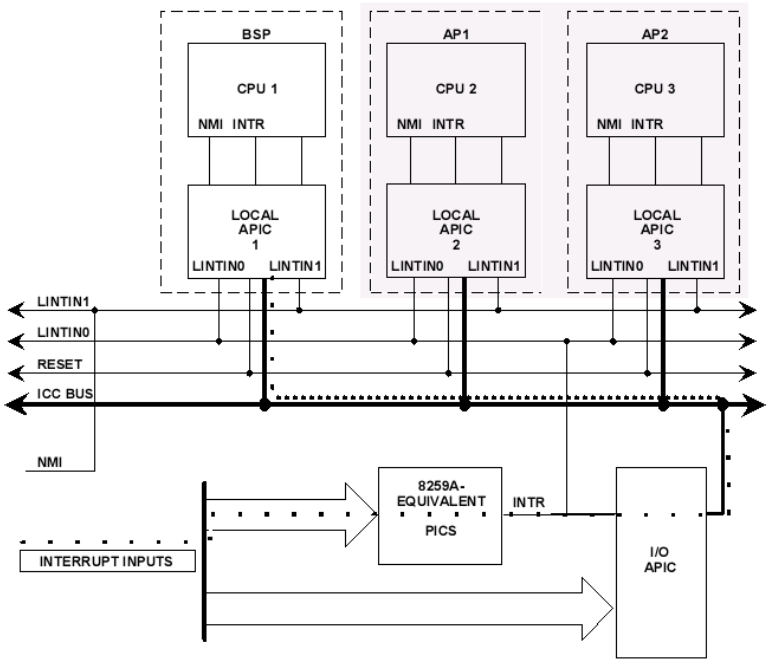


图 3-2 Virtual Wire Mode —— IOAPIC

如图虚线，外部中断通过 PIC 连接的 IOAPIC 管脚，最终到达 BSP。当然，连接 PIC 的这个管脚需要被配置成 ExtINT 模式。

笔者：spec 对 IOAPIC 用作 Virtual Wire 的描述只有寥寥数语。根据前面介绍 IOAPIC 的内容，我们来猜测一下 IOAPIC 是什么样的。首先，IOAPIC 的 PRT 表已经配置好了，不

然无法向 LAPIC 发中断消息。其次由于外部中断即连接 PIC 又连接 IOAPIC，为了避免出现中断复接的情况，IOAPIC 中除了连接 PIC 的管脚，其它的都应该被 mask。连接 PIC 的管脚 delivery mode 配置成 ExtINT。再来猜猜 PIC 是什么样的。图中没有话出，但 PIC 的 INTA 脚肯定是连到了 BSP 的，因为对于配置成 ExtINT 的中断消息，CPU 要通过 INTA 脚应答 PIC，并接收中断的 vector …… 以上纯属个人猜测

LAPIC 用作 Virtual Wire 的情况：

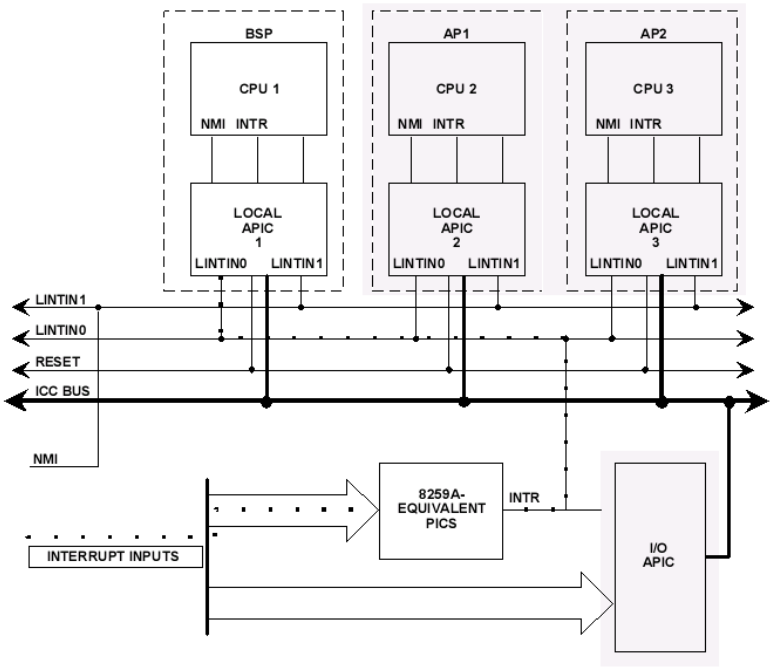


图 3-3 Virtual Wire Mode —— LAPIC

还是看图中虚线，这次 PIC 接的是 BSP 的 LINT0 脚，当然，该管脚也要配置成 ExtINT 模式。

笔者：MP spec 没说如何从 Virtual Wire Mode 切换到 APIC 模式。我们再猜测一下，重新配置模式为 ExtINT（LAPIC 或 IOAPIC）的管脚，将 PIC 的所有管脚 mask 掉？

3.1.3 APIC mode

没什么好说的，直接看图：

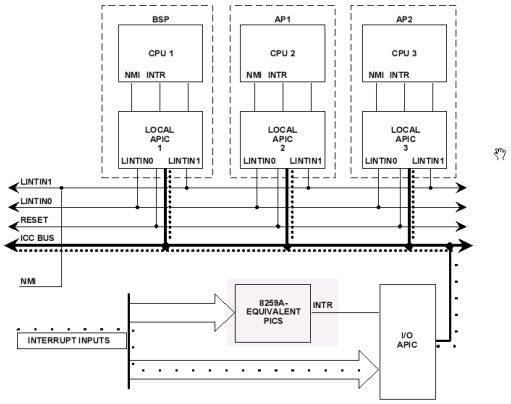


图 3-4 APIC mode

一个要求，进入 APIC 模式后要将 PIC 的所有管脚 mask 掉。

3.2 PCI 中断转 ISA 中断协议

PCI 为啥要转 ISA 中断？这是很堂皇问题，毕竟，任何退步都是可耻的。但在计算机领域，兼容才是王道，才是市场。否则，Intel 就不会栽个跟头让 AMD 风光好几年 …… 我的废话好像有点多了。

话归正题，也是个很堂皇的答案，PIC 是为 ISA 设计的，不是为你 PCI 设计的。当然，你可以接 APIC，不过在旧社会是没有它滴。旧社会，PCI 要接 PIC。

ISA 中断和 PCI 中断最大的不同在哪儿？ISA 是上升沿触发的（low-to-high edge sensitive），PCI 是低电平有效、电平触发的(active low, level sensitive)。（我们不要那么麻烦，记住 ISA 是 edge 触发，PCI 是 level 触发就好了）。PCI 要接 PIC，就需要把 level 触发转换成 edge 触发。本章介绍 PCI 中断转 ISA 中断协议，它是中断路由可以产生的基础（中断路由见下一章）。

Intel 公司 Albert R.Nelson 等人的《PCI to ISA Interrtup Protocol Converter and Selection Mechanism》一文讲解了实现。

先来看副图：

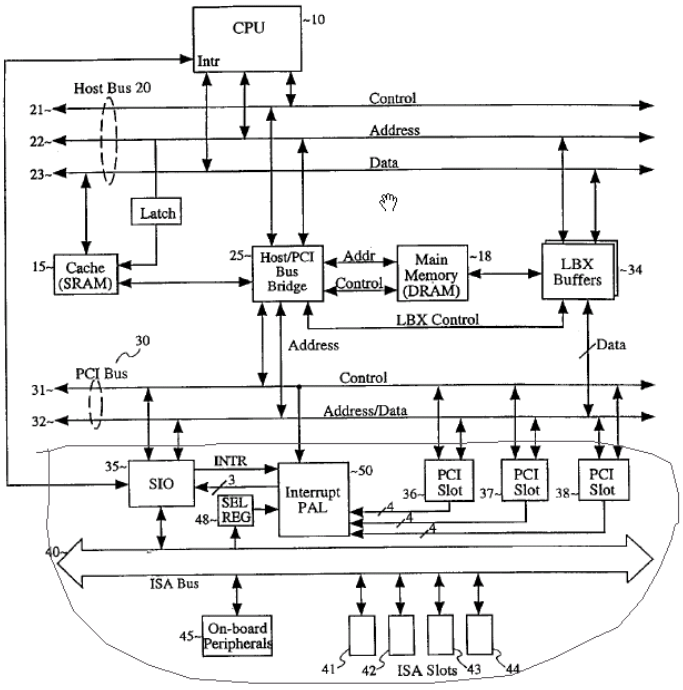


图 3-5 PCI 转 ISA 概要

内容有点多，不用全看，看下面画框的部分就可以了（我也想用红色画个框，但不知道这是什么鬼图，硬上不了色！）。

图中有几个主要部件，一个是 ISA 总线，它的中断线连接到 SIO（SIO 在本章中看作 PIC 即可）；一个是 Interrupt PAL（简称 PAL，它是完成 PCI 中断转换到 ISA 中断的关键），连接 3 个 PCI slot 和 SIO；还有一个 SEL REG。三者中的关键是 PAL，只要明白它的结构，整个协议就清楚了，其结构见下图：

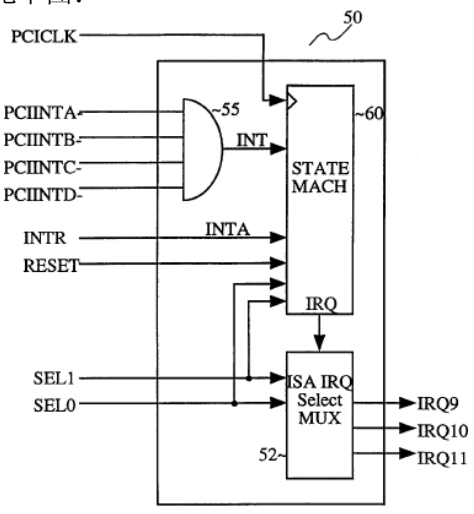


图 3-6 PAL 结构

图中，PCI 的 4 条中断线 INTA、INTB、INTC、INTD 通过一个或门产生 INT 信号，接入状态机。INTR（即 PIC 的 INTR 脚）和一个 RESET 脚一起接入状态机（STATE MACH）。这里 INTR 脚在图中又标为 INTA，为了避免和 PIC 中的 INTA（有上划线那个）混淆，下文

仍称它为 INTR。

> 笔者：这万恶的图啊，搞一个这么容易混淆的标注，我就把它和 PIC 的 INTA 混淆了，直接导致后来在看状态机时想不通其原理，还自己提出 N 种说法来解释，真是 E$%^#$&$%&%^&*$%&&%

SEL1、SEL0 对应 SEL REG，用于控制中断通过哪条 IRQ 线连到 PIC（图中为 IRQ9、IRQ10、IRQ11）。图中的选择器 ISA IRQ Select MUX 还有一个输入为 IRQ，它是状态机的输出。整个 PAL 的运作，主要是靠状态机控制，下图为所有状态的转换情况：

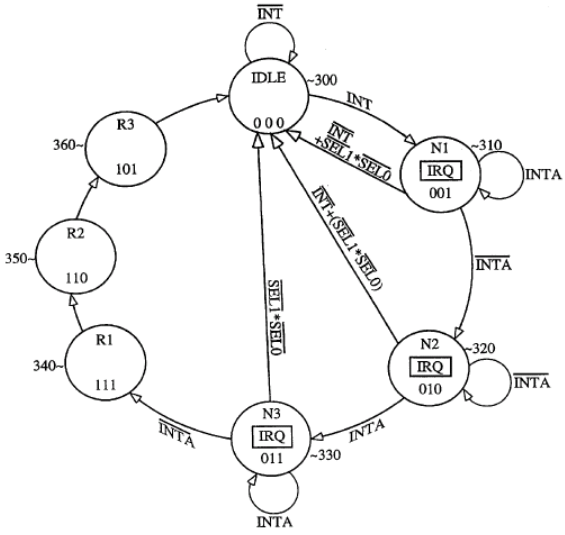


图 3-7 状态转换图

> 笔者：再强调一次，图中的 INTA 就是 PIC 的 INTR 脚。并且图中的各个管脚，我们统一用高电平表示有效电平。否则英文中一个 active，我就要打 4 个字，太麻烦了。

状态机 RESET 过后处于 IDLE 状态，此时没有中断。当某个 PCI 设备把自身中断线拉到有效电平后，INTA/B/C/D 变为高电平，通过或门后 INT 变高，从而 IRQ 变高。状态机在下一个时钟上升沿时（状态机由图中 PCICLK 信号驱动）进入 N1 状态。此时 INTR 脚如果拉高，说明 CPU 正在处理 PIC 提交的中断，故停留在 N1 状态等待。此时，INT 拉低或 SEL0、SEL1 清零将导致返回 IDLE 状态。

> 笔者：PCI 设备中断线 active，但在有效时间内又 in-active 的情况少，但确实存在。对于 SEL0、SEL1 清零即 SEL REG 清零。文章称这代表 BIOS（操作系统）不愿意接收 PCI 中断或表明这是一个特殊的中断，或默认中断。不太明白是什么意思。在我看来，SEL REG 的用处本身就不明了，完全可以在布线时选一条没有 ISA 设备使用的管脚硬连线。提供这种动态选线的功能有什么好处？

当 INTR 被拉低（如果它之前是高电平），代表 CPU 已经处理完先前中断，此时 N1 状态前进到 N2 状态。如果 INT、SEL0、SEL1 发生了和 N1 状态相同的变化，就回到 IDLE。如果 INTR 仍为低电平，表示 PIC 还没有中断 CPU，状态机停留在 N2，INT 和 IRQ 都维持

在原电平。

笔者：原文这样描述：When the INTA signal again goes active, it indicates that CPU is responding to an interrupt which may be the interrupt generated by the PCI agent …… 搞不清楚这里的"is responding to"该怎么翻译，如果翻译成"应答"（即 acknowledge）显然不合适。因为 PIC 必须先拉高 INTR，CPU 才能通过 INTA 脚应答。这里的 N2 状态中，INTR 变高，只能说明 PIC 中断 CPU，不能说明 CPU 应答了 …… 唉，英语太烂，只能这样讲了。

INTR 变高，便进入了 N3 状态，且该状态将一直维持到 INTR 拉低。该状态代表 CPU 正在处理一个可能是自己发起的中断。N3 状态中，SEL0、SEL1 清零将导致回答到 IDLE。在此状态下，状态机的 IRQ 脚继续维持在高电平。一旦 INTR 拉低，状态机从 N3 转移到 R1、R2、R3 三个状态，在每个状态停留一个 PCICLK 周期，最后回到 IDLE。如果此时 INT 脚仍然为高，说明 N3 状态中，CPU 处理的不是自己发出的 PCI 中断，状态机迁移到 N1 继续前面的过程。

OK，明白没？没明白只有看原文了。

笔者：文章强调："It should be understood that the series of states following the N3 state through the R1, R2 and R3 state, back through the IDLE state and to N1 state provides a necessary delay. It is this delay and the switch off and back on of the IRQ signal from the interrupt PAL that converts the PCI compliant level sensitive interrupt received by the interrupt PAL to the low-to-high edge sensitive interrupt driven to the interrupt controller of the SIO component as an ISA IRQ compliant signal"。我看不出来从 N3 到 R1、R2、R3、IDLE 再到 N1 提供了一个多么必要的 delay。这个 delay 只有在转换完一个 PCI 中断，马上转换下一个时才 necessary。也就是 INT 脚在状态机遍历完所有状态后仍然为高电平，需要再次进行 PCI to ISA 转换的情况。为了构建一个上升沿，我们需要一段时间维持在低电平，这就是 R1、R2、R3 三个延时状态的作用。

与之相比，我认为" the switch off and back on of the IRQ signal"比较关键。Edge 触发和 level 触发毕竟不相同，虽然说一个高电平有效的 level 触发也有一个上升沿，但 ISA IRQ Select MUX 的输出总不能一直在高电平吧。什么时候 switch off 的呢？我认为是在 N3 中。尽管此时状态机的 IRQ 输出仍为高电平，但我们注意到，N3 回到 IDLE 的情况中，INT 变低这一情况已经没有了。在 N1、N2 中它都有的。这说明此时 INT 变低已经不影响大局，由此推断，ISA IRQ Select MUX 在 N3 中已经 switch off 了它的输出，完成了一次 level 到 edge 的转换。

3.2.1 PIRQ Table

忘了上面这个复杂的信号转换吧，它会让我们偏离我们的目标——我们仅仅需要知道的是：需要把 PCI 中断映射到 ISA 中断，需要把电平触发转换为边缘触发，剩下的事情，是电

子工程师的领域，作为一个附录应该不错：）

为了描述 PCI 到 ISA 中断的映射，微软公司在最开始就提出了 PCI Interrupt Routing Table 的数据结构，它的作用就是用来描述在使用 8259 中断控制器的系统下，PCI 中断（INTA-INTD）如何路由到 ISA 的 IRQ。

不说废话了，给出 PIRQ Table 的数据结构：

表 3-1 PIRQ Table 数据结构

Byte Offset	Size in Bytes	Name
0	4	Signature
4	2	Version
6	2	Table Size
8	1	PCI Interrupt Router's Bus
9	1	PCI Interrupt Router's DevFunc
10	2	PCI Exclusive IRQs
12	4	Compatible PCI Interrupt Router
16	4	Miniport Data
20	11	Reserved (Zero)
31	1	Checksum
32	16	First Slot Entry
48	16	Second Slot Entry
(N + 1) * 16	16	Nth Slot Entry

每个 slot entry 的结构如下：

表 3-2 Slot 字段

Byte Offset	Size in Bytes	Name
0	Byte	PCI Bus Number
1	Byte	PCI Device Number (in upper five bits)
2	Byte	Link Value for INTA#
3	Word	IRQ Bitmap for INTA#
5	Byte	Link Value for INTB#
6	Word	IRQ Bitmap for INTB#
8	Byte	Link Value for INTC#
9	Word	IRQ Bitmap for INTC#
11	Byte	Link Value for INTD#
12	Word	IRQ Bitmap for INTD#
14	Byte	Slot Number
15	Byte	Reserved

捡几个重要的字段解释：

PCI Interrupt Router's Bus：

PCI Interrupt Router's DevFunc:

PCI Interrupt Router 是一个硬件设备，而且是一个 PCI 设备！这两个字段描述该设备的 BDF 地址。

PCI Exclusive IRQs:

并不是所有 ISA 的 IRQ 都能用来作为 PCI 的映射 IRQ，因此这个字段描述了那些只能被 PCI 使用的 IRQ。换句话说，这些 IRQ 不能被 ISA 设备使用。

IRQ Bitmap for INTA#:

每个 INTx 可能会有多个 IRQ 可供选择，因此需要一个 bitmap

容易看出，PIRQ Table 描述了每个 PCI 设备（BDF 地址指定）的每个 INTx 对应的可能的 IRQ 值。下面是我在虚拟机上得到的结果：

Entry	Location	Bus	Device	Pin	Link	IRQs
0	slot 1	0	3	A	0x60	0 1 4 6 8 9 10 11 12 15
0	slot 1	0	3	B	0x61	1 2 4 6 8 9 10 11 12 15
0	slot 1	0	3	C	0x62	0 3 4 6 8 9 10 11 12 15
0	slot 1	0	3	D	0x63	2 3 4 6 8 9 10 11 12 15
1	slot 2	0	4	A	0x61	0 1 5 6 8 9 10 11 12 15
1	slot 2	0	4	B	0x62	1 2 5 6 8 9 10 11 12 15
1	slot 2	0	4	C	0x63	0 3 5 6 8 9 10 11 12 15
1	slot 2	0	4	D	0x60	2 3 5 6 8 9 10 11 12 15
2	slot 3	0	4	A	0x62	0 1 4 5 6 8 9 10 11 12 15
2	slot 3	0	4	B	0x63	1 2 4 5 6 8 9 10 11 12 15
2	slot 3	0	4	C	0x60	0 3 4 5 6 8 9 10 11 12 15
2	slot 3	0	4	D	0x61	2 3 4 5 6 8 9 10 11 12 15
3	slot 4	0	4	A	0x63	0 1 7 8 9 10 11 12 15
3	slot 4	0	4	B	0x60	1 2 7 8 9 10 11 12 15
3	slot 4	0	4	C	0x61	0 3 7 8 9 10 11 12 15
3	slot 4	0	4	D	0x62	2 3 7 8 9 10 11 12 15
4	slot 5	0	4	A	0x60	0 1 4 7 8 9 10 11 12 15
4	slot 5	0	4	B	0x61	1 2 4 7 8 9 10 11 12 15
4	slot 5	0	4	C	0x62	0 3 4 7 8 9 10 11 12 15
4	slot 5	0	4	D	0x63	2 3 4 7 8 9 10 11 12 15
5	slot 6	0	5	A	0x61	0 1 5 7 8 9 10 11 12 15

```
5    slot 6      0    5    B    0x62   1 2 5 7 8 9 10 11 12 15
5    slot 6      0    5    C    0x63   0 3 5 7 8 9 10 11 12 15
5    slot 6      0    5    D    0x60   2 3 5 7 8 9 10 11 12 15
6    embedded    0    0    A    0x60   0 1 4 5 7 8 9 10 11 12 15
6    embedded    0    0    B    0x61   1 2 4 5 7 8 9 10 11 12 15
6    embedded    0    0    C    0x62   0 3 4 5 7 8 9 10 11 12 15
6    embedded    0    0    D    0x63   2 3 4 5 7 8 9 10 11 12 15
7    embedded    0    1    A    0x60   0 1 6 7 8 9 10 11 12 15
7    embedded    0    1    B    0x61   1 2 6 7 8 9 10 11 12 15
7    embedded    0    1    C    0x62   0 3 6 7 8 9 10 11 12 15
7    embedded    0    1    D    0x63   2 3 6 7 8 9 10 11 12 15
8    embedded    0    0    A    0x60   0 1 4 6 7 8 9 10 11 12 15
8    embedded    0    0    B    0x61   1 2 4 6 7 8 9 10 11 12 15
```

看到这里，提出两个问题：

1、 PCI Interrupt Router 的作用是映射 PCI 中断到 ISA 中断，而它自己也是一个 PCI 设备，会不会出现“先有鸡还是先有蛋”的问题呢？

不会，PCI Interrupt Router 其实是作为南桥的一个功能单元而存在，访问这个设备是通过配置寄存器实现的，因此只需要 BDF 即可。下面是 Intel 主板的一个实现：

4.1.10. PIRQRC[A:D]—PIRQX ROUTE CONTROL REGISTERS (FUNCTION 0)

Address Offset : 60h (PIRQRCA#)–63h (PIRQRCD#)
Default Value: 80h
Attribute: R/W

These registers control the routing of the PIRQ[A:D]# signals to the IRQ inputs of the interrupt controller. Each PIRQx# can be independently routed to any one of 11 interrupts. All four PIRQx# lines can be routed to the same IRQx input. Note that the IRQ that is selected through bits [3:0] must be set to level sensitive mode in the corresponding ELCR Register. When a PIRQ signal is routed to an interrupt controller IRQ, PIIX4 masks the corresponding IRQ signal.

Bit	Description
7	**Interrupt Routing Enable.** 0=Enable; 1=Disable.
6:4	**Reserved.** Read as 0s.
3:0	**Interrupt Routing.** When bit 7=0, this field selects the routing of the PIRQx to one of the interrupt controller interrupt inputs.

Bits[3:0]	IRQ Routing	Bits[3:0]	IRQ Routing	Bits[3:0]	IRQ Routing
0000	Reserved	0110	IRQ6	1011	IRQ11
0001	Reserved	0111	IRQ7	1100	IRQ12
0010	Reserved	1000	Reserved	1101	Reserved
0011	IRQ3	1001	IRQ9	1110	IRQ14
0100	IRQ4	1010	IRQ10	1111	IRQ15
0101	IRQ5				

不同芯片组支持的设备可能不同，因此这些可能的 IRQ 也可能会有差别。通过这个 4 个寄存器，我们甚至可以动态地将 PCI 的某个中断映射到不同的 IRQ 值（虽然并不多见）。

2、 BIOS 怎么知道那些 IRQ 可用？

这个表是 BIOS 提供给 OS 使用的，但是 BIOS 也不能凭空捏造吧。其实说 BIOS“凭空捏造”也未尝不可，因为 BIOS 都是这些主板厂商自己写的，他们太了解这些东西了，看看下面 BIOS 的一段代码就恍然大悟了：

```
const struct irq_routing_table intel_irq_routing_table = {
        PIRQ_SIGNATURE, /* u32 signature */
        PIRQ_VERSION,     /* u16 version      */
        32+16*4,        /* there can be total 5 devices on the bus */
        0x00,           /* Bus 0 */
        0x08,           /* Device 1, Function 0 */
        0x0000,         /* reserve IRQ 11, 9, 5, for PCI */
        0x1039,         /* Silicon Integrated System */
        0x0008,         /* SiS 85C503/5513 ISA Bridge */
        0x00,           /*    u8 miniport_data    - "crap" */
        { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }, /* u8 rfu[11] */
        CHECKSUM,     /*    u8 checksum             - mod 256 checksum must give zero */
        {
              /* bus, devfn, {link, bitmap}, {link, bitmap}, {link, bitmap}, {link, bitmap}, slot, rfu
*/
              {0x00, 0x08, {{0x41, 0xdcb8}, {0x42, 0xdcb8}, {0x43, 0xdcb8}, {0x44, 0xdcb8}},
               0x00, 0x00},
              {0x00, 0x10, {{0x41, 0xdcb8}, {0x42, 0xdcb8}, {0x43, 0xdcb8}, {0x44, 0xdcb8}},
               0x00, 0x00},
              {0x00, 0x48, {{0x41, 0xdcb8}, {0x42, 0xdcb8}, {0x43, 0xdcb8}, {0x44, 0xdcb8}},
               0x01, 0x00},
              {0x00, 0x58, {{0x43, 0xdcb8}, {0x44, 0xdcb8}, {0x41, 0xdcb8}, {0x42, 0xdcb8}},
               0x02, 0x00},
        }
};
printk(KERN_INFO "Copying IRQ routing tables...");
        memcpy((char            *)          RTABLE_DEST,          &intel_irq_routing_table,
intel_irq_routing_table.size);
printk(KERN_INFO "done.\n");
```

原来，在 BIOS 的代码中是将这个 PIRQ Table 写死的，然后 BIOS 将这个表拷贝到一个合适的地方，供 OS 使用。

笔者：感谢 bluesky_jxc 同学为我们带来 PIRQ Table 的知识。PIRQ Table 由微软提出，描述了从 OS 的观点，如何看待中断路由的问题，它并没有规定硬件厂商如何实现。PRIQ spec 和 MP spec 一起构成了早期 MP 平台中断几乎所有的内容，但往事如尘埃，俱往矣。ACPI 替代了它们。从长远的眼光看，我们在学习相关内容时，应该尽可能的多着眼于 ACPI，对于历史，了解即可。

关于 PIRQ table 的详细内容可参考：

http://www.microsoft.com/whdc/archive/pciirq.mspx

3.3 多 IOAPIC 情况下的中断路由

有没有操作系统不支持 APIC？有，某些嵌入式系统。但在很多年前，Windows95/98、Unix、Linux、Solaris and so on 都不支持。APIC 这样的高档货，总得有人用吧，Intel 虽然习惯为新技术推出一些实现范例，但还没闲到写一个支持 APIC 的操作系统供大家参考的程度。这时，微软走在了前面，Windows NT，以及基于 NT 架构的 Windows2000 率先支持 APIC 系统。那是一个 PIC、APIC 混存的时代，必须提供一个让不能使用 APIC 的 OS 回到 PIC 的途径，这就是中断路由。

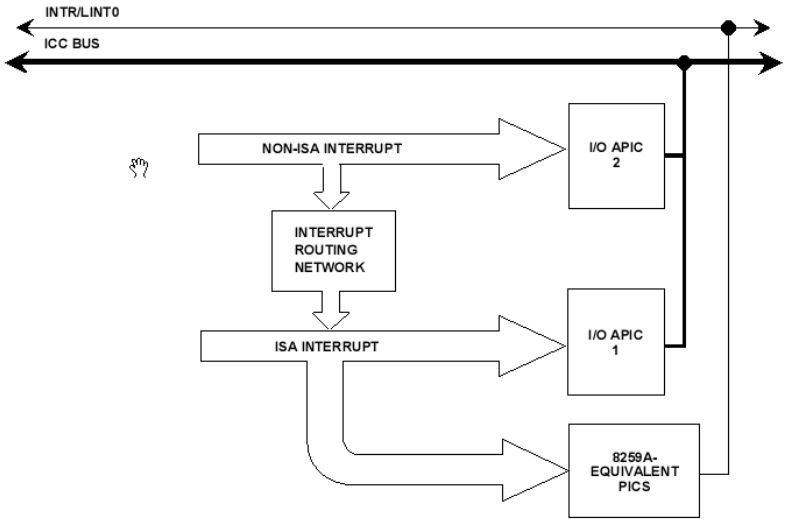


图 3-8 中断路由

笔者：MP spec 描述了这样一种情况：系统中有两个 IOAPIC1、IOAPIC2（如图 15）。IOAPIC1 接 ISA 设备，IOAPIC2 接 PCI 设备。其中硬盘使用 PCI controller，连接 IOAPIC2。根据 MP spec 对 PIC、APIC 共存情况的规定，系统启动首先进入 PIC mode 或 Virtual Wire mode，此时 IOAPIC2 不可用。若需要从硬盘读数据怎么办？路由，把 IOAPIC2 上的非 ISA 中断路由到 IOAPIC1 和 PIC 的 ISA 中断管脚来。这种情况是合理的，但会让人产生如下误解（当然，可能你不会误解，但俺着实误解了好久）：1）中断路由是为多 IOAPIC 系统引入的。当然不是，它出现的原因前面说了。2）IOAPIC2 的中断路由到 PIC 就行了，干嘛要接

到 IOAPIC1 上？思来想去也觉得这是个实现问题，或许硬件上有某些我们不知道的原因。

要详细了解中断路由，一篇好的文章是 Intel 公司 Joseph A.Bennett 所著的《Interrupt Routing Mechanism For Routing Interrupts From Peripheral Bus to Interrupt Controller》，此文讲述了如何构造中断路由网络。不过从一个软件程序员的角度来看，这篇文章的废话着实多了点~~~~~~~~~~~~~~

MP spec 规定了两种模式：Variable Routing 和 Fixing Routing。

3.3.1 Variable Routing

动态路由，或者称可控路由。使用这种方式，系统可以在进入 APIC 模式后，关闭路由网络，此时 PCI 中断就不在映射到 ISA 中断管脚上，而通过自己的 IOAPIC 发送中断。

在有 IMCR 的系统中，当软件通过 IMCR 切换到 APIC 模式时，路由网络被关闭。

对于系统中没有 IMCR，或者有一个（多个）IOAPIC 不在 IMCR 控制下时，由硬件保证进入 APIC 模式后路由网络关闭。这过程要求对系统软件（BIOS、OS）透明，无需它们做额外的工作。

笔者：spec 没有指明硬件如何在没有 IMCR 情况下关闭路由网络，这是个实现相关的问题。但 spec 又明确的暗示，硬件必须探测到操作系统对 IOAPIC 的编程动作，并关闭路由网络。故我们是不是可以认为，当我们开始操作 IOAPIC 时，路由网络就关闭了……《Interrupt Routing Mechanism For Routing Interrupts From Peripheral Bus to Interrupt Controller》一文中把 RTE 的 mask bit 引出来做非运算同 IRQ line 过或门的逻辑很有启发，对硬件感兴趣的朋友可以研究一下。

3.3.2 Fixing Routing:

这是一种老的路由方式，用于没有实现 Variable Routing 的系统。计算机中的“老方式”，显而易见的等同于“笨拙”、“不方便”、“万恶”……

在 Fixing Routing 的情况下，路由网络是无法从硬件上关闭的，只能通过软件的方法来避免 APIC 模式下的中断复接（Duplicate Interrupt）。

笔者：中断复接（Duplicate Interrupt）是指在多 IOAPIC 系统中，如图 15，非 ISA 中断路由到接 ISA 中断管脚的 IOAPIC，在路由网络无法关闭时，进入 APIC 模式后同一中断连接两个 IOAPIC 的情况。前面我们已经提出疑问了，只路由到 PIC 上不就好了 ……

BIOS 可以通过对 MP table 进行适当的配置来避免中断复接，根据硬件连接不同，又可以分为两种情况：

路由到的 ISA 管脚接有 ISA 设备:

这种情况下，BIOS 只为该 ISA 中断管脚设置 MP table entry，而不为 PCI 设备连接到的

IOAPIC 管脚设置 entry。简言之，即进入 APIC 模式后，PCI 设备连接到 IOAPIC2 的管脚不起作用，统一使用 ISA 中断管脚发送中断。

笔者：无论是 MP spec，还是《Interrupt Routing Mechanism For Routing Interrupts From Peripheral Bus to Interrupt Controller》，都透露出一个意思：中断路由并非每个 PCI 管脚路由到单独的 ISA 管脚，而是所有 PCI 管脚都路由到同一个 ISA 管脚。其结果是 PCI、ISA 设备共享中断，性能当然好不了（并且还有我们前面提到的 ISA 中断共享的问题）。当然，避免路由到不能共享中断的 ISA 管脚是硬件厂商的义务。

路由到的 ISA 管脚没有 ISA 设备:

这种情况下，BIOS 为 PCI 设备连接的 IOAPIC2 管脚设置 MP table entry，而不为路由到的 ISA 管脚设置 entry。和情况 1 相比，这在进入 APIC 模式后，就没有那么重的中断共享负担了。

题外话 —— Fixing Routing 对 OS 的暗示

一句话，OS 应该根据 MP table 报告的中断 entry 配置自己的中断系统，而不应该自己去探测 IOAPIC 的管脚，那些没有出现在 MP table 中的管脚都应该被 mask 住 …… 这是典型的硬件厂商思维，须知 OS 厂商从来都不相信 BIOS 厂商，它们经常为 BIOS 厂商背黑锅。几乎没有用户在遇到系统崩溃时会认为这是个 BIOS BUG …… 于是，微软说：要自由！ Intel 说：给你自由！ …… ACPI 诞生了 ……

笔者：MP spec 毕竟已经是过时的东西，微软有一篇文章为 ACPI 下的中断路由提供了范例。可惜俺不懂 ASL 语言，有兴趣的朋友可以参考：

《PCI IRQ Routing on a Multiprocessor ACPI System》

http://www.microsoft.com/taiwan/whdc/archive/acpi-mp.mspx

写在后面的话

如果你看到这里，非常感谢你忍受我通篇的唠叨。写文章太痛苦了，比看代码痛苦十倍！还好，写完了，都结束了，不想再写了 ……