|  |
| --- |
| 浅谈PCI Express体系结构--序 |
| 发布时间：2013-03-18 14:40:54 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

我以“浅谈”开始以下的文字，因为PCIe规范博大足使天下无书。正如大家所知，我之前写过一本《PCI Experss体系结构导读》，每次重温这本书籍时，总是遗憾着书中出现的错误。纸质的图书一旦出现问题，便很难有更改的余地。这一次我们不会再有这些遗憾。

|  |
| --- |
| 第I篇：PCI体系结构概述 |
| 发布时间：2013-03-18 14:45:18 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI(Peripheral Component Interconnect)总线的诞生与PC(Personal Computer)的蓬勃发展密切相关。在处理器体系结构中，PCI总线属于局部总线(Local Bus)。局部总线作为系统总线的延伸，主要功能是为了连接外部设备。

处理器主频的不断提升，要求速度更快，带宽更高的局部总线。起初PC使用8位的XT总线作为局部总线，并很快升级到16位的ISA(Industry Standard Architecture)总线，逐步发展到32位的EISA(Extended Industry Standard Architecture)、VESA(Video Electronics Standards Association)和MCA(Micro Channel Architecture)总线。

PCI总线规范在上世纪九十年代提出。这条总线推出之后，很快得到了各大主流半导体厂商的认同，迅速统一了当时并存的各类局部总线。EISA、VESA等其他32位总线很快就被PCI总线淘汰了。从那时起，PCI总线一直在处理器体系结构中占有重要地位。

在此后相当长的一段时间里，PC处理器系统的大多数外部设备都是直接或者间接地与PCI总线相连。即使目前PCI Express总线逐步取代了PCI总线成为PC局部总线的主流，也不能掩盖PCI总线的光芒。从软件层面上看，PCI Express总线与PCI总线基本兼容；从硬件层面上看，PCI Express总线在很大程度上继承了PCI总线的设计思路。因此PCI总线依然是软硬件工程师在进行处理器系统的开发与设计时，必须要掌握的一条局部总线。

PCI总线V1.0规范仅针对在一个PCB(Printed Circuit Board)环境内的，器件之间的互连，而1993年4月30日发布的V2.0规范增加了对PCI插槽的支持。1995年6月1日，PCI V2.1总线规范发布，这个规范具有里程碑意义。正是这个规范使得PCI总线大规模普及，至此PCI总线完成了对(E)ISA和MCA总线的替换。

至1996年，VESA总线也逐渐离开了人们的视线，当然PCI总线并不能完全提供显卡所需要的带宽，真正替代VESA总线的是AGP总线。随后PCISIG(PCI Special Interest Group)陆续发布了PCI总线V2.2，V2.3规范，并最终将PCI总线规范定格在V3.0。

除了PCI总线规范外，PCISIG还定义了一些与PCI总线相关的规范，如PCMCIA(Personal Computer Memory Card International Association)规范和MiniPCI规范。其中PCMCIA规范主要针对Laptop应用，后来PCMCIA升级为PC Card(Cardbus)规范，而PC Card又升级为ExpressCard规范。

PC Card规范基于32位，33MHz的PCI总线；而ExpressCard规范基于PCI Express和USB 2.0。这两个规范都在Laptop领域中获得了成功。除了PCMCIA规范外，Mini PCI总线也非常流行，与标准PCI插槽相比，Mini PCI插槽占用面积较小，适用于一些对尺寸有要求的应用。

除了以上规范之外，PCISIG还推出了一系列和PCI总线直接相关的规范。如PCI-to-PCI桥规范、PCI电源管理规范、PCI热插拔规范和CompactPCI总线规范。其中PCI-to-PCI桥规范最为重要，理解PCI-to-PCI桥是理解PCI体系结构的基础；而CompactPCI总线规范多用于具有背板结构的大型系统，并支持热拔插。

PCISIG在PCI总线规范的基础上，进一步提出PCI-X规范。与PCI总线相比，PCI-X总线规范可以支持133MHz、266MHz和533MHz的总线频率，并在传送规则上做了一些改动。虽然PCI-X总线还没有得到大规模普及就被PCI Express总线替代，但是在PCI-X总线中提出的许多设计思想仍然被PCI Express总线继承。

PCI总线规范是Intel对PC领域做出的一个巨大贡献。Intel也在PCI总线规范留下了深深的印记，PCI总线规范的许多内容都与基于IA (Intel Architecture)架构的x86处理器密切相关。但是这并不妨碍其他处理器系统使用PCI总线，事实上PCI总线在非x86处理器系统上也取得了巨大的成功。目前绝大多数处理器系统都使用PCI/PCI Express总线连接外部设备，特别是一些通用外设。

随着时间的推移，PCI和PCI-X总线逐步遇到瓶颈。PCI和PCI-X总线使用单端并行信号进行数据传递，由于单端信号容易被外部系统干扰，其总线频率很难进一步提高。目前，为了获得更高的总线频率以提高总线带宽，高速串行总线逐步替代了并行总线。PCI Express总线也逐渐替代PCI总线成为主流。但是从系统软件的角度上看，PCI Express总线仍然基于PCI总线。理解PCI Express总线的一个基础是深入理解PCI总线，同时PCI Express总线也继承了PCI总线的许多概念。本篇将详细介绍与处理器体系结构相关的，一些必备的PCI总线知识。

为简化起见，本篇主要介绍PCI总线的32位地址模式。在实际应用中，使用64位地址模式的PCI设备非常少。而且在PCI Express总线逐渐取代PCI总线的大趋势之下，将来也很难会有更多的，使用64位地址的PCI设备。如果读者需要掌握PCI总线的64位地址模式，请自行阅读PCI总线的相关规范。实际上，如果读者真正掌握了PCI总线的32位地址模式之后，理解64位地址模式并不困难。

为节省篇幅，下文将PCI Express总线简称为PCIe总线，PCI-to-PCI桥简称为PCI桥，PCI Express-to-PCI桥简称为PCIe桥，Host-to-PCI主桥简称为HOST主桥。值得注意的是许多书籍将HOST主桥称为PCI主桥或者PCI总线控制器。

|  |
| --- |
| **第1章 PCI总线的基本知识** |
| 发布时间：2013-03-19 09:47:32 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI总线作为处理器系统的局部总线，主要目的是为了连接外部设备，而不是作为处理器的系统总线连接Cache和主存储器。但是PCI总线、系统总线和处理器体系结构之间依然存在着紧密的联系。

PCI总线作为系统总线的延伸，其设计考虑了许多与处理器相关的内容，如处理器的Cache共享一致性和数据完整性，以及如何与处理器进行数据交换等一系列内容。其中Cache共享一致性和数据完整性是现代处理器局部总线的设计的重点和难点，也是本书将重点讲述的主题之一。

独立地研究PCI总线并不可取，因为PCI总线仅是处理器系统的一个组成部分。深入理解PCI总线需要了解一些与处理器体系结构相关的知识。这些知识是本书所侧重描述的，同时也是PCI总线规范所忽略的内容。脱离实际的处理器系统，不容易也不可能深入理解PCI总线规范。

对于今天的读者来说，PCI总线提出的许多概念略显过时，也有许多不足之处。但是在当年，PCI总线与之前的存在其他并行局部总线如ISA、EISA和MCA总线相比，具有许多突出的优点，是一个全新的设计。

(1) PCI总线空间与处理器空间隔离

PCI设备具有独立的地址空间，即PCI总线地址空间，该空间与存储器地址空间通过HOST主桥隔离。处理器需要通过HOST主桥才能访问PCI设备，而PCI设备需要通过HOST主桥才能主存储器。在HOST主桥中含有许多缓冲，这些缓冲使得处理器总线与PCI总线工作在各自的时钟频率中，彼此互不干扰。HOST主桥的存在也使得PCI设备和处理器可以方便地共享主存储器资源。

处理器访问PCI设备时，必须通过HOST主桥进行地址转换；而PCI设备访问主存储器时，也需要通过HOST主桥进行地址转换。HOST主桥的一个重要作用就是将处理器访问的存储器地址转换为PCI总线地址。PCI设备使用的地址空间是属于PCI总线域的，而与存储器地址空间不同。

x86处理器对PCI总线域与存储器域的划分并不明晰，这也使得许多程序员并没有准确地区分PCI总线域地址空间与存储器域地址空间。而本书将反复强调存储器地址和PCI总线地址的区别，因为这是理解PCI体系结构的重要内容。

PCI规范并没有对HOST主桥的设计进行约束。每一个处理器厂商使用的HOST主桥，其设计都不尽相同。HOST主桥是联系PCI总线与处理器的核心部件，掌握HOST主桥的实现机制是深入理解PCI体系结构的前提。

本书将以Freescale的PowerPC处理器和Intel的x86处理器为例，说明各自HOST主桥的实现方式，值得注意的是本书涉及的PowerPC处理器仅针对Freescale的PowerPC处理器，而不包含IBM和AMCC的Power和PowerPC处理器。而且如果没有特别说明，本书中涉及的x86处理器特指Intel的处理器，而不是其他厂商的x86处理器。

(2) 可扩展性

PCI总线具有很强的扩展性。在PCI总线中，HOST主桥可以直接推出一条PCI总线，这条总线也是该HOST主桥的所管理的第一条PCI总线，该总线还可以通过PCI桥扩展出一系列PCI总线，并以HOST主桥为根节点，形成1颗PCI总线树。这些PCI总线都可以连接PCI设备，但是在1颗PCI总线树上，最多只能挂接256个PCI设备(包括PCI桥)。

在同一条PCI总线上的设备间可以直接通信，并不会影响其他PCI总线上设备间的数据通信。隶属于同一颗PCI总线树上的PCI设备，也可以直接通信，但是需要通过PCI桥进行数据转发。

PCI桥是PCI总线的一个重要组成部件，该部件的存在使得PCI总线极具扩展性。PCI桥也是有别于其他局部总线的一个重要部件。在“以HOST主桥为根节点”的PCI总线树中，每一个PCI桥下也可以连接一个PCI总线子树，PCI桥下的PCI总线仍然可以使用PCI桥继续进行总线扩展。

PCI桥可以管理这个PCI总线子树，PCI桥的配置空间含有一系列管理PCI总线子树的配置寄存器。在PCI桥的两端，分别连接了两条总线，分别是上游总线(Primary Bus)和下游总线(Secondary Bus)。其中与处理器距离较近的总线被称为上游总线，另一条被称为下游总线。这两条总线间的通信需要通过PCI桥进行。PCI桥中的许多概念被PCIe总线采纳，理解PCI桥也是理解PCIe体系结构的基础。

(3) 动态配置机制

PCI设备使用的地址可以根据需要由系统软件动态分配。PCI总线使用这种方式合理地解决了设备间的地址冲突，从而实现了“即插即用”功能。从而PCI总线不需要使用ISA或者EISA接口卡为解决地址冲突而使用的硬件跳线。

每一个PCI设备都有独立的配置空间，在配置空间中含有该设备在PCI总线中使用的基地址，系统软件可以动态配置这个基地址，从而保证每一个PCI设备使用的物理地址并不相同。PCI桥的配置空间中含有其下PCI子树所能使用的地址范围。

(4) 总线带宽

PCI总线与之前的局部总线相比，极大提高了数据传送带宽，32位/33MHz的PCI总线可以提供132MB/s的峰值带宽，而64位/66MHz的PCI总线可以提供的峰值带宽为532MB/s。虽然PCI总线所能提供的峰值带宽远不能和PCIe总线相比，但是与之前的局部总线ISA、EISA和MCA总线相比，仍然具有较大的优势。

ISA总线的最高主频为8MHz，位宽为16，其峰值带宽为16MB/s；EISA总线的最高主频为8.33MHz，位宽为32，其峰值带宽为33MB/s；而MCA总线的最高主频为10MHz，最高位宽为32，其峰值带宽为40MB/s。PCI总线提供的峰值带宽远高于这些总线。

(5) 共享总线机制

PCI设备通过仲裁获得PCI总线的使用权后，才能进行数据传送，在PCI总线上进行数据传送，并不需要处理器进行干预。

PCI总线仲裁器不在PCI总线规范定义的范围内，也不一定是HOST主桥和PCI桥的一部分。虽然绝大多数HOST主桥和PCI桥都包含PCI总线仲裁器，但是在某些处理器系统的设计中也可以使用独立的PCI总线仲裁器。如在PowerPC处理器的HOST主桥中含有PCI总线仲裁器，但是用户可以关闭这个总线仲裁器，而使用独立的PCI总线仲裁器。

PCI设备使用共享总线方式进行数据传递，在同一条总线上，所有PCI设备共享同一总线带宽，这将极大地影响PCI总线的利用率。这种机制显然不如PCIe总线采用的交换结构，但是在PCI总线盛行的年代，半导体的工艺、设计能力和制作成本决定了采用共享总线方式是当时的最优选择。

(6) 中断机制

PCI总线上的设备可以通过四根中断请求信号INTA~D#向处理器提交中断请求。与ISA总线上的设备不同，PCI总线上的设备可以共享这些中断请求信号，不同的PCI设备可以将这些中断请求信号“线与”后，与中断控制器的中断请求引脚连接。PCI设备的配置空间记录了该设备使用这四根中断请求信号的信息。

PCI总线进一步提出了MSI(Message Signal Interrupt)机制，该机制使用存储器写总线事务传递中断请求，并可以使用x86处理器FSB(Front Side Bus)总线提供的Interrupt Message总线事务，从而提高了PCI设备的中断请求效率。

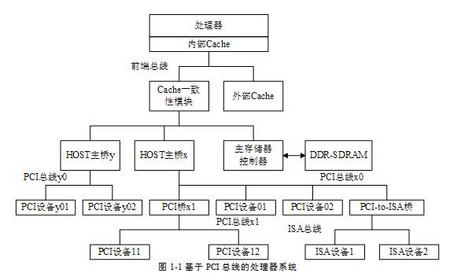
虽然从现代总线技术的角度上看，PCI总线仍有许多不足之处，但也不能否认PCI总线已经获得了巨大的成功，不仅x86处理器将PCI总线作为标准的局部总线连接各类外部设备，PowerPC、MIPS和ARM[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011445.HTM" \l "_ftn1" \o ")处理器也将PCI总线作为标准局部总线。除此之外，基于PCI总线的外部设备，如以太网控制器、声卡、硬盘控制器等，也已经成为主流。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011445.HTM" \l "_ftnref1" \o ") 在ARM处理器中，使用SoC平台总线，即AMBA总线，连接片内设备。但是某些ARM生产厂商，依然使用AMBA-to-PCI桥推出PCI总线，以连接PCI设备。

|  |
| --- |
| **1.1 PCI总线的组成结构** |
| 发布时间：2013-03-20 09:02:28 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

如上文所述，PCI总线作为处理器系统的局部总线，是处理器系统的一个组成部件，讲述PCI总线的组成结构不能离开处理器系统这个大环境。在一个处理器系统中，与PCI总线相关的模块如图1‑1所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130320090350635.jpg)

如图1‑1所示在一个处理器系统中，与PCI总线相关的模块包括，HOST主桥、PCI总线、PCI桥和PCI设备。PCI总线由HOST主桥和PCI桥推出，HOST主桥与主存储器控制器在同一级总线上，PCI设备可以方便地通过HOST主桥访问主存储器，即进行DMA操作。

值得注意的是，PCI设备的DMA操作需要与处理器系统的Cache进行一致性操作，当PCI设备通过HOST主桥访问主存储器时，Cache一致性模块将进行地址监听，并根据监听的结果改变Cache的状态。

在一些简单的处理器系统中，可能不含有PCI桥，此时所有PCI设备都是连接在HOST主桥推出的PCI总线上，此外在一些处理器系统中可能含有多个HOST主桥，如在图1‑1所示的处理器系统中含有HOST主桥x和HOST主桥Y。

**1.1.1 HOST主桥**

HOST主桥是一个很特别的桥片，其主要功能是隔离处理器系统的存储器域与处理器系统的PCI总线域，管理PCI总线域，并完成处理器与PCI设备间的数据交换。处理器与PCI设备间的数据交换主要由“处理器访问PCI设备的地址空间”和“PCI设备使用DMA机制访问主存储器”这两部分组成。

为简便起见，下文将处理器系统的存储器域简称为存储器域，而将处理器系统的PCI总线域称为PCI总线域，存储器域和PCI总线域的详细介绍见第2.1节。值得注意的是，在一个处理器系统中，有几个HOST主桥，就有几个PCI总线域。

HOST主桥在处理器系统中的位置并不相同，如PowerPC处理器将HOST主桥与处理器集成在一个芯片中。而有些处理器不进行这种集成，如x86处理器使用南北桥结构，处理器内核在一个芯片中，而HOST主桥在北桥中。但是从处理器体系结构的角度上看，这些集成方式并不重要。

PCI设备通过HOST主桥访问主存储器时，需要与处理器的Cache进行一致性操作，因此在设计HOST主桥时需要重点考虑Cache一致性操作。在HOST主桥中，还含有许多数据缓冲，以支持PCI总线的预读机制。

HOST主桥是联系处理器与PCI设备的桥梁。在一个处理器系统中，每一个HOST主桥都管理了一颗PCI总线树，在同一颗PCI总线树上的所有PCI设备属于同一个PCI总线域。如图1‑1所示，HOST主桥x之下的PCI设备属于PCI总线x域，而HOST主桥y之下的PCI设备属于PCI总线y域。在这颗总线树上的所有PCI设备的配置空间都由HOST主桥通过配置读写总线周期访问。

如果HOST主桥支持PCI V3.0规范的Peer-to-Peer数据传送方式，那么分属不同PCI总线域的PCI设备可以直接进行数据交换。如图1‑1所示，如果HOST主桥y支持Peer-to-Peer数据传送方式，PCI设备y01可以直接访问PCI设备01或者PCI设备11，而不需要通过处理器的参与。但是这种跨越总线域的数据传送方式在PC架构中并不常用，在PC架构中，重点考虑的是PCI设备与主存储器之间的数据交换，而不是PCI设备之间的数据交换。此外在PC架构中，具有两个HOST主桥的处理器系统也并不多见。

在PowerPC处理器中，HOST主桥可以通过设置Inbound寄存器，使得分属于不同PCI总线域的设备可以直接通信。许多PowerPC处理器都具有多个HOST主桥，有关PowerPC处理器使用的HOST主桥详见第2.2节。

**1.1.2 PCI总线**

在处理器系统中，含有PCI总线和PCI总线树这两个概念。这两个概念并不相同，在一颗PCI总线树中可能具有多条PCI总线，而具有血缘关系的PCI总线组成一颗PCI总线树。如在图1‑1所示的处理器系统中，PCI总线x树具有两条PCI总线，分别为PCI总线x0和PCI总线x1。而PCI总线y树中仅有一条PCI总线。

PCI总线由HOST主桥或者PCI桥管理，用来连接各类设备，如声卡、网卡和IDE接口卡等。在一个处理器系统中，可以通过PCI桥扩展PCI总线，并形成具有血缘关系的多级PCI总线，从而形成PCI总线树型结构。在处理器系统中有几个HOST主桥，就有几颗这样的PCI总线树，而每一颗PCI总线树都与一个PCI总线域对应。

与HOST主桥直接连接的PCI总线通常被命名为PCI总线0。考虑到在一个处理器系统中可能有多个主桥，图1‑1将HOST主桥x推出的PCI总线命名为x0总线，而将PCI桥x1扩展出的PCI总线称之为x1总线；而将HOST主桥y推出的PCI总线称为y0~yn。分属不同PCI总线树的设备，其使用的PCI总线地址空间分属于不同的PCI总线域空间。

**1.1.3 PCI设备**

在PCI总线中有三类设备，PCI主设备、PCI从设备和桥设备。其中PCI从设备只能被动地接收来自HOST主桥，或者其他PCI设备的读写请求；而PCI主设备可以通过总线仲裁获得PCI总线的使用权，主动地向其他PCI设备或者主存储器发起存储器读写请求。而桥设备的主要作用是管理下游的PCI总线，并转发上下游总线之间的总线事务。

一个PCI设备可以即是主设备也是从设备，但是在同一个时刻，这个PCI设备或者为主设备或者为从设备。PCI总线规范将PCI主从设备统称为PCI Agent设备。在处理器系统中常见的PCI网卡、显卡、声卡等设备都属于PCI Agent设备。

在PCI总线中，HOST主桥是一个特殊的PCI设备，该设备可以获取PCI总线的控制权访问PCI设备，也可以被PCI设备访问。但是HOST主桥并不是PCI设备。PCI规范也没有规定如何设计HOST主桥。

在PCI总线中，还有一类特殊的设备，即桥设备。桥设备包括PCI桥、PCI-to-(E)ISA桥和PCI-to-Cardbus桥。本篇重点介绍PCI桥，而不关心其他桥设备的实现原理。PCI桥的存在使PCI总线极具扩展性，处理器系统可以使用PCI桥进一步扩展PCI总线。

PCI桥的出现使得采用PCI总线进行大规模系统互连成为可能。但是在目前已经实现的大规模处理器系统中，并没有使用PCI总线进行处理器系统与处理器系统之间的大规模互连。因为PCI总线是一个以HOST主桥为根的树型结构，使用主从架构，因而不易实现多处理器系统间的对等互连。

即便如此PCI桥仍然是PCI总线规范的精华所在，掌握PCI桥是深入理解PCI体系结构的基础。PCI桥可以连接两条PCI总线，上游PCI总线和下游PCI总线，这两个PCI总线属于同一个PCI总线域，使用PCI桥扩展的所有PCI总线都同属于一个PCI总线域。

其中对PCI设备配置空间的访问可以从上游总线转发到下游总线，而数据传送可以双方向进行。在PCI总线中，还存在一种非透明PCI桥，该桥片不是PCI总线规范定义的标准桥片，但是适用于某些特殊应用，本篇将在第2.5节中详细介绍这种桥片。在本书中，如不特别强调，PCI桥是指透明桥，透明桥也是PCI总线规范定义的标准桥片。

PCI-to-(E)ISA桥和PCI-to-Cardbus桥的主要作用是通过PCI总线扩展(E)ISA和Cardbus总线。在PCI总线推出之后，(E)ISA总线并没有在处理器系统中立即消失，此时需要使用PCI-(E)ISA桥扩展(E)ISA总线，而使用PCI-to-Cardbus桥用来扩展Cardbus总线，本篇并不关心(E)ISA和Cardbus总线的设计与实现。

**1.1.4 HOST处理器**

PCI总线规定在同一时刻内，在一颗PCI总线树上有且只有一个HOST处理器。这个HOST处理器可以通过HOST主桥，发起PCI总线的配置请求总线事务，并对PCI总线上的设备和桥片进行配置。

在PCI总线中，HOST处理器是一个较为模糊的概念。在SMP(symmetric multiprocessing)处理器系统中，所有CPU都可以通过HOST主桥访问其下的PCI总线树，这些CPU都可以作为HOST处理器。但是值得注意的是，HOST主桥才是PCI总线树的实际管理者，而不是HOST处理器。

在HOST主桥中，设置了许多寄存器，HOST处理器通过操作这些寄存器管理这些PCI设备。如在x86处理器的HOST主桥中设置了0xCF8和0xCFC这两个I/O端口访问PCI设备的配置空间，而PowerPC处理器的HOST主桥设置了CFG\_ADDR和CFG\_DATA寄存器访问PCI设备的配置空间。值得注意的是，在PowerPC处理器中并没有I/O端口，因此使用存储器映像寻址方式访问外部设备的寄存器空间。

**1.1.5 PCI总线的负载**

PCI总线的所能挂接的负载与总线频率相关，其中总线频率越高，所能挂接的负载越少。下文以 PCI总线和PCI-X总线为例说明总线频率、峰值带宽和负载能力之间的关系，如表1‑1所示。

表1‑1 PCI总线频率、带宽与负载之间的关系

|  |  |  |  |
| --- | --- | --- | --- |
| 总线类型 | 总线频率 | 峰值带宽 | 负载能力 |
| PCI | 33MHz | 133MB/s | 4-5个插槽 |
| 66MHz | 266MB/s | 1-2个插槽 |
| PCI-X | 66MHz | 266MB/s | 4个插槽 |
| 133MHz | 533MB/s | 2个插槽 |
| 266MHz | 1066MB/s | 1个插槽 |
| 533MHz | 2131MB/s | 1个插槽 |

由表1‑1所示，PCI总线频率越高，所能挂接的负载越少，但是整条总线所能提供的带宽越大。值得注意的是，PCI-X总线与PCI总线的传送协议略有不同，因此66MHz的PCI-X总线的负载数较大，PCI-X总线的详细说明见第1.5节。当PCI-X总线频率为266MHz和533MHz时，该总线只能挂接一个PCI-X插槽。在PCI总线中，一个插槽相当于两个负载，接插件和插卡各算为一个负载，在表1‑1中，33MHz的PCI总线可以挂接4~5个插槽，相当于直接挂接8~10个负载。

|  |
| --- |
| **1.2 PCI总线的信号定义** |
| 发布时间：2013-03-20 09:30:37 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI总线是一条共享总线，在一条PCI总线上可以挂接多个PCI设备。这些PCI设备通过一系列信号与PCI总线相连，这些信号由地址/数据信号、控制信号、仲裁信号、中断信号等多种信号组成。

PCI总线是一个同步总线，每一个设备都具有一个CLK信号，其发送设备与接收设备使用这个CLK信号进行同步数据传递。PCI总线可以使用33MHz或者66MHz的时钟频率，而PCI-X总线可以使用133MHz、266MHz或者533MHz的时钟频率。

除了RST#、INTA~D#、PME#和CLKRUN#等信号之外，PCI设备使用的绝大多数信号需要与CLK信号同步。其中RST#是复位信号，PCI设备使用INTA~D#信号进行中断请求。本篇并不关心PME#和CLKRUN#信号。

**1.2.1 地址和数据信号**

在PCI总线中，与地址和数据相关的信号如下所示。

(1) AD[31:0]信号

PCI总线复用地址与数据信号。PCI总线事务在启动后的第一个时钟周期传送地址，这个地址是PCI总线域的存储器地址或者I/O地址；而在下一个时钟周期传送数据[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011481.HTM#_ftn1)。传送地址的时钟周期也被称为地址周期，而传送数据的时钟周期也被称为数据周期。PCI总线支持突发传送，即在一个地址周期之后，可以紧跟多个数据周期。

(2) PAR信号

PCI总线使用奇偶校验机制，保证地址和数据信号在进行数据传递时的正确性。PAR信号是AD[31:0]和C/BE[3:0]的奇偶校验信号。PCI主设备在地址周期和数据周期中，使用该信号为地址和数据信号线提供奇偶校验位。

(3) C/BE[3:0]#信号

PCI总线复用命令与字节选通引脚。在地址周期中，C/BE[3:0]信号表示PCI总线的命令。而在数据周期，C/BE[3:0]引脚输出字节选通信号，其中C/BE3、C/BE2、C/BE1和C/BE0与数据的字节3、2、1和0对应。使用这组信号可以对PCI设备进行单个字节、字和双字访问。PCI总线通过C/BE[3:0]#信号定义了多个总线事务，这些总线事务如表1‑2所示。

表1‑2 PCI总线事务

|  |  |  |
| --- | --- | --- |
| C/BE[3:0]# | 命令类型 | 说明 |
| 0000 | Interrupt Acknowledge | 中断响应总线事务读取当前挂接在PCI总线上的中断控制器的中断向量号。目前大多数处理器系统的中断控制器都不挂接在PCI总线上，因此这种总线事务很少被使用。 |
| 0001 | Special Cycle | HOST主桥可以使用Special Cycle事务在PCI总线上，进行信息广播。 |
| 0010 | I/O Read | HOST主桥可以使用该总线事务对PCI设备的I/O地址空间进行读操作。目前多数PCI设备都不支持I/O地址空间，而仅支持存储器地址空间，但是仍有部分PCI设备同时包含I/O地址空间和存储器地址空间。 |
| 0011 | I/O Write | 对PCI总线的I/O地址空间进行写操作。 |
| 0100 | Reserved | 保留 |
| 0101 | Reserved | 保留 |
| 0110 | Memory Read | HOST主桥可以使用该总线事务对PCI设备的存储器空间进行读操作。PCI设备也可以使用该总线事务读取处理器的存储器空间。 |
| 0111 | Memory Write | HOST主桥可以使用该总线事务对PCI设备的存储器空间进行写操作。PCI设备也可以使用该总线事务向处理器的存储器空间进行写操作。 |
| 1000 | Reserved | 保留 |
| 1001 | Reserved | 保留 |
| 1010 | Configuration Read | HOST主桥可以对PCI设备的配置空间进行读操作。每一个PCI设备都有独立的配置空间。在多功能PCI设备中，每一个子设备(Function)也有一个独立的配置空间。该总线事务只能由HOST主桥发出，PCI桥可以转发该总线事务。 |
| 1011 | Configuration Write | HOST主桥对PCI设备的配置空间进行写操作。 |
| 1100 | Memory Read Multiple | HOST主桥可以使用该总线事务对PCI设备的存储器空间进行多行读操作，这种操作并不多见。该总线事务的主要用途是供PCI设备使用，读取主存储器。这个读操作与Memory Read操作(C/BE[3:0]为0x0110时)略有不同，详见第3.4.5节。 |
| 1101 | Dual Address Cycle | PCI总线支持64位地址，处理器或者其他PCI设备访问64位PCI总线地址时，必须使用双地址周期产生64位的PCI总线地址。PCI设备使用DMA读写方式访问64位的存储器地址时，也可以使用该总线事务。 |
| 1110 | Memory Read Line | HOST主桥可以使用该总线事务对PCI设备的存储器空间进行单行读操作，这种操作并不多见。该总线事务的主要用途是供PCI设备使用，读取主存储器。详见第3.4.5节。 |
| 1111 | Memory Write and Invalidate | 存储器写并无效操作，与存储器写不同，PCI设备可以使用该总线事务对主存储器空间进行写操作。该总线事务将数据写入主存储器的同时，将对应Cache行中的数据“使无效”，详见第3.3.4节。 |

**1.2.2 接口控制信号**

在PCI总线中，接口控制信号的主要作用是保证数据的正常传递，并根据PCI主从设备的状态，暂停、终止或者正常完成当前总线事务，其主要信号如下。

(1) FRAME#信号

该信号指示一个PCI总线事务的开始与结束。当PCI设备获得总线的使用权后，将置该信号有效，即置为低，启动PCI总线事务，当结束总线事务时，将置该信号无效，即置为高。PCI设备(HOST主桥)只有通过仲裁获得当前PCI总线的使用权后，才能驱动该信号。

(2) IRDY#信号

该信号由PCI主设备(包括HOST主桥)驱动，该信号有效时表示PCI主设备的数据已经准备完毕。如果当前PCI总线事务为写事务，表示数据已经在AD[31:0]上有效；如果为读事务，表示PCI目标设备已经准备好接收缓冲，目标设备可以将数据发送到AD[31:0]上。

(3) TRDY#信号

该信号由目标设备驱动，该信号有效时表示目标设备已经将数据准备完毕。如果当前PCI总线事务为写事务，表示目标设备已经准备好接收缓冲，可以将AD[31:0]上的数据写入目标设备；如果为读事务，表示PCI设备需要的数据已经在AD[31:0]上有效。

该信号可以和IRDY#信号联合使用，在PCI总线事务上插入等待周期，对PCI总线的数据传送进行控制。

(4) STOP#信号

该信号有效时表示目标设备请求主设备停止当前PCI总线事务。一个PCI总线事务除了可以正常结束外，目标设备还可以使用该信号终止当前PCI总线事务。目标设备可以根据不同的情况，要求主设备对当前PCI总线事务进行重试(Retry)、断连(Disconnect)，也可以向主设备报告目标设备夭折(Target Abort)。

目标设备要求主设备Retry和Disconnect并不意味着当前PCI总线事务出现错误。当目标设备没有将数据准备好时，可以使用Retry周期使主设备重试当前PCI总线事务。有时目标设备不能接收来自主设备较长的Burst操作时，可以使用Disconnect周期，将一个较长的Burst操作，分解为多个Burst操作。当主设备访问的地址越界时，目标设备可以使用Disconnect周期，终止主设备的越界访问。

而Target Abort表示在数据传送中出现错误。处理器系统必须要对这种情况进行处理。在PCI总线中，出现Abort一般意味着当前PCI总线域出现了较为严重的错误。

(5) IDSEL信号

PCI总线在进行配置读写总线事务时，使用该信号选择PCI目标设备。配置读写总线事务与存储器读写总线事务在实现上略有不同。在PCI总线中，存储器读写总线事务使用地址译码方式访问外部设备。而配置读写总线事务使用“ID译码方式”访问PCI设备，即通过PCI设备的总线号、设备号和寄存器号访问PCI设备的配置空间。

IDSEL信号与PCI设备的设备号相关，相当于PCI设备配置空间的片选信号，这部分内容将在第2.4.4节中详细介绍。

(6) DEVSEL#信号

该信号有效时表示PCI总线的目标设备准备好，该信号与TRDY#信号不同之处在于该信号有效仅表示目标设备已经完成了地址译码。目标设备使用该信号通知PCI主设备，其访问对象在当前PCI总线上，但是并不表示目标设备可以与主设备进行数据交换。而TRDY#信号表示数据有效，PCI主设备可以向目标设备写入或者从目标设备读取数据。

PCI总线规范根据设备进行译码速度的快慢，将PCI设备分为快速、中速和慢速三种设备。在PCI总线上还有一种特殊的设备，即负向译码设备，在一条PCI总线上当快速、中速和慢速三种设备都不能响应PCI总线事务的地址时，负向译码设备将被动地接收这个PCI总线事务。如果在PCI主设备访问的PCI总线上，没有任何设备可以置DEVSEL#信号为有效，主设备将使用Master Abort周期结束当前总线事务。

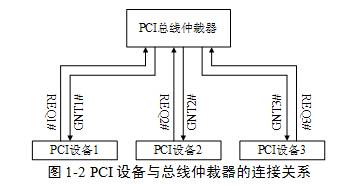
(7) LOCK#信号

PCI主设备可以使用该信号，将目标设备的某个存储器或者I/O资源锁定，以禁止其他PCI主设备访问此资源，直到锁定这个资源的主设备将其释放。PCI总线使用LOCK#信号实现LOCK总线事务，只有HOST主桥、PCI桥或者其他桥片可以使用LOCK#信号。在PCI总线的早期版本中，PCI Agent设备也可以使用LOCK#信号，而目前PCI总线使用LOCK#信号仅是为防止死锁和向前兼容。LOCK总线事务将严重影响PCI总线的传送效率，在实际应用中，设计者应当尽量避免使用该总线事务。

**1.2.3 仲裁信号**

PCI设备使用该组信号进行总线仲裁，并获得PCI总线的使用权。只有PCI主设备需要使用该组信号，而PCI从设备可以不使用总线仲裁信号。这组信号由REQ#和GNT#组成。其中PCI主设备的REQ#和GNT#信号与PCI总线的仲裁器直接相连。

PCI主设备的总线仲裁信号与PCI总线仲裁器的连接关系如图1‑2所示。值得注意的是，每一个PCI主设备都具有独立的总线仲裁信号，并与PCI总线仲裁器一一相连。而总线仲裁器需要保证在同一个时间段内，只有一个PCI设备可以使用当前总线。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130320092722785.jpg)

在一个处理器系统中，一条PCI总线可以挂接PCI主设备的数目，除了与负载能力相关之外，还与PCI总线仲裁器能够提供的仲裁信号数目直接相关。

在一颗PCI总线树中，每一条PCI总线上都有一个总线仲裁器。一个处理器系统可以使用PCI桥扩展出一条新的PCI总线，这条新的PCI总线也需要一个总线仲裁器，通常在PCI桥中集成了这个总线仲裁器。多数HOST主桥也集成了一个PCI总线仲裁器，但是PCI总线也可以使用独立的PCI总线仲裁器。

PCI主设备使用PCI总线进行数据传递时，需要首先置REQ#信号有效，向PCI总线仲裁器发出总线申请，当PCI总线仲裁器允许PCI主设备获得PCI总线的使用权后，将置GNT#信号为有效，并将其发送给指定的PCI主设备。而PCI主设备在获得总线使用权之后，将可以置FRAME#信号有效，与PCI从设备进行数据通信。

**1.2.4 中断请求等其他信号**

PCI总线提供了INTA#、INTB#、INTC#和INTD#四个中断请求信号，PCI设备借助这些中断请求信号，使用电平触发方式向处理器提交中断请求。当这些中断请求信号为低时，PCI设备将向处理器提交中断请求；当处理器执行中断服务程序清除PCI设备的中断请求后，PCI设备将该信号置高[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011481.HTM" \l "_ftn2" \o ")，结束当前中断请求。

PCI总线规定单功能设备只能使用INTA#信号，而多功能设备才能使用INTB#/C#/D#信号。PCI设备的这些中断请求信号可以通过某种规则进行线与，之后与中断控制器的中断请求信号线相连。而处理器系统需要预先知道这个规则，以便正确处理来自不同PCI设备的中断请求，这个规则也被称为中断路由表，有关中断路由表的详细描述见第1.4.2节。

PCI总线在进行数据传递过程时，难免会出现各种各样的错误，因此PCI总线提供了一些错误信号，如PERR#和SERR#信号。其中当PERR#信号有效时，表示数据传送过程中出现奇偶校验错(Special Cycle周期除外)；而当SERR#信号有效时，表示当前处理器系统出现了三种错误可能，分别为地址奇偶校验错，在Special Cycle周期中出现数据奇偶校验错，或者系统出现其他严重错误。

如果PCI总线支持64位模式，还需要提供AD[63:32]、C/BE[7:4]、REQ64、ACK64和PAR64这些信号。此外PCI总线还有一些与JTAG、SMBCLK以及66MHz使能等信号，本章并不关心这些信号。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011481.HTM#_ftnref1) 双地址周期在第一、二个时钟周期，都传送地址。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011481.HTM" \l "_ftnref2" \o ") INTx#这组信号为开漏输出，当所有的驱动源不驱动该信号时，该信号由上拉电阻驱动为高。

|  |
| --- |
| **1.3 PCI总线的存储器读写总线事务** |
| 发布时间：2013-03-21 08:39:31 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

总线的基本任务是实现数据传送，将一组数据从一个设备传送到另一个设备，当然总线也可以将一个设备的数据广播到多个设备。在处理器系统中，这些数据传送都要依赖一定的规则，PCI总线并不例外。

PCI总线使用单端并行数据线，采用地址译码方式进行数据传递，而采用ID译码方式进行配置信息的传递。其中地址译码方式使用地址信号，而ID译码方式使用PCI设备的ID号，包括Bus Number、Device Number、Function Number和Register Number。下文将以图1‑1中的处理器系统为例，简要介绍PCI总线支持的总线事务及其传送方式。

如表1‑2所示，PCI总线支持多种总线事务。而本节重点介绍存储器读写总线事务，I/O读写总线事务，并在第2.4节详细介绍配置读写总线事务。值得注意的是，PCI设备只有在系统软件初始化配置空间之后，才能够被其他主设备访问。

当PCI设备的配置空间被初始化之后，该设备在当前的PCI总线树上将拥有一个独立的PCI总线地址空间，即BAR((Base Address Register)寄存器所描述的空间，有关BAR寄存器的详细说明见第2.3.2节。

处理器与PCI设备进行数据交换，或者PCI设备之间进行存储器数据交换时，都将通过PCI总线地址完成。而PCI设备与主存储器进行DMA操作时，使用的也是PCI总线域的地址，而不是存储器域的地址，此时HOST主桥将完成PCI总线地址到存储器域地址的转换，不同的HOST主桥进行地址转换时使用的方法并不相同。

PCI总线的配置读写总线事务与HOST主桥与PCI桥相关，因此读者需要了解HOST主桥和PCI桥的详细实现机制之后，才能深入理解这部分内容。本篇将在第2.4节详细介绍这些内容。在下文中，我们假定所使用的PCI设备的配置空间已经被系统软件初始化。

PCI总线支持以下几类存储器读写总线事务。

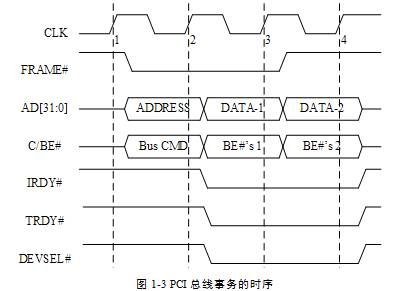
(1) HOST处理器对PCI设备的BAR空间进行数据读写，BAR空间可以使用存储器或者I/O译码方式。HOST处理器使用PCI总线的存储器读写总线事务和I/O读写总线事务访问PCI设备的BAR空间。

(2) PCI设备之间的数据传递。在PCI总线上的两个设备可以直接通信，如一个PCI设备可以访问另外一个设备的BAR空间。不过这种数据传递在PC处理器系统中，较少使用。

(3) PCI设备对主存储器进行读写，即DMA读写操作。DMA读写操作在所有处理器系统中都较为常用，也是PCI总线数据传送的重点所在。在多数情况下，DMA读写操作结束后将伴随着中断的产生。PCI设备可以使用INTA#、INTB#、INTC#和INTD#信号提交中断请求，也可以使用MSI机制提交中断请求。

**1.3.1 PCI总线事务的时序**

PCI总线使用第1.2节所述的信号进行数据和配置信息的传递，一个PCI总线事务的基本访问时序如图1‑3所示，与PCI总线事务相关的控制信号有FRAME#、IRDY#、TRDY#、DEVSEL#等其他信号。

[](http://download.bbs.ednchina.com/images/attachments/201303/original/466978_TIME_1363827055579.jpg)

当一个PCI主设备需要使用PCI总线时，需要首先发送REQ#信号，通过总线仲裁获得总线使用权，即GNT#信号有效后，使用以下步骤完成一个完整PCI总线事务，对目标设备进行存储器或者I/O地址空间的读写访问。

(1) 当PCI主设备获得总线使用权之后，将在CLK1的上升沿置FRAME#信号有效，启动PCI总线事务。当PCI总线事务结束后，FRAME#信号将被置为无效。

(2) PCI总线周期的第一个时钟周期(CLK1的上升沿到CLK2的上升沿之间)为地址周期。在地址周期中，PCI主设备将访问的目的地址和总线命令分别驱动到AD[31:0]和C/BE#信号上。如果当前总线命令是配置读写，那么IDSEL信号线也被置为有效，IDSEL信号与PCI总线的AD[31:11]相连，详见第2.4.4节。

(3) 当IRDY#、TRDY#和DEVSEL#信号都有效后，总线事务将使用数据周期，进行数据传递。当IRDY#和TRDY#信号没有同时有效时，PCI总线不能进行数据传递，PCI总线使用这两个信号进行传送控制。

(4) PCI总线支持突发周期，因此在地址周期之后可以有多个数据周期，可以传送多组数据。而目标设备并不知道突发周期的长度，如果目标设备不能继续接收数据时，可以disconnect当前总线事务。值得注意的是，只有存储器读写总线事务可以使用突发周期。

一个完整的PCI总线事务远比上述过程复杂的多，因为PCI总线还支持许多传送方式，如双地址周期、fast back-to-back(快速背靠背)、插入等待状态、重试和断连、总线上的错误处理等一系列总线事务。本篇不一一介绍这些传送方式。

**1.3.2 Posted和Non-Posted传送方式**

PCI总线规定了两类数据传送方式，分别是Posted和Non-Posted数据传送方式。其中使用Posted数据传送方式的总线事务也被称为Posted总线事务；而使用Non-Posted数据传送方式的总线事务也被称为Non-Posted总线事务。

其中Posted总线事务指PCI主设备向PCI目标设备进行数据传递时，当数据到达PCI桥后，即由PCI桥接管来自上游总线的总线事务，并将其转发到下游总线。采用这种数据传送方式，在数据还没有到达最终的目的地之前，PCI总线就可以结束当前总线事务，从而在一定程度上解决了PCI总线的拥塞。

而Non-Posted总线事务是指PCI主设备向PCI目标设备进行数据传递时，数据必须到达最终目的地之后，才能结束当前总线事务的一种数据传递方式。

显然采用Posted传送方式，当这个Posted总线事务通过某条PCI总线后，就可以释放PCI总线的资源；而采用Non-Posted传送方式，PCI总线在没有结束当前总线事务时必须等待。这种等待将严重阻塞当前PCI总线上的其他数据传送，因此PCI总线使用Delayed总线事务处理Non-Posted数据请求，使用Delayed总线事务可以相对缓解PCI总线的拥塞。Delayed总线事务的详细介绍见第1.3.5节。

PCI总线规定只有存储器写请求(包括存储器写并无效请求)可以采用Posted总线事务，下文将Posted存储器写请求简称为PMW(Posted Memory Write)，而存储器读请求、I/O读写请求、配置读写请求只能采用Non-Posted总线事务。

下文以图1‑1的处理器系统中的PCI设备11向存储器进行DMA写操作为例，说明Posted传送方式的实现过程。PCI设备11进行DMA写操作时使用存储器写总线事务，当PCI设备11获得PCI总线x1的使用权后，将发送存储器写总线事务到PCI总线x1。当PCI桥1发现这个总线事务的地址不在该桥管理的地址范围内将首先接收这个总线事务，并结束PCI总线x1的总线事务。

此时PCI总线x1使用的资源已被释放，PCI设备11和PCI设备12可以使用PCI总线x1进行通信。PCI桥1获得PCI总线x0的使用权后，将转发这个存储器写总线事务到PCI总线x0，之后HOST主桥x将接收这个存储器写总线事务，并最终将数据写入主存储器。

由以上过程可以发现，Posted数据请求在通过PCI总线之后，将逐级释放总线资源，因此PCI总线的利用率较高。而使用Non-Posted方式进行数据传送的处理过程与此不同，Non-Posted数据请求在通过PCI总线时，并不会及时释放总线资源，从而在某种程度上影响PCI总线的使用效率和传送带宽。

**1.3.3 HOST处理器访问PCI设备**

HOST处理器对PCI设备的数据访问主要包含两方面内容，一方面是处理器向PCI设备发起存储器和I/O读写请求；另一方面是处理器对PCI设备进行配置读写。

在PCI设备的配置空间中，共有6个BAR寄存器。每一个BAR寄存器都与PCI设备使用的一组PCI总线地址空间对应，BAR寄存器记录这组地址空间的基地址。本书将与BAR寄存器对应的PCI总线地址空间称为BAR空间，在BAR空间中可以存放I/O地址空间，也可以存放存储器地址空间。

PCI设备可以根据需要，有选择地使用这些BAR空间。值得注意的是，在BAR寄存器中存放的是PCI设备使用的“PCI总线域”的物理地址，而不是“存储器域”的物理地址，有关BAR寄存器的详细介绍见第2.3.2节。

HOST处理器访问PCI设备I/O地址空间的过程，与访问存储器地址空间略有不同。有些处理器，如x86处理器，具有独立的I/O地址空间。x86处理器可以将PCI设备使用的I/O地址映射到存储器域的I/O地址空间中，之后处理器可以使用IN，OUT等指令对存储器域的I/O地址进行访问，然后通过HOST主桥将存储器域的I/O地址转换为PCI总线域的I/O地址，最后使用PCI总线的I/O总线事务对PCI设备的I/O地址进行读写访问。在x86处理器中，存储器域的I/O地址与PCI总线域的I/O地址相同。

对于有些没有独立I/O地址空间的处理器，如PowerPC处理器，需要在HOST主桥初始化时，将PCI设备使用的I/O地址空间映射为处理器的存储器地址空间。PowerPC处理器对这段“存储器域”的存储器空间进行读写访问时，HOST主桥将存储器域的这段存储器地址转换为PCI总线域的I/O地址，然后通过PCI总线的I/O总线事务对PCI设备的I/O地址进行读写操作。

在PCI总线中，存储器读写事务与I/O读写事务的实现较为类似。首先HOST处理器在初始化时，需要将PCI设备使用的BAR空间映射到“存储器域”的存储器地址空间。之后处理器通过存储器读写指令访问“存储器域”的存储器地址空间，HOST主桥将“存储器域”的读写请求翻译为PCI总线的存储器读写总线事务之后，再发送给目标设备。

值得注意的是，存储器域和PCI总线域的概念，PCI设备能够直接使用的地址为PCI总线域的地址，在PCI总线事务中出现的地址也为PCI总线域的地址；而处理器能够直接使用的地址为存储器域的地址。理解存储器域与PCI总线域的区别对于理解PCI总线至关重要，本篇将在第2.1节专门讨论这两个概念。

以上对PCI总线的存储器与I/O总线事务的介绍并没有考虑PCI桥的存在，如果将PCI桥考虑进来，情况将略微复杂一些。下文将以图1‑1为例说明处理器如何通过HOST主桥和PCI桥1对PCI设备11进行存储器读写操作。当处理器对PCI设备11进行存储器写操作时，这些数据需要通过HOST主桥x和PCI桥x1，最终到达PCI设备11，其访问步骤如下。值得注意的是，以下步骤忽略PCI总线的仲裁过程。

(1) 首先处理器将要传递的数据放入通用寄存器中，之后向PCI设备11映射到的存储器域的地址进行写操作。值得注意的是，处理器并不能直接访问PCI设备11的PCI总线地址空间，因为这些地址空间是属于PCI总线域的，处理器所能直接访问的空间是存储器域的地址空间。处理器必须通过HOST主桥将存储器域的数据访问转换为PCI总线事务才能对PCI总线地址空间进行访问。

(2) HOST主桥x接收来自处理器的存储器写请求，之后处理器结束当前存储器写操作，释放系统总线。HOST主桥x将存储器域的存储器地址转换为PCI总线域的PCI总线地址。并向PCI总线x0发起PCI写请求总线事务。值得注意的是，虽然在许多处理器系统中，存储器地址和PCI总线地址完全相等，但其含义完全不同。

(3) PCI总线x0上的PCI设备01、PCI设备02和PCI桥1将同时监听这个PCI写总线事务。最后PCI桥x1接收这个写总线事务，并结束来自PCI总线x0的PCI总线事务。之后PCI桥x1向PCI总线x1发起新的PCI总线写总线事务。

(4) PCI总线x1上的PCI设备11和PCI设备12同时监听这个PCI写总线事务。最后PCI设备11通过地址译码方式接收这个写总线事务，并结束来自PCI总线x1上的PCI总线事务。

由以上过程可以发现，由于存储器写总线事务使用Posted传送方式，因此数据通过PCI桥后都将结束上一级总线的PCI总线事务，从而上一级PCI总线可以被其他PCI设备使用。如果使用Non-Posted传送方式，直到数据发送到PCI设备11之后，PCI总线x1和x0才能依次释放，从而在某种程度上将造成PCI总线的拥塞。

处理器对PCI设备11进行I/O写操作时只能采用Non-Posted方式进行，与Posted方式相比，使用Non-Posted方式，当数据到达目标设备后，目标设备需要向主设备发出“回应[1] ”，当主设备收到这个“回应”后才能结束整个总线事务。本节不再讲述处理器如何对PCI设备进行I/O写操作，请读者思考这个过程。

处理器对PCI设备11进行存储器读时，这个读请求需要首先通过HOST主桥x和PCI桥x1到达PCI设备，之后PCI设备将读取的数据再次通过PCI桥x1和HOST主桥x传递给HOST处理器，其步骤如下所示。我们首先假设PCI总线没有使用Delayed传送方式处理Non-Posted总线事务，而是使用纯粹的Non-Posted方式。

(1) 首先处理器准备接收数据使用的通用寄存器，之后向PCI设备11映射到的存储器域的地址进行读操作，

(2) HOST主桥x接收来自处理器的存储器读请求。HOST主桥x进行存储器地址到PCI总线地址的转换，之后向PCI总线x0发起存储器读总线事务。

(3) PCI总线x0上的PCI设备01、PCI设备02和PCI桥x1将监听这个存储器读请求，之后PCI桥1接收这个存储器读请求。然后PCI桥x1向PCI总线x1发起新的PCI总线读请求。

(4) PCI总线x1上的PCI设备11和PCI设备12监听这个PCI读请求总线事务。最后PCI设备11接收这个存储器读请求总线事务，并将这个读请求总线事务转换为存储器读完成总线事务之后，将数据传送到PCI桥x1，并结束来自PCI总线x1上的PCI总线事务。

(5) PCI桥x1将接收到的数据通过PCI总线x0，继续上传到HOST主桥x，并结束PCI总线x0上的PCI总线事务。

(6) HOST主桥x将数据传递给处理器，最终结束处理器的存储器读操作。

显然这种方式与Posted传送方式相比，PCI总线的利用率较低。因为只要HOST处理器没有收到来自目标设备的“回应”，那么HOST处理器到目标设备的传送路径上使用的所有PCI总线都将被阻塞。因而PCI总线x0和x1并没有被充分利用。

由以上例子，我们可以发现只有“读完成”依次通过PCI总线x1和x0之后，存储器读总线事务才不继续占用PCI总线x1和x0的资源，显然这种数据传送方式并不合理。因此PCI总线使用Delayed传送方式解决这个总线拥塞问题，有关Delayed传送方式的实现机制见第1.3.5节。

**1.3.4 PCI设备读写主存储器**

PCI设备与存储器直接进行数据交换的过程也被称为DMA。与其他总线的DMA过程类似，PCI设备进行DMA操作时，需要获得数据传送的目的地址和传送大小。支持DMA传递的PCI设备可以在其BAR空间中设置两个寄存器，分别保存这个目标地址和传送大小。这两个寄存器也是PCI设备DMA控制器的组成部件。

值得注意的是，PCI设备进行DMA操作时，使用的目的地址是PCI总线域的物理地址，而不是存储器域的物理地址，因为PCI设备并不能识别存储器域的物理地址，而仅能识别PCI总线域的物理地址。

HOST主桥负责完成PCI总线地址到存储器域地址的转换。HOST主桥需要进行合理设置，将存储器的地址空间映射到PCI总线之后，PCI设备才能对这段存储器空间进行DMA操作。PCI设备不能直接访问没有经过主桥映射的存储器空间。

许多处理器允许PCI设备访问所有存储器域地址空间，但是有些处理器可以设置PCI设备所能访问的存储器域地址空间，从而对存储器域地址空间进行保护。例如PowerPC处理器的HOST主桥可以使用Inbound寄存器组，设置PCI设备访问的存储器地址范围和属性，只有在Inbound寄存器组映射的存储器空间才能被PCI设备访问，本篇将在第2.2节详细介绍PowerPC处理器的这组寄存器。

由上所述，在一个处理器系统中，并不是所有存储器空间都可以被PCI设备访问，只有在PCI总线域中有映像的存储器空间才能被PCI设备访问。经过HOST主桥映射的存储器，具有两个“地址”，一个是在存储器域的地址，一个是在PCI总线域的PCI总线地址。当处理器访问这段存储器空间时，使用存储器地址；而PCI设备访问这段内存时，使用PCI总线地址。在多数处理器系统中，存储器地址与PCI总线地址相同，但是系统程序员需要正确理解这两个地址的区别。

下文以PCI设备11向主存储器写数据为例，说明PCI设备如何进行DMA写操作。

(1) 首先PCI设备11将存储器写请求发向PCI总线x1，注意这个写请求使用的地址是PCI总线域的地址。

(2) PCI总线x1上的所有设备监听这个请求，因为PCI设备11是向处理器的存储器写数据，所以PCI总线x1上的PCI Agent设备都不会接收这个数据请求。

(3) PCI桥x1发现当前总线事务使用的PCI总线地址不是其下游设备使用的PCI总线地址，则接收这个数据请求，有关PCI桥的Secondary总线接收数据的过程见第3.2.1节。此时PCI桥x1将结束来自PCI设备11的Posted存储器写请求，并将这个数据请求推到上游PCI总线上，即PCI总线x0上。

(4) PCI总线x0上的所有PCI设备包括HOST主桥将监听这个请求。PCI总线x0上的PCI Agent设备也不会接收这个数据请求，此时这个数据请求将由HOST主桥x接收，并结束PCI桥x1的Posted存储器写请求。

(5) HOST主桥x发现这个数据请求发向存储器，则将来自PCI总线x0的PCI总线地址转换为存储器地址，之后通过存储器控制器将数据写入存储器，完成PCI设备的DMA写操作。

PCI设备进行DMA读过程与DMA写过程较为类似。不过PCI总线的存储器读总线事务只能使用Non-Posted总线事务，其过程如下。

(1) 首先PCI设备11将存储器读请求发向PCI总线x1。

(2) PCI总线x1上的所有设备监听这个请求，因为PCI设备11是从存储器中读取数据，所以PCI总线x1上的设备，如PCI设备12，不会接收这个数据请求。PCI桥x1发现下游PCI总线没有设备接收这个数据请求，则接收这个数据请求，并将这个数据请求推到上游PCI总线上，即PCI总线x0上。

(3) PCI总线x0上的设备将监听这个请求。PCI总线x0上的设备也不会接收这个数据请求，最后这个数据请求将由HOST主桥x接收。

(4) HOST主桥x发现这个数据请求是发向主存储器，则将来自PCI总线x0的PCI总线地址转换为存储器地址，之后通过存储器控制器将数据读出，并转发到HOST主桥x。

(5) HOST主桥x将数据经由PCI桥x1传递到PCI设备11，PCI设备11接收到这个数据后结束DMA读。

以上过程仅是PCI设备向存储器读写数据的一个简单流程。如果考虑处理器中的Cache，这些存储器读写过程较为复杂。

PCI总线还允许PCI设备之间进行数据传递，PCI设备间的数据交换较为简单。在实际应用中，PCI设备间的数据交换并不常见。下文以图1‑1为例，简要介绍PCI设备11将数据写入PCI设备01的过程；请读者自行考虑PCI设备11从PCI设备01读取数据的过程。

(1) 首先PCI设备11将PCI写总线事务发向PCI总线x1上。PCI桥x1和PCI设备12同时监听这个写总线事务。

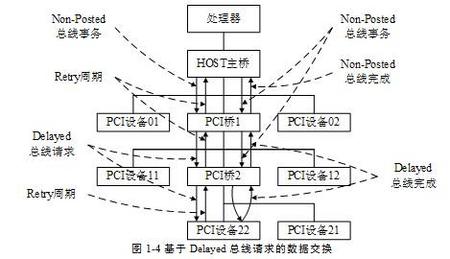
(2) PCI桥x1将接收这个PCI写请求总线事务，并将这个PCI写总线事务上推到PCI总线x0。

(3) PCI总线x0上的所有设备将监听这个PCI写总线事务，最后由PCI设备01接收这个数据请求，并完成PCI写事务。

**1.3.5 Delayed传送方式**

如上文所述，如果处理器使用Non-Posted总线周期对PCI设备进行读操作，或者PCI设备使用Non-Posted总线事务对存储器进行读操作时，如果数据没有到达目的地，那么在这个读操作路径上的所有PCI总线都不能被释放，这将严重影响PCI总线的使用效率。

为此PCI桥需要对Non-Posted总线事务进行优化处理，并使用Delayed总线事务处理这些Non-Posted总线事务，PCI总线规定只有Non-Posted总线事务可以使用Delayed总线事务。PCI总线的Delay总线事务由Delay读写请求和Delay读写完成总线事务组成，当Delay读写请求到达目的地后，将被转换为Delay读写完成总线事务。基于Delay总线请求的数据交换如图1‑4所示。

[](http://download.bbs.ednchina.com/images/attachments/201303/original/466978_TIME_1363827018502.jpg)

假设处理器通过存储器读、I/O读写或者配置读写访问PCI设备22时，首先经过HOST主桥进行存储器域与PCI总线域的地址转换，并由HOST主桥发起PCI总线事务，然后通过PCI桥1、2，最终到达PCI设备22。其详细步骤如下。

(1) HOST主桥完成存储器域到PCI总线域的转换，然后启动PCI读总线事务。

(2) PCI桥1接收这个读总线事务，并首先使用Retry周期，使HOST主桥择时重新发起相同的总线周期。此时PCI桥1的上游PCI总线将被释放。值得注意的是PCI桥并不会每一次都使用Retry周期，使上游设备择时进行重试操作。在PCI总线中，有一个“16 Clock”原则，即FRAME#信号有效后，必须在16个时钟周期内置为无效，如果PCI桥发现来自上游设备的读总线事务不能在16个时钟周期内结束时，则使用Retry周期终止该总线事务。

(3) PCI桥1使用Delayed总线请求继续访问PCI设备22。

(4) PCI桥2接收这个总线请求，并将这个Delayed总线请求继续传递。此时PCI桥2也将首先使用Retry周期，使PCI桥1择时重新发起相同的总线周期。此时PCI桥2的上游PCI总线被释放。

(5) 这个数据请求最终到达PCI设备22，如果PCI设备22没有将数据准备好时，也可以使用Retry周期，使PCI桥2择时重新发起相同的总线周期；如果数据已经准备好，PCI设备22将接收这个数据请求，并将这个Delayed总线请求转换为Delayed总线完成事务。如果Delayed总线请求是读请求，则Delayed总线完成事务中含有数据，否则只有完成信息，而不包含数据。

(6) Delayed总线完成事务将“数据或者完成信息”传递给PCI桥2，当PCI桥1重新发出Non-Posted总线请求时，PCI桥2将这个“数据或者完成信息”传递给PCI桥1。

(7) HOST主桥重新发出存储器读总线事务时，PCI桥1将“数据或者完成信息”传递给HOST主桥，最终完成整个PCI总线事务。

由以上分析可知，Delayed总线周期由Delayed总线请求和Delayed总线完成两部分组成。下文将Delayed读请求总线事务简称为DRR(Delayed Read Request)，Delayed读完成总线事务简称为DRC(Delayed Read Completion)；而将Delayed写请求总线事务简称为DWR(Delayed Write Request)，Delayed写完成总线事务简称为DWC(Delayed Write Completion)。

PCI总线使用Delayed总线事务，在一定程度上可以提高PCI总线的利用率。因为在进行Non-Posted总线事务时，Non-Posted请求在通过PCI桥之后，可以暂时释放PCI总线，但是采用这种方式，HOST/PCI桥将会择时进行重试操作。在许多情况下，使用Delayed总线事务，并不能取得理想的效果，因为过多的重试周期也将大量消耗PCI总线的带宽。

为了进一步提高Non-Posted总线事务的执行效率，PCI-X总线将PCI总线使用的Delayed总线事务，升级为Split总线事务。采用Split总线事务可以有效解决HOST/PCI桥的这些重试操作。Split总线事务的基本思想是发送端首先将Non-Posted总线请求发送给接收端，然后再由接收端主动地将数据传递给发送端。

除了PCI-X总线可以使用Split总线事务进行数据传送之外，有些处理器，如x86和PowerPC处理器的FSB(Front Side Bus)总线也支持这种Split总线事务，因此这些HOST主桥也可以发起这种Split总线事务。在PCIe总线中，Non-Posted数据传送都使用Split总线事务完成，而不再使用Delayed总线事务。本章将在第1.5.1节简要介绍Split总线事务和PCI-X总线对PCI总线的一些功能上的增强。

[1] 如果是存储器、I/O读或者配置读总线事务，这个回应包含数据；如果是I/O写或者配置写，这个回应不包含数据。

|  |
| --- |
| **1.4 PCI总线的中断机制** |
| 发布时间：2013-03-22 09:31:27 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI总线使用INTA#、INTB#、INTC#和INTD#信号向处理器发出中断请求。这些中断请求信号为低电平有效，并与处理器的中断控制器连接。在PCI体系结构中，这些中断信号属于边带信号(Sideband Signals)，PCI总线规范并没有明确规定在一个处理器系统中如何使用这些信号，因为这些信号对于PCI总线是可选信号。PCI设备还可以使用MSI机制向处理器提交中断请求，而不使用这组中断信号。有关MSI机制的详细说明见第8章。

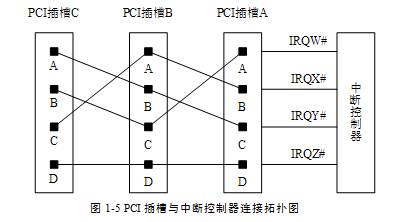
**1.4.1 中断信号与中断控制器的连接关系**

不同的处理器使用的中断控制器不同，如x86处理器使用APIC(Advanced Programmable Interrupt Controller)中断控制器，而PowerPC处理器使用MPIC(Multiprocessor Interrupt Controller)中断控制器。这些中断控制器都提供了一些外部中断请求引脚IRQ\_PINx#。外部设备，包括PCI设备可以使用这些引脚向处理器提交中断请求。

但是PCI总线规范没有规定PCI设备的INTx信号如何与中断控制器的IRQ\_PINx#信号相连，这为系统软件的设计带来了一定的困难，为此系统软件使用中断路由表存放PCI设备的INTx信号与中断控制器的连接关系。在x86处理器系统中，BIOS可以提供这个中断路由表，而在PowerPC处理器中Firmware也可以提供这个中断路由表。

在一些简单的嵌入式处理器系统中，Firmware并没有提供中断路由表，此时系统软件开发者需要事先了解PCI设备的INTx信号与中断控制器的连接关系。此时外部设备与中断控制器的连接关系由硬件设计人员指定。

我们假设在一个处理器系统中，共有3个PCI插槽(分别为PCI插槽A、B和C)，这些PCI插槽与中断控制器的IRQ\_PINx引脚(分别为IRQW#、IRQX#、IRQY#和IRQZ#)可以按照图1‑5所示的拓扑结构进行连接。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130322092911730.jpg)

采用图1‑5所示的拓扑结构时，PCI插槽A、B、C的INTA#、INTB#和INTC#信号将分散连接到中断控制器的IRQW#、IRQX#和IRQY#信号，而所有INTD#信号将共享一个IRQZ#信号。采用这种连接方式时，整个处理器系统使用的中断请求信号，其负载较为均衡。而且这种连接方式保证了每一个插槽的INTA#信号都与一根独立的IRQx#信号对应，从而提高了PCI插槽中断请求的效率。

在一个处理器系统中，多数PCI设备仅使用INTA#信号，很少使用INTB#和INTC#信号，而INTD#信号更是极少使用。在PCI总线中，PCI设备配置空间的Interrupt Pin寄存器记录该设备究竟使用哪个INTx信号，该寄存器的详细介绍见第2.3.2节。

**1.4.2 中断信号与PCI总线的连接关系**

在PCI总线中，INTx信号属于边带信号。所谓边带信号是指这些信号在PCI总线中是可选信号，而且只能在一个处理器系统的内部使用，并不能离开这个处理器环境。PCI桥也不会处理这些边带信号。这给PCI设备将中断请求发向处理器带来了一些困难，特别是给挂接在PCI桥之下的PCI设备进行中断请求带来了一些麻烦。

在一些嵌入式处理器系统中，这个问题较易解决。因为嵌入式处理器系统很清楚在当前系统中存在多少个PCI设备，这些PCI设备使用了哪些中断资源。在多数嵌入式处理器系统中，PCI设备的数量小于中断控制器提供的外部中断请求引脚数，而且在嵌入式系统中，多数PCI设备仅使用INTA#信号提交中断请求。

在这类处理器系统中，可能并不含有PCI桥，因而PCI设备的中断请求信号与中断控制器的连接关系较易确定。而在这类处理器系统中，即便存在PCI桥，来自PCI桥之下的PCI设备的中断请求也较易处理。

在多数情况下，嵌入式处理器系统使用的PCI设备仅使用INTA#信号进行中断请求，所以只要将这些INTA#信号挂接到中断控制器的独立IRQ\_PIN#引脚上即可。这样每一个PCI设备都可以独占一个单独的中断引脚。

而在x86处理器系统中，这个问题需要BIOS参与来解决。在x86处理器系统中，有许多PCI插槽，处理器系统并不知道在这些插槽上将要挂接哪些PCI设备，而且也并不知道这些PCI设备到底需不需要使用所有的INTx#信号线。因此x86处理器系统必须要对各种情况进行处理。

x86处理器系统还经常使用PCI桥进行PCI总线扩展，扩展出来的PCI总线还可能挂接一些PCI插槽，这些插槽上INTx#信号仍然需要处理。PCI桥规范并没有要求桥片传递其下PCI设备的中断请求。事实上多数PCI桥也没有为下游PCI总线提供中断引脚INTx#，管理其下游总线的PCI设备。但是PCI桥规范推荐使用表1‑3建立下游PCI设备的INTx信号与上游PCI总线INTx信号之间的映射关系。

表1‑3 PCI设备INTx#信号与PCI总线INTx#信号的映射关系

|  |  |  |
| --- | --- | --- |
| 设备号 | PCI设备的INTx#信号 | PCI总线的INTx#信号 |
| 0, 4, 8, 12, 16, 20, 24, 28 | INTA# | INTA# |
| INTB# | INTB# |
| INTC# | INTC# |
| INTD# | INTD# |
| 1, 5, 9, 13, 17, 21, 25, 29 | INTA# | INTB# |
| INTB# | INTC# |
| INTC# | INTD# |
| INTD# | INTA# |
| 2, 6, 10, 14, 18, 22, 26, 30 | INTA# | INTC# |
| INTB# | INTD# |
| INTC# | INTA# |
| INTD# | INTB# |
| 3, 7, 11, 15, 19, 23, 27, 31 | INTA# | INTD# |
| INTB# | INTA# |
| INTC# | INTB# |
| INTD# | INTC# |

我们举例说明该表的含义。在PCI桥下游总线上的PCI设备，如果其设备号为0，那么这个设备的INTA#引脚将和PCI总线的INTA#引脚相连；如果其设备号为1，其INTA#引脚将和PCI总线的INTB#引脚相连；如果其设备号为2，其INTA#引脚将和PCI总线的INTC#引脚相连；如果其设备号为3，其INTA#引脚将和PCI总线的INTD#引脚相连。

在x86处理器系统中，由BIOS或者APCI表记录PCI总线的INTA~D#信号与中断控制器之间的映射关系，保存这个映射关系的数据结构也被称为中断路由表。大多数BIOS使用表1‑3中的映射关系，这也是绝大多数BIOS支持的方式。如果在一个x86处理器系统中，PCI桥下游总线的PCI设备使用的中断映射关系与此不同，那么系统软件程序员需要改动BIOS中的中断路由表。

BIOS初始化代码根据中断路由表中的信息，可以将PCI设备使用的中断向量号写入到该PCI设备配置空间的Interrupt Line register寄存器中，该寄存器将在第2.3.2节中介绍。

**1.4.3 中断请求的同步**

在PCI总线中，INTx信号是一个异步信号。所谓异步是指INTx信号的传递并不与PCI总线的数据传送同步，即INTx信号的传递与PCI设备使用的CLK#信号无关。这个“异步”信号给系统软件的设计带来了一定的麻烦。

系统软件程序员需要注意“异步”这种事件，因为几乎所有“异步”事件都会带来系统的“同步”问题。以图1‑1为例，当PCI设备11使用DMA写方式，将一组数据写入存储器时，该设备在最后一个数据离开PCI设备11的发送FIFO时，会认为DMA写操作已经完成。此时这个设备将通过INTx信号，通知处理器DMA写操作完成。

此时处理器(驱动程序的中断服务例程)需要注意，因为INTx信号是一个异步信号，当处理器收到INTx信号时，并不意味着PCI设备11已经将数据写入存储器中，因为PCI设备11的数据传递需要通过PCI桥1和HOST主桥，最终才能到达存储器控制器。

而INTx信号是“异步”发送给处理器的，PCI总线并不知道这个“异步”事件何时被处理。很有可能处理器已经接收到INTx信号，开始执行中断处理程序时，该PCI设备还没有完全将数据写入存储器。

因为“PCI设备向处理器提交中断请求”与“将数据写入存储器”分别使用了两个不同的路径，处理器系统无法保证哪个信息率先到达。从而在处理器系统中存在“中断同步”的问题，PCI总线提供了以下两种方法解决这个同步问题。

(1) PCI设备保证在数据到达目的地之后，再提交中断请求。

显然这种方法不仅加大了硬件的开销，而且也不容易实现。如果PCI设备采用Posted写总线事务，PCI设备无法单纯通过硬件逻辑判断数据什么时候写入到存储器。此时为了保证数据到达目的地后，PCI设备才能提交中断请求，PCI设备需要使用“读刷新”的方法保证数据可以到达目的地，其方法如下。

PCI设备在提交中断请求之前，向DMA写的数据区域发出一个读请求，这个读请求总线事务将被PCI设备转换为读完成总线事务，当PCI设备收到这个读完成总线事务后，再向处理器提交中断请求。PCI总线的“序”机制保证这个存储器读请求，会将DMA数据最终写入存储器，有关PCI序的详细说明见第9.3节。

PCI总线规范要求HOST主桥和PCI桥必须保证这种读操作可以刷新写操作。但问题是，没有多少芯片设计者愿意提供这种机制，因为这将极大地增加他们的设计难度。除此之外，使用这种方法也将增加中断请求的延时。

(2) 中断服务例程使用“读刷新”方法。

中断服务例程在使用“PCI设备写入存储器”的这些数据之前，需要对这个PCI设备进行读操作。这个读操作也可以强制将数据最终写入存储器，实际上是将数据写到存储器控制器中。这种方法利用了PCI总线的传送序规则，这种方法与第1种方法基本相同，只是使用这种方法使用软件方式，而第1种方式使用硬件方式。第9.3节将详细介绍这个读操作如何将数据刷新到存储器中。

第2种方法也是绝大多数处理器系统采用的方法。程序员在书写中断服务例程时，往往都是先读取PCI设备的中断状态寄存器，判断中断产生原因之后，才对PCI设备写入的数据进行操作。这个读取中断状态寄存器的过程，一方面可以获得设备的中断状态，另一方面是保证DMA写的数据最终到达存储器。如果驱动程序不这样做，就可能产生数据完整性问题。产生这种数据完整性问题的原因是INTx这个异步信号。

这里也再次提醒系统程序员注意PCI总线的“异步”中断所带来的数据完整性问题。在一个操作系统中，即便中断处理程序没有首先读取PCI设备的寄存器，也多半不会出现问题，因为在操作系统中，一个PCI设备从提交中断到处理器开始执行设备的中断服务例程，所需要的时间较长，处理器系统基本上可以保证此时数据已经写入存储器。

但是如果系统程序员不这样做，这个驱动程序依然有Bug存在，尽管这个Bug因为各种机缘巧合，始终不能够暴露出来，而一旦这些Bug被暴露出来将难以定位。为此系统程序员务必要重视设计中出现的每一个实现细节，当然仅凭谨慎小心是远远不够的，因为重视细节的前提是充分理解这些细节。

PCI总线V2.2规范还定义了一种新的中断机制，即MSI中断机制。MSI中断机制采用存储器写总线事务向处理器系统提交中断请求，其实现机制是向HOST处理器指定的一个存储器地址写指定的数据。这个存储器地址一般是中断控制器规定的某段存储器地址范围，而且数据也是事先安排好的数据，通常含有中断向量号。

HOST主桥会将MSI这个特殊的存储器写总线事务进一步翻译为中断请求，提交给处理器。目前PCIe和PCI-X设备必须支持MSI中断机制，但是PCI设备并不一定都支持MSI中断机制。

目前MSI中断机制虽然在PCIe总线上已经成为主流，但是在PCI设备中并不常用。即便是支持MSI中断机制的PCI设备，在设备驱动程序的实现中也很少使用这种机制。首先PCI设备具有INTx#信号可以传递中断，而且这种中断传送方式在PCI总线中根深蒂固。其次PCI总线是一个共享总线，传递MSI中断需要占用PCI总线的带宽，需要进行总线仲裁等一系列过程，远没有使用INTx#信号线直接。

但是使用MSI中断机制可以取消PCI总线这个INTx#边带信号，可以解决使用INTx中断机制所带来的数据完整性问题。而更为重要的是，PCI设备使用MSI中断机制，向处理器系统提交中断请求的同时，还可以通知处理器系统产生该中断的原因，即通过不同中断向量号表示中断请求的来源。当处理器系统执行中断服务例程时，不需要读取PCI设备的中断状态寄存器，获得中断请求的来源，从而在一定程度上提高了中断处理的效率。本书将在第8章详细介绍MSI中断机制。

|  |
| --- |
| **1.5 PCI-X总线简介** |
| 发布时间：2013-03-25 09:07:58 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI-X总线仍采用并行总线技术。PCI-X总线使用的大多数总线事务基于PCI总线，但是在实现细节上略有不同。PCI-X总线将工作频率提高到533MHz，并首先引入了PME(Power Management Event)机制。除此之外，PCI-X总线还提出了许多新的特性。

### 1.5.1 Split总线事务

Split总线事务是PCI-X总线一个重要特性。该总线事务替代了PCI总线的Delayed数据传送方式，从而提高了Non-Posted总线事务的传送效率。下文以存储器读为例，说明PCI-X设备如何使用Split总线事务。

PCI-X总线在进行存储器读总线事务时，总线事务的发起方(Requester)使用Split总线事务与总线事务接收端(Completer)进行数据交换，其步骤如下。

(1) Requester向Completer发起存储器读请求总线事务。

(2) 这个存储器读请求在到达Completer之前，可能会经过多级PCI-X桥。这些PCI-X桥使用Split Response周期结束当前总线事务，释放上游PCI总线。之后继续转发这个存储器读请求，直到Completer认领这个存储器读请求总线事务。

(3) Completer认领存储器读请求总线事务后，将记录将Requester的ID号，并使用Split Response周期结束存储器读请求总线事务。

(4) Completer准备好数据后，将重新申请总线，并使用存储器读完成总线事务主动地将数据传送给Requester。在这个完成报文中包含Requester的ID号，因为完成报文使用ID路由而不是地址路由。

(5) 这些完成报文根据ID路由方式，最终到达Requester。Requester从完成报文中接收数据并完成整个存储器读请求。

与Delayed总线事务相比，Requester获得的数据是Completer将数据完全准备好后，由Completer主动传递的，而不是通过Requester通过多次重试获得的，因此能够提高PCI-X总线的使用效率。PCI-X总线提出的Split总线事务被PCIe总线继承。

### 1.5.2 总线传送协议

PCI-X总线改变了PCI总线使用的传送协议。目标设备可以将主设备发送的命令锁存，然后在下一个时钟周期进行译码操作。与PCI总线事务相比，PCI-X总线采用的这种方式，虽然在总线时序中多使用了一个时钟周期，但是可以有效提高PCI-X总线的运行频率。

因为主设备通过数据线将命令发送到目标设备需要一定的延时。如果PCI总线频率较高，目标设备很难在一个时钟周期内接收完毕总线命令，并同时完成译码工作。而如果目标设备能够将主设备发出的命令先进行锁存，然后在下一个时钟周期进行译码则可以有效解决这个译码时间Margin不足的问题，从而提高PCI-X总线的频率。PCI-X 1.0总线可以使用的最高总线频率为133MHz，而PCI-X 2.0总线可以使用的最高总线频率为533Mhz，远比PCI总线使用的总线频率高。

除了信号传送协议外，PCI-X总线在进行DMA读写时，可以不进行Cache共享一致性操作，而PCI总线进行DMA读写时必须进行Cache一致性操作。在某些特殊情况下，DMA读写时进行Cache共享一致性不但不能提高总线传送效率，反而会降低。第3.3节将详细讨论与Cache一致性相关的PCI总线事务。

此外PCI-X总线还支持乱序总线事务，即Relaxed Ordering，该总线事务被PCIe总线继承。对于某些应用，PCI-X设备使用Relaxed ordering方式，可以有效地提高数据传送效率。但是支持Relaxed Ordering的设备，需要较多的数据缓存和硬件逻辑处理这些乱序，这为PCI-X设备的设计带来了不小的困难。

### 1.5.3 基于数据块的突发传送

在PCI总线中，一次突发传送的大小为2个以上的双字，一次突发传送所携带的数据越多时，突发传送的总线利用率也越高。

而PCI总线的突发传送仍然存在缺陷。在PCI总线中，数据发送端知道究竟需要发送多少字节的数据，但是接收端并不清楚到底需要接收多少数据。这种不确定性，为接收端的缓冲管理带来了较大的挑战。

为此PCI-X总线使用基于数据块的突发传送方式，发送端以ADB(Allowable Disconnect Boundary)为单位，将数据发送给接收端，一次突发读写为一个以上的ADB。采用这种方式，接收端可以事先预知是否有足够的接收缓冲，接收来自发送端的数据，从而可以及时断连当前总线周期，以节约PCI-X总线的带宽。在PCI-X总线中，ADB的大小为128B。

由于ADB的引入，PCI总线与Cache相关的总线事务如Memory Read Line、Memory Read Multiline和Memory Write and Invalidate，都被PCI-X总线使用与ADB相关的总线事务替代。因为通过ADB，PCI-X桥(HOST主桥)可以准确地预知即将访问的数据在Cache中的分布情况。

PCI-X总线还增加了一些其他特性，如在总线事务中增加传送字节计数，限制等待状态等机制，并增强了奇偶校验的管理方式。但是PCI-X总线还没有普及，就被PCIe总线替代。因此在PC领域和嵌入式领域很少有基于PCI-X总线的设备，PCI-X设备仅在一些高端服务器上出现。因此本节不对PCI-X总线做进一步描述。事实上，PCI-X总线的许多特性都被PCIe总线继承。

### 1.6 小结

本章主要介绍了PCI总线的基本组成部件，PCI设备如何提交中断请求，以及PCI-X总线对PCI总线的功能增强。本章的重点在于PCI总线的Posted和Non-Posted总线事务，以及PCI总线如何使用Delayed传送方式处理Non-Posted总线事务，请读者务必深入理解这两种总线事务的不同。

|  |
| --- |
| **第2章 PCI总线的桥与配置** |
| 发布时间：2013-03-26 13:35:23 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

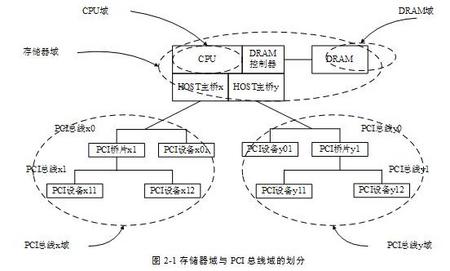
在PCI体系结构中，含有两类桥片，一个是HOST主桥，另一个是PCI桥。在每一个PCI设备中(包括PCI桥)都含有一个配置空间。这个配置空间由HOST主桥管理，而PCI桥可以转发来自HOST主桥的配置访问。在PCI总线中，PCI Agent设备使用的配置空间与PCI桥使用的配置空间有些差别，但这些配置空间都是由处理器通过HOST主桥管理。

|  |
| --- |
| **2.1 存储器域与PCI总线域** |
| 发布时间：2013-03-26 13:41:40 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

HOST主桥的实现因处理器系统而异。PowerPC处理器和x86处理器的HOST主桥除了集成方式不同之外，其实现机制也有较大差异。但是这些HOST主桥所完成的最基本功能依然是分离存储器域与PCI总线域，完成PCI总线域到存储器域，存储器域到PCI总线域之间的数据传递，并管理PCI设备的配置空间。

上文曾经多次提到在一个处理器系统中，存在PCI总线域与存储器域，深入理解这两个域的区别是理解HOST主桥的关键所在。在一个处理器系统中，存储器域、PCI总线域与HOST主桥的关系如图2‑1所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130326133913734.jpg)

上图所示的处理器系统由一个CPU，一个DRAM控制器和两个HOST主桥组成。在这个处理器系统中，包含CPU域、DRAM域、存储器域和PCI总线域地址空间。其中HOST主桥x和HOST主桥y分别管理PCI总线x域与PCI总线y域。PCI设备访问存储器域时，也需要通过HOST主桥，并由HOST主桥进行PCI总线域到存储器域的地址转换；CPU访问PCI设备时，同样需要通过HOST主桥进行存储器域到PCI总线域的地址转换。

如果HOST主桥支持Peer-to-Peer传送机制，PCI总线x域上的设备可以与PCI总线y域上的设备直接通信，如PCI设备x11可以直接与PCI设备y11通信。为简化模型，在本书中，PCI总线仅使用32位地址空间。

**2.1.1 CPU域、DRAM域与存储器域**

CPU域地址空间指CPU所能直接访问的地址空间集合。在本书中，CPU、处理器与处理器系统的概念不同。如MPC8548处理器的内核是E500 V2[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM#_ftn1)，本书将这个处理器内核称为CPU；处理器由一个或者多个CPU、外部Cache、中断控制器和DRAM控制器组成；而处理器系统由一个或者多个处理器和外部设备组成。

在CPU域中有一个重要概念，即CPU域边界，所谓CPU域边界，即CPU所能控制的数据完整性边界。CPU域的边界由Memory Fence指令[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM#_ftn2)的作用范围确定，CPU域边界的划分对数据完整性(Data Consistency)非常重要。与CPU域相关的数据完整性知识较为复杂，可以独立出书，因此本篇对数据完整性不做进一步介绍。笔者有计划再更新完PCIe总线部分的资料后，书体系结构的两方面内容，一个是Cache层次结构，一个是以Weakly Ordered Memory Modle为基础书写数据完整性。

严格的讲CPU域仅在CPU内核中有效，CPU访问主存储器时，首先将读写命令放入读写指令缓冲中，然后将这个命令发送到DRAM控制器或者HOST主桥。DRAM控制器或者HOST主桥将CPU地址转换为DRAM或者PCI总线地址，分别进入DRAM域或者PCI总线域后，再访问相应的地址空间。

DRAM域地址空间指DRAM控制器所能访问的地址空间集合。目前处理器系统的DRAM一般由DDR-SDRAM组成，有的书籍也将这部分内存称为主存储器。在有些处理器系统中，DRAM控制器能够访问的地址空间，并不能被处理器访问，因此在这类处理器系统中，CPU域与DRAM域地址空间并不等同。

比如有些CPU可以支持36位的物理地址，而有些DRAM控制器仅支持32位的物理地址，此时CPU域包含的地址空间大于DRAM域地址空间。但是这并不意味着DRAM域一定包含在CPU域中，在某些处理器系统中，CPU并不能访问在DRAM域中的某些数据区域。而CPU域中除了包含DRAM域外，还包含外部设备空间。

在多数处理器系统中，DRAM域空间是CPU域空间的一部分，但是也有例外。比如显卡控制器可能会借用一部分主存储器空间，这些被借用的空间不能被CPU访问，而只能被DRAM控制器，更为准确地说是显卡通过DRAM控制器访问，因此这段空间不属于CPU域，严格地讲，这段空间属于外部设备域。

本书使用存储器域统称CPU域与DRAM域。存储器域包括CPU内部的通用寄存器，存储器映像寻址的寄存器，主存储器空间和外部设备空间。在Intel的x86处理器系统中，外部设备空间与PCI总线域地址空间等效，因为在x86处理器系统中，使用PCI总线统一管理全部外部设备。为简化起见，本书使用PCI总线域替代外部设备域。

值得注意的是，存储器域的外部设备空间，在PCI总线域中还有一个地址映射。当处理器访问PCI设备时，首先访问的是这个设备在存储器域上的PCI设备空间，之后HOST主桥将这个存储器域的PCI总线地址转换为PCI总线域的物理地址[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM" \l "_ftn3" \o ")，然后再通过PCI总线事务访问PCI总线域的地址空间。

**2****.1.2 PCI总线域**

在x86处理器系统中，PCI总线域是外部设备域的重要组成部分。实际上在Intel的x86处理器系统中，所有的外部设备都使用PCI总线管理。而AMD的x86处理器系统中还存在一条HT(HyperTransport)总线，在AMD的x86处理器系统中还存在HT总线域。本书对HT总线不做进一步介绍。

PCI总线域(PCI Segment)由PCI设备所能直接访问的地址空间组成。在一个处理器系统中，可能存在多个HOST主桥，因此也存在多个PCI总线域。如在图2‑1所示的处理器系统中，具有两个HOST主桥，因而在这个处理器系统中存在PCI总线x和y域。

在多数处理器系统中，分属于两个PCI总线域的PCI设备并不能直接进行数据交换，而需要通过FSB进行数据交换。值得注意的是，如果某些处理器的HOST主桥支持Peer-to-Peer数据传送，那么这个HOST主桥可以支持不同PCI总线域间的数据传送。

PowerPC处理器使用了OCeaN技术连接两个HOST主桥，OCeaN可以将属于x域的PCI数据请求转发到y域，OCeaN支持PCI总线的Peer-to-Peer数据传送。有关OCeaN技术的详细说明见第2.2节。

**2.1.3 处理器域**

处理器域是指一个处理器系统能够访问的地址空间集合。处理器系统能够访问的地址空间由存储器域和外部设备域组成。其中存储器域地址空间较为简单，而在不同的处理器系统中，外部设备域的组成结构并不相同。如在x86处理器系统中，外部设备域主要由PCI总线域组成，因为大多数外部设备都是挂接在PCI总线[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM" \l "_ftn4" \o ")上的，而在PowerPC处理器和其他处理器系统中，有相当多的设备与FSB直接相连，而不与PCI总线相连。

本书仅介绍PCI总线域而不对其他外部设备域进行说明。其中存储器域与PCI总线域之间由HOST主桥联系在一起。深入理解这些域的关系是深入理解PCI体系结构的关键所在，实际上这也是理解处理器体系结构的基础。

通过HOST主桥，处理器系统可以将处理器域划分为存储器域与PCI总线域。其中存储器域与PCI总线域，彼此独立，并通过HOST主桥进行数据交换。HOST主桥是联系存储器域与PCI总线域的桥梁，是PCI总线域实际的管理者。

有些书籍认为HOST处理器是PCI总线域的管理者，这种说法并不精确。假设在一个SMP(symmetric multiprocessing)处理器系统中，存在4个CPU而只有一个HOST主桥，这4个CPU将无法判断究竟谁是HOST处理器。不过究竟是哪个处理器作为HOST处理器并不重要，因为在一个处理器系统中，是HOST主桥管理PCI总线域，而不是HOST处理器。当一个处理器系统中含有多个CPU时，如果这些CPU都可以访问HOST主桥，那么这些CPU都可以作为这个HOST主桥所管理PCI总线树的HOST处理器。

在一个处理器系统中，CPU所能访问的PCI总线地址一定在存储器域中具有地址映射；而PCI设备能访问的存储器域的地址也一定在PCI总线域中具有地址映射。当CPU访问PCI域地址空间时，首先访问存储器域的地址空间，然后经过HOST主桥转换为PCI总线域的地址，再通过PCI总线事务进行数据访问。而当PCI设备访问主存储器时，首先通过PCI总线事务访问PCI总线域的地址空间，然后经过HOST主桥转换为存储器域的地址后，再对这些空间进行数据访问。

由此可见，存储器域与PCI总线域的转换关系由HOST主桥统一进行管理。有些处理器提供了一些寄存器进行这种地址映射，如PowerPC处理器使用Inbound和Outbound寄存器组保存存储器域与PCI总线域的地址映射关系；而有些处理器并没有提供这些寄存器，但是存储器域到PCI总线域的转换关系依然存在。

HOST主桥进行不同地址域间的数据交换时，需要遵循以下规则。为区别存储器域到PCI总线域的地址映射，下文将PCI总线域到存储器域的地址映射称为反向映射。

(1) 处理器访问PCI总线域地址空间时，首先需要访问存储器域的地址空间，之后通过HOST主桥将存储器地址转换为PCI总线地址，之后才能进入PCI总线域进行数据交换。PCI设备使用的地址空间保存在各自的PCI配置寄存器中，即BAR寄存器中。这些PCI总线地址空间需要在初始化时映射成为存储器域的存储器地址空间，之后处理器才能访问这些地址空间。在有些处理器的HOST主桥中，具有独立的寄存器保存这个地址映射规则，如PowerPC处理器的Outbound寄存器组；而有些处理器，如在x86处理器中，虽然没有这样的寄存器组，但是在HOST主桥的硬件逻辑中仍然存在这个地址转换的概念。

(2) PCI设备访问存储器域时，首先需要访问PCI总线域的地址空间，之后通过HOST主桥将PCI总线地址转换为存储器地址，之后才能穿越HOST主桥进行数据交换。为此处理器需要通过HOST主桥将这个PCI总线地址反向映射为存储器地址。PCI设备不能访问在PCI总线域中没有进行这种反向映射的存储器域地址空间。PowerPC处理器使用Inbound寄存器组存放PCI设备所能访问的存储器空间，而在x86处理器中并没有这样的寄存器组，但是依然存在这个地址转换的概念。

(3) 如果HOST主桥不支持Peer-to-Peer传送方式，那么分属不同PCI总线域的PCI设备间不能直接进行数据交换。在32位的PCI总线中，每一个PCI总线域的地址范围都是0x0000-0000~0xFFFF-FFFF，但是这些地址没有直接联系。PCI总线x域上的PCI总线地址0x0000-0000与PCI总线y域上的PCI总线地址0x0000-0000并不相同，而且这两个PCI总线地址经过HOST主桥反向映射后，得到的存储器地址也不相同。

本篇在第2.2节中，主要以PowerPC处理器为例说明HOST主桥的实现机制，并在第2.2.4节简要说明了x86处理器中的南北桥构架。尽管部分读者对PowerPC处理器并不感兴趣，笔者仍然强烈建议读者仔细阅读第2.2节的全部内容。

在PowerPC处理器中，HOST主桥的实现比较完整，尤其是PCI总线域与存储器域的映射关系比较明晰，便于读者准确掌握这个重要的概念。而x86处理器由于考虑向前兼容，设计中包含了太多的不得已，x86处理器有时不得不保留原设计中的不完美，向前兼容是Intel的重要成就，也是一个沉重的十字架。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM#_ftnref1) MPC8548处理器基于E500 V2内核。目前E500内核包括V1，V2和mc(MultiCore)三个版本。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM#_ftnref2) x86处理器的Memory Fence指令为MFENCE，LFENCE和SFENCE，而PowerPC处理器的Memory Fence指令为msync和mbar。

[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM" \l "_ftnref3" \o ") PCI总线域只含有物理地址，因此下文将直接使用PCI总线地址，而不使用PCI总线物理地址。

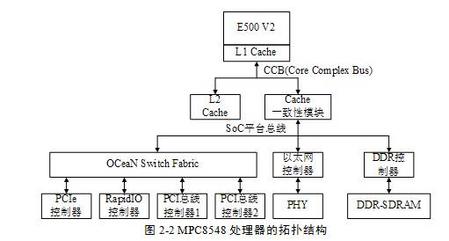
[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011598.HTM" \l "_ftnref4" \o ") AMD的x86处理器中的某些外部设备，可能是基于HT总线，而不使用PCI总线。

|  |
| --- |
| **2.2 HOST主桥(1)** |
| 发布时间：2013-03-28 10:11:49 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

本节以MPC8548处理器为例说明HOST主桥在PowerPC处理器中的实现机制，并简要介绍x86处理器系统使用的HOST主桥。

MPC8548处理器是Freescale基于E500 V2内核的一个PowerPC处理器，该处理器中集成了DDR控制器、多个eTSEC(Enhanced Three-Speed Ethernet Controller)、PCI/PCI-X和PCIe总线控制器等一系列接口。MPC8548处理器的拓扑结构如图2‑2所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130328094519236.jpg)

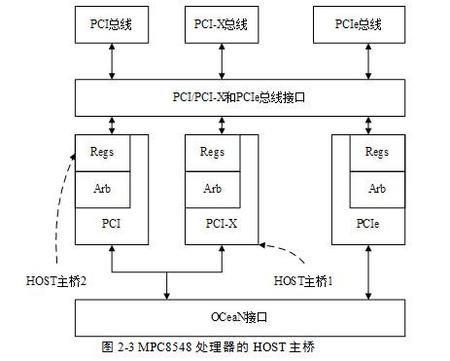
如上图所示，MPC8548处理器的L1 Cache在E500 V2内核中，而L2 Cache与FSB[1]直接相连，不属于E500内核。值得注意的是有些高端PowerPC处理器的L2 Cache也在CPU中，而L3 Cache与CCB总线直接相连。

在MPC8548处理器中，所有外部设备，如以太网控制器、DDR控制器和OCeaN连接的总线控制器都与SoC平台总线[2]直接连接。而SoC平台总线通过Cache共享一致性模块与FSB连接。

在MPC8548处理器中，具有一个32位的PCI总线控制器、一个64位的PCI/PCI-X总线控制器，还有多个PCIe总线控制器。MPC8548处理器使用OCeaN连接这些PCI、PCI-X和PCIe总线控制器。在MPC8548处理器系统中，PCI设备进行DMA操作时，首先通过OCeaN，之后经过SoC平台总线到达DDR控制器。

OCeaN是MPC8548处理器中连接快速外设使用的交叉互连总线，OCeaN不仅可以连接PCI、PCI-X和PCIe总线控制器，而且可以连接RapidIO[3]总线控制器。使用OCeaN进行互连的总线控制器可以直接通信，而不需要通过SoC平台总线。

如来自HOST主桥1的数据报文可以通过OCeaN直接发向HOST主桥2，而不需要将数据通过SoC平台总线，再进行转发，从而减轻了SoC平台总线的负担。OCeaN部件的拓扑结构如图2‑3所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130328094621789.jpg)

在MPC8548处理器中，有两个HOST主桥，分别是HOST主桥1和HOST主桥2，其中HOST主桥1可以支持PCI-X总线，而HOST主桥2只能支持PCI总线。此外该处理器还含有多个PCIe总线控制器。

本节仅介绍HOST主桥，即MPC8548处理器中的PCI总线控制器，而不介绍该处理器的PCIe总线控制器。因为从软件层面上看，MPC8548处理器的PCIe总线控制器与PCI/PCI-X总线控制器功能类似。

MPC8548处理器即可以作为PCI总线的HOST处理器，也可以作为PCI总线的从设备，本节仅讲述MPC8548处理器如何作为PCI总线的HOST处理器管理PCI总线树，而并不关心MPC8548处理器作为从设备的情况。

在MPC8548处理器的HOST主桥中，定义了一系列与系统软件相关的寄存器。本节将通过介绍这些寄存器，说明这个HOST主桥的功能。为节约篇幅，本节仅介绍与HOST主桥1相关的寄存器，HOST主桥2使用的寄存器与HOST主桥1使用的寄存器类似。

### 2.2.1 PCI设备配置空间的访问机制

PCI总线规定访问配置空间的总线事务，即配置读写总线事务，使用ID号进行寻址。PCI设备的ID号由总线号(Bus Number)、设备号(Device Number)和功能号(Function Number)组成。

其中总线号在HOST主桥遍历PCI总线树时确定。PCI总线可以使用PCI桥扩展PCI总线，并形成一颗PCI总线树。在一颗PCI总线树上，有几个PCI桥(包括HOST主桥)，就有几条PCI总线。在一颗PCI总线树中，总线号由系统软件决定，通常与HOST主桥直接相连的PCI总线编号为0，系统软件使用DFS(Depth-First Search)算法扫描PCI总线树上的所有PCI总线，并依次进行编号。

一条PCI总线的设备号由PCI设备的IDSEL信号与PCI总线地址线的连接关系确定，而功能号与PCI设备的具体设计相关。在一个PCI设备中最多有8个功能设备，而且每一个功能设备都有各自的PCI配置空间，而在绝大多数PCI设备中只有一个功能设备。HOST主桥使用寄存器号，访问PCI设备配置空间的某个寄存器。

在MPC8548处理器的HOST主桥中，与PCI设备配置空间相关的寄存器由CFG\_ADDR、CFG\_DATA和INT\_ACK寄存器组成。系统软件使用CFG\_ADDR和CFG\_DATA寄存器访问PCI设备的配置空间，而使用INT\_ACK寄存器访问挂接在PCI总线上的中断控制器的中断向量，这3个寄存器的地址偏移和属性如表2‑1所示。

表2‑1 PCI总线配置寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| Offset | 寄存器 | 属性 | 复位值 |
| 0x0\_8000 | CFG\_ADDR | 可读写 | 0x0000-0000 |
| 0x0\_8004 | CFG\_DATA | 可读写 | 0x0000-0000 |
| 0x0\_8008 | INT\_ACK | 只读 | 0x0000-0000 |

在MPC8548处理器中，所有内部寄存器都使用存储器映射方式进行寻址，并存放在以BASE\_ADDR[4]变量为起始地址的“1MB连续的物理地址空间”中。PowerPC处理器可以通过BASE\_ADDR+Offset的方式访问表2‑1中的寄存器。

MPC8548处理器使用CFG\_ADDR寄存器和CFG\_DATA寄存器访问PCI设备的配置空间，其中用CFG\_ADDR寄存器保存PCI设备的ID号和寄存器号，该寄存器的各个字段的详细说明如下所示。

* Enable位。当该位为1时，HOST主桥使能对PCI设备配置空间的访问，当HOST处理器对CFG\_DATA寄存器进行访问时，HOST主桥将对这个寄存器的访问转换为PCI配置读写总线事务并发送到PCI总线上。
* Bus Number字段记录PCI设备所在的总线号。
* Device Number字段记录PCI设备的设备号。
* Function Number字段记录PCI设备的功能号。
* Register Number字段记录PCI设备的配置寄存器号。

MPC8548处理器访问PCI设备的配置空间时，首先需要在CFG\_ADDR寄存器中设置这个PCI设备对应的总线号、设备号、功能号和寄存器号，然后使能Enable位。之后当MPC8548处理器对CFG\_DATA寄存器进行读写访问时， HOST主桥将这个存储器读写访问转换为PCI配置读写请求，并发送到PCI总线上。如果Enable位没有使能，处理器对CFG\_DATA的访问不过是一个普通的I/O访问，HOST主桥并不能将其转换为PCI配置读写请求。

HOST主桥根据CFG\_ADDR寄存器中的ID号，生成PCI配置读写总线事务，并将这个读写总线事务，通过ID译码方式发送到指定的PCI设备。PCI设备将接收来自配置写总线事务的数据，或者为配置读总线事务提供数据。

值得注意的是，在PowerPC处理器中，在CFG\_DATA寄存器中保存的数据采用大端方式进行编址，而PCI设备的配置寄存器采用小端编址，因此HOST主桥需要进行端模式转换。我们以源代码2‑1为例说明PowerPC处理器如何访问PCI配置空间。

源代码2‑1 PowerPC处理器访问PCI配置空间

|  |
| --- |
| stw r0, 0(r1)  ld r3, 0(r2) |

我们首先假设寄存器r1的初始值为BASE\_ADDR+0x0\_8000(即CFG\_ADDR寄存器的地址) ，寄存器r0的初始值为0x8000-0008，寄存器r2的初始值为BASE\_ADDR+0x0\_8004 (即CFG\_DATA寄存器的地址)，而指定PCI设备(总线号、设备号、功能号都为0)的配置寄存器的0x0B~0x08中的值为0x9988-7766。

这段源代码的执行步骤如下。

(1) 将r0寄存器赋值到r1寄存器所指向的地址空间中，即初始化CFG\_ADDR寄存器为0x8000-0008。

(2) 从r2寄存器所指向的地址空间中读取数据到r3寄存器中，即从CFG\_DATA寄存器中读取数据到r3寄存器。

在MPC8548处理器中，源代码2‑1执行完毕后，寄存器r3保存的值为0x6677-8899，而不是0x9988-6677。系统程序员在使用这个返回值时，一定要注意大小端模式的转换。值得注意的是，源代码2‑1可以使用lwbrx指令进行优化，该指令可以在读取数据的同时，进行大小端模式的转换。

处理器读取INT\_ACK寄存器时，HOST主桥将这个读操作转换为PCI总线中断响应事务。PCI总线中断响应事务的作用是通过PCI总线读取中断控制器的中断向量号，这样做的前提是中断控制器需要连接在PCI总线上。

PowerPC处理器使用的MPIC中断控制器不是挂接在PCI总线上，而是挂接在SoC平台总线上的，因此PCI总线提供的中断应答事务在这个处理器系统中并没有太大用途。但是并不排除某些PowerPC处理器系统使用了挂接在PCI总线上的中断控制器，比如PCI南桥芯片，此时PowerPC处理器系统需要使用中断应答事务读取PCI南桥中的中断控制器，以获取中断向量号。

### 2.2.2 存储器域地址空间到PCI总线域地址空间的转换

MPC8548处理器使用ATMU (Address Translation and Mapping Unit)寄存器组进行存储器域到PCI总线域，以及PCI总线域到存储器域的地址映射。ATMU寄存器组由两大组寄存器组成，分别为Outbound和Inbound寄存器组。其中Outbound寄存器组将存储器域的地址转换为PCI总线域的地址，而Inbound寄存器组将PCI总线域的地址转换为存储器域的地址。

在MPC8548处理器中，只有当CPU读写访问的地址范围在Outbound寄存器组管理的地址空间之内时，HOST主桥才能接收CPU的读写访问，并将CPU在存储器域上的读写访问转换为PCI总线域上的读写访问，然后才能对PCI设备进行读写操作。

如图2‑2所示，CPU对存储器域的地址访问，首先使用CCB总线事务，如果所访问的地址在Cache中命中时，则从Cache中直接获得数据，否则将从存储器域中获取数据。而在绝大多数情况下，外部设备使用的地址空间是不可Cache[5]的，所以在绝大多数情况之下，发向PCI设备的CCB总线事务并不会与Cache进行数据交换。

如果CCB总线事务使用的地址在HOST主桥的Outbound寄存器窗口中命中时，HOST主桥将接收这个CCB总线事务，并将其转换为PCI总线事务之后，再发送到PCI总线上。MPC8548处理器的每一个HOST主桥都提供了5个Outbound寄存器窗口来实现存储器域地址到PCI总线域地址的映射，其映射过程如图2‑4所示。

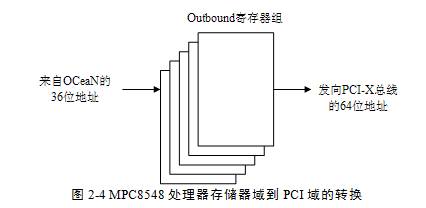
[](http://bbs.ednchina.com/images/attachments/201303/original/20130328094659715.jpg)  
在介绍MPC8548处理器如何使用Outbound寄存器组进行存储器域地址空间到PCI总线域地址空间的转换之前，本节将首先介绍Outbound寄存器组中的相应寄存器。Outbound寄存器组的地址偏移、属性和复位值如表2‑2所示。

表2‑2 PCI/X ATMU Outbound寄存器组

|  |  |  |  |
| --- | --- | --- | --- |
| 地址偏移 | 寄存器名 | 属性 | 复位值 |
| 0x0\_8C00/20/40/60/80 | POTARn | 可读写 | 0x0000-0000 |
| 0x0\_8C04/24/44/64/84 | POTEARn | 可读写 | 0x0000-0000 |
| 0x0\_8C28/48/68/88 | POWBARn | 可读写 | 0x0000-0000 |
| 0x0\_8C30/50/70/90 | POWARn | 可读写 | 0x0000-0000 |

#### 1 POTARn和POTEARn寄存器

在POTARn和POTEARn寄存器中保存当前Outbound窗口在PCI总线域中的64位地址空间的基地址。这两个寄存器的主要字段如下。

* POTARn寄存器的TEA字段，第0~11位，保存PCI总线地址空间的43~32位。
* POTARn寄存器的TA字段，第12~31位，保存PCI总线地址空间的31~12位[6] 。
* POTEARn寄存器的TEA字段，第12~31位，保存PCI总线地址空间的63~44位。

#### 2 POWBARn寄存器和POWARn寄存器

而POWBARn寄存器保存当前Outbound窗口在存储器域中的36位地址空间的基地址[7] ，其主要字段如下。

* WBEA字段保存存储器域地址的第0~3位。
* WBA字段保存存储器域地址的第4~23 [8]位。
* POWARn寄存器描述Outbound窗口的属性，其主要字段如下。
* EN位，第0位。该位是Outbound窗口的使能位，为1表示当前Outbound寄存器组描述的存储器地址空间到PCI总线地址空间的映射关系有效；为0表示无效。
* RTT字段，第12~15位，该字段描述当前窗口的读传送类型，为0b0100表示存储器读，为0b1000表示I/O读。
* WTT字段，第16~19位，该字段描述当前窗口的写传送类型，为0b0100表示存储器写，为0b1000表示I/O写。在PCIe总线控制器中，RTT字段和WTT字段还可以支持对配置空间的读写操作。
* OWS字段，第26~31位，该字段描述当前窗口的大小，Outbound窗口的大小在4KB~64GB之间，其值为2OWS+1。

[1]MPC8548也将FSB称为CCB(Core Complex Bus)。

[2]PowerPC处理器并没有公开其SoC平台总线的设计规范。ARM提出的AMBA总线是一条典型的SoC平台总线。

[3]RapidIO总线由Mercury Computer System和Motorola Semiconductor(目前的Freescale)共同提出，用于解决背板互连的一条外部总线。

[4]在MPC8548处理器中，BASE\_ADDR存放在CCSRBAR寄存器中。

[5]PCI设备使用的ROM空间可以是“可Cache”的地址空间。

[6]POTARn寄存器没有保存PCI总线的11~0位，因为Outbound窗口大小至少为4KB。

[7]MPC8548处理器的物理地址为36位。注意在PowerPC处理器中，第0位是地址的最高位。

|  |
| --- |
| **2.2 HOST主桥(2)** |
| 发布时间：2013-03-28 10:11:54 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

#### 3 使用Outbound寄存器访问PCI总线地址空间

MPC8548处理器使用Outbound寄存器组访问PCI总线地址空间的步骤如下。

(1) 首先MPC8548处理器需要将程序使用的32位有效地址EA (Effective Address)转换为41位的虚拟地址VA (Virtual Address)。E500 V2内核不能关闭MMU(Memory Management Unit)，因此不能直接访问物理地址。

(2) MPC8548处理器通过MMU将41位的虚拟地址转换为36位的物理地址。在E500 V2内核中，物理地址是36位(缺省是32位，需要使能)。

(3) 检查LAWBAR和LAWAR寄存器，判断当前36位的物理地址是否属于PCI总线空间。在MPC8548中定义了一组LAWBAR和LAWAR寄存器对，每一对寄存器描述当前物理空间是与PCI总线、PCIe总线、DDR还是RapidIO空间对应。该组寄存器的详细说明见MPC8548 PowerQUICC III™ Integrated Host Processor Family Reference Manual。如果CPU访问的空间为PCI总线空间，则执行第(4)步，否则处理器将不会访问PCI地址空间。

(4) 判断当前36位物理地址是否在POWBARn寄存器1~4描述的窗口中，如果在则将36位的处理器物理地址通过寄存器POTARn和POTEARn转换为64位的PCI总线地址，然后HOST主桥将来自处理器的读写请求发送到PCI总线上；如果不在POWBARn寄存器1~4描述的窗口中，POWBAR0寄存器作为缺省窗口，接管这个存储器访问，并使用寄存器POTAR0和POTEAR0，将处理器物理地址转换为PCI总线地址，当然在正常设计中很少出现这种情况。

许多系统软件，将Outbound窗口两边的寄存器使用“直接相等”的方法进行映射，将存储器域的地址与PCI总线地址设为相同的值。但是系统软件程序员务必注意这个存储器地址与PCI总线地址是分属于存储器域与PCI总线域的，这两个值虽然相等，但是所代表的地址并不相同，一个属于存储器域，而另一个属于PCI总线域。

### 2.2.3 PCI总线域地址空间到存储器域地址空间的转换

MPC8548处理器使用Inbound寄存器组将PCI总线域地址转换为存储器域的地址。PCI设备进行DMA读写时，只有访问的地址在Inbound窗口中时，HOST主桥才能接收这些读写请求，并将其转发到存储器控制器。MPC8548处理器提供了3组Inbound寄存器，即提供3个Inbound寄存器窗口，实现PCI总线地址到存储器地址的反向映射。

从PCI设备的角度上看，PCI设备访问存储器域的地址空间时，首先需要通过Inbound窗口将PCI总线地址转换为存储器域的地址；而从处理器的角度上看，处理器必须要将存储器地址通过Inbound寄存器组反向映射为PCI总线地址空间，才能被PCI设备访问。

PCI设备只能使用PCI总线地址访问PCI总线域的地址空间。HOST主桥将这段地址空间通过Inbound窗口转换为存储器域的地址之后，PCI设备才能访问存储器域地址空间。这个地址转换过程如图2‑5所示。

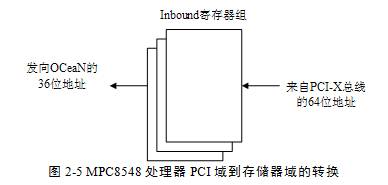
[](http://bbs.ednchina.com/images/attachments/201303/original/20130328095901203.jpg)  
在介绍MPC8548处理器如何使用Inbound寄存器组进行PCI总线域地址空间到存储器域地址空间的转换之前，我们首先简要介绍Inbound寄存器组中的相应寄存器。该组寄存器的地址偏移、属性和复位值如表2‑3所示。

表2‑3 PCI/X ATMU Inbound寄存器组

|  |  |  |  |
| --- | --- | --- | --- |
| Offset | 寄存器名 | 属性 | 复位值 |
| 0x0\_8DA0/C0/E0 | PITARn | 可读写 | 0x0000-0000 |
| 0x0\_8DA8/C8/E8 | PIWBARn | 可读写 | 0x0000-0000 |
| 0x0\_8DAC/CC | PIWBEARn | 可读写 | 0x0000-0000 |
| 0x0\_8DB0/D0/F0 | PIWARn | 可读写 | 0x0000-0000 |

值得注意的是，Inbound寄存器组除了可以进行PCI总线地址空间到存储器域地址空间的转换之外，还可以转换分属不同PCI总线域的地址空间，以支持PCI总线的Peer-to-Peer数据传送方式。

#### 1 PITARn寄存器

PITARn寄存器保存当前Inbound窗口在存储器域中的36位地址空间的基地址，其地址窗口的大小至少为4KB，因此在该寄存器中仅存放存储器域地址的第0~23位，该寄存器的其主要字段如下所示。

* TEA字段存放存储器地址空间的第0~3位。
* TA字段存放存储器地址空间的第4~23位。

#### 2 PIWBARn和PIWBEARn寄存器

* PIWBARn和PIWBEARn寄存器保存当前Inbound窗口在PCI总线域中的64位地址空间的基地址的第63~12位，Inbound窗口使用的最小地址空间为4KB，因此在这两个寄存器中不含有PCI总线地址空间的第11~0位。这两个寄存器的主要字段如下所示。
* PIWBARn寄存器的BEA字段存放PCI总线地址空间的第43~32位。
* PIWBARn寄存器的BA字段存放PCI总线地址空间的第31~12位。
* PIWBEARn寄存器的BEA字段存放PCI总线地址空间的第63~44位。

#### 3 PIWARn寄存器

PIWARn寄存器描述当前Inbound窗口的属性，该寄存器由以下位和字段组成。

* EN位，第0位。该位是Inbound窗口的使能位，为1表示当前Inbound寄存器组描述的存储器地址空间到PCI总线地址空间的映射关系有效；为0表示无效。
* PF位，第2位。该位为1表示当前Inbound窗口描述的存储区域支持预读；为0表示不支持预读。
* TGI字段，第8~11位。该字段为0b0010表示当前Inbound窗口描述的存储区域属于PCIe总线域地址空间；为0b1100表示当前Inbound窗口描述的存储区域属于RapidIO总线域地址空间。该字段对于OCeaN实现不同域间的报文转发非常重要，如果当前Inbound窗口的TGI字段为0b0010，此时PCI总线上的设备可以使用该Inbound窗口，通过OCeaN直接读取PCIe总线的地址空间，而不需要经过SoC平台总线。如果TGI字段为0b1111表示Inbound窗口描述的存储器区域属于主存储器地址空间，这也是最常用的方式。使用该字段可以实现HOST主桥的Peer-to-Peer数据传送方式。
* RTT字段和WTT字段，分别为该寄存器的第12~15位和第16~19位。Inbound窗口的RTT/WTT字段的含义与Outbound窗口的RTT/WTT字段基本类似。只是在Inbound窗口中可以规定PCI设备访问主存储器时，是否需要进行Cache一致性操作(Cache Lock and Allocate)，在进行DMA写操作时，数据是否可以直接进入到Cache中。该字段是PowerPC处理器对PCI总线规范的有效补充，由于该字段的存在，PowerPC处理器的PCI设备可以将数据直接写入Cache，也可以视情况决定DMA操作是否需要进行Cache共享一致性操作。
* IWS字段，第26~31位。该字段描述当前窗口的大小，Inbound窗口的大小在4KB~16GB之间，其值为2IWS+1。

#### 4 使用Inbound寄存器组进行DMA操作

PCI设备使用DMA操作访问主存储器空间，或者访问其他PCI总线域地址空间时，需要通过Inbound窗口，其步骤如下。

(1) PCI设备在访问主存储器空间时，将首先检查当前PCI总线地址是否在PIWBARn和PIWBEARn寄存器描述的窗口中。如果在这个窗口中，则将这个PCI总线地址通过PITARn寄存器转换为存储器域的地址或者其他PCI总线域的地址；如果不在将禁止本次访问。

(2) 如果PCI设备访问的是存储器地址空间，HOST主桥将来自PCI总线的读写请求发送到存储器空间，进行存储器读写操作，并根据Inbound寄存器组的RTT/WTT位决定是否需要进行Cache一致性操作，或者将数据直接写入到Cache中。

结合Outbound寄存器组，可以发现PCI总线地址空间与存储器地址空间是有一定联系的。如果存储器域地址空间被Inbound寄存器组反向映射到PCI空间，这个存储器地址具有两个地址，一个是在存储器域的地址，一个是在PCI总线域的地址；同理PCI总线空间的地址如果使用Outbound寄存器映射到寄存器地址空间，这个PCI总线地址也具有两个地址，一个是在PCI总线域的地址，一个是在存储器域的地址。

能够被处理器和PCI总线同时访问的地址空间，一定在PCI总线域和存储器域中都存在地址映射。再次强调，绝大多数操作系统将同一个空间的PCI总线域地址和存储器地址设为相同的值，但是这两个相同的值所代表的含义不同。

由此可以看出，如果MPC8548处理器的某段存储器区域没有在Inbound窗口中定义时，PCI设备将不能使用DMA机制访问这段存储器空间；同理如果PCI设备的空间不在Outbound窗口，HOST处理器也不能访问这段PCI地址空间。

在绝大多数PowerPC处理器系统中，PCI设备地址空间都在HOST主桥的Outbound窗口中建立了映射；而MPC8548处理器可以选择将哪些主存储器空间共享给PCI设备，从而对主存储器空间进行保护。

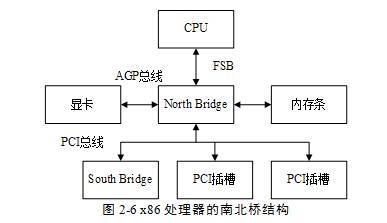
### 2.2.4 x86处理器的HOST主桥

x86处理器使用南北桥结构连接CPU和PCI设备。其中北桥(North Bridge)连接快速设备，如显卡、和内存条，并推出PCI总线，HOST主桥包含在北桥中。而南桥(South Bridge)连接慢速设备。x86处理器使用的南北桥结构如图2‑6所示。

Intel使用南北桥概念统一PC架构。但是从体系结构的角度上看，南北桥架构并不重要，北桥中存放的主要部件不过是存储器控制器、显卡控制器和HOST主桥而已，而南桥存放的是一些慢速设备，如ISA总线和中断控制器等。

不同的处理器系统集成这些组成部件的方式并不相同，如PowerPC、MIPS和ARM处理器系统通常将CPU和主要外部设备都集成到一颗芯片中，组成一颗基于SoC架构的处理器系统。这些集成方式并不重要，每一个处理器系统都有其针对的应用领域，不同应用领域的需求对处理器系统的集成方式有较大的影响。Intel采用的南北桥架构针对x86处理器的应用领域而设计，并不能说采用这种结构一定比MPC8548处理器中即含有HOST-to-PCI主桥也含有HOST-to-PCIe主桥更为合理。

在许多嵌入式处理器系统中，即含有PCI设备也含有PCIe设备，为此MPC8548处理器同时提供了PCI总线和PCIe总线接口，在这个处理器系统中，PCI设备可以与PCI总线直接相连，而PCIe设备可以与PCIe总线直接相连，因此并不需要使用PCIe桥扩展PCI总线，从而在一定程度上简化了嵌入式系统的设计。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130328100008969.jpg)

嵌入式系统所面对的应用千姿百态，进行芯片设计时所要考虑的因素相对较多，因而在某种程度上为设计带来了一些难度。而x86处理器系统所面对的应用领域针对个人PC\*\*务器，向前兼容和通用性显得更加重要。在多数情况下，一个通用处理器系统的设计难过专用处理器系统的设计，Intel为此付出了极大的代价。

在一些相对较老的北桥中，如Intel 440系列芯片组中包含了HOST主桥，从系统软件的角度上看HOST-to-PCI主桥实现的功能与HOST-to-PCIe主桥实现的功能相近。本节仅简单介绍Intel的HOST-to-PCI主桥如何产生PCI的配置周期，有关Intel HOST-to-PCIe主桥[9]的详细信息参见第5章。

x86处理器定义了两个I/O端口寄存器，分别为CONFIG\_ADDRESS和CONFIG\_DATA寄存器，其地址为0xCF8和0xCFC。x86处理器使用这两个I/O端口访问PCI设备的配置空间。PCI总线规范也以这个两个寄存器为例，说明处理器如何访问PCI设备的配置空间。其中CONFIG\_ADDRESS寄存器存放PCI设备的ID号，而CONFIG\_DATA寄存器存放进行配置读写的数据。

CONFIG\_ADDRESS寄存器与PowerPC处理器中的CFG\_ADDR寄存器的使用方法类似，而CONFIG\_DATA寄存器与PowerPC处理器中的CFG\_DATA寄存器的使用方法类似。CONFIG\_ADDRESS寄存器的结构如图2‑7所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130328100054859.jpg)

CONFIG\_ADDRESS寄存器的各个字段和位的说明如下所示，

* Enable位，第31位。该位为1时，对CONFIG\_DATA寄存器进行读写时将引发PCI总线的配置周期。
* Bus Number字段，第23~16位，记录PCI设备的总线号。
* Device Number字段，第15~11位，记录PCI设备的设备号。
* Function Number字段，第10~8位，记录PCI设备的功能号。
* Register Number字段，第7~2位，记录PCI设备的寄存器号。

当x86处理器对CONFIG\_DATA寄存器进行I/O读写访问，且CONFIG\_ADDR寄存器的Enable位为1时，HOST主桥将这个I/O读写访问转换为PCI配置读写总线事务，然后发送到PCI总线上，PCI总线根据保存在CONFIG\_ADDR寄存器中的ID号，将PCI配置读写请求发送到指定PCI设备的指定配置寄存器中。

x86处理器使用小端地址模式，因此从CONFIG\_DATA寄存器中读出的数据不需要进行模式转换，这点和PowerPC处理器不同，此外x86处理器的HOST主桥也实现了存储器域到PCI总线域的地址转换，但是这个概念在x86处理器中并不明晰。

本书将在第5章以HOST-to-PCIe主桥为例，详细介绍Intel处理器的存储器地址与PCI总线地址的转换关系，而在本节不对x86处理器的HOST主桥做进一步说明。x86处理器系统的升级速度较快，目前在x86的处理器体系结构中，已很难发现HOST主桥的身影。

目前Intel对南北桥架构进行了升级，其中北桥被升级为MCH(Memory Controller Hub)，而南桥被升级为ICH(I/O Controller Hub)。x86处理器系统在MCH中集成了存储器控制器、显卡芯片和HOST-to-PCIe主桥，并通过Hub Link与ICH相连；而在ICH中集成了一些相对低速总线接口，如AC’97、LPC(Low Pin Count)、IDE和USB总线，当然也包括一些低带宽的PCIe总线接口。

在Intel最新的Nehelem[10]处理器系统中，MCH被一份为二，存储器控制器和图形控制器已经与CPU内核集成在一个Die中，而MCH剩余的部分与ICH合并成为PCH(Peripheral Controller Hub)。但是从体系结构的角度上看，这些升级与整合并不重要。

目前Intel在Menlow[11]平台基础上，计划推出基于SoC架构的x86处理器，以进军手持设备市场。在基于SoC构架的x86处理器中将逐渐淡化Chipset的概念，其拓扑结构与典型的SoC处理器，如ARM和PowerPC处理器，较为类似。 

[8]WBA字段并没有保存存储器域的第24~35位地址，因为Outbound窗口大小至少为4KB。

[9]这个HOST-to-PCIe主桥也是RC(Root Complex)的一部分。

[10] Nehelem处理器也被称为Core i7处理器。

[11] Menlow平台于2008年3月发布，其目标应用为MID(Mobile Internet Device)设备。Menlow平台基于低功耗处理器内核Atom。

|  |
| --- |
| **2.3 PCI桥与PCI设备的配置空间** |
| 发布时间：2013-03-29 10:38:45 |
|  |

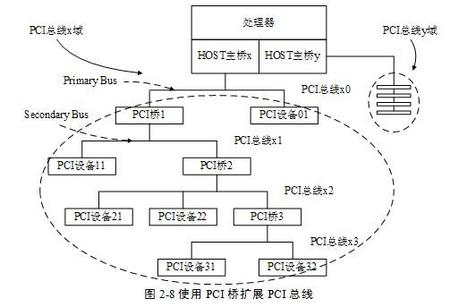
|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI设备都有独立的配置空间，HOST主桥通过配置读写总线事务访问这段空间。PCI总线规定了三种类型的PCI配置空间，分别是PCI Agent设备使用的配置空间，PCI桥使用的配置空间和Cardbus桥片使用的配置空间。

本节重点介绍PCI Agent和PCI桥使用的配置空间，而并不介绍Cardbus桥片使用的配置空间。值得注意的是，在PCI设备配置空间中出现的地址都是PCI总线地址，属于PCI总线域地址空间。

**2.3.1 PCI桥**

PCI桥的引入使PCI总线极具扩展性，也极大地增加了PCI总线的复杂度。PCI总线的电气特性决定了在一条PCI总线上挂接的负载有限，当PCI总线需要连接多个PCI设备时，需要使用PCI桥进行总线扩展，扩展出的PCI总线可以连接其他PCI设备，包括PCI桥。在一颗PCI总线树上，最多可以挂接256个PCI设备，包括PCI桥。PCI桥在PCI总线树中的位置如图2‑8所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130329101108624.jpg)

PCI桥作为一个特殊的PCI设备，具有独立的配置空间。但是PCI桥配置空间的定义与PCI Agent设备有所不同。PCI桥的配置空间可以管理其下PCI总线子树的PCI设备，并可以优化这些PCI设备通过PCI桥的数据访问。PCI桥的配置空间在系统软件遍历PCI总线树时配置，系统软件不需要专门的驱动程序设置PCI桥的使用方法，这也是PCI桥被称为透明桥的主要原因。

在某些处理器系统中，还有一类PCI桥，叫做非透明桥。非透明桥不是PCI总线定义的标准桥片，但是在使用PCI总线挂接另外一个处理器系统时非常有用，非透明桥片的主要作用是连接两个不同的PCI总线域，进而连接两个处理器系统，本章将在第2.5节中详细介绍PCI非透明桥。

使用PCI桥可以扩展出新的PCI总线，在这条PCI总线上还可以继续挂接多个PCI设备。PCI桥跨接在两个PCI总线之间，其中距离HOST主桥较近的PCI总线被称为该桥片上游总线(Primary Bus)，距离HOST主桥较远的PCI总线被称为该桥片的下游总线(Secondary Bus)。如图2‑8所示，PCI桥1的上游总线为PCI总线x0，而PCI桥1的下游总线为PCI总线x1。这两条总线间的数据通信需要通过PCI桥1。

通过PCI桥连接的PCI总线属于同一个PCI总线域，在图2‑8中，PCI桥1、2和3连接的PCI总线都属于PCI总线x域。在这些PCI总线域上的设备可以通过PCI桥直接进行数据交换而不需要进行地址转换；而分属不同PCI总线域的设备间的通信需要进行地址转换，如与PCI非透明桥两端连接的设备之间的通信。

如图2‑8所示，每一个PCI总线的下方都可以挂接一个到多个PCI桥，每一个PCI桥都可以推出一条新的PCI总线。在同一条PCI总线上的设备之间的数据交换不会影响其他PCI总线。如PCI设备21与PCI设备22之间的数据通信仅占用PCI总线x2的带宽，而不会影响PCI总线x0、x1与x3，这也是引入PCI桥的另一个重要原因。

由图2‑8我们还可以发现PCI总线可以通过PCI桥组成一个胖树结构，其中每一个桥片都是父节点，而PCI Agent设备只能是子节点。当PCI桥出现故障时，其下的设备不能将数据传递给上游总线，但是并不影响PCI桥下游设备间的通信。当PCI桥1出现故障时，PCI设备11、PCI设备21和PCI设备22将不能与PCI设备01和存储器进行通信，但是PCI设备21和PCI设备22之间的通信可以正常进行。

使用PCI桥可以扩展一条新的PCI总线，但是不能扩展新的PCI总线域。如果当前系统使用32位的PCI总线地址，那么这个系统的PCI总线域的地址空间为4GB大小，在这个总线域上的所有设备将共享这个4GB大小的空间。如在PCI总线x域上的PCI桥1、PCI设备01、PCI设备11、PCI桥2、PCI设备21和PCI设备22等都将共享一个4GB大小的空间。再次强调这个4GB空间是PCI总线x域的“PCI总线地址空间”，和存储器域地址空间和PCI总线y域没有直接联系。

处理器系统可以通过HOST主桥扩展出新的PCI总线域，如MPC8548处理器的HOST主桥x和y可以扩展出两个PCI总线域x和y。这两个PCI总线域x和y之间的PCI空间在正常情况下不能直接进行数据交换，但是PowerPC处理器可以通过设置PIWARn寄存器的TGI字段使得不同PCI总线域的设备直接通信，详见第2.2.3节。

许多处理器系统使用的PCI设备较少，因而并不需要使用PCI桥。因此在这些处理器系统中，PCI设备都是直接挂接在HOST主桥上，而不需要使用PCI桥扩展新的PCI总线。即便如此读者也需要深入理解PCI桥的知识。

PCI桥对于理解PCI和PCIe总线都非常重要。在PCIe总线中，虽然在物理结构上并不含有PCI桥，但是与PCI桥相关的知识在PCIe总线中无处不在，比如在PCIe总线的Switch中，每一个端口都与一个虚拟PCI桥对应，Switch使用这个虚拟PCI桥管理其下PCI总线子树的地址空间。

**2.3.2 PCI Agent设备的配置空间**

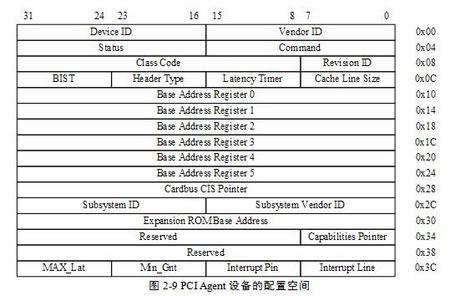
在一个具体的处理器应用中，PCI设备通常将PCI配置信息存放在E2PROM中。PCI设备进行上电初始化时，将E2PROM中的信息读到PCI设备的配置空间中作为初始值。这个过程由硬件逻辑完成，绝大多数PCI设备使用这种方式初始化其配置空间。

读者可能会对这种机制产生一个疑问，如果系统软件在PCI设备将E2PROM中的信息读到配置空间之前，就开始操作配置空间，会不会带来问题？因为此时PCI设备的初始值并不“正确”，仅仅是PCI设备使用的复位值。

读者的这种担心是多余的，因为PCI设备在配置寄存器没有初始化完毕之前，即E2PROM中的内容没有导入PCI设备的配置空间之前，可以使用PCI总线规定的“Retry”周期使HOST主桥在合适的时机重新发起配置读写请求。

在x86处理器中，系统软件使用CONFIG\_ADDR和CONFIG\_DATA寄存器，读取PCI设备配置空间的这些初始化信息，然后根据处理器系统的实际情况使用DFS算法，初始化处理器系统中所有PCI设备的配置空间。

在PCI Agent设备的配置空间中包含了许多寄存器，这些寄存器决定了该设备在PCI总线中的使用方法，本节不会全部介绍这些寄存器，因为系统软件只对部分配置寄存器感兴趣。PCI Agent设备使用的配置空间如图2‑9所示。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130329101143606.jpg)

在PCI Agent设备配置空间中包含的寄存器如下所示。

(1) Device ID和Vendor ID寄存器

这两个寄存器的值由PCISIG分配，只读。其中Vendor ID代表PCI设备的生产厂商，而Device ID代表这个厂商所生产的具体设备。如Intel公司的基于82571EB芯片的系列网卡，其Vendor ID为0x8086[1]，而Device ID为0x105E[2]。

(2) Revision ID和Class Code寄存器

这两个寄存器只读。其中Revision ID寄存器记载PCI设备的版本号。该寄存器可以被认为是Device ID寄存器的扩展。

(3) Header Type寄存器

该寄存器只读，由8位组成。

* 第7位为1表示当前PCI设备是多功能设备，为0表示为单功能设备。
* 第6~0位表示当前配置空间的类型，为0表示该设备使用PCI Agent设备的配置空间，普通PCI设备都使用这种配置头；为1表示使用PCI桥的配置空间，PCI桥使用这种配置头；为2表示使用Cardbus桥片的配置空间，Card Bus桥片使用这种配置头，本篇对这类配置头不感兴趣。

系统软件需要使用该寄存器区分不同类型的PCI配置空间，该寄存器的初始化必须与PCI设备的实际情况对应，而且必须为一个合法值。

(4) Cache Line Size寄存器

该寄存器记录HOST处理器使用的Cache行长度。在PCI总线中和Cache相关的总线事务，如存储器写并无效和Cache多行读等总线事务需要使用这个寄存器。值得注意的是，该寄存器由系统软件设置，但是在PCI设备的运行过程中，只有其硬件逻辑才会使用该寄存器，比如PCI设备的硬件逻辑需要得知处理器系统Cache行的大小，才能进行存储器写并无效总线事务，单行读和多行读总线事务。

如果PCI设备不支持与Cache相关的总线事务，系统软件可以不设置该寄存器，此时该寄存器为初始值0x00。对于PCIe设备，该寄存器的值无意义，因为PCIe设备在进行数据传送时，在其报文中含有一次数据传送的大小，PCIe总线控制器可以使用这个“大小”，判断数据区域与Cache行的对应关系。

(5) Subsystem ID和Subsystem Vendor ID寄存器

这两个寄存器和Device ID和Vendor ID类似，也是记录PCI设备的生产厂商和设备名称。但是这两个寄存器和Device ID与Vendor ID寄存器略有不同。下文以一个实例说明Subsystem ID和Subsystem Vendor ID的用途。

Xilinx公司在FGPA中集成了一个PCIe总线接口的IP核，即LogiCORE。用户可以使用LogiCORE设计各种各样基于PCIe总线的设备，但是这些设备的Device ID都是0x10EE，而Vendor ID为0x0007[3]。

(6) Expansion ROM base address寄存器

有些PCI设备在处理器还没有运行操作系统之前，就需要完成基本的初始化设置，比如显卡、键盘和硬盘等设备。为了实现这个“预先执行”功能，PCI设备需要提供一段ROM程序，而处理器在初始化过程中将运行这段ROM程序，初始化这些PCI设备。Expansion ROM base address记载这段ROM程序的基地址。

(7) Capabilities Pointer寄存器

在PCI设备中，该寄存器是可选的，但是在PCI-X和PCIe设备中必须支持这个寄存器，Capabilities Pointer寄存器存放Capabilities寄存器组的基地址，PCI设备使用Capabilities寄存器组存放一些与PCI设备相关的扩展配置信息。该组寄存器的详细说明见第4.3节。

(8) Interrupt Line寄存器

这个寄存器是系统软件对PCI设备进行配置时写入的，该寄存器记录当前PCI设备使用的中断向量号，设备驱动程序可以通过这个寄存器，判断当前PCI设备使用处理器系统中的哪个中断向量号，并将驱动程序的中断服务例程注册到操作系统中[4]。

该寄存器由系统软件初始化，其保存的值与8259A中断控制器相关，该寄存器的值也是由PCI设备与8259A中断控制器的连接关系决定的。如果在一个处理器系统中，没有使用8259A中断控制器管理PCI设备的中断，则该寄存器中的数据并没有意义。

在多数PowerPC处理器系统中，并不使用8259A中断控制器管理PCI设备的中断请求，因此该寄存器没有意义。即使在x86处理器系统中，如果使用I/O APIC中断控制器，该寄存器保存的内容仍然无效。目前在绝大多数处理器系统中，并没有使用该寄存器存放PCI设备使用的中断向量号。

(9) Interrupt Pin寄存器

这个寄存器保存PCI设备使用的中断引脚，PCI总线提供了四个中断引脚INTA#、INTB#、INTC#和INTD#。Interrupt Pin寄存器为1时表示使用INTA#引脚向中断控制器提交中断请求，为2表示使用INTB#，为3表示使用INTC#，为4表示使用INTD#。

如果PCI设备只有一个子设备时，该设备只能使用INTA#；如果有多个子设备时，可以使用INTB~D#信号。如果PCI设备不使用这些中断引脚，向处理器提交中断请求时，该寄存器的值必须为0。值得注意的是，虽然在PCIe设备中并不含有INTA~D#信号，但是依然可以使用该寄存器，因为PCIe设备可以使用INTx中断消息，模拟PCI设备的INTA~D#信号，详见第6.3.4节。

(10) Base Address Register 0~5寄存器

该组寄存器简称为BAR寄存器，BAR寄存器保存PCI设备使用的地址空间的基地址，该基地址保存的是该设备在PCI总线域中的地址。其中每一个设备最多可以有6个基址空间，但多数设备不会使用这么多组地址空间。

在PCI设备复位之后，该寄存器将存放PCI设备需要使用的基址空间大小，这段空间是I/O空间还是存储器空间[5]，如果是存储器空间该空间是否可预取，有关PCI总线预读机制的详细说明见第3.4.5节。

系统软件对PCI总线进行配置时，首先获得BAR寄存器中的初始化信息，之后根据处理器系统的配置，将合理的基地址写入相应的BAR寄存器中。系统软件还可以使用该寄存器，获得PCI设备使用的BAR空间的长度，其方法是向BAR寄存器写入0xFFFF-FFFF，之后再读取该寄存器。

处理器访问PCI设备的BAR空间时，需要使用BAR寄存器提供的基地址。值得注意的是，处理器使用存储器域的地址，而BAR寄存器存放PCI总线域的地址。因此处理器系统并不能直接使用“BAR寄存器+偏移”的方式访问PCI设备的寄存器空间，而需要将PCI总线域的地址转换为存储器域的地址。

如果x86处理器系统使能了IOMMU后，这两个地址也并不一定相等，因此处理器系统直接使用这个PCI总线域的物理地址，并不能确保访问PCI设备的BAR空间的正确性。除此之外在Linux系统中，ioremap函数的输入参数为存储器域的物理地址，而不能使用PCI总线域的物理地址。

而在pci\_devàresource[bar].start参数中保存的地址已经经过PCI总线域到存储器域的地址转换，因此在编写Linux系统的设备驱动程序时，需要使用pci\_devàresource[bar].start参数中的物理地址，然后再经过ioremap函数将物理地址转换为“存储器域”的虚拟地址。

(11) Command寄存器

该寄存器为PCI设备的命令寄存器，该寄存器在初始化时，其值为0，此时这个PCI设备除了能够接收配置请求总线事务之外，不能接收任何存储器或者I/O请求。系统软件需要合理设置该寄存器之后，才能访问该设备的存储器或者I/O空间。在Linux系统中，设备驱动程序调用pci\_enable\_device函数，使能该寄存器的I/O和Memory Space位之后，才能访问该设备的存储器或者I/O地址空间。

(12) Status寄存器

该寄存器的绝大多数位都是只读位，保存PCI设备的状态。

(13) Latency Timer寄存器

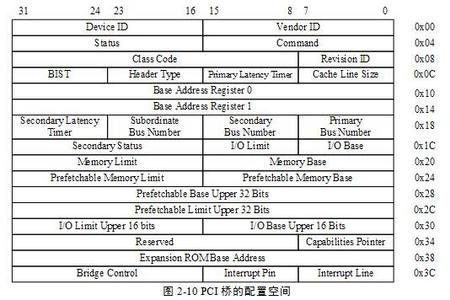
在PCI总线中，多个设备共享同一条总线带宽。该寄存器用来控制PCI设备占用PCI总线的时间，当PCI设备获得总线使用权，并使能Frame#信号后，Latency Timer寄存器将递减，当该寄存器归零后，该设备将使用超时机制停止[6]对当前总线的使用。

如果当前总线事务为Memeory Write and Invalidate时，需要保证对一个完整Cache行的操作结束后才能停止当前总线事务。对于多数PCI设备而言，该寄存器的值为32或者64，以保证一次突发传送的基本单位为一个Cache行。

PCIe设备不需要使用该寄存器，该寄存器的值必须为0。因为PCIe总线的仲裁方法与PCI总线不同，使用的连接方法也与PCI总线不同。

**2.3.3 PCI桥的配置空间**

PCI桥使用的配置空间的寄存器如图2‑10所示。PCI桥作为一个PCI设备，使用的许多配置寄存器与PCI Agent的寄存器是类似的，如Device ID、Vendor ID、Status、Command、Interrupt Pin、Interrupt Line寄存器等，本节不再重复介绍这些寄存器。下文将重点介绍在PCI桥中与PCI Agent的配置空间不相同的寄存器。

[](http://bbs.ednchina.com/images/attachments/201303/original/20130329101220555.jpg)

与PCI Agent设备不同，在PCI桥中只含有两组BAR寄存器，Base Address Register 0~1寄存器。这两组寄存器与PCI Agent设备配置空间的对应寄存器的含义一致。但是在PCI桥中，这两个寄存器是可选的。如果在PCI桥中不存在私有寄存器，那么可以不使用这组寄存器设置BAR空间。

在大多数PCI桥中都不存在私有寄存器，操作系统也不需要为PCI桥提供专门的驱动程序，这也是这类桥被称为透明桥的原因。如果在PCI桥中不存在私有空间时，PCI桥将这两个BAR寄存器初始化为0。在PCI桥的配置空间中使用两个BAR寄存器的目的是这两个32位的寄存器可以组成一个64位地址空间。

在PCI桥的配置空间中，有许多寄存器是PCI桥所特有的。PCI桥除了作为PCI设备之外，还需要管理其下连接的PCI总线子树使用的各类资源，即Secondary Bus所连接PCI总线子树使用的资源。这些资源包括存储器、I/O地址空间和总线号。

在PCI桥中，与Secondary bus相关的寄存器包括两大类。一类寄存器管理Secondary Bus之下PCI子树的总线号，如Secondary和Subordinate Bus Number寄存器；另一类寄存器管理下游PCI总线的I/O和存储器地址空间，如I/O和Memory Limit、I/O和Memory Base寄存器。在PCI桥中还使用Primary Bus寄存器保存上游的PCI总线号。

其中存储器地址空间还分为可预读空间和不可预读空间，Prefetchable Memory Limit和Prefetchable Memory Base寄存器管理可预读空间，而Memory Limit、Memory Base管理不可预读空间。在PCI体系结构中，除了了ROM地址空间之外，PCI设备使用的地址空间大多都是不可预读的。

(1) Subordinate Bus Number、Secondary Bus Number和Primary Bus Number寄存器

PCI桥可以管理其下的PCI总线子树。其中Subordinate Bus Number寄存器存放当前PCI子树中，编号最大的PCI总线号。而Secondary Bus Number寄存器存放当前PCI桥Secondary Bus使用的总线号，这个PCI总线号也是该PCI桥管理的PCI子树中编号最小的PCI总线号。因此一个PCI桥能够管理的PCI总线号在Secondary Bus Number~Subordinate Bus Number之间。这两个寄存器的值由系统软件遍历PCI总线树时设置。

Primary Bus Number寄存器存放该PCI桥上游的PCI总线号，该寄存器可读写。Primary Bus Number、Subordinate Bus Number和Secondary Bus Number寄存器在初始化时必须为0，系统软件将根据这几个寄存器是否为0，判断PCI桥是否被配置过。

不同的操作系统使用不同的Bootloader引导，有的Bootloader可能会对PCI总线树进行遍历，此时操作系统可以不再重新遍历PCI总线树。在x86处理器系统中，BIOS会遍历处理器系统中的所有PCI总线树，操作系统可以直接使用BIOS的结果，也可以重新遍历PCI总线树。而PowerPC处理器系统中的Bootloader，如U-Boot并没有完全遍历PCI总线树，此时操作系统必须重新遍历PCI总线树。

(2) Secondary Status寄存器

该寄存器的含义与PCI Agent配置空间的Status寄存器的含义相近，PCI桥的Secondary Status寄存器记录Secondary Bus的状态，而不是PCI桥作为PCI设备时使用的状态。在PCI桥配置空间中还存在一个Status寄存器，该寄存器保存PCI桥作为PCI设备时的状态。

(3) Secondary Latency Timer寄存器

该寄存器的含义与PCI Agent配置空间的Latency Timer寄存器的含义相近，PCI桥的Secondary Latency Timer寄存器管理Secondary Bus的超时机制，即PCI桥发向下游的总线事务；在PCI桥配置空间中还存在一个Latency Timer寄存器，该寄存器管理PCI桥发向上游的总线事务。

(4) I/O Limit和I/O Base寄存器

在PCI桥管理的PCI子树中包含许多PCI设备，而这些PCI设备可能会使用I/O地址空间。PCI桥使用这两个寄存器，存放PCI子树中所有设备使用的I/O地址空间集合的基地址和大小。

(5) Memory Limit和Memory Base寄存器

在PCI桥管理的PCI子树中有许多PCI设备，这些PCI设备可能会使用存储器地址空间。这两个寄存器存放所有这些PCI设备使用的，存储器地址空间集合的基地址和大小，PCI桥规定这个空间的大小至少为1MB。

(6) Prefetchable Memory Limit和Prefetchable Memory Base寄存器

在PCI桥管理的PCI子树中有许多PCI设备，如果这些PCI设备支持预读，则需要从PCI桥的可预读空间中获取地址空间。PCI桥的这两个寄存器存放这些PCI设备使用的，可预取存储器空间的基地址和大小。

如果PCI桥不支持预读，则其下支持预读的PCI设备需要从Memory Base寄存器为基地址的存储器空间中获取地址空间。如果PCI桥支持预读，其下的PCI设备需要根据情况，决定使用可预读空间，还是不可预读空间。PCI总线建议PCI设备支持预读，但是支持预读的PCI设备并不多见。

(7) I/O Base Upper 16 Bits and I/O Limit Upper 16寄存器

如果PCI桥仅支持16位的I/O端口，这组寄存器只读，且其值为0。如果PCI桥支持32位I/O端口，这组寄存器可以提供I/O端口的高16位地址。

(8) Bridge Control Register。

该寄存器用来管理PCI桥的Secondary Bus，其主要位的描述如下。

* Secondary Bus Reset位，第6位，可读写。当该位为1时，将使用下游总线提供的RST#信号复位与PCI桥的下游总线连接的PCI设备。通常情况下与PCI桥下游总线连接的PCI设备，其复位信号需要与PCI桥提供的RST#信号连接，而不能与HOST主桥提供的RST#信号连接。
* Primary Discard Timer位，第8位，可读写。PCI桥支持Delayed传送方式，当PCI桥的Primary总线上的主设备使用Delayed方式进行数据传递时，PCI桥使用Retry周期结束Primary总线的Non-Posted数据请求，并将这个Non-Posted数据请求转换为Delayed数据请求，之后主设备需要择时重试相同的Non-Posted数据请求。当该位为1时，表示在Primary Bus上的主设备需要在210个时钟周期之内重试这个数据请求，为0时，表示主设备需要在215个时钟周期之内重试这个数据请求，否则PCI桥将丢弃Delayed数据请求。
* Secondary Discard Timer位，第9位，可读写。当该位为1时，表示在Secondary Bus上的主设备需要在210个时钟周期之内重试这个数据请求，为0时，表示主设备需要在215个时钟周期之内重试这个数据请求，如果主设备在规定的时间内没有进行重试时，PCI桥将丢弃Delayed数据请求。

|  |
| --- |
| **2.4 PCI总线的配置** |
| 发布时间：2013-04-02 18:03:56 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI总线定义了两类配置请求，一个是Type 00h配置请求，另一个是Type 01h配置请求。PCI总线使用这些配置请求访问PCI总线树上的设备配置空间，包括PCI桥和PCI Agent设备的配置空间。

其中HOST主桥或者PCI桥使用Type 00h配置请求，访问与HOST主桥或者PCI桥直接相连的PCI Agent设备或者PCI桥[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftn1)；而HOST主桥或者PCI桥使用Type 01h配置请求，需要至少穿越一个PCI桥，访问没有与其直接相连的PCI Agent设备或者PCI桥。如图2‑8所示，HOST主桥可以使用Type 00h配置请求访问PCI设备01，而使用Type 01h配置请求通过PCI桥1、2或者3转换为Type 00h配置请求之后，访问PCI总线树上的PCI设备11、21、22、31和32[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftn2)。

当x86处理器对CONFIG\_DATA寄存器进行读写操作时，HOST主桥将决定向PCI总线发送Type 00h配置请求还是Type 01h配置请求。在PCI总线事务的地址周期中，这两种配置请求总线事务的不同反映在PCI总线的AD[31:0]信号线上。

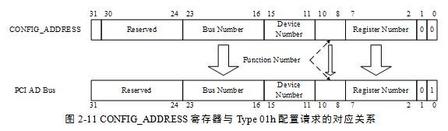
值得注意的是，PCIe总线还可以使用ECAM(Enhanced Configuration Access Mechanism)机制访问PCIe设备的扩展配置空间，使用这种方式可以访问PCIe设备256B~4KB之间的扩展配置空间。但是本节仅介绍如何使用CONFIG\_ADDRESS和CONFIG\_FATA寄存器产生Type 00h和Type 01h配置请求。有关ECAM机制的详细说明见第5.3.2节。

处理器首先将目标PCI设备的ID号保存在CONFIG\_ADDRESS寄存器中，之后HOST主桥根据该寄存器的Bus Number字段，决定是产生Type 00h配置请求，还是Type 01h配置请求。当Bus Number字段为0时，将产生Type 00h配置请求，因为与HOST主桥直接相连的总线号为0；大于0时，将产生Type 01h配置请求。

### 2.4.1 Type 01h和Type 00h配置请求

本节首先介绍Type 01h配置请求，并从PCI总线使用的信号线的角度上，讲述HOST主桥如何生成Type 01配置请求。在PCI总线中，只有PCI桥能够接收Type 01h配置请求。Type 01h配置请求不能直接发向最终的PCI Agent设备，而只能由PCI桥将其转换为Type 01h继续发向其他PCI桥，或者转换为Type 00h配置请求发向PCI Agent设备。PCI桥还可以将Type 01h配置请求转换为Special Cycle总线事务(HOST主桥也可以实现该功能)，本节对这种情况不做介绍。

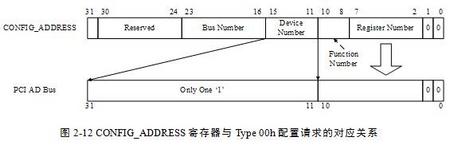
在地址周期中，HOST主桥使用配置读写总线事务，将CONFIG\_ADDRESS寄存器的内容拷贝到PCI总线的AD[31:0]信号线中。CONFIG\_ADDRESS寄存器与Type 01h配置请求的对应关系如图2‑11所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130402180103879.jpg)

从图2‑11中可以发现，CONFIG\_ADDRESS寄存器的内容基本上是原封不动的拷贝到PCI总线的AD[31:0]信号线上的[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftn3)。其中CONFIG\_ADDRESS的Enable位不被拷贝，而AD总线的第0位为必须为1，表示当前配置请求是Type 01h。

当PCI总线接收到Type 01配置请求时，将寻找合适的PCI桥[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftn4)接收这个配置信息。如果这个配置请求是直接发向PCI桥下的PCI设备时，PCI桥将接收个Type 01配置请求，并将其转换为Type 00h配置请求；否则PCI桥将当前Type 01h配置请求原封不动的传递给下一级PCI总线。

如果HOST主桥或者PCI桥发起的是Type 00h配置请求，CONFIG\_ADDRESS寄存器与AD[31:0]的转换如图2‑12所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130402180137406.jpg)

此时处理器对CONFIG\_DATA寄存器进行读写时，处理器将CONFIG\_ADDRESS寄存器中的Function Number和Register Number字段拷贝到PCI的AD总线的第10~2位；将AD总线的第1~0位赋值为0b00。PCI总线在配置请求总线事务的地址周期根据AD[1:0]判断当前配置请求是Type 00h还是Type 01h，如果AD[1:0]等于0b00表示是Type 00h配置请求，如果AD[1:0]等于0b01表示是Type 01h配置请求。

而AD[31:11]与CONFIG\_ADDRESS的Device Number字段有关，在Type 00h配置请求的地址周期中，AD[31:11]位有且只有一位为1，其中AD[31:11]的每一位选通一个PCI设备的配置空间。如第1.2.2节所述，PCI设备配置空间的片选信号是IDSEL，因此AD[31:11]将与PCI设备的IDSEL信号对应相连。

当以下两种请求之一满足时，HOST主桥或者PCI桥将生成Type 00h配置头，并将其发送到指定的PCI总线上。

(1) CONFIG\_ADDRESS寄存器的Bus Number字段为0时，处理器访问CONFIG\_DATA寄存器时，HOST主桥将直接向PCI总线0发出Type 00h配置请求。因为与HOST主桥直接相连的PCI总线号为0，此时表示HOST主桥需要访问与其直接相连的PCI设备。

(2) 当PCI桥收到Type 01h配置头时，将检查Type 01配置头的Bus Number字段，如果这个Bus Number与PCI桥的Secondary Bus Number相同，则将这个Type 01配置头转换为Type 00h配置头，并发送到该PCI桥的Secondary总线上。

### 2.4.2 PCI总线配置请求的转换原则

当CONFIG\_ADDRESS寄存器的Enable位为1，系统软件访问CONFIG\_DATA寄存器时，HOST主桥可以产生两类PCI总线配置读写总线事务，分别为Type 00h和Type 01h配置读写总线事务。在配置读写总线事务的地址周期和数据周期中，CONFIG\_ADDRESS和CONFIG\_DATA寄存器中的数据将被放置到PCI总线的AD总线上。其中Type 00h和Type 01h配置读写总线事务映射到AD总线的数据并不相同。

其中Type 00h配置请求可以直接读取PCI Agent设备的配置空间，而Type 01h配置请求在通过PCI桥时，最终将被转换为Type 00h配置请求，并读取PCI Agent设备的配置寄存器。本节重点讲述PCI桥如何将Type 01h配置请求转换为Type 00h配置请求。

首先Type 00h配置请求不会被转换成Type 01h配置请求，因为Type 00h配置请求是发向最终PCI Agent设备，这些PCI Agent设备不会转发这些配置请求。

当CONFIG\_ADDRESS寄存器的Bus Number字段为0时，处理器对CONFIG\_DATA寄存器操作时，HOST主桥将直接产生Type 00h配置请求，挂接在PCI总线0上的某个设备将通过ID译码接收这个Type 00h配置请求，并对配置寄存器进行读写操作。如果PCI总线上没有设备接收这个Type 00h配置请求，将引发Master Abort，详情见PCI总线规范，本节对此不做进一步说明。

如果CONFIG\_ADDRESS寄存器的Bus Number字段为n(n≠0)，即访问的PCI设备不是直接挂接在PCI总线0上的，此时HOST主桥对CONFIG\_DATA寄存器操作时，将产生Type 01h配置请求，PCI总线0将遍历所有在这条总线上的PCI桥，确定由哪个PCI桥接收这个Type 01h配置请求。

如果n大于等于某个PCI桥的Secondary Bus Number寄存器，而且小于等于Subordinate Bus number寄存器，那么这个PCI桥将接收在当前PCI总线上的Type 01配置请求，并采用以下规则进行递归处理。

(1) 开始。

(2) 遍历当前PCI总线的所有PCI桥。

(3) 如果n等于某个PCI桥的Secondary Bus Number寄存器，说明这个Type 01配置请求的目标设备直接连接在该PCI桥的Secondary bus上。此时PCI桥将Type 01配置请求转换为Type 00h配置请求，并将这个配置请求发送到PCI桥的Secondary Bus上，Secondary Bus上的某个设备将响应这个Type 00h配置请求，并与HOST主桥进行配置信息的交换，转(5)。

(4) 如果n大于PCI桥的Secondary Bus Number寄存器，而且小于等于PCI桥的Subordinate Bus number寄存器，说明这个Type 01配置请求的目标设备不与该PCI桥的Secondary Bus直接相连，但是由这个PCI桥下游总线上的某个PCI桥管理。此时PCI桥将首先认领这个Type 01配置请求，并将其转发到Secondary Bus，转(2)。

(5) 结束。

下文将举例说明PCI总线配置请求的转换原则，并以图2‑8为例说明处理器如何访问PCI设备01和PCI设备31的配置空间。PCI设备01直接与HOST主桥相连，因此HOST主桥可以使用Type 00h配置请求访问该设备。

而HOST主桥需要经过多级PCI桥才能访问PCI设备31，因此HOST主桥需要首先使用Type 01h配置请求，之后通过PCI桥1、2和3将Type 01h配置请求转换为Type 00h配置请求，最终访问PCI设备31。

#### 1 PCI设备01

这种情况较易处理，当HOST处理器访问PCI设备01的配置空间时，发现PCI设备01与HOST主桥直接相连，所以将直接使用Type 00h配置请求访问该设备的配置空间，具体步骤如下。

首先HOST处理器将CONFIG\_ADDRESS寄存器的Enable位置1，Bus Number号置为0，并对该寄存器的Device、Function和Register Number字段赋值。当处理器对CONFIG\_DATA寄存器访问时，HOST主桥将存放在CONFIG\_ADDRESS寄存器中的数值，转换为Type 00h配置请求，并发送到PCI总线0上，PCI设备01将接收这个Type 00h配置请求，并与处理器进行配置信息交换。

#### 2 PCI设备31

HOST处理器对PCI设备31进行配置读写时，需要通过HOST主桥、PCI桥1、2和3，最终到达PCI设备31。

当处理器访问PCI设备31时，首先将CONFIG\_ADDRESS寄存器的Enable位置1，Bus Number字段置为3，并对Device、Function和Register Number字段赋值。之后当处理器对CONFIG\_DATA寄存器进行读写访问时，HOST主桥、PCI桥1、2和3将按照以下步骤进行处理，最后PCI设备31将接收这个配置请求。

(1) HOST主桥发现Bus Number字段的值为3，该总线号并不是与HOST主桥直接相连的PCI总线的Bus Number，所以HOST主桥将处理器对CONFIG\_DATA寄存器的读写访问直接转换为Type 01h配置请求，并将这个配置请求发送到PCI总线0上。PCI总线规定Type 01h配置请求只能由PCI桥负责处理。

(2) 在PCI总线0上，PCI桥1的Secondary Bus Number为1而Subordinate Bus Number为3。而1< Bus Number <= 3，所以PCI桥1将接收来自PCI总线0的Type 01h配置请求，并将这个配置请求直接下推到PCI总线1。

(3) 在PCI总线1上，PCI桥2的Secondary Bus Number为2而Subordinate Bus Number为3。而1< Bus Number <= 3，所以PCI桥2将接收来自PCI总线0的Type 01h配置请求，并将这个配置请求直接下推到PCI总线2。

(4) 在PCI总线2上，PCI桥3的Secondary Bus Number为3，因此PCI桥3将“来自PCI总线2的Type 01h配置请求”转换为Type 00h配置请求，并将其下推到PCI总线3。PCI总线规定，如果PCI桥的Secondary Bus Number与Type 01h配置请求中包含的Bus Number相同时，该PCI桥将接收的Type 01h配置请求转换为Type 00h配置请求，然后再发向其Secondary Bus。

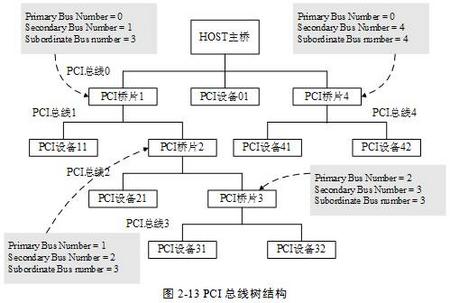
(5) 在PCI总线3上，有两个设备PCI设备31和PCI设备32。在这两个设备中，必然有一个设备将要响应这个Type 00h配置请求，从而完成整个配置请求周期。本篇在第2.4.1节中，讨论了究竟是PCI设备31还是PCI设备32接收这个配置请求，这个问题涉及PCI总线如何分配PCI设备使用的设备号。

### 2.4.3 PCI总线树Bus号的初始化

在一个处理器系统中，每一个HOST主桥都推出一颗PCI总线树。在一颗PCI总线树中有多少个PCI桥(包括HOST主桥)，就含有多少条PCI总线。系统软件在遍历当前PCI总线树时，需要首先对这些PCI总线进行编号，即初始化PCI桥的Primary、Secondary和Subordinate Bus Number寄存器。

在一个处理器系统中，一般将与HOST主桥直接相连的PCI总线被命名为PCI总线0。然后系统软件使用DFS(Depth First Search)算法，依次对其他PCI总线进行编号。值得注意的是，与HOST主桥直接相连的PCI总线，其编号都为0，因此当处理器系统中存在多个HOST主桥时，将有多个编号为0的PCI总线，但是这些编号为0的PCI总线分属不同的PCI总线域，其含义并不相同。

在一个处理器系统中，PCI总线树的结构如图2‑13所示。当然在一个实际的处理器系统中，很少会出现这样复杂的PCI总线树结构，本节采用这个结构的目的是便于说明PCI总线号的分配过程。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130402180207875.jpg)

在PCI总线中，系统软件使用深度优先DFS算法对PCI总线树进行遍历，DFS算法和广度优先BFS(Breadth First Search)算法是遍历树型结构的常用算法。与BFS算法相比，DFS算法的空间复杂度较低，因此绝大多数系统系统在遍历PCI总线树时，都使用DFS算法而不是BFS算法。

DFS是搜索算法的一种，其实现机制是沿着一颗树的深度遍历各个节点，并尽可能深地搜索树的分支，DFS的算法为线性时间复杂度，适合对拓扑结构未知的树进行遍历。在一个处理器系统的初始化阶段，PCI总线树的拓扑结构是未知的，适合使用DFS算法进行遍历。下文以图2‑13为例，说明系统软件如何使用DFS算法，分配PCI总线号，并初始化PCI桥中的Primary Bus Number、Secondary Bus Number和Subordinate Bus number寄存器。所谓DFS算法是指按照深度优先的原则遍历PCI胖树，其步骤如下。

(1) HOST主桥扫描PCI总线0上的设备。系统软件首先忽略所有这条总线上的PCI Agent设备，因为在这些设备之下不会挂接新的PCI总线。例如PCI设备01下不可能挂接新的PCI总线。

(2) HOST主桥首先发现PCI桥1，并将PCI桥1的Secondary Bus命名为PCI总线1。系统软件将初始化PCI桥1的配置空间，将PCI桥1的Primary Bus Number寄存器赋值为0，而将Secondary Bus Number寄存器赋值为1，即PCI桥1的上游PCI总线号为0，而下游PCI总线号为1。

(3) 扫描PCI总线1，发现PCI桥2，并将PCI桥2的Secondary Bus命名为PCI总线2。系统软件将初始化PCI桥2的配置空间，将PCI桥2的Primary Bus Number寄存器赋值为1，而将Secondary Bus Number寄存器赋值为2。

(4) 扫描PCI总线2，发现PCI桥3，并将PCI桥3的Secondary Bus命名为PCI总线3。系统软件将初始化PCI桥3的配置空间，将PCI桥3的Primary Bus Number寄存器赋值为2，而将Secondary Bus Number寄存器赋值为3。

(5) 扫描PCI总线3，没有发现任何PCI桥，这表示PCI总线3下不可能有新的总线，此时系统软件将PCI桥3的Subordinate Bus number寄存器赋值为3。系统软件在完成PCI总线3的扫描后，将回退到PCI总线3的上一级总线，即PCI总线2，继续进行扫描。

(6) 在重新扫描PCI总线2时，系统软件发现PCI总线2上除了PCI桥3之外没有发现新的PCI桥，而PCI桥3之下的所有设备已经完成了扫描过程，此时系统软件将PCI桥2的Subordinate Bus number寄存器赋值为3。继续回退到PCI总线1。

(7) PCI总线1上除了PCI桥2外，没有其他桥片，于是继续回退到PCI总线0，并将PCI桥1的Subordinate Bus number寄存器赋值为3。

(8) 在PCI总线0上，系统软件扫描到PCI桥4，则首先将PCI桥4的Primary Bus Number寄存器赋值为0，而将Secondary Bus Number寄存器赋值为4，即PCI桥1的上游PCI总线号为0，而下游PCI总线号为4。

(9) 系统软件发现PCI总线4上没有任何PCI桥，将结束对PCI总线4的扫描，并将PCI桥4的Subordinate Bus number寄存器赋值为4，之后回退到PCI总线4的上游总线，即PCI总线0继续进行扫描。

(10) 系统软件发现在PCI总线0上的两个桥片PCI总线0和PCI总线4都已完成扫描后，将结束对PCI总线的DFS遍历全过程。

从以上算法可以看出，PCI桥的Primary Bus和Secondary Bus号的分配在遍历PCI总线树的过程中从上向下分配，而Subordinate Bus号是从下向上分配的，因为只有确定了一个PCI桥之下究竟有多少条PCI总线后，才能初始化该PCI桥的Subordinate Bus号。

### 2.4.4 PCI总线Device号的分配

一条PCI总线会挂接各种各样的PCI设备，而每一个PCI设备在PCI总线下具有唯一的设备号。系统软件通过总线号和设备号定位一个PCI设备之后，才能访问这个PCI设备的配置寄存器。值得注意的是，系统软件使用“地址寻址方式”访问PCI设备的存储器和I/O地址空间，这与访问配置空间使用的“ID寻址方式”不同。

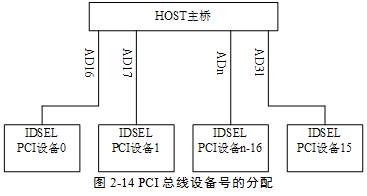
PCI设备的IDSEL信号与PCI总线的AD[31:0]信号的连接关系决定了该设备在这条PCI总线的设备号。如上文所述，每一个PCI设备都使用独立的IDSEL信号，该信号将与PCI总线的AD[31:0]信号连接，IDSEL信号的含义见第1.2.2节。

在此我们简要回顾PCI的配置读写事务使用的时序。如图1‑3所示，PCI总线事务由一个地址周期加若干个数据周期组成。在进行配置读写请求总线事务时，C/BE#信号线的值在地址周期中为0x1010或者为0x1011，表示当前总线事务为配置读或者配置写请求。此时出现在AD[31:0]总线上的值并不是目标设备的PCI总线地址，而是目标设备的ID号，这与PCI总线进行I/O或者存储器请求时不同，因为PCI总线使用ID号而不是PCI总线地址对配置空间进行访问。

如图2‑12所示，在配置读写总线事务的地址周期中，AD[10:0]信号已经被Function Number和Register Number使用，因此PCI设备的IDSEL只能与AD[31:11]信号连接。

认真的读者一定可以发现在CONFIG\_ADDRESS寄存器中Device Number字段一共有5位可以表示32个设备，而AD[31:11]只有21位，显然在这两者之间无法建立一一对应的映射关系。因此在一条PCI总线上如果有21个以上的PCI设备，那么总是有几个设备无法与AD[31:11]信号线连接，从而PCI总线无法访问这些设备。因为PCI总线在配置请求的地址周期中，只能使用第31~11这些AD信号，所以在一条总线上最多也只能挂接21个PCI设备。这21个设备可能是从0到20，也可能是从11到31排列。从而系统软件在遍历PCI总线时，还是需要从0到31遍历整条PCI总线。

在实际的应用中，一条PCI总线能够挂接21个设备已经足够了，实际上由于PCI总线的负载能力有限，即便总线频率为33MHz的情况下，在一条PCI总线中最多也只能挂接10个负载，一条PCI总线所能挂接的负载详见表1‑1。AD信号线与PCI设备IDSEL线的连接关系如图2‑14所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130402180238319.jpg)

PCI总线推荐了一种Device Number字段与AD[31:16]之间的映射关系。其中PCI设备0与Device Number字段的0b00000对应；PCI设备1与Device Number字段的0b00001对应，并以此类推，PCI设备15与Device Number字段的0b01111对应。

在这种映射关系之下，一条PCI总线中，与信号线AD16相连的PCI设备其设备号为0；与信号线AD17相连的PCI设备其设备号为1；以此类推，与信号线AD31相连的PCI设备其设备号为15。在Type 00h配置请求中，设备号并没有像Function Number和Register Number那样以编码的形式出现在AD总线上，而是与AD信号一一对应，如图2‑12所示。

这里有一个原则需要读者注意，就是对PCI设备的配置寄存器进行访问时，一定要有确定的Bus Number、Device Number、Function Number和Register Number，这“四元组”缺一不可。在Type 00h配置请求中，Device Number由AD[31:11]信号线与PCI设备IDSEL信号的连接关系确定；Function Number保存在AD[10:8]字段中；而Register Number保存在AD[7:0]字段中；在Type 01h配置请求中，也有完整的四元组信息。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftnref1) 此时PCI桥作为一个PCI设备，接收访问其配置空间的读写请求。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftnref2) 最终Type 01h配置请求将会被转换为Type 00h配置请求，然后访问PCI Agent设备。

[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftnref3) Type 01h配置头信息存在于PCI总线事务的地址周期中。

[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011776.HTM#_ftnref4) PCI桥根据Subordinate Bus Number和Secondary Bus Number寄存器，决定是否接收当前配置请求。

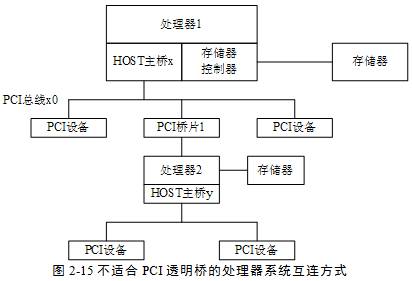
|  |
| --- |
| **2.5 非透明PCI桥** |
| 发布时间：2013-04-08 14:09:12 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI桥规范定义了透明桥的实现规则，本篇在第2.3.1节中详细介绍了这种桥片。通过透明桥，处理器系统可以以HOST主桥为根节点，建立一颗PCI总线树，在这个树上的PCI设备共享同一个PCI总线域上的地址空间。

但是在某些场合下PCI透明桥并不适用。在图2‑15所示的处理器系统中存在两个处理器，此时使用PCI桥1连接处理器2并不利于整个处理器系统的配置与管理。我们假定PCI总线使用32位地址空间，而处理器1和处理器2所使用的存储器大小都为2GB，同时我们假定处理器1和处理器2使用的存储器都可以被PCI设备访问。

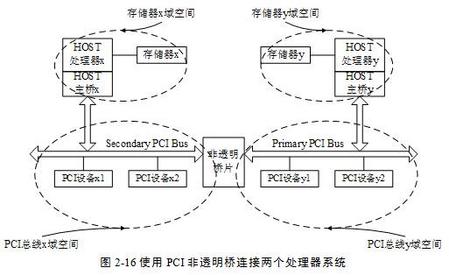
此时处理器1和2使用的存储器空间必须映射到PCI总线的地址空间中，而32位的PCI总线只能提供4GB地址空间，此时PCI总线x0的地址空间将全部被处理器1和2的存储器空间占用，而没有额外的空间分配给PCI设备。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130408140243602.jpg)

此外有些处理器不能作为PCI Agent设备，因此不能直接连接到PCI桥上，比如x86处理器就无法作为PCI Agent设备，因此使用PCI透明桥无法将两个x86处理器直接相连。如果处理器2有两个以上的PCI接口，其中一个可以与PCI桥1相连(此时处理器2将作为PCI Agent设备)，而另一个作为HOST主桥y连接PCI设备。此时HOST主桥y挂接的PCI设备将无法被处理器1直接访问。

使用透明桥也不便于解决处理器1与处理器2间的地址冲突，如图2‑15所示的处理器系统，如果处理器1和2都将各自的存储器映射到PCI总线地址空间中，有可能会出现地址冲突。虽然PowerPC处理器可以使用Inbound寄存器，将存储器地址空间映射到不同的PCI总线地址空间中，但是不是所有的处理器都具有这种映射机制。许多处理器的存储器地址与PCI总线地址使用了“简单相等”这种映射方法，如果PCI总线连接了两个这样的处理器，将不可避免地出现PCI总线地址的映射冲突。

采用非透明桥将有效解决以上这些问题，非透明桥并不是PCI总线定义的标准桥片，但是这类桥片在连接两个处理器系统中得到了广泛的应用。一个使用非透明桥连接两个处理器系统的实例如图2‑16所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130408140316409.jpg)

使用非透明PCI桥可以方便地连接两个处理器系统。从上图中我们可以发现非透明桥可以将PCI总线x域与PCI总线y域进行隔离。值得注意的是，非透明PCI桥的作用是对不同PCI总线域地址空间进行隔离，而不是隔离存储器域地址空间。而HOST主桥的作用才是将存储器域与PCI总线域进行隔离。

非透明PCI桥可以连接两条独立的PCI总线，一条被称为Secondary PCI总线，另一条被称为Primary PCI总线，但是这两条总线没有从属关系，两边是对等的[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftn1)。从处理器x的角度上看，与非透明PCI桥右边连接的总线叫Secondary PCI总线；而从处理器y的角度上看，非透明PCI桥左边连接的总线叫Secondary PCI总线。

HOST处理器x和PCI设备可以通过非透明PCI桥，直接访问PCI总线y域的地址空间，并通过HOST主桥y与访问存储器y；HOST处理器y和PCI设备也可以通过非透明PCI桥，直接访问PCI总线x域的地址空间，并通过HOST主桥x访问存储器x。为此非透明PCI桥需要对分属不同PCI总线域的地址空间进行转换。

目前有许多厂商可以提供非透明PCI桥的芯片，在具体实现上各有差异，但是其基本原理类似，下文以Intel 21555为例说明非透明PCI桥。值得注意的是，在PCIe体系结构中，也存在非透明PCI桥的概念。

### 2.5.1 Intel 21555中的配置寄存器

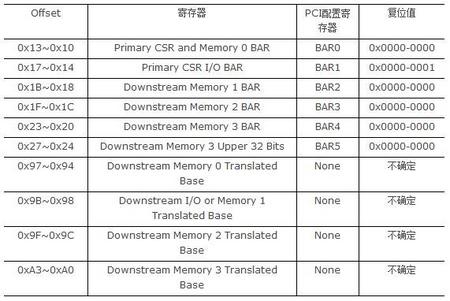
Intel 21555非透明PCI桥源于DEC21554[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftn2)，并在此基础上做了一些改动。Intel 21555桥片与其他透明桥在系统中的位置相同。如图2‑16所示，这个桥片一边与Primary PCI总线相连，另一边与Secondary PCI总线相连。

在Intel 21555桥片中，包含两个PCI 设备配置空间，分别是Primary PCI总线配置空间和Secondary PCI总线配置空间，处理器可以使用Type 00h配置请求访问这些配置空间。在大多数情况之下，在Primary PCI总线上的HOST处理器管理Primary PCI配置空间；在Secondary PCI总线上的HOST处理器管理Secondary PCI配置空间[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftn3)。

在Intel 21555桥片中，还有一组私有寄存器CSR(Control and Status Register)，系统软件使用这组寄存器对非透明桥进行管理并获得桥片的一些信息，这组寄存器可以被映射成为PCI总线的存储器地址空间或者I/O地址空间。

本章仅介绍Primary PCI总线这一边的配置寄存器，Secondary PCI总线的配置寄存器虽然与Primary PCI总线的这些寄存器略有不同，但是基本对等，因此本节对此不做介绍。Primary PCI总线的主要寄存器如表2‑6所示。

表2‑6 Primary PCI总线的配置寄存器

[](http://bbs.ednchina.com/images/attachments/201304/original/20130408140349374.jpg)

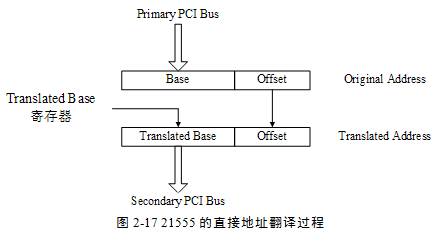
从表2‑6中，我们可以发现Primary PCI总线的这些配置寄存器共分为两组，一组寄存器与PCI设备的配置寄存器的BAR0~5对应，这些寄存器与标准PCI配置寄存器BAR0~5的功能相同；另一组寄存器是Translated Base寄存器，这组寄存器的主要作用是将来自Primary PCI总线的数据访问转换到Secondary PCI总线。

其中BAR0~5寄存器在系统初始化时由Primary PCI总线上的HOST处理器进行配置，配置过程与PCI总线上的普通设备完全相同。只是Intel 21555规定，BAR0只能映射为32位存储器空间。

CSR寄存器可以根据需要映射在BAR0空间中，此时BAR0空间最小为4KB。CSR寄存器也可以根据需要使用BAR1寄存器映射为I/O地址空间，同时BAR1寄存器还可以映射其他I/O空间；BAR2~3只能映射为32位存储器地址空间；而BAR4~5用来映射64位的存储器地址空间。

对于Primary PCI总线，所有BAR0~5寄存器映射的地址空间都将占用Primary PCI总线域，然而这些地址空间中所对应的数据并不在Primary PCI总线域中，而是在Secondary PCI总线域中。Translated Base寄存器实现不同PCI总线域地址空间的转换，Intel 21555将不同PCI总线域地址空间的转换过程称为“地址翻译”。

Intel 21555支持两种地址翻译方法，一个是直接地址翻译，一个是查表翻译。Primary PCI总线的BAR空间只支持直接地址翻译，而Secondary PCI总线的Memory 2 BAR空间支持查表翻译，本节仅介绍直接地址翻译方法，对查表翻译有兴趣的读者请阅读Intel 21555的数据手册[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftn4)。直接地址翻译过程如图2‑17所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130408140417578.jpg)

当Primary PCI总线对非透明桥21555的BAR0~5地址空间进行数据请求时，这个数据请求将被转换为对Secondary PCI总线的数据请求。Translated Base寄存器将完成这个地址翻译过程，下节将结合实例说明这个直接地址翻译过程。

### 2.5.2 通过非透明桥片进行数据传递

下文以图2‑16中处理器x访问处理器y存储器地址空间的实例，说明非透明桥21555如何将PCI总线x域与PCI总线y域联系在一起。

处理器x需要访问处理器y的存储器空间之前，需要做一些必要的准备工作。

(1) 首先确定由哪一个BAR寄存器空间映射处理器y的存储器地址空间。本节假定使用BAR2寄存器映射处理器y的存储器地址空间。

(2) BAR2寄存器使用Downstream Memory 2 Translated Base寄存器，将来自Primary PCI总线的访问转换为对Secondary PCI总线地址空间的访问。其中Downstream Memory 2 Translated Base寄存器可以由处理器x或者处理器y根据需要进行设置。

假定处理器x和y的HOST主桥使用“直接相等”策略，建立存储器域与PCI总线域间的映射；而处理器x使用BAR2地址空间访问处理器y存储器空间0x1000-000~0x1FFF-FFFF；处理器x的系统软件事先将BAR2寄存器设置完毕。处理器x访问处理器y的这段存储器空间的步骤如下，读者可参考图2‑18理解这一步骤。

(1) 首先处理器x访问在处理器x域中，且与非透明桥的BAR2空间相对应的存储器地址空间。

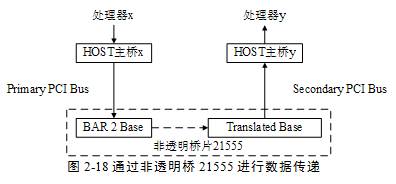
(2) HOST主桥将进行存储器域到PCI总线域的转换，并将这个请求发送到Primary PCI总线上。

(3) 非透明桥发现这个数据请求发向BAR2地址空间，则接收这个数据请求，并在桥片中暂存这个数据请求。

(4) 非透明桥根据Downstream Memory 2 Translated Base寄存器的内容，按照图2‑17所示的规则进行地址转换。假设Downstream Memory 2 Translated Base寄存器的基地址被预先设置为0x1000-0000，大小为256MB(这个物理地址属于处理器y的主存储器地址空间)。

(5) 经过非透明桥的转换后，这个数据请求将穿越非透明桥，从Primary PCI总线域进入Secondary PCI总线域，然后访问处理器y的基地址为0x1000-0000的存储器区域。

(6) 处理器y的HOST主桥接收这个存储器访问请求，并最终将数据请求发向处理器y的存储器中。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130408140706853.jpg)

非透明桥21555除了可以支持存储器到存储器之间的数据传递，还支持PCI总线域到存储器域，以及PCI总线域之间的数据传递，此外非透明桥21555还可以通过I2O和Doorbell寄存器进行Primary PCI总线与Secondary PCI总线之间的中断信号传递。本节对这部分内容不做进一步介绍。

非透明桥有效解决了使用PCI总线连接两个处理器存在的问题，因而得到了广泛的应用。在PCIe体系结构中，也存在非透明PCI桥的概念。如在PLX的Switch芯片中，各个端口都可以设置为非透明模式，

### 2.6 小结

本章介绍了在PCI总线中使用的桥，包括HOST主桥和PCI桥，并较为详细介绍了如何使用这些桥访问PCI设备的配置空间。

其中HOST主桥并不在PCI总线规范的约束范围内，不同的处理器可以根据需要设计出不同的HOST主桥。本章更加侧重介绍PowerPC处理器使用的HOST主桥，在该主桥的设计中，提出了许多新的概念，并极大促进了PCI总线的发展，在这个桥片中出现的许多新的思想被PCI V3.0总线规范采纳。

在PowerPC处理器的HOST主桥中，明确了存储器域与PCI总线域的概念。而区分存储器域与PCI总线域也是本章的书写重点，本书将始终强调这两个域的不同。有些处理器系统并没有明确区分这两个域的差别，因此许多读者忽略了PCI总线域的存在，并错误地认为PCI总线域是存储器域的一部分。

在本章中，还重点介绍了PCI桥的实现机制。在许多较为简单处理器系统中，并不包含PCI桥，但是读者仍然需要深入理解PCI桥这一重要概念。深入理解PCI桥的运行机制，是理解PCI体系结构的重要基础。 

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftnref1) 有些非透明桥，如DEC21554的两边并不是完全对等的，尤其是在处理64位地址空间时，本文对此不做详细说明。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftnref2) DEC21554是Digital公司的产品。

[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftnref3) Intel 21555非透明桥片两边的HOST处理器都可以访问Primary和Secondary总线的配置寄存器。

[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3011901.HTM#_ftnref4) 多数半导体厂商提供两类芯片手册，分别是Datasheet和User manual。其中Datasheet偏重硬件电气特性，User Manual侧重芯片使用原理。

|  |
| --- |
| **第3章 PCI总线的数据交换** |
| 发布时间：2013-04-09 18:01:45 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI Agent设备之间，以及HOST处理器和PCI Agent设备之间可以使用存储器读写和I/O读写等总线事务进行数据传送。在大多数情况下，PCI桥不直接与PCI设备或者HOST主桥进行数据交换，PCI桥仅转发来自PCI Agent设备或者HOST主桥的数据。

PCI Agent设备间的数据交换，并不是本章讨论的重点。本章更侧重讲述PCI Agent设备使用DMA机制读写主存储器的数据，以及HOST处理器如何访问PCI设备的BAR空间。本章还使用了一定的篇幅讨论在PCI总线中与Cache相关的总线事务，并在最后一节介绍预读机制。

|  |
| --- |
| **3.1 PCI设备BAR空间的初始化** |
| 发布时间：2013-04-09 18:15:01 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

在PCI Agent设备进行数据传送之前，系统软件需要初始化PCI Agent设备的BAR0~5寄存器和PCI桥的Base、Limit寄存器。系统软件使用DFS算法对PCI总线进行遍历时，完成这些寄存器的初始化，即分配这些设备在PCI总线域的地址空间。当这些寄存器初始化完毕后，PCI设备可以使用PCI总线地址进行数据传递。

值得注意的是，PCI Agent设备的BAR0~5寄存器和PCI桥的Base寄存器保存的地址都是PCI总线地址。而这些地址在处理器系统的存储器域中具有映像，如果一个PCI设备的BAR空间在存储器域中没有映像，处理器将不能访问该PCI设备的BAR空间。

如上文所述，处理器通过HOST主桥将PCI总线域与存储器域隔离。当处理器访问PCI设备的地址空间时，需要首先访问该设备在存储器域中的地址空间，并通过HOST主桥将这个存储器域的地址空间转换为PCI总线域的地址空间之后，再使用PCI总线事务将数据发送到指定的PCI设备中。

PCI设备访问存储器域的地址空间，即进行DMA操作时，也是首先访问该存储器地址空间所对应的PCI总线地址空间，之后通过HOST主桥将这个PCI总线地址空间转换为存储器地址空间，再由DDR控制器对存储器进行读写访问。

不同的处理器系统采用不同的机制实现存储器域和PCI总线域的转换。如PowerPC处理器使用Outbound寄存器组实现存储器域到PCI总线域间的转换，并使用Inbound寄存器组实现PCI总线域到存储器域间的转换。

而x86处理器没有这种地址空间域的转换机制，因此从PCI设备的角度上看，PCI设备可以直接访问存储器地址；从处理器的角度上看，处理器可以直接访问PCI总线地址空间。但是读者需要注意，在x86处理器的HOST主桥中仍然有存储器域与PCI总线域这个概念。只是在x86处理器的HOST主桥中，存储器域的存储器地址与PCI总线地址相等，这种“简单相等”也是一种映射关系。

**3.1.1 存储器地址与PCI总线地址的转换**

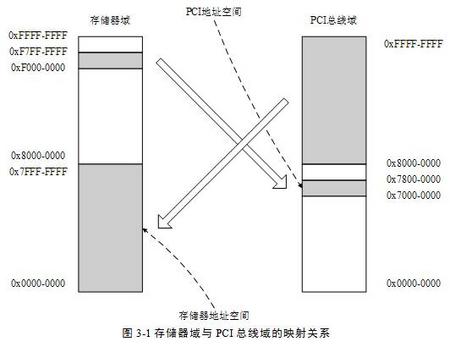
下文根据PowerPC和x86处理器的主桥，抽象出一个虚拟的HOST主桥，并以此为例讲述PCI Agent设备之间，以及PCI Agent设备与主存储器间的数据传送过程。

我们假设在一个32位处理器中，其存储器域的0xF000-0000~0xF7FF-FFFF(共128MB)这段物理地址空间与PCI总线的地址空间存在映射关系。

当处理器访问这段存储器地址空间时，HOST主桥将会认领这个存储器访问，并将这个存储器访问使用的物理地址空间转换为PCI总线地址空间，并与0x7000-0000~0x77FF-FFFF这段PCI总线地址空间对应。

为简化起见，我们假定在存储器域中只映射了PCI设备的存储器地址空间，而不映射PCI设备的I/O地址空间。而PCI设备的BAR空间使用0x7000-0000~0x77FF-FFFF这段PCI总线域的存储器地址空间。

在这个HOST主桥中，存储器域与PCI总线域的对应关系如图3‑1所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130409180721875.jpg)

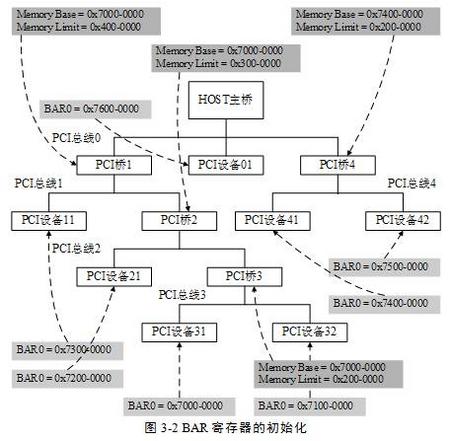
当PCI设备使用DMA机制，访问存储器域地址空间时，处理器系统同样需要将存储器域的地址空间反向映射到PCI总线地址空间。假设在一个处理器系统中，如果主存储器大小为2GB，其在存储器域的地址范围为0x0000-0000~0x7FFF-FFFF，而这段地址在PCI总线域中对应的“PCI总线地址空间”为0x8000-0000~0xFFFF-FFFF。

因此PCI设备进行DMA操作时，必须使用0x8000-0000~0xFFFF-FFFF这段PCI总线域的地址，HOST主桥才能认领这个PCI总线事务，并将这个总线事务使用的PCI总线地址转换为存储器地址，并与0x0000-0000~0x7FFF-FFFF这段存储器区域进行数据传递。

在一个实际的处理器系统中，很少有系统软件采用这样的方法，实现存储器域与PCI总线域之间的映射，“简单相等”还是最常用的映射方法。本章采用图3‑1的映射关系，虽然增加了映射复杂度，却便于读者深入理解存储器域到PCI总线域之间的映射关系。下文将以这种映射关系为例，详细讲述PCI设备BAR0~5寄存器的初始化。

**3.1.2 PCI设备BAR寄存器和PCI桥Base、Limit寄存器的初始化**

PCI桥的Base、Limit寄存器保存“该桥所管理的PCI子树”的存储器或者I/O空间的基地址和长度。值得注意的是，PCI桥也是PCI总线上的一个设备，在其配置空间中也有BAR寄存器，本节不对PCI桥BAR寄存器进行说明，因为在多数情况下透明桥并不使用其内部的BAR寄存器。下文以图3‑2所示的处理器系统为例说明上述寄存器的初始化过程，该处理器系统使用的存储器域与PCI总线域的映射关系如图3‑1所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130409180751894.jpg)

在PCI设备的BAR寄存器中，包含该设备使用的PCI总线域的地址范围。在PCI设备的配置空间\*\*有6个BAR寄存器，因此一个PCI设备最多可以使用6组32位的PCI总线地址空间，或者3组64位的PCI总线地址空间。这些BAR空间可以保存PCI总线域的存储器地址空间或者I/O地址空间，目前多数PCI设备仅使用存储器地址空间。而在通常情况下，一个PCI设备使用2到3个BAR寄存器就足够了。

为简化起见，我们首先假定在图3‑2中所示的PCI总线树中，所有PCI Agent设备只使用了BAR0寄存器，其申请的数据空间大小为16M字节(即0x1000000字节)而且不可预读，而且PCI桥不占用PCI总线地址空间，即PCI桥不含有BAR空间。并且假定当前HOST主桥已经完成了对PCI总线树的编号。

根据以上假设，系统软件该PCI总线树的遍历过程如下所示。

(1) 系统软件根据DFS算法，系统软件率先寻找到第一组PCI设备，分别为PCI设备31和PCI设备32[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011934.HTM#_ftn1)，并根据这两个PCI设备需要的PCI空间大小，从PCI总线地址空间中(0x7000-0000~0x77FF-FFFF)为这两个PCI设备的BAR0寄存器分配基地址，分别为0x7000-0000和0x7100-0000。

(2) 当系统软件完成PCI总线3下所有设备的BAR空间的分配后，将初始化PCI桥3的配置空间。这个桥片的Memory Base寄存器保存其下所有PCI设备使用的“PCI总线域地址空间的基地址”，而Memory Limit寄存器保存其下PCI设备使用的“PCI总线域地址空间的大小”。系统软件将Memory Base寄存器赋值为0x7000-0000，而将Memory Limit寄存器赋值为0x200-0000。

(3) 系统软件回朔到PCI总线2，并找到PCI总线2上的PCI设备21，并将PCI设备21的BAR0寄存器赋值为0x7200-0000。

(4) 完成PCI总线2的遍历后，系统软件初始化PCI桥2的配置寄存器，将Memory Base寄存器赋值为0x7000-0000，Memory Limit寄存器赋值为0x300-0000。

(5) 系统软件回朔到PCI总线1，并找到PCI设备11，并将这个设备的BAR0寄存器赋值为0x7300-0000。并将PCI桥1的Memory Base寄存器赋值为0x7000-0000，Memory Limit寄存器赋值为0x400-0000。

(6) 系统软件回朔到PCI总线0，并在这条总线上发现另外一个PCI桥，即PCI桥4。并使用DFS算法继续遍历PCI桥4。首先系统软件将遍历PCI总线4，并发现PCI设备41和PCI设备42，并将这两个PCI设备的BAR0寄存器分别赋值为0x7400-0000和0x7500-0000。

(7) 系统软件初始化PCI桥4的配置寄存器，将Memory Base寄存器赋值为0x7400-0000，Memory Limit寄存器赋值为0x200-0000。系统软件再次回到PCI总线0，这一次系统软件没有发现新的PCI桥，于是将初始化这条总线上的所有PCI设备。

(8) PCI总线0上只有一个PCI设备，PCI设备01。系统软件将这个设备的BAR0寄存器赋值为0x7600-0000，并结束整个DFS遍历过程。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3011934.HTM#_ftnref1) HOST主桥下的第一个桥片是PCI桥片1，PCI桥片1下的第一个桥片是PCI桥片2，而PCI桥片2下的第一个桥片是PCI桥片3，因而第一组PCI设备为PCI总线3下的PCI设备。不同的系统软件查找第一组PCI设备的方法不同，Linux认为第一组PCI设备为PCI总线0下的PCI设备。

|  |
| --- |
| **3.2 PCI设备的数据传递** |
| 发布时间：2013-04-17 13:51:52 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI设备的数据传递使用地址译码方式，当一个存储器读写总线事务到达PCI总线时，在这条总线上的所有PCI设备将进行地址译码，如果当前总线事务使用的地址在某个PCI设备的BAR空间中时，该PCI设备将使能DEVSEL#信号，认领这个总线事务。

如果PCI总线上的所有设备都不能通过地址译码，认领这个总线事务时，这条总线的“负向译码”设备将认领这个总线事务，如果在这条PCI总线上没有“负向译码”设备，该总线事务的发起者将使用Master Abort总线周期结束当前PCI总线事务。

### 3.2.1 PCI设备的正向译码与负向译码

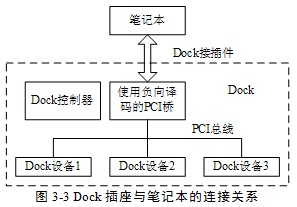
如上文所示PCI设备使用“地址译码”方式接收存储器读写总线请求。在PCI总线中定义了两种“地址译码”方式，一个是正向译码，一个是负向译码。

我们仍以图3‑2所示的处理器系统为例，说明数据传送使用的寻址方法。当HOST主桥通过存储器或者I/O读写总线事务访问其下所有PCI设备时，PCI总线0下的所有PCI设备都将对出现在地址周期中的PCI总线地址进行译码。如果这个地址在某个PCI设备的BAR空间中命中时，这个PCI设备将接收这个PCI总线请求。这个过程也被称为PCI总线的正向译码，这种方式也是大多数PCI设备所采用的译码方式。

但是在PCI总线上的某些设备，如PCI-to-(E)ISA桥并不使用正向译码接收来自PCI总线的请求，PCI-to-ISA桥在处理器系统中的位置如图1‑1所示。PCI总线0上的总线事务在三个时钟周期后，没有得到任何PCI设备响应时(即总线请求的PCI总线地址不在这些设备的BAR空间中)，PCI-to-ISA桥将被动地接收这个数据请求。这个过程被称为PCI总线的负向译码。可以进行负向译码的设备也被称为负向译码设备。

在PCI总线中，除了PCI-to-(E)ISA桥可以作为负向译码设备，PCI桥也可以作为负向译码设备，但是PCI桥并不是在任何时候都可以作为负向译码设备。在绝大多数情况下，PCI桥无论是处理“来自上游总线”，还是处理“来自下游总线”的总线事务时，都使用正向译码方式，但是在某些特殊应用中，PCI桥也可以作为负向译码设备。

如笔记本在连接Dock插座时，也使用了PCI桥。因为在大多数情况下，笔记本与Dock插座是分离使用的，而且Dock插座上连接的设备多为慢速设备，此时用于连接Dock插座的PCI桥使用负向译码。Dock插座在笔记本系统中的位置如图3‑3所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130417134933770.jpg)

当笔记本与Dock建立连接之后，如果处理器需要访问Dock中的外部设备时，Dock中的PCI桥将首先使用负向译码方式接收PCI总线事务，之后将这个PCI总线事务转发到Dock的PCI总线中，然后再访问相应的PCI设备。

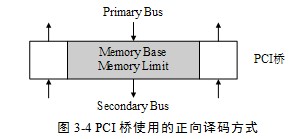
在Dock中使用负向译码PCI桥的优点是，该桥管理的设备并不参与处理器系统对PCI总线的枚举过程。当笔记本插入到Dock之后，系统软件并不需要重新枚举Dock中的设备并为这些设备分配系统资源，而仅需要使用负向译码PCI桥管理好其下的设备即可，从而极大降低了Dock对系统软件的影响。

当HOST处理器访问Dock中的设备时，负向译码PCI桥，将首先接管这些存储器读写总线事务，然后发送到Dock设备中。值得注意的是，在许多笔记本的Dock实现中，并没有使用负向译码PCI桥，而使用PCI-to-ISA桥。

PCI总线规定使用负向译码的PCI桥，其Base Class Code寄存器为0x06，Sub Class Code寄存器为0x04，而Interface寄存器为0x01；使用正向译码方式的PCI桥的Interface寄存器为0x00。系统软件(E2PROM)在初始化Interface寄存器时务必注意这个细节。

综上所述，在PCI总线中有两种负向译码设备，PCI-to-E(ISA)桥和PCI桥。但是PCI桥并不是在任何时候都是负向译码设备，只有PCI桥连接Dock插座时，PCI桥的Primary Bus才使用负向译码方式。而这个PCI桥的Secondary Bus在接收Dock设备的请求时仍然使用正向译码方式。

PCI桥使用的正向译码方式与PCI设备使用的正向译码方式有所不同。如图3‑4所示，当一个总线事务是从PCI桥的Primary Bus到Secondary Bus时，PCI桥使用的正向译码方式与PCI设备使用的方式类似。如果该总线事务使用的地址在PCI桥任意一个Memory Base窗口[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012117.HTM#_ftn1)命中时，该PCI桥将使用正向译码方式接收该总线事务，并根据实际情况决定是否将这个总线事务转发到Secondary Bus。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130417135039985.jpg)

当一个总线事务是从PCI桥的Secondary Bus到Primary Bus时，如果该总线事务使用的地址没有在PCI桥所有的Memory Base窗口命中时，表明当前总线事务不是访问该PCI桥管理的PCI子树中的设备，因此PCI桥将接收当前总线事务，并根据实际情况决定是否将这个总线事务转发到Primary Bus。

以图3‑2为例，当PCI设备11访问主存储器空间时，首先将存储器读写总线事务发送到PCI总线1上，而这个存储器地址显然不会在PCI总线1的任何PCI设备的BAR空间中，此时PCI桥1将认领这个PCI总线的数据请求，并将这个总线事务转发到PCI总线0上。最后HOST主桥将接收这个总线事务，并将PCI总线地址转换为存储器域的地址，与主存储器进行读写操作。

值得注意的是，PCI总线并没有规定HOST主桥使用正向还是负向译码方式接收这个存储器读写总线事务，但是绝大多数HOST主桥使用正向译码方式接收来自下游的存储器读写总线事务。在PowerPC处理器中，如果当前存储器读写总线事务使用的地址在Inbound窗口内时，HOST主桥将接收这个总线事务，并将其转换为存储器域的读写总线事务，与主存储器进行数据交换。

### 3.3.2 处理器到PCI设备的数据传送

下文以图3‑2所示的处理器系统为例，说明处理器向PCI设备11进行存储器写的数据传送过程。处理器向PCI设备进行读过程与写过程略有区别，因为存储器写使用Posted方式，而存储器读使用Non-Posted方式，但是存储器读使用的地址译码方式与存储器写类似，因此本节对处理器向PCI设备进行存储器读的过程不做进一步介绍。

PCI设备11在PCI总线域的地址范围是0x7300-0000~0x73FF-FFFF。这段空间在存储器域中对应的地址范围是0xF300-0000~0xF3FF-FFFF。下文我们假设处理器使用存储器写指令，访问0xF300-0008这个存储器地址，其步骤如下。

(1) 存储器域将0xF300-0008这个地址发向HOST主桥，0xF000-0000~0xF7FF-FFFF这段地址已经由HOST主桥映射到PCI总线域地址空间，所以HOST主桥认为这是一个对PCI设备的访问。因此HOST主桥将首先接管这个存储器写请求。

(2) HOST主桥将存储器域的地址0xF300-0008转换为PCI总线域的地址0x7300-0008，并通过总线仲裁获得PCI总线0的使用权，启动PCI存储器写周期，并将这个存储器写总线事务发送到PCI总线0上。值得注意的是，这个存储器读写总线事务使用的地址为0x7300-0008，而不是0xF300-0008。

(3) PCI总线0的PCI桥1发现0x7300-0008在自己管理的地址范围内，于是接管这个存储器写请求，并通过总线仲裁逻辑获得PCI总线1的使用权，并将这个请求转发到PCI总线1上。

(4) PCI总线1的PCI设备11发现0x7300-0008在自己的BAR0寄存器中命中，于是接收这个PCI写请求，并完成存储器写总线事务。

### 3.2.3 PCI设备的DMA操作

下文以图3‑2所示的处理器系统为例，说明PCI设备11向存储器进行DMA写的数据传送过程。PCI设备的DMA写使用Posted方式而DMA读使用Non-Posted方式。本节不介绍PCI设备进行DMA读的过程，而将这部分内容留给读者分析。

假定PCI设备11需要将一组数据，发送到0x1000-0000~0x1000-FFFF这段存储器域的地址空间中。由上文所述，存储器域的0x0000-0000~0x7FFF-FFFF这段存储器空间与PCI总线域的0x8000-0000~0xFFFF-FFFF这段PCI总线地址空间对应。

PCI设备11并不能直接操作0x1000-0000~0x1000-FFFF这段存储器域的地址空间，PCI设备11需要对PCI总线域的地址空间0x9000-0000~0x9000-FFFF进行写操作，因为PCI总线地址空间0x9000-0000~0x9000-FFFF已经被HOST主桥映射到0x1000-0000~0x1000-FFFF这段存储器域。这个DMA写具体的操作流程如下。

(1) 首先PCI设备11通过总线仲裁逻辑获得PCI总线1的使用权，之后将存储器写总线事务发送到PCI总线1上。值得注意的是，这个存储器写总线事务的目的地址是PCI总线域的地址空间0x9000-0000~0x9000-FFFF，这个地址是主存储器在PCI总线域的地址映像。

(2) PCI总线1上的设备将进行地址译码，确定这个写请求是不是发送到自己的BAR空间，在PCI总线1上的设备除了PCI设备11之外，还有PCI桥2和PCI桥1。

(3) 首先PCI桥1、2和PCI设备11对这个地址同时进行正向译码。PCI桥1发现这个PCI地址并不在自己管理的PCI总线地址范围之内，因为PCI桥片1所管理的PCI总线地址空间为0x7000-0000~0x73FF-FFFF。此时PCI桥1将接收这个存储器写总线事务，因为PCI桥1所管理的PCI总线地址范围并不包含当前存储器写总线事务的地址，所以其下所有PCI设备都不可能接收这个存储器写总线事务。

(4) PCI桥1发现自己并不能处理当前这个存储器写总线事务，则将这个存储器写总线事务转发到上游总线。PCI桥1首先通过总线仲裁逻辑获得PCI总线0的使用权后，然后将这个总线事务转发到PCI总线0。

(5) HOST主桥发现0x9000-0000~0x9000-FFFF这段PCI总线地址空间与存储器域的存储器地址空间0x1000-0000~0x1000-FFFF对应，于是将这段PCI总线地址空间转换成为存储器域的存储器地址空间，并完成对这段存储器的写操作。

(6) 存储器控制器将从HOST主桥接收数据，并将其写入到主存储器。

PCI设备间的数据传递与PCI设备到存储器的数据传送大体类似。我们以PCI设备11将数据传递到PCI设备42为例说明这个转递过程。我们假定PCI设备11将一组数据发送到PCI设备42的PCI总线地址0x7500-0000~0x7500-FFFF这段地址空间中。这个过程与PCI设备11将数据发送到存储器的第1~5步基本类似，只是第5、6步不同。PCI设备11将数据发送到PCI设备42的第5、6步如下所示。

(1) PCI总线0发现其下的设备PCI桥4能够处理来自PCI总线0的数据请求，则PCI桥4将接管这个PCI写请求，并通过总线仲裁逻辑获得PCI总线4的使用权，之后将这个存储器写请求发向PCI总线4。此时HOST主桥不会接收当前存储器写总线事务，因为0x7500-0000~0x7500-FFFF这段地址空间并不是HOST主桥管理的地址范围。

(2) PCI总线4的PCI设备42将接收这个存储器写请求，并完成这个PCI存储器写请求总线事务。

PCI总线树内的数据传送始终都在PCI总线域中进行，因此不存在不同域之间的地址转换，因此PCI设备11向PCI设备42进行数据传递时，并不会进行PCI总线地址空间到存储器地址空间的转换。

### 3.2.4 PCI桥的Combining、Merging和Collapsing

由上文所述，PCI设备间的数据传递有时将通过PCI桥。在某些情况下，PCI桥可以合并一些数据传递，以提高数据传递的效率。PCI桥可以采用Combining、Merging和Collapsing三种方式，优化数据通过PCI桥的效率。

#### 1 Combining

PCI桥可以将接收到的多个存储器写总线事务合并为一个突发存储器写总线事务。PCI桥进行这种Combining操作时需要注意数据传送的“顺序”。当PCI桥接收到一组物理地址连续的存储器写访问时，如对PCI设备的某段空间的DW1、2和4进行存储器写访问时，PCI桥可以将这组访问转化为一个对DW1~4的突发存储器写访问，并使用字节使能信号C/BE[3:0]#进行控制，其过程如下所示。

PCI桥将在数据周期1中，置C/BE[3:0]#信号为有效表示传递数据DW1；在数据周期2中，置C/BE[3:0]#信号为有效表示传递数据DW2；在数据周期3中，置C/BE[3:0]#信号为无效表示在这个周期中所传递的数据无效，从而跳过DW3；并在数据周期4中，置C/BE[3:0]#信号为有效表示传递数据DW4。

目标设备将最终按照发送端的顺序，接收DW1、DW2和DW4，采用这种方法在不改变传送序的前提下，提高了数据的传送效率。值得注意的是，有些HOST主桥也提供这种Combining方式，合并多次数据访问。如果目标设备不支持突发传送方式，该设备可以使用Disconnect周期，终止突发传送，此时PCI桥/HOST主桥可以使用多个存储器写总线事务分别传送DW1、DW2和DW4，而不会影响数据传送。

如果PCI桥收到“乱序”的存储器写访问，如对PCI设备的某段空间的DW4、3和1进行存储器写访问时，PCI桥不能将这组访问转化为一个对DW1~4的突发存储器写访问，此时PCI桥必须使用三个存储器写总线事务转发这些存储器写访问。

#### 2 Merge

PCI桥可以将收到的多个对同一个DW地址的Byte、Word进行的存储器写总线事务，合并为一个对这个DW地址的存储器写总线事务。PCI规范并没有要求这些对Byte、Word进行的存储器写在一个DW的边界之内，但是建议PCI桥仅处理这种情况。本节也仅介绍在这种情况下，PCI桥的处理过程。

PCI规范允许PCI桥进行Merge操作的存储器区域必须是可预读的，而可预读的存储器区域必须支持这种Merge操作。Merge操作可以不考虑访问顺序，可以将对Byte0、Byte1、Byte3的存储器访问合并为一个DW，也可以将对Byte3、Byte1、Byte0的存储器访问合并为一个DW。在这种情况下，PCI总线事务仅需屏蔽与Byte2相关的字节使能信号C/BE2#即可。

如果PCI设备对Byte1进行存储器写、然后再对Byte1、Byte2、Byte3进行存储器写时，PCI桥不能将这两组存储器写合并为一次对DW进行存储器写操作。但是PCI桥可以合并后一组存储器写，即首先对Byte1进行存储器写，然后合并后一组存储器写(Byte1、Byte2和Byte3)为一个DW写，并屏蔽相应的C/BE0#信号。Combining与Merge操作之间没有直接联系，PCI桥可以同时支持这两种方式，也可以支持任何一种方式。

#### 3 Collapsing

Collapsing指PCI桥可以将对同一个地址进行的Byte、Word和DW存储器写总线事务合并为一个存储器写操作。使用PCI桥的Collapsing方式是，具有某些条件限制，在多数情况下，PCI桥不能使用Collapsing方式合并多个存储器写总线事务。

当PCI桥收到一个对“DW地址X”的Byte3进行的存储器写总线事务，之后又收到一个对“DW地址X”的Byte、Word或者DW存储器写总线事务时，而且后一个对DW地址X进行的存储器写仍然包含Byte3时，如果PCI桥支持Collapsing方式，就可以将这两个存储器写合并为一个存储器写。

PCI桥在绝大多数情况下不能支持这种方式，因为很少有PCI设备支持这种数据合并方式。通常情况下，对PCI设备的同一地址的两次写操作代表不同的含义，因此PCI桥不能使用Collapsing方式将这两次写操作合并。PCI规范仅是提出了Collapsing方式的概念，几乎没有PCI桥支持这种数据合并方式。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012117.HTM#_ftnref1) PCI桥除了具有Memory Base窗口外，还有I/O Base窗口和Prefetchable Memory Base窗口。

|  |
| --- |
| **3.3 与Cache相关的PCI总线事务** |
| 发布时间：2013-04-17 14:01:52 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI总线规范定义了一系列与Cache相关的总线事务，以提高PCI设备与主存储器进行数据交换的效率，即DMA读写的效率。当PCI设备使用DMA方式向存储器进行读写操作时，一定需要经过HOST主桥，而HOST主桥通过FSB总线[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012118.HTM#_ftn1)向存储器控制器进行读写操作时，需要进行Cache共享一致性操作。

PCI设备与主存储器进行的Cache共享一致性增加了HOST主桥的设计复杂度。在高性能处理器中Cache状态机的转换模型十分复杂。而HOST主桥是FSB上的一个设备，需要按照FSB规定的协议处理这个Cache一致性，而多级Cache的一致性和状态转换模型一直是高性能处理器设计中的难点。

不同的HOST主桥处理PCI设备进行的DMA操作时，使用的Cache一致性的方法并不相同。因为Cache一致性操作不仅与HOST主桥的设计相关，而且主要与处理器和Cache Memory系统设计密切相关。

PowerPC和x86处理器可以对PCI设备所访问的存储器进行设置，其设置方法并不相同。其中PowerPC处理器，如MPC8548处理器，可以使用Inbound寄存器的RTT字段和WTT字段，设置在PCI设备进行DMA操作时，是否需要进行Cache一致性操作，是否可以将数据直接写入Cache中。RTT字段和WTT字段的详细说明见第2.2.3节。

而x86处理器可以使用MTRR(Memory Type Range Registers)设置物理存储器区间的属性，是否为可Cache空间。下文分别讨论在PowerPC与x86处理器中，PCI设备进行DMA写操作时，如何进行Cache一致性操作。

但是与PowerPC处理器相比，x86处理器在处理PCI设备的Cache一致性上略有不足，特别是网络设备与存储器系统进行数据交换的效率。因为x86处理器所重点优化的是PCIe设备，目前x86处理器使用的IOAT(I/O Acceleration Technology)技术，极大增强了PCIe设备与主存储器进行数据通信的效率，但是这种技术仍然不能与一些Data Plane处理器，如XLP832，P4080处理器优化I/O访问的技术相提并论。毕竟x86处理器所适用的领域依然是PC、服务器等计算和控制领域，并不是Data Plane处理器领域。

|  |
| --- |
| **3.3.1 Cache一致性的基本概念** |
| 发布时间：2013-04-19 17:45:53 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI设备对可Cache的存储器空间进行DMA读写的操作的过程较为复杂，有关Cache一致性的话题可以独立成书。而不同的处理器系统使用的Cache Memory的层次结构和访问机制有较大的差异，这部分内容也是现代处理器系统设计的重中之重。

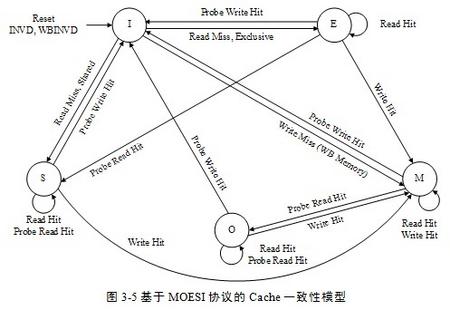
本节仅介绍在Cache Memory系统中与PCI设备进行DMA操作相关的，一些最为基础的概念。在多数处理器系统中，使用了以下概念描述Cache一致性的实现过程。

#### 1 Cache一致性协议

多数SMP处理器系统使用了MESI协议处理多个处理器之间的Cache一致性。该协议也被称为Illinois protocol，MESI协议在SMP处理器系统中得到了广泛的应用。MESI协议使用四个状态位描述每一个Cache行。

* M(Modified)位。M 位为1 时表示当前Cache 行中包含的数据与存储器中的数据不一致，而且它仅在本CPU的Cache 中有效，不在其他CPU的Cache 中存在拷贝，在这个Cache行的数据是当前处理器系统中最新的数据拷贝。当CPU对这个Cache行进行替换操作时，必然会引发系统总线的写周期，将Cache行中数据与内存中的数据同步。
* E(Exclusive)位。E 位为1 时表示当前Cache行中包含的数据有效，而且该数据仅在当前CPU的Cache中有效，而不在其他CPU的Cache中存在拷贝。在该Cache行中的数据是当前处理器系统中最新的数据拷贝，而且与存储器中的数据一致。
* S(Shared)位。S 位为1 表示Cache行中包含的数据有效，而且在当前CPU和至少在其他一个CPU中具有副本。在该Cache行中的数据是当前处理器系统中最新的数据拷贝，而且与存储器中的数据一致。
* I(Invalid)位。I 位为1 表示当前Cache 行中没有有效数据或者该Cache行没有使能。MESI协议在进行Cache行替换时，将优先使用I位为1的Cache行。

MESI协议还存在一些变种，如MOESI协议和MESIF协议。基于MOESI协议的Cache一致性模型如图3‑5所示，该模型基于AMD处理器使用的MOESI协议。不同的处理器在实现MOESI协议时，状态机的转换原理类似，但是在处理上仍有细微区别。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130419174437193.jpg)

MOESI协议引入了一个O(Owned)状态，并在MESI协议的基础上，进行了重新定义了S状态，而E、M和I状态和MESI协议的对应状态相同。

* O位。O位为1表示在当前Cache 行中包含的数据是当前处理器系统最新的数据拷贝，而且在其他CPU中一定具有该Cache行的副本，其他CPU的Cache行状态为S。如果主存储器的数据在多个CPU的Cache中都具有副本时，有且仅有一个CPU的Cache行状态为O，其他CPU的Cache行状态只能为S。与MESI协议中的S状态不同，状态为O的Cache行中的数据与存储器中的数据并不一致。
* S位。在MOESI协议中，S状态的定义发生了细微的变化。当一个Cache行状态为S时，其包含的数据并不一定与存储器一致。如果在其他CPU的Cache中不存在状态为O的副本时，该Cache行中的数据与存储器一致；如果在其他CPU的Cache中存在状态为O的副本时，Cache行中的数据与存储器不一致。

在一个处理器系统中，主设备(CPU或者外部设备)进行存储器访问时，将试图从存储器系统(主存储器或者其他CPU的Cache)中获得最新的数据拷贝。如果该主设备访问的数据没有在本地命中时，将从其他CPU的Cache中获取数据，如果这些数据仍然没有在其他CPU的Cache中命中，主存储器将提供数据。外设设备进行存储器访问时，也需要进行Cache共享一致性。

在MOESI模型中，“Probe Read”表示主设备从其他CPU中获取数据拷贝的目的是为了读取数据；而“Probe Write”表示主设备从其他CPU中获取数据拷贝的目的是为了写入数据；“Read Hit”和“Write Hit”表示主设备在本地Cache中获得数据副本；“Read Miss”和“Write Miss”表示主设备没有在本地Cache中获得数据副本；“Probe Read Hit”和“Probe Write Hit”表示主设备在其他CPU的Cache中获得数据副本。

本节为简便起见，仅介绍CPU进行存储器写和与O状态相关的Cache行状态迁移，CPU进行存储器读的情况相对较为简单，请读者自行分析这个过程。

当CPU对一段存储器进行写操作时，如果这些数据在本地Cache中命中时，其状态可能为E、S、M或者O。

* 状态为E或者M时，数据将直接写入到Cache中，并将状态改为M。
* 状态为S时，数据将直接写入到Cache中，并将状态改为M，同时其他CPU保存该数据副本的Cache行状态将从S或者O迁移到I(Probe Write Hit)。
* 状态为O时，数据将直接写入到Cache中，并将状态改为M，同时其他CPU保存该数据副本的Cache行状态将从S迁移到I(Probe Write Hit)。

当CPU A对一段存储器进行写操作时，如果这些数据没有在本地Cache中命中时，而在其他CPU，如CPU B的Cache中命中时，其状态可能为E、S、M或者O。其中CPU A使用CPU B在同一个Cache共享域中。

* Cache行状态为E时，CPU B将该Cache行状态改为I；而CPU A将从本地申请一新的个Cache行，将数据写入，并该Cache行状态更新为M。
* Cache行状态为S时，CPU B将该Cache行状态改为I，而且具有同样副本的其他CPU的Cache行也需要将状态改为I；而CPU A将从本地申请一个Cache行，将数据写入，并该Cache行状态更新为M。
* Cache行状态为M时，CPU B将原Cache行中的数据回写到主存储器，并将该Cache行状态改为I；而CPU A将从本地申请一个Cache行，将数据写入，并该Cache行状态更新为M。
* Cache行状态为O时，CPU B将原Cache行中的数据回写到主存储器，并将该Cache行状态改为I，具有同样数据副本的其他CPU的Cache行也需要将状态从S更改为I；CPU A将从本地申请一个Cache行，将数据写入，并该Cache行状态更新为M。

Cache行状态可以从M迁移到O。例如当CPU A读取的数据从CPU B中命中时，如果在CPU B中Cache行的状态为M时，将迁移到O，同时CPU B将数据传送给CPU A新申请的Cache行中，而且CPU A的Cache行状态将被更改为S。

当CPU读取的数据在本地Cache中命中，而且Cache行状态为O时，数据将从本地Cache获得，并不会改变Cache行状态。如果CPU A读取的数据在其他Cache中命中，如在CPU B的Cache中命中而且其状态为O时，CPU B将该Cache行状态保持为O，同时CPU B将数据传送给CPU A新申请的Cache行中，而且CPU A的Cache行状态将被更改为S。

在某些应用场合，使用MOESI协议将极大提高Cache的利用率，因为该协议引入了O状态，从而在发送Read Hit的情况时，不必将状态为M的Cache回写到主存储器，而是直接从一个CPU的Cache将数据传递到另外一个CPU。目前MOESI协议在AMD和RMI公司的处理器中得到了广泛的应用。

Intel提出了另外一种MESI协议的变种，即MESIF协议，该协议与MOESI协议有较大的不同，也远比MOESI协议复杂，该协议由Intel的QPI(QuickPath Interconnect)技术引入，其主要目的是解决“基于点到点的全互连处理器系统”的Cache共享一致性问题，而不是“基于共享总线的处理器系统”的Cache共享一致性问题。

在基于点到点互连的NUMA(Non-Uniform Memroy Architecture)处理器系统中，包含多个子处理器系统，这些子处理器系统由多个CPU组成。如果这个处理器系统需要进行全机Cache共享一致性，该处理器系统也被称为ccNUMA(Cache Cohenrent NUMA)处理器系统。MESIF协议主要解决ccNUMA处理器结构的Cache共享一致性问题，这种结构通常使用目录表，而不使用总线监听处理Cache的共享一致性。

MESIF协议引入了一个F(Forware)状态。在ccNUMA处理器系统中，可能在多个处理器的Cache中存在相同的数据副本，在这些数据副本中，只有一个Cache行的状态为F，其他Cache行的状态都为S。Cache行的状态位为F时，Cache中的数据与存储器一致。

当一个数据请求方读取这个数据副本时，只有状态为F的Cache行，可以将数据副本转发给数据请求方，而状态位为S的Cache不能转发数据副本。从而MESIF协议有效解决了在ccNUMA处理器结构中，所有状态位为S的Cache同时转发数据副本给数据请求方，而造成的数据拥塞。

在ccNUMA处理器系统中，如果状态位为F的数据副本，被其他CPU拷贝时，F状态位将会被迁移，新建的数据副本的状态位将为F，而老的数据副本的状态位将改变为S。当状态位为F的Cache行被改写后，ccNUMA处理器系统需要首先Invalidate状态位为S其他的Cache行，之后将Cache行的状态更新为M。

独立地研究MESIF协议并没有太大意义，该协议由Boxboro-EX处理器系统[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012167.HTM#_ftn1)引入，目前Intel并没有公开Boxboro-EX处理器系统的详细设计文档。MESIF协议仅是解决该处理器系统中Cache一致性的一个功能，该功能的详细实现与QPI的Protocal Layer相关，QPI由多个层次组成，而Protocal Layer是QPI的最高层。

对MESIF协议QPI互连技术有兴趣的读者，可以在深入理解“基于目录表的Cache一致性协议”的基础上，阅读Robert A. Maddox, Gurbir Singh and Robert J. Safranek合著的书籍“Weaving High Performance Multiprocessor Fabric”以了解该协议的实现过程和与QPI互连技术相关的背景知识。

值得注意的是，MESIF协议解决主要的问题是ccNUMA架构中SMP子系统与SMP子系统之间Cache一致性。而在SMP处理器系统中，依然需要使用传统的MESI协议。Nehelem EX处理器也可以使用MOESI协议进一步优化SMP系统使用的Cache一致性协议，但是并没有使用该协议。

为简化起见，本章假设处理器系统使用MESI协议进行Cache共享一致性，而不是MOESI协议或者MESIF协议。

#### 2 HIT#和HITM#信号

在SMP处理器系统中，每一个CPU都使用HIT#和HITM#信号反映HOST主桥访问的地址是否在各自的Cache中命中。当HOST主桥访问存储器时，CPU将驱动HITM#和HIT#信号，其描述如表3‑1所示。

表3‑1 HITM#和HIT#信号的含义

|  |  |  |
| --- | --- | --- |
| HITM# | HIT# | 描述 |
| 1 | 1 | 表示HOST主桥访问的地址没有在CPU的Cache中命中。 |
| 1 | 0 | 表示HOST主桥访问的地址在CPU的Cache中命中，而且Cache的状态为S(Shared)或者E(Exclusive)，即Cache中的数据与存储器的数据一致。 |
| 0 | 1 | 表示HOST主桥访问的地址在CPU的Cache中命中，而且Cache的状态为M(Modified)，即Cache中的数据与存储器的数据不一致，在Cache中保存最新的数据拷贝。 |
| 0 | 0 | MESI协议规定这种情况不允许出现，但是在有些处理器系统中仍然使用了这种状态，表示暂时没有获得是否在Cache命中的信息，需要等待几拍后重试。 |

HIT#和HITM#信号是FSB中非常重要的两个信号，各个CPU的HIT#和HITM#信号通过“线与方式”直接相连[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012167.HTM#_ftn2)。而在一个实际FSB中，还包括许多信号，本节并不会详细介绍这些信号。

#### 3 Cache一致性协议中使用的Agent

在处理器系统中，与Cache一致性相关的Agent如下所示。

* Request Agent。FSB总线事务的发起设备。在本节中，Request Agent特指HOST主桥。实际上在FSB总线上的其他设备也可以成为Request Agent，但这些Request Agent并不是本节的研究重点。Request Agent需要进行总线仲裁后，才能使用FSB，在多数处理器的FSB中，需要对地址总线与数据总线分别进行仲裁。
* Snoop Agents。FSB总线事务的监听设备。Snoop Agents为CPU，在一个SMP处理器系统中，有多个CPU共享同一个FSB，此时这些CPU都是这条FSB上的Snoop Agents。Snoop Agents监听FSB上的存储器读写事务，并判断这些总线事务访问的地址是否在Cache中命中。Snoop Agents通过HIT#和HITM#信号向FSB通知Cache命中的结果。在某些情况下，Snoop Agents需要将Cache中的数据回写到存储器，同时为Request Agent提供数据。
* Response Agent。FSB总线事务的目标设备。在本节中，Response Agent特指存储器控制器。Response Agent根据Snoop Agents提供的监听结果，决定如何接收数据或者向Request Agent设备提供数据。在多数情况下，当前数据访问没有在Snoop Agents中命中时，Response Agent需要提供数据，此外Snoop Agents有时需要将数据回写到Response Agent中。

#### 4 FSB的总线事务

一个FSB的总线事务由多个阶段组成，包括Request Phase、Snoop Phase、Response Phase和Data Phase。目前在多数高端处理器中，FSB支持流水操作，即在同一个时间段内，不同的阶段可以重叠，如图3‑6所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130419174505188.jpg)

在一个实际的FSB中，一个总线事务还可能包含Arbitration Phase和Error Phase。而本节仅讲述图3‑6中所示的4个基本阶段。

* Request Phase。Request Agent在获得FSB的地址总线的使用权后，在该阶段将访问数据区域的地址和总线事务类型发送到FSB上。
* Snoop Phase。Snoop Agents根据访问数据区域在Cache中的命中情况，使用HIT#和HITM#信号，向其他Agents通知Cache一致性的结果。有时Snoop Agent需要将数据回写到存储器。
* Reponse Phase。Response Agent根据Request和Snoop Phase提供的信号，可以要求Request Agent重试(Retry)，或者Response Agent延时处理(Defer)当前总线事务。在FSB总线事务的各个阶段中，该步骤的处理过程最为复杂。本章将在下文结合PCI设备的DMA读写执行过程，说明该阶段的实现原理。
* Data Phase。一些不传递数据的FSB总线事务不包含该阶段。该阶段用来进行数据传递，包括Request Agent向Response Agent写入数据；Response Agent为Request Agent提供数据；和Snoop Agent将数据回写到Response Agent。

下文将使用本小节中的概念，描述在PCI总线中，与Cache相关的总线事务，并讲述相关的FSB的操作流程。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012167.HTM#_ftnref1) Boxboro-EX处理器系统由多个Nehalem EX处理器组成，而Nehalem EX处理器由两个SMP处理器系统组成，一个SMP处理器系统由4个CPU组成，而每一个CPU具有2个线程。其中SMP处理器系统之间使用QPI进行连接，而在一个SMP处理器内部的各个CPU仍然使用FSB连接。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012167.HTM#_ftnref2) HIT#和HITM#信号是Open Drain（开漏）信号，Open Drain信号可以直接相连，而不用使用逻辑门。

|  |
| --- |
| **3.3.2 PCI设备对不可Cache的存储器空间进行DMA读写** |
| 发布时间：2013-04-19 17:50:45 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

在x86处理器和PowerPC处理器中，PCI设备对“不可Cache的存储器空间”进行DMA读写的过程并不相同。其中PowerPC处理器对“不可Cache的存储器空间”进行DMA读写进行了专门的处理，而x86处理器在对这类空间操作时，效率相对较低。

#### 1 x86处理器

x86处理器使用MTRR(Memory Type Range Register)寄存器设置存储器空间的属性，如果存储器空间为“可Cache空间”，x86处理器还可以进一步设置这段空间为“Write Through”、“Write Combining”、“Write Protect”和“Write Back”。但是这些设置与PCI设备进行DMA操作时，是否进行Cache一致性操作并没有直接关系。

在x86处理器系统中，一个PCI设备进行DMA写操作，可以将数据从PCI设备写入到主存储器中。这个数据首先需要通过HOST主桥，然后经过FSB发送到存储器控制器。虽然在x86处理器系统中，CPU知道这个存储器区域是否为“可Cache的”，但是HOST主桥并不知道PCI设备访问的存储器地址是否为“可Cache的”，因此都需要使用“Cache一致”的FSB总线传送事务[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012168.HTM#_ftn1)进行存储器写操作，从而数据在发向FSB时，CPU必须要进行总线监听，通知FSB总线这段空间是“不可Cache的”。

在x86处理器中，PCI设备向不可Cache的存储器空间进行读操作时，CPU也必须进行Cache共享一致性操作，而这种没有必要的Cache共享一致性操作将影响PCI总线的传送效率。当PCI设备所访问的存储器空间没有在CPU的Cache命中时，CPU会通知FSB，数据没有在Cache中命中，此时PCI设备访问的数据将从存储器中直接读出。

x86处理器在前端总线上进行Cache共享一致性操作时，需要使用Snoop Phase，如果PCI设备能事先得知所访问的存储器是“不可Cache的”，就可以不在前端总线上进行Cache共享一致性操作，即FSB总线事务可以不包含Snoop Phase，从而可以提高前端总线的使用效率。但是x86处理器并不支持这种方式。

在x86处理器系统中，无论PCI设备访问的存储器空间是否为“不可Cache的”，都需要进行Cache共享一致性操作。这也是PCI总线在x86处理器使用中的一个问题。而PCIe总线通过在数据报文中设置“Snooping”位解决了这个问题，有关PCIe总线Snooping位的内容参见第6.1.3节。

#### 2 PowerPC处理器

在MPC8548处理器中，HOST主桥可以通过PIWARn寄存器[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012168.HTM#_ftn2)的RTT字段和WTT字段预知PCI设备访问的存储器空间是否为可Cache空间。当HOST主桥访问“不可Cache空间时”，可以使用FSB总线的“不进行Cache一致性”的总线事务。

此时PowerPC处理器不会在FSB总线中进行Cache一致性操作，即忽略FSB总线事务的Snoop Phase。PCI设备进行DMA写时，数据将直接进入主存储器，而PCI设备进行DMA读所读取的数据将直接从主存储器获得。与x86处理器相比，PowerPC处理器可以忽略CPU进行总线监听的动作，从而提高了FSB传送效率。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012168.HTM#_ftnref1) FSB总线定义了许多总线事务，有的FSB总线提供了一个Snoop信号，该信号为1时表示当前FSB的总线事务需要进行Cache共享一致性，为0时不需要进行Cache共享一致性。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012168.HTM#_ftnref2) 该寄存器在Inbound寄存器组中，详见第2.2.3节。

|  |
| --- |
| **3.3.3 PCI设备对可Cache的存储器空间进行DMA读写** |
| 发布时间：2013-04-25 11:32:14 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCI设备向“可Cache的存储器空间”进行读操作的过程相对简单。对于x86处理器或者PowerPC处理器，如果访问的数据在Cache中命中，CPU会通知FSB总线，PCI设备所访问的数据在Cache中。

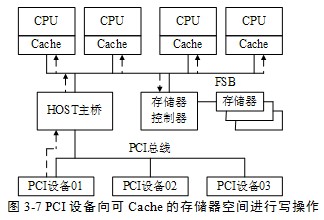
首先HOST主桥发起存储器读总线事务，并在Request Phase中，提供地址。Snoop Agent在Snoop Phase进行总线监听，并通过HIT#和HITM#信号将监听结果通知给Response Agent。如果Cache行的状态为E时，Response Agent将提供数据，而CPU不必改变Cache行状态。如果Snoop Agent可以直接将数据提供给HOST主桥，无疑数据访问的延时更短，但是采用这种方法无疑会极大的提高Cache Memory系统的设计难度，因此采用这种数据传送方式的处理器[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012263.HTM#_ftn1)并不多。

如果Cache行的状态为M时，Response Agent在Response Phase阶段，要求Snoop Agents将Cache中数据回写到存储器，并将Cache行状态更改为E。Snoop Agents在Data Phase，将Cache中数据回写给存储器控制器，同时为HOST主桥提供数据。Snoop Agents也可以直接将数据提供给HOST主桥，不需要进行数据回写过程，也不更改Cache行状态，但是采用这种方法会提高Cache Memory系统的设计难度。

如果PCI设备访问的数据没有在Cache中命中，Snoop Agents会通知FSB总线，PCI设备所访问的数据不在Cache中，此时存储器控制器(Response Agent)将在Data Phase向HOST主桥提供数据。

PCI设备向“可Cache的”存储器区域进行写操作，无论对于PowerPC处理器还是x86处理器，都较为复杂。当HOST主桥通过FSB将数据发送给存储器控制器时，在这个系统总线上的所有CPU都需要对这个PCI写操作进行监听，并根据监听结果，合理地改动Cache行状态，并将数据写入存储器。

下文以图3‑7所示的SMP处理器系统为例，说明PCI设备对“可Cache的存储器空间”进行DMA写的实现过程。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130425112613495.jpg)

在图3‑7所示的处理器系统中，存在4个CPU，这4个CPU通过一条FSB连接在一起，而CPU之间使用MESI协议进行Cache一致性处理，而HOST主桥和存储器控制器与FSB直接相连。HOST主桥向存储器控制器传递数据时，需要处理Cache的一致性。

在这个处理器系统中，当PCI设备，如PCI设备01，进行DMA写操作时，数据将首先到达HOST主桥，而HOST主桥将首先接管该PCI设备数据访问并将其转换为FSB总线事务，并在Request Phase中，提供本次FSB总线事务的地址。CPU将在Snoop Phase对这个地址进行监听，判断当前地址在Cache中的命中情况。

当HOST主桥访问的地址不在Cache中命中时，此时在处理器系统中，所有CPU都没有驱动HIT#和HITM#信号，HIT#和HITM#信号都为1，表示HOST主桥访问的地址没有在CPU的Cache中命中，HOST主桥可以简单地将数据写入存储器。当HOST主桥访问的存储器地址在Cache中命中时，Cache行的状态可以为S、E或者为M，此时处理器系统的处理过程相对较为复杂，下一节将专门讨论这种情况。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012263.HTM#_ftnref1) 目前Cortex A8/A9和Intel的Nehelem处理器支持这种方式。

|  |
| --- |
| **3.3.4 PCI设备进行DMA写时发生Cache命中** |
| 发布时间：2013-04-25 11:41:55 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

如果PCI设备访问的地址在某个CPU的Cache行中命中时，可能会出现三种情况。

第一种情况是命中的Cache行其状态为E，即Cache行中的数据与存储器中的数据一致；而第二种情况是命中的Cache行其状态为S。其中E位为1表示该数据在SMP处理器系统中，有且仅有一个CPU的Cache中具有数据副本；而S位为1表示在SMP处理器系统中，该数据至少在两个以上CPU的Cache中具有数据副本。

当Cache行状态为E时，这种情况比较容易处理。因为PCI设备(通过HOST主桥)写入存储器的信息比Cache行中的数据新，而且PCI设备在进行DMA写操作之前，存储器与Cache中数据一致，此时CPU仅需要在Snoop Phase使无效(Invalidate)这个Cache行，然后FSB总线事务将数据写入存储器即可。当然如果FSB总线事务可以将数据直接写入Cache，并将Cache行的状态更改为M，也可提高DMA写的效率，这种方式的实现难度较大，第3.3.5节将介绍这种优化方式。

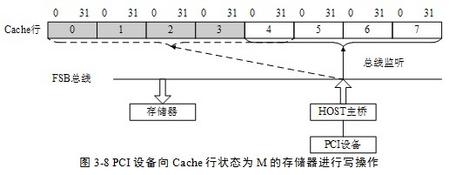
Cache行状态为S时的处理情况与状态为E时的处理情况大同小异，PCI设备在进行写操作时也将数据直接写入主存储器，并使无效状态为S的Cache行。

第三种情况是命中的Cache行其状态为M(Modified)，即Cache行中的数据与存储器的数据不一致，Cache行中保存最新的数据拷贝，主存储器中的部分数据无效。对于SMP系统，此时有且仅有一个CPU中的Cache行的状态为M，因为MESI协议规定存储器中的数据不能在多个CPU的Cache行中的状态为M。

我们假定一个处理器的Cache行长度为32B，即256b。当这个Cache行的状态为M时，表示这个Cache行的某个字节、双字、几个双字、或者整个Cache行中的数据比主存储器中含有的数据新。

假设HOST主桥访问的地址，在Snoop Phase，通过CPU进行总线监听后，发现其对应的Cache行状态为M。此时HOST主桥进行存储器写操作时，处理情况较为复杂，此时这些状态为M的数据需要回写到主存储器。

我们考虑如图3‑8所示的实例。假定处理器的Cache使用回写(Write-Back)策略进行更新。在这个实例中，HOST主桥对存储器的某个地址进行写操作，而所有CPU通过FSB总线进行总线监听时发现，HOST主桥使用的这个目的地址在某个CPU的Cache行命中，此时这个CPU将置HITM#信号为0，并置HIT#信号为1，表示当前Cache行中含有的数据比存储器中含有的数据更新。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130425113742367.jpg)

我们假设此时在Cache行中，阴影部分的数据比存储器中的数据新，而其他数据与存储器保持一致，即在这个Cache行中第0~3个双字的数据是当前处理器系统中最新的数据，而第4~7个双字中的数据与存储器保持一致。

如果PCI设备向存储器写的数据区域可以完全覆盖这些阴影部分，如对第0~5个双字进行写操作时，这种情况不难处理。此时CPU只需在总线监听阶段，将这个Cache行使无效，然后将数据写入存储器即可。因为完成这个存储器写操作之后，PCI设备写入的数据是最新的，而且这个最新的数据将完全覆盖在Cache行中阴影部分的数据，所以CPU只需要简单地将这个Cache行使无效即可。

然而PCI设备(HOST主桥)无法预先知道这些Cache行中的数据哪些是有效的，哪些是无效的，而仅知道命中了一个“被修改过”的Cache行，从而PCI设备(HOST主桥)无法保证能够对Cache行中有效数据进行覆盖。因此PCI设备对存储器进行写操作时，不能简单地使无效(Invalid)状态位为M的Cache行。

我们仍然以图3‑8为例，考虑一个PCI设备将4个双字(第4~7个双字)的数据写入到一个存储器中，这4个双字所访问的数据在某个CPU的Cache行中命中，而且该Cache行的状态为M，而且这个Cache行的前4个双字曾被处理器修改过。

此时CPU对FSB总线监听时，不能简单将当前Cache行使无效，因为这个使无效操作将丢失阴影部分的有效数据。这个阴影部分中的有效数据并没有被PCI设备重新写入，因此在整个处理器系统中，这个阴影部分仍然包含最新的数据。将最新的数据丢弃显然是一种错误做法，将会导致处理器系统的崩溃。

为此HOST主桥需要专门处理这种情况，不同的HOST主桥采用了不同的方法处理这种情况，但无外乎以下三种方法。

(1) CPU进行总线监听后发现，HOST主桥访问的数据命中了一个状态位为M的Cache行，此时存储器控制器将通知HOST主桥重试或者延时处理，并暂时停止HOST主桥发起的这次存储器写操作。随后CPU将状态位为M的Cache行与存储器进行同步后，再使无效这个Cache行。之后HOST主桥在合适的时机，重新发起被HOST主桥要求重试的总线事务，此时CPU再次进行总线监听时不会再次出现Cache命中的情况，因此HOST主桥可以直接将数据写入存储器。许多HOST主桥使用这种方法处理PCI设备的存储器写总线事务。

(2) 首先HOST主桥将接收PCI设备进行DMA写的数据，并将这些数据放入存储器控制器的一个缓冲区中，同时结束PCI设备的存储器写总线事务。之后CPU进行总线监听，如果CPU发现HOST主桥访问的数据命中了一个状态位为M的Cache行时，则这个Cache行放入存储器控制器的另一个缓冲区后，使无效这个Cache行。最后存储器控制器将这两个缓冲区的数据合并然后统一写入到存储器中。

(3) HOST主桥并不结束当前PCI总线周期，而直接进行总线监听，如果CPU进行总线监听发现HOST主桥访问的数据命中了一个状态位为M的Cache行时，则将这个Cache行整体写入存储器控制器的缓冲区后使无效这个Cache行，之后HOST主桥开始从PCI设备接收数据，并将这些数据直接写入这个缓冲区中。最后HOST主桥结束PCI设备的存储器写总线周期，同时存储器控制器将这个缓冲区内的数据写入存储器。

以上这几种情况是PCI设备进行存储器写时，HOST主桥可能的处理情况，其中第1种方法最常用。而x86处理器使用的implicit writeback方式，与第2种方法基本类似。第3种方法与第2种方法并没有本质不同。

但是如果PCI设备对一个或者多个完整Cache行的存储器区域进行写操作时，上述过程显得多余。对完整Cache行进行写操作，可以保证将Cache行对应的存储器区域完全覆盖，此时Cache行中的数据在PCI设备完成这样的操作后，在处理器系统中将不再是最新的。PCI设备进行这样的存储器写操作时，可以直接将数据写入存储器，同时直接使无效状态为M的Cache行。

PCI总线使用存储器写并无效(Memory Write and Invalidate)总线事务，支持这种对一个完整Cache行进行的存储器写总线事务。PCI设备使用这种总线事务时，必须要事先知道当前处理器系统中CPU使用的Cache行大小，使用这种总线事务时，一次总线事务传递数据的大小必须以Cache行为单位对界。为此PCI设备必须使用配置寄存器Cache Line Size保存当前Cache行的大小，Cache Line Size寄存器在PCI配置空间的位置见图2‑9。

存储器读(Memory Read)、存储器多行读(Memory Read Multiple)和存储器单行读(Memory Read Line)总线事务也是PCI总线中的重要总线事务，这些总线事务不仅和Cache有关，还和PCI总线的预读机制有关，本篇在第3.4.5节中重点介绍这些总线事务。

|  |
| --- |
| **3.3.5 DMA写时发生Cache命中的优化** |
| 发布时间：2013-04-27 11:34:36 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

在许多高性能处理器中，还提出了一些新的概念，以加速外设到存储器的DMA写过程。如Freescale的I/O Stashing和Intel的IOAT技术。

如图3‑8所示，当设备进行存储器写时，如果可以对Cache直接进行写操作时，即便这个存储器写命中了一个状态为M的Cache行，可以不将该Cache行的数据回写到存储器中，而是直接将数据写入Cache，之后该Cache行的状态依然为M。采用这种方法可以有效提高设备对存储器进行写操作的效率。采用直接向Cache行写的方法，PCI设备对存储器写命中一个状态为M的Cache行时，将执行以下操作。

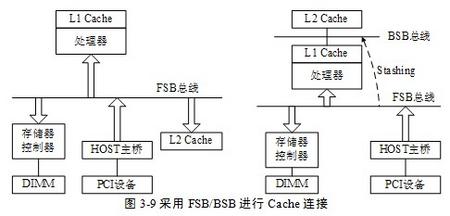
(1) HOST主桥将对存储器的写请求发送到FSB总线上。

(2) CPU通过对FSB监听，发现该写请求在某个Cache行中命中，而且该Cache行的状态为M。

(3) HOST主桥将数据直接写入到Cache行中，并保持Cache行的状态为M。注意此时设备不需要将数据写入存储器中。

从原理上看，这种方法并没有奇特之处，仅需Cache能够提供一个接口，使外部设备能够直接写入即可。但是从具体实现上看，设备直接将数据写入Cache中，还是有相当大的难度。特别是考虑在一个处理器中，可能存在多级Cache，当CPU进行总线监听时，可能是在L1、L2或者L3 Cache中命中，此时的情况较为复杂，多级Cache间的协议状态机远比FSB总线协议复杂的多。

在一个处理器系统中，如果FSB总线事务在“与FSB直接相连的Cache”中命中时，这种情况相对容易处理；但是在与BSB(Back-Side Bus)直接相连的Cache命中时，这种情况较难处理。下文分别对这两种情况进行讨论，在一个处理器中，采用FSB和BSB连接Cache的拓扑如图3‑9所示。

[](http://bbs.ednchina.com/images/attachments/201304/original/20130427113345279.jpg)

当采用FSB总线连接L2 Cache时，L2 Cache直接连接到FSB总线上，设备通过FSB总线向L2 Cache进行写操作并不难实现，MPC8548处理器就是采用了这种结构将L2 Cache直接连接到FSB总线上。

但是由于FSB总线的频率远低于BSB总线频率，因此采用这种结构将影响L2 Cache的访问速度，为此高端处理器多采用BSB总线连接L2 Cache，x86处理器在Pentium Pro之后的高性能处理器都使用BSB总线连接L2 Cache，Freescale的G4系列处理器和最新的P4080处理器也使用BSB总线连接L2 Cache。

当L2 Cache没有直接连接到FSB上时，来自外部设备的数据并不容易到达BSB总线。除了需要考虑Cache连接在BSB总线的情况外，在外部设备进行DMA操作时，还需要考虑多处理器系统的Cache共享一致性协议。设计一个专用通道，将数据从外部设备直接写入到处理器的Cache中并不容易实现。Intel的IOAT和Freescale的I/O Stashing可能使用了这种专用通道技术，直接对L1和L2 Cache进行写操作，并在极大增加了设计复杂度的前提下，提高了处理器系统的整体效率。

以上对Cache进行直接写操作，仅是Intel的IOAT和Freescale的I/O Stashing技术的一个子集。目前Intel和Freescale没有公开这些技术的具体实现细节。在一个处理器系统中，可能存在多级Cache，这些Cache的层次组成结构和状态机模型异常复杂，本章对这些内容不做进一步说明。

|  |
| --- |
| **3.4 预读机制2** |
| 发布时间：2013-05-03 18:14:47 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

软件预读机制由来已久，首先实现预读指令的处理器是Motorola的88110处理器，这颗处理器首先实现了“touch load”指令，这条指令是PowerPC处理器dcbt指令[5]的雏形。88110处理器是Motorola第一颗RISC处理器，具有里程碑意义，这颗处理器从内核到外部总线的设计都具有许多亮点。这颗处理器是Motorola对PowerPC架构做出的巨大贡献，PowerPC架构中著名的60X总线也源于88110处理器。

后来绝大多数处理器都采用这类指令进行软件预读，Intel在i486处理器中提出了Dummy Read指令，这条指令也是后来x86处理器中PREFETCHh指令[6]的雏形。

这些软件预读指令都有一个共同的特点，就是在处理器真正需要数据之前，向存储器发出预读请求，这个预读请求[7]不需要等待数据真正到达存储器之后，就可以执行完毕。从而处理器可以继续执行其他指令，以实现存储器访问与处理器运算同步进行，从而提高了程序的整体执行效率。由此可见，处理器采用软件预读可以有效提高程序的执行效率。我们考虑源代码3‑1所示的实例。

源代码3‑1没有采用软件预读机制的程序

3.4.3 软件预读

软件预读机制由来已久，首先实现预读指令的处理器是Motorola的88110处理器，这颗处理器首先实现了“touch load”指令，这条指令是PowerPC处理器dcbt指令[5]的雏形。88110处理器是Motorola第一颗RISC处理器，具有里程碑意义，这颗处理器从内核到外部总线的设计都具有许多亮点。这颗处理器是Motorola对PowerPC架构做出的巨大贡献，PowerPC架构中著名的60X总线也源于88110处理器。

后来绝大多数处理器都采用这类指令进行软件预读，Intel在i486处理器中提出了Dummy Read指令，这条指令也是后来x86处理器中PREFETCHh指令[6]的雏形。

这些软件预读指令都有一个共同的特点，就是在处理器真正需要数据之前，向存储器发出预读请求，这个预读请求[7]不需要等待数据真正到达存储器之后，就可以执行完毕。从而处理器可以继续执行其他指令，以实现存储器访问与处理器运算同步进行，从而提高了程序的整体执行效率。由此可见，处理器采用软件预读可以有效提高程序的执行效率。我们考虑源代码3‑1所示的实例。

源代码3‑1没有采用软件预读机制的程序

int ip, a[N], b[N];

for (i = 0; i < N; i++)

ip = ip + a[i]\*b[i];

这个例子在对数组进行操作时被经常使用，这段源代码的作用是将int类型的数组a和数组b的每一项进行相乘，然后赋值给ip，其中数组a和b都是Cache行对界的。源代码3‑1中的程序并没有使用预读机制进行优化，因此这段程序在执行时会因为a[i]和b[i]中的数据不在处理器的Cache中，而必须启动存储器读操作。因此在这段程序在执行过程中，必须要等待存储器中的数据后才能继续，从而降低了程序的执行效率。为此我们将程序进行改动，如源代码3‑2所示。

源代码3‑2 采用软件预读机制的程序

int ip, a[N], b[N];  
for (i = 0; i < N; i++) {  
fetch(&a[i+1]);  
fetch(&b[i+1]);  
ip = ip + a[i]\*b[i];  
}

以上程序对变量ip赋值之前，首先预读数组a和b，当对变量ip赋值时，数组a和b中的数据已经在Cache中，因而不需要进行再次进行存储器操作，从而在一定程度上提高了代码的执行效率。以上代码仍然并不完美，首先ip，a[0]和b[0]并没有被预读，其次在一个处理器，预读是以Cache行为单位进行的，因此对a[0]，a[1]进行预读时都是对同一个Cache行进行预读[8]，从而这段代码对同一个Cache行进行了多次预读，从而影响了执行效率。为此我们将程序再次进行改动，如源代码3‑3所示。

源代码3‑3软件预读机制的改进程序

int ip, a[N], b[N];  
fetch(&ip);  
fetch(&a[0]);  
fetch(&b[0]);  
for (i = 0; i < N-4; i+=4) {  
fetch(&a[i+4]);  
fetch(&b[i+4]);  
ip = ip + a[i]\*b[i];  
ip = ip + a[i+1]\*b[i+1];  
ip = ip + a[i+2]\*b[i+2];  
ip = ip + a[i+3]\*b[i+3];  
}  
for (; i < N; i++)  
ip = ip + a[i]\*b[i];

对于以上这个例子，采用这种预读方法可以有效提高执行效率，对此有兴趣的读者可以对以上几个程序进行简单的对比测试。但是提醒读者注意，有些较为先进的编译器，可以自动的加入这些预读语句，程序员可以不手工加入这些预读指令。实际上源代码3‑3中的程序还可以进一步优化。这段程序的最终优化如源代码3‑4所示。

源代码3‑4软件预读机制的改进程序

int ip, a[N], b[N];  
fetch( &ip);  
for (i = 0; i < 12; i += 4){  
fetch( &a[i]);  
fetch( &b[i]);  
}  
for (i = 0; i < N-12; i += 4){  
fetch( &a[i+12]);  
fetch( &b[i+12]);  
ip = ip + a[i] \*b[i];  
ip = ip + a[i+1]\*b[i+1];  
ip = ip + a[i+2]\*b[i+2];  
ip = ip + a[i+3]\*b[i+3];  
}  
for ( ; i < N; i++)  
ip = ip + a[i]\*b[i];

因为我们还可以对ip、数据a和b进行充分预读之后；再一边预读数据，一边计算ip的值；最后计算ip的最终结果。使用这种方法可以使数据预读和计算充分并行，从而优化了整个任务的执行时间。

由以上程序可以发现，采用软件预读机制可以有效地对矩阵运算进行优化，因为矩阵运算进行数据访问时非常有规律，便于程序员或编译器进行优化，但是并不是所有程序都能如此方便地使用软件预读机制。此外预读指令本身也需要占用一个机器周期，在某些情况下，采用硬件预读机制更为合理。

### 3.4.4 硬件预读

采用硬件预读的优点是不需要软件进行干预，也不需要浪费一条预读指令来进行预读。但硬件预读的缺点是预读结果有时并不准确，有时预读的数据并不是程序执行所需要的。在许多处理器中这种硬件预读通常与指令预读协调工作。硬件预读机制的历史比软件预读更为久远，在IBM 370/168处理器系统中就已经支持硬件预读机制。

大多数硬件预读仅支持存储器到Cache的预读，并在程序执行过程中，利用数据的局部性原理进行硬件预读。其中最为简单的硬件预读机制是OBL(One Block Lookahead)机制，采用这种机制，当程序对数据块b进行读取出现Cache Miss时，将数据块b从存储器更新到Cache中，同时对数据块b+1也进行预读并将其放入Cache中；如果数据块b+1已经在Cache中，将不进行预读。

这种OBL机制有很多问题，一个程序可能只使用数据块b中的数据，而不使用数据块b+1中的数据，在这种情况下，采用OBL预读机制没有任何意义。而且使用这种预读机制时，每次预读都可能伴随着Cache Miss，这将极大地影响效率。有时预读的数据块b+1会将Cache中可能有用的数据替换出去，从而造成Cache污染。有时仅预读数据块b+1可能并不足够，有可能程序下一个使用的数据块来自数据块b+2。

为了解决OBL机制存在的问题，有许多新的预读方法涌现出来，如“tagged预读机制”。采用这种机制，将设置一个“tag位”，处理器访问数据块b时，如果数据块b没有在Cache中命中，则将数据块b从存储器更新到Cache中，同时对数据块b+1进行预读并将其放入Cache中；如果数据块b已经在Cache中，但是这个数据块b首次被处理器使用，此时也将数据块b+1预读到Cache中；如果数据块b已经在Cache中，但是这个数据块b已经被处理器使用过，此时不将数据块b+1预读到Cache中。

这种“tagged预读机制”还有许多衍生机制，比如可以将数据块b+1，b+2都预读到Cache中，还可以根据程序的执行信息，将数据块b-1，b-2预读到Cache中。

但是这些方法都无法避免因为预读而造成的Cache污染问题，于是Stream buffer机制被引入。采用该机制，处理器可以将预读的数据块放入Stream Buffer中，如果处理器使用的数据没有在Cache中，则首先在Stream Buffer中查找，采用这种方法可以消除预读对Cache的污染，但是增加了系统设计的复杂性。

与软件预读机制相比，硬件预读机制可以根据程序执行的实际情况进行预读操作，是一种动态预读方法；而软件预读机制需要对程序进行静态分析，并由编译器自动或者由程序员手工加入软件预读指令来实现。

### 3.4.5 PCI总线的预读机制

在一个处理器系统中，预读的目标设备并不仅限于存储器，程序员还可以根据实际需要对外部设备进行预读。但并不是所有的外部设备都支持预读，只有“well-behavior”存储器支持预读。处理器使用的内部存储器，如基于SDRAM、DDR-SDRAM或者SRAM的主存储器是“well-behavior”存储器，有些外部设备也是“well-behavior”存储器。这些well-behavior存储器具有以下特点。

(1) 对这些存储器设备进行读操作时不会改变存储器的内容。显然主存储器具有这种性质。如果一个主存储器的一个数据为0，那么读取这个数据100次也不会将这个结果变为1。但是在外部设备中，一些使用存储器映像寻址的寄存器具有读清除的功能。比如某些中断状态寄存器[[9]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftn9" \o ")。当设备含有未处理的中断请求时，该寄存器的中断状态位为1，对此寄存器进行读操作时，硬件将自动地把该中断位清零，这类采用存储映像寻址的寄存器就不是well-behavior存储器。

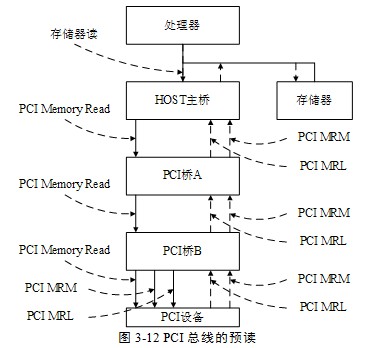
(2) 对“well-behavior”存储器的多次读操作，可以合并为一次读操作。如向这个设备的地址n，n+4，n+8和n+12地址处进行四个双字的读操作，可以合并为对n地址的一次突发读操作(大小为4个双字)。

(3) 对“well-behavior”存储器的多次写操作，可以合并为一次写操作。如向这个设备的地址n，n+4，n+8和n+12地址处进行四个双字的写操作，可以合并为对n地址的一次突发写操作。对于主存储器，进行这种操作不会产生副作用，但是对于有些外部设备，不能进行这种操作。

(4) 对“well-behavior”的存储器写操作，可以合并为一次写操作。向这个设备的地址n，n+1，n+2和n+3地址处进行四个单字的写操作，可以合并为对n地址的一次DW写操作。对主存储器进行这种操作不会出现错误，但是对于有些外部设备，不能进行这种操作。

如果外部设备满足以上四个条件，该外部设备被称为“well-behavior”。PCI配置空间的BAR寄存器中有一个“Prefectchable”位，该位为1时表示这个BAR寄存器所对应的存储器空间支持预读。PCI总线的预读机制需要HOST主桥、PCI桥和PCI设备的共同参与。在PCI总线中，预读机制需要分两种情况进行讨论，一个是HOST处理器通过HOST主桥和PCI桥访问最终的PCI设备；另一个是PCI设备使用DMA机制访问存储器。

PCI总线预读机制的拓扑结构如图3‑12所示。

[](http://bbs.ednchina.com/images/attachments/201305/original/20130503175719618.jpg)

由上图所示，HOST处理器预读PCI设备时，需要经过HOST主桥，并可能通过多级PCI桥，最终到达PCI设备，在这个数据传送路径上，有的PCI桥支持预读，有的不支持预读。而PCI设备对主存储器进行预读时也将经过多级PCI桥。PCI设备除了可以对主存储器进行预读之外，还可以预读其他PCI设备，但是这种情况在实际应用中极少出现，本节仅介绍PCI设备预读主存储器这种情况。

#### 1 HOST处理器预读PCI设备

PCI设备的BAR寄存器可以设置预读位，首先支持预读的BAR寄存器空间必须是一个Well-behavior的存储器空间，其次PCI设备必须能够接收来自PCI桥和HOST主桥的MRM(Memory Read Multiple)和MRL(Memory Read Line)总线事务。

如果PCI设备支持预读，那么当处理器对这个PCI设备进行读操作时，可以通过PCI桥启动预读机制(该PCI桥也需要支持预读)，使用MRM和MRL总线事务，对PCI设备进行预读，并将预读的数据暂时存放在PCI桥的预读缓冲中。

之后当PCI主设备继续读取PCI设备的BAR空间时，如果访问的数据在PCI桥的预读缓冲中，PCI桥可以不对PCI设备发起存储器读总线事务，而是直接从预读缓冲中获取数据，并将其传递给PCI主设备。当PCI主设备完成读总线事务后，PCI桥必须丢弃预读的数据以保证数据的完整性。此外当PCI桥预读的地址空间超越了PCI设备可预读BAR空间边界时，PCI设备需要“disconnect”该总线事务。

如果PCI桥支持“可预读”的存储器空间，而且其下挂接的PCI设备BAR空间也支持预读时，系统软件需要从PCI桥“可预读”的存储器空间中为该PCI设备分配空间。此时PCI桥可以将从PCI设备预读的数据暂存在PCI桥的预读缓冲中。

PCI总线规定，如果下游PCI桥地址空间支持预读，则其上游PCI桥地址空间可以支持也可以不支持预读机制。如图3‑12所示，如果PCI桥B管理的PCI子树使用了可预读空间时，PCI桥A可以不支持可预读空间，此时PCI桥A只能使用存储器读总线事务读取PCI设备，而PCI桥B可以将这个存储器读总线事务转换为MRL或者MRM总线事务，预读PCI设备的BAR空间(如果PCI设备的BAR空间支持预读)，并将预读的数据保存在PCI桥B的数据缓冲中。

但是PCI总线不允许PCI桥A从其“可预读”的地址空间中，为PCI桥B的“不可预读”区域预留空间，因为这种情况将影响数据的完整性。

大多数HOST主桥并不支持对PCI设备的预读，这些HOST主桥并不能向PCI设备发出MRL或者MRM总线事务。由于在许多处理器系统中，PCI设备是直接挂接到HOST主桥上的，如果连HOST主桥也不支持这种预读，即便PCI设备支持了预读机制也没有实际作用。而且如果PCI设备支持预读机制，硬件上需要增加额外的开销，这也是多数PCI设备不支持预读机制的原因。

尽管如此本节仍需要对HOST处理器预读PCI设备进行探讨。假设在图3‑12所示的处理器系统中，HOST主桥和PCI桥A不支持预读，而PCI桥B支持预读，而且处理器的Cache行长度为32B(0x20)。

如果HOST处理器对PCI设备的0x8000-0000~0x8000-0003这段地址空间进行读操作时。HOST主桥将使用存储器读总线事务读取PCI设备的“0x8000-0000~0x8000-0003这段地址空间”，这个存储器读请求首先到达PCI桥A，并由PCI桥A转发给PCI桥B。

PCI桥B发现“0x8000-0000~0x8000-0003这段地址空间”属于自己的可预读存储器区域，即该地址区域在该桥的Prefetchable Memory Base定义的范围内，则将该存储器读请求转换为MRL总线事务，并使用该总线事务从PCI设备[[10]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftn10" \o ")中读取0x8000-0000~0x8000-001F这段数据，并将该数据存放到PCI桥B的预读缓冲中。MRL总线事务将从需要访问的PCI设备的起始地址开始，一直读到当前Cache行边界。

之后当HOST处理器读取0x8000-0004~0x8000-001F这段PCI总线地址空间的数据时，将从PCI桥B的预读缓冲中直接获取数据，而不必对PCI设备进行读取。

#### 2 PCI设备读取存储器

PCI设备预读存储器地址空间时，需要使用MRL或者MRM总线事务。与MRL总线周期不同，MRM总线事务将从需要访问的存储器起始地址开始，一直读到下一个Cache行边界为止。

对于一个Cache行长度为32B(0x20)的处理器系统，如果一个PCI设备对主存储器的0x1000-0000~0x1000-0007这段存储器地址空间进行读操作时，由于这段空间没有跨越Cache行边界，此时PCI设备将使用MRL总线事务对0x1000-0000~0x1000-001F这段地址区域发起存储器读请求。

如果一个PCI设备对主存储器的0x1000-001C~0x1000-0024这段存储器地址空间进行读操作时，由于这段空间跨越了Cache行边界，此时PCI设备将使用MRM总线事务对0x1000-001C~0x1000-002F这段地址空间发起存储器读请求。

在图3‑12所示的例子中，PCI设备读取0x1000-001C~0x1000-0024这段存储器地址空间时，首先将使用MRM总线事务发起读请求，该请求将通过PCI桥B和A最终到达HOST主桥。HOST主桥[[11]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftn11" \o ")将从主存储器中读取0x1000-001C~0x1000-002F这段地址空间的数据。如果PCI桥A也支持下游总线到上游总线的预读，这段数据将传递给PCI桥A；如果PCI桥A和B都支持这种预读，这段数据将到达PCI桥B的预读缓冲。

如果PCI桥A和B都不支持预读，0x1000-0024~0x1000-002F这段数据将缓存在HOST主桥中，HOST主桥仅将0x1000-001C~0x1000-0024这段数据通过PCI桥A和B传递给PCI设备。之后当PCI设备需要读取0x1000-0024~0x1000-002F这段数据时，该设备将根据不同情况，从HOST主桥、PCI桥A或者B中获取数据而不必读取主存储器。值得注意的是，PCI设备在完成一次数据传送后，暂存在HOST主桥中的预读数据将被清除。PCI设备采用这种预读方式，可以极大提高访问主存储器的效率。

PCI总线规范有一个缺陷，就是目标设备并不知道源设备究竟需要读取或者写入多少个数据。例如PCI设备使用DMA读方式从存储器中读取4KB大小的数据时，只能通过PCI突发读方式，一次读取一个或者多个Cache行。

假定PCI总线一次突发读写只能读取32B大小的数据，此时PCI设备读取4KB大小的数据，需要使用128次突发周期才能完成全部数据传送。而HOST主桥只能一个一个的处理这些突发传送，从而存储器控制器并不能准确预知何时PCI设备将停止读取数据。在这种情况下，合理地使用预读机制可以有效地提高PCI总线的数据传送效率。

我们首先假定PCI设备一次只能读取一个Cache行大小的数据，然后释放总线，之后再读取一个Cache行大小的数据。如果使用预读机制，虽然PCI设备在一个总线周期内只能获得一个Cache行大小的数据，但是HOST主桥仍然可以从存储器获得2个Cache行以上的数据，并将这个数据暂存在HOST主桥的缓冲中，之后PCI设备再发起突发周期时，HOST主桥可以不从存储器，而是从缓冲中直接将数据传递给PCI设备，从而降低了PCI设备对存储器访问的次数，提高了整个处理器系统的效率。

以上描述仅是实现PCI总线预读的一个例子，而且仅仅是理论上的探讨。实际上绝大多数半导体厂商都没有公开HOST主桥预读存储器系统的细节，在多数处理器中，HOST主桥以Cache行为单位读取主存储器的内容，而且为了支持PCI设备的预读功能HOST主桥需要设置必要的缓冲部件，这些缓冲的管理策略较为复杂。

目前PCI总线已经逐渐退出历史舞台，进一步深入研究PCI桥和HOST主桥，意义并不重大，不过读者依然可以通过学习PCI体系结构，获得处理器系统中有关外部设备的必要知识，并以此为基础，学习PCIe体系结构。

## 3.5 小结

本章重点介绍了PCI总线的数据交换。其中最重要的内容是与Cache相关的PCI总线事务和预读机制。虽然与Cache相关的PCI总线事务并不多见，但是理解这些内容对于理解PCI和处理器体系结构，非常重要。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM#_ftnref1) 为简便起见，下文将转移指令成功进行转移称为“Taken”；而将不进行转移称为“Not Taken”。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM#_ftnref2) 假定从访问Cache到发现Cache Miss需要一个时钟周期。

[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM#_ftnref3) PowerPC处理器使用dcbt指令，而x86处理器使用PREFETCHh指令，实现这种软件预读。

[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM#_ftnref4) 假定从Cache中获得数据需要一个时钟周期。

[[5]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref5" \o ") dcbt指令是PowerPC处理器的一条存储器预读指令，该指令可以将内存中的数据预读到L1或者L2 Cache中。

[[6]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref6" \o ") PREFETCHh指令是x86处理器的一条存储器预读指令。

[[7]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref7" \o ") 预读指令在一个时钟周期内就可以执行完毕。

[[8]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref8" \o ") 假定这个处理器系统的Cache行长度为4个双字，即128位。

[[9]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref9" \o ") 假设中断状态寄存器支持读清除功能。

[[10]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref10" \o ") 此时PCI设备的这段区域一定是可预读的存储器区域。

[[11]](http://bbs.ednchina.com/BLOG_ARTICLE_3012420.HTM" \l "_ftnref11" \o ") 假设HOST主桥读取存储器时支持预读，多数HOST主桥都支持这种预读。

|  |
| --- |
| **第II篇PCI Express体系结构概述** |
| 发布时间：2013-05-13 11:54:30 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

虽然PCI总线取得了巨大的成功，但是随着处理器主频的不断提高，PCI总线提供的带宽愈发显得捉襟见肘。PCI总线也在不断地进行升级，其位宽和频率从最初的32位/33MHz扩展到64位/66MHz，而PCI-X总线更是将总线频率提高到533MHz，能够提供的最大理论带宽为4263MB。但是PCI总线仍无法解决其体系结构中存在的一些缺陷。PCI总线面临着一系列挑战，包括带宽、流量控制和数据传送质量等。

PCI总线的最高工作频率为66M，最大位宽为64b，从理论上讲，PCI总线可以提供的最大传输带宽为532MB。然而PCI总线作为一个共享总线，在其上的所有PCI设备必须要共享PCI总线的带宽。同时由于PCI总线的协议开销，导致PCI总线可以实际利用的数据带宽远小于其峰值带宽。

PCI总线采用提高总线位宽和频率的方法增加其传输带宽。但是这种方法从性能价格比的角度上看，并不是最优的。数据总线位宽的提高将直接影响芯片的生产成本，64位的PCI总线接口需要设计者使用更多的芯片引脚，从而导致64位的PCI总线接口芯片的价格远高于32位的PCI总线接口芯片。与32位PCI总线接口相比，设计者还需要使用更多的印制板层数来实现64位PCI总线接口。

而提高总线频率，除了给硬件工程师带来了一系列信号完整性的问题之外，更直接影响PCI总线的负载能力。一条33MHz的PCI总线最多可以驱动10个负载，而66Mhz的PCI总线最多只能驱动4个负载。因此片面提高PCI总线的频率和位宽，并不能有效地提高PCI总线的带宽。

除此之外PCI总线在设计之初并没有考虑服务质量的问题。有些实时数据采集卡，音频或者视频的多媒体应用需要PCI总线提供额定带宽，而PCI总线上的设备只能轮流使用PCI总线，当一个设备长期占用PCI总线时，将阻止其他PCI设备使用PCI总线，从而影响了PCI总线的传送质量。

基于以上几个原因，PCI总线在某种程度上说并不能完全适应现代处理器系统的需要，而使用PCIe总线可以有效解决PCI总线存在的一些问题。首先PCIe总线可以提供更大的总线带宽，PCIe V3.0支持的最高总线频率为4GHz，远高于PCI-X总线提供的最高总线频率。

其次PCIe总线支持虚通路VC(Virtual Channel)技术，优先级不同的数据报文可以使用不同的虚通路，而每一路虚通路可以独立设置缓冲，从而相对合理地解决了数据传送过程中存在的服务质量问题。

PCIe总线由若干层次组成，包括事务层、数据链路层和物理层。PCIe总线使用数据报文进行数据传递，这些数据报文需要通过PCIe总线的这些层次。PCIe总线的这种数据传递方式与互联网使用TCP/IP协议进行数据传递有类似之处。

实际上在互联网中存在的许多概念也存在于PCIe总线中，如交换、路由和仲裁机制等，不过这两者之间在实现上的最大不同在于前者主要使用软件程序实现其协议栈，而后者使用硬件逻辑实现。

半导体工艺的逐步提高，使得更多的软件算法可以使用硬件逻辑来实现，这给从事IC Design的工程师带来了巨大的挑战，因为他们使用Verilog/VHDL程序书写的算法，之前是使用C或者是C++这样的高别语言实现的。

PCIe总线在系统软件级与PCI总线兼容，基于PCI总线的系统软件几乎可以不经修改直接移植到PCIe总线中。绝大多数PCI/PCI-X总线使用的总线事务都被PCIe总线保留，而PCI设备使用的配置空间也被PCIe总线继承。基于PCI体系结构的系统编程模型，几乎可以在没有本质变化的前提下，直接在PCIe体系结构中使用。

但是从体系系统的角度上看，PCIe总线还是增加了一些新的特性，其中一些特性不仅仅是称呼上的变化，而且在功能上也得到了增强。如在PCIe体系结构中出现的RC(Root Complex)。RC的主要功能与PCI总线中的HOST主桥类似，但是在HOST主桥的基础上增加了许多功能。

在不同处理器系统中，RC的实现方式不同，因此仅仅用PCIe总线控制器称呼RC是不够的，实际上PCIe总线规范对RC并没有一个合适的解释。RC本身也是随处理器系统的不同而不同，是一个很模糊的概念。

Intel并没有使用PCIe总线控制器，而是使用RC管理PCIe总线，基于深层次的考虑。在x86处理器体系结构中，RC并不仅仅管理PCIe设备的数据访问，而且还包含访问控制、错误处理和虚拟化技术等一系列内容。因此使用PCIe总线控制器统称RC，在x86处理器体系结构中，并不合适。

在PCIe总线中，还有一些特性与PCIe总线协议的实现相关。与PCI总线相比，PCIe总线使用端到端的连接方式，添加流量控制机制，并对“访问序”做出了进一步优化。虽然从系统软件的角度上看，PCI总线与PCIe总线基本一致。但是从硬件设计的角度上看PCIe总线完全不同于PCI总线，基于PCIe总线各类设备的硬件设计难度远大于基于PCI总线的对应设备的设计难度。

目前PCIe总线规范，依然在迅猛发展，但并不是所有PCIe设备都支持这些在PCIe总线的最新规范中提及的概念。一般说来，PCIe总线规范提出的新的概念，最先在x86处理器系统的Chipset和Intel设计的EP中出现。

|  |
| --- |
| **第4章 PCIe总线概述** |
| 发布时间：2013-05-13 12:16:22 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

随着现代处理器技术的发展，在互连领域中，使用高速差分总线替代并行总线是大势所趋。与单端并行信号相比，高速差分信号可以使用更高的时钟频率，从而使用更少的信号线，完成之前需要许多单端并行数据信号才能达到的总线带宽。

PCI总线使用并行总线结构，在同一条总线上的所有外部设备共享总线带宽，而PCIe总线使用了高速差分总线，并采用端到端的连接方式，因此在每一条PCIe链路中只能连接两个设备。这使得PCIe与PCI总线采用的拓扑结构有所不同。PCIe总线除了在连接方式上与PCI总线不同之外，还使用了一些在网络通信中使用的技术，如支持多种数据路由方式，基于多通路的数据传递方式，和基于报文的数据传送方式，并充分考虑了在数据传送中出现服务质量QoS (Quality of Service)问题。

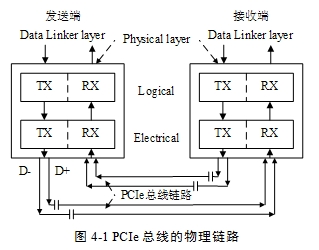
|  |
| --- |
| **4.1 PCIe总线的基础知识** |
| 发布时间：2013-05-23 11:20:29 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

与PCI总线不同，PCIe总线使用端到端的连接方式，在一条PCIe链路的两端只能各连接一个设备，这两个设备互为是数据发送端和数据接收端。PCIe总线除了总线链路外，还具有多个层次，发送端发送数据时将通过这些层次，而接收端接收数据时也使用这些层次。PCIe总线使用的层次结构与网络协议栈较为类似。

### 4.1.1 端到端的数据传递

PCIe链路使用“端到端的数据传送方式”，发送端和接收端中都含有TX(发送逻辑)和RX(接收逻辑)，其结构如图4‑1所示。

  
由上图所示，在PCIe总线的物理链路的一个数据通路(Lane)中，由两组差分信号，共4根信号线组成。其中发送端的TX部件与接收端的RX部件使用一组差分信号连接，该链路也被称为发送端的发送链路，也是接收端的接收链路；而发送端的RX部件与接收端的TX部件使用另一组差分信号连接，该链路也被称为发送端的接收链路，也是接收端的发送链路。一个PCIe链路可以由多个Lane组成。

高速差分信号电气规范要求其发送端串接一个电容，以进行AC耦合。该电容也被称为AC耦合电容。PCIe链路使用差分信号进行数据传送，一个差分信号由D+和D-两根信号组成，信号接收端通过比较这两个信号的差值，判断发送端发送的是逻辑“1”还是逻辑“0”。

与单端信号相比，差分信号抗干扰的能力更强，因为差分信号在布线时要求“等长”、“等宽”、“贴近”，而且在同层。因此外部干扰噪声将被“同值”而且“同时”加载到D+和D-两根信号上，其差值在理想情况下为0，对信号的逻辑值产生的影响较小。因此差分信号可以使用更高的总线频率。

此外使用差分信号能有效抑制电磁干扰EMI(Electro Magnetic Interference)。由于差分信号D+与D-距离很近而且信号幅值相等、极性相反。这两根线与地线间耦合电磁场的幅值相等，将相互抵消，因此差分信号对外界的电磁干扰较小。当然差分信号的缺点也是显而易见的，一是差分信号使用两根信号传送一位数据；二是差分信号的布线相对严格一些。

PCIe链路可以由多条Lane组成，目前PCIe链路可以支持1、2、4、8、12、16和32个Lane，即×1、×2、×4、×8、×12、×16和×32宽度的PCIe链路。每一个Lane上使用的总线频率与PCIe总线使用的版本相关。

第1个PCIe总线规范为V1.0，之后依次为V1.0a，V1.1，V2.0和V2.1。目前PCIe总线的最新规范为V2.1，而V3.0正在开发过程中，预计在2010年发布。不同的PCIe总线规范所定义的总线频率和链路编码方式并不相同，如表4‑1所示。

表4‑1 PCIe总线规范与总线频率和编码的关系

|  |  |  |  |
| --- | --- | --- | --- |
| PCIe总线规范 | 总线频率[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012735.HTM#_ftn1) | 单Lane的峰值带宽 | 编码方式 |
| 1.x | 1.25GHz | 2.5GT/s | 8/10b编码 |
| 2.x | 2.5GHz | 5GT/s | 8/10b编码 |
| 3.0 | 4GHz | 8GT/s | 128/130b编码 |

如上表所示，不同的PCIe总线规范使用的总线频率并不相同，其使用的数据编码方式也不相同。PCIe总线V1.x和V2.0规范在物理层中使用8/10b编码，即在PCIe链路上的10 bit中含有8 bit的有效数据；而V3.0规范使用128/130b编码方式，即在PCIe链路上的130 bit中含有128 bit的有效数据。

由上表所示，V3.0规范使用的总线频率虽然只有4GHz，但是其有效带宽是V2.x的两倍。下文将以V2.x规范为例，说明不同宽度PCIe链路所能提供的峰值带宽，如表4‑2所示。

表4‑2 PCIe总线的峰值带宽

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| PCIe总线的数据位宽 | ×1 | ×2 | ×4 | ×8 | ×12 | ×16 | ×32 |
| 峰值带宽(GT/s) | 5 | 10 | 20 | 40 | 60 | 80 | 160 |

由上表所示，×32的PCIe链路可以提供160GT/s的链路带宽，远高于PCI/PCI-X总线所能提供的峰值带宽。而即将推出的PCIe V3.0规范使用4GHz的总线频率，将进一步提高PCIe链路的峰值带宽。

在PCIe总线中，使用GT(Gigatransfer)计算PCIe链路的峰值带宽。GT是在PCIe链路上传递的峰值带宽，其计算公式为总线频率×数据位宽×2。

在PCIe总线中，影响有效带宽的因素有很多，因而其有效带宽较难计算，这部分内容详见第8.4.1节。尽管如此，PCIe总线提供的有效带宽还是远高于PCI总线。PCIe总线也有其弱点，其中最突出的问题是传送延时。

PCIe链路使用串行方式进行数据传送，然而在芯片内部，数据总线仍然是并行的，因此PCIe链路接口需要进行串并转换，这种串并转换将产生较大的延时。除此之外PCIe总线的数据报文需要经过事务层、数据链路层和物理层，这些数据报文在穿越这些层次时，也将带来延时。本书将在第8.4节详细讨论PCIe总线的延时与带宽之间的联系。

在基于PCIe总线的设备中，×1的PCIe链路最为常见，而×12的PCIe链路极少出现，×4和×8的PCIe设备也不多见。Intel通常在ICH中集成了多个×1的PCIe链路用来连接低速外设，而在MCH中集成了一个×16的PCIe链路用于连接显卡控制器。而PowerPC处理器通常能够支持×8、×4、×2和×1的PCIe链路。

PCIe总线物理链路间的数据传送使用基于时钟的同步传送机制，但是在物理链路上并没有时钟线，PCIe总线的接收端含有时钟恢复模块CDR(Clock Data Recovery)，CDR将从接收报文中提取接收时钟，从而进行同步数据传递。

值得注意的是，在一个PCIe设备中除了需要从报文中提取时钟外，还使用了REFCLK+和REFCLK-信号对作为本地参考时钟，这个信号对的描述见下文。

### 4.1.2 PCIe总线使用的信号

PCIe设备使用两种电源信号供电，分别是Vcc与Vaux，其额定电压为3.3V。其中Vcc为主电源，PCIe设备使用的主要逻辑模块均使用Vcc供电，而一些与电源管理相关的逻辑使用Vaux供电。在PCIe设备中，一些特殊的寄存器通常使用Vaux供电，如Sticky Register，此时即使PCIe设备的Vcc被移除，这些与电源管理相关的逻辑状态和这些特殊寄存器的内容也不会发生改变。

在PCIe总线中，使用Vaux的主要原因是为了降低功耗和缩短系统恢复时间。因为Vaux在多数情况下并不会被移除，因此当PCIe设备的Vcc恢复后，该设备不用重新恢复使用Vaux供电的逻辑，从而设备可以很快地恢复到正常工作状状态。

PCIe链路的最大宽度为×32，但是在实际应用中，×32的链路宽度极少使用。在一个处理器系统中，一般提供×16的PCIe插槽，并使用PETp0~15、PETn0~15和PERp0~15、PERn0~15共64根信号线组成32对差分信号，其中16对PETxx信号用于发送链路，另外16对PERxx信号用于接收链路。除此之外PCIe总线还使用了下列辅助信号。

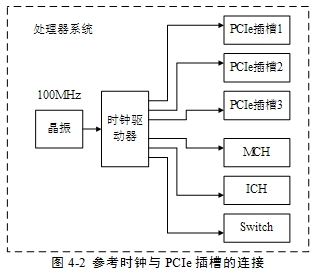
#### 1 PERST#信号

该信号为全局复位信号，由处理器系统提供，处理器系统需要为PCIe插槽和PCIe设备提供该复位信号。PCIe设备使用该信号复位内部逻辑。当该信号有效时，PCIe设备将进行复位操作。PCIe总线定义了多种复位方式，其中Cold Reset和Warm Reset这两种复位方式的实现与该信号有关，详见第4.1.5节。

#### 2 REFCLK+和REFCLK-信号

在一个处理器系统中，可能含有许多PCIe设备，这些设备可以作为Add-In卡与PCIe插槽连接，也可以作为内置模块，与处理器系统提供的PCIe链路直接相连，而不需要经过PCIe插槽。PCIe设备与PCIe插槽都具有REFCLK+和REFCLK-信号，其中PCIe插槽使用这组信号与处理器系统同步。

在一个处理器系统中，通常采用专用逻辑向PCIe插槽提供REFCLK+和REFCLK-信号，如图4‑2所示。其中100Mhz的时钟源由晶振提供，并经过一个“一推多”的差分时钟驱动器生成多个同相位的时钟源，与PCIe插槽一一对应连接。



PCIe插槽需要使用参考时钟，其频率范围为100MHz±300ppm。处理器系统需要为每一个PCIe插槽、MCH、ICH和Switch提供参考时钟。而且要求在一个处理器系统中，时钟驱动器产生的参考时钟信号到每一个PCIe插槽(MCH、ICH和Swith)的距离差在15英寸之内。通常信号的传播速度接近光速，约为6英寸/ns，由此可见，不同PCIe插槽间REFCLK+和REFCLK-信号的传送延时差约为2.5ns。

当PCIe设备作为Add-In卡连接在PCIe插槽时，可以直接使用PCIe插槽提供的REFCLK+和REFCLK-信号，也可以使用独立的参考时钟，只要这个参考时钟在100MHz±300ppm范围内即可。内置的PCIe设备与Add-In卡在处理REFCLK+和REFCLK-信号时使用的方法类似，但是PCIe设备可以使用独立的参考时钟，而不使用REFCLK+和REFCLK-信号。

在PCIe设备配置空间的Link Control Register中，含有一个“Common Clock Configuration”位。当该位为1时，表示该设备与PCIe链路的对端设备使用“同相位”的参考时钟；如果为0，表示该设备与PCIe链路的对端设备使用的参考时钟是异步的。

在PCIe设备中，“Common Clock Configuration”位的缺省值为0，此时PCIe设备使用的参考时钟与对端设备没有任何联系，PCIe链路两端设备使用的参考时钟可以异步设置。这个异步时钟设置方法对于使用PCIe链路进行远程连接时尤为重要。

在一个处理器系统中，如果使用PCIe链路进行机箱到机箱间的互连，因为参考时钟可以异步设置，机箱到机箱之间进行数据传送时仅需要差分信号线即可，而不需要参考时钟，从而极大降低了连接难度。

#### 3 WAKE#信号

当PCIe设备进入休眠状态，主电源已经停止供电时，PCIe设备使用该信号向处理器系统提交唤醒请求，使处理器系统重新为该PCIe设备提供主电源Vcc。在PCIe总线中，WAKE#信号是可选的，因此使用WAKE#信号唤醒PCIe设备的机制也是可选的。值得注意的是产生该信号的硬件逻辑必须使用辅助电源Vaux供电。

WAKE#是一个Open Drain信号，一个处理器的所有PCIe设备可以将WAKE#信号进行线与后，统一发送给处理器系统的电源控制器。当某个PCIe设备需要被唤醒时，该设备首先置WAKE#信号有效，然后在经过一段延时之后，处理器系统开始为该设备提供主电源Vcc，并使用PERST#信号对该设备进行复位操作。此时WAKE#信号需要始终保持为低，当主电源Vcc上电完成之后，PERST#信号也将置为无效并结束复位，WAKE#信号也将随之置为无效，结束整个唤醒过程。

PCIe设备除了可以使用WAKE#信号实现唤醒功能外，还可以使用Beacon信号实现唤醒功能。与WAKE#信号实现唤醒功能不同，Beacon使用In-band信号，即差分信号D+和D-实现唤醒功能。Beacon信号DC平衡，由一组通过D+和D-信号生成的脉冲信号组成。这些脉冲信号宽度的最小值为2ns，最大值为16us。当PCIe设备准备退出L2状态(该状态为PCIe设备使用的一种低功耗状态)时，可以使用Beacon信号，提交唤醒请求。

#### 4 SMCLK和SMDAT信号

SMCLK和SMDAT信号与x86处理器的SMBus(System Mangement Bus)相关。SMBus于1995年由Intel提出，SMBus由SMCLK和SMDAT信号组成。SMBus源于I2C总线，但是与I2C总线存在一些差异。

SMBus的最高总线频率为100KHz，而I2C总线可以支持400KHz和2MHz的总线频率。此外SMBus上的从设备具有超时功能，当从设备发现主设备发出的时钟信号保持低电平超过35ms时，将引发从设备的超时复位。在正常情况下，SMBus的主设备使用的总线频率最低为10KHz，以避免从设备在正常使用过程中出现超时。

在SMbus中，如果主设备需要复位从设备时，可以使用这种超时机制。而I2C总线只能使用硬件信号才能实现这种复位操作，在I2C总线中，如果从设备出现错误时，单纯通过主设备是无法复位从设备的。

SMBus还支持Alert Response机制。当从设备产生一个中断时，并不会立即清除该中断，直到主设备向0b0001100地址发出命令。

上文所述的SMBus和I2C总线的区别还是局限于物理层和链路层上，实际上SMBus还含有网络层。SMBus还在网络层上定义了11种总线协议，用来实现报文传递。

SMBus在x86处理器系统中得到了大规模普及，其主要作用是管理处理器系统的外部设备，并收集外设的运行信息，特别是一些与智能电源管理相关的信息。PCI和PCIe插槽也为SMBus预留了接口，以便于PCI/PCIe设备与处理器系统进行交互。

在Linux系统中，SMBus得到了广泛的应用，ACPI也为SMBus定义了一系列命令，用于智能电池、电池充电器与处理器系统之间的通信。在Windows操作系统中，有关外部设备的描述信息，也是通过SMBus获得的。

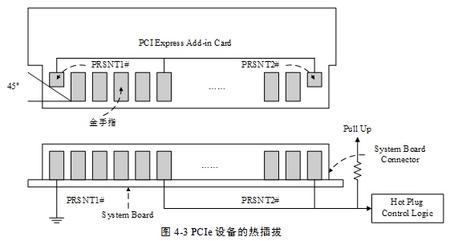
#### 5 JTAG信号

JTAG(Joint Test Action Group)是一种国际标准测试协议，与IEEE 1149.1兼容，主要用于芯片内部测试。目前绝大多数器件都支持JTAG测试标准。JTAG信号由TRST#、TCK、TDI、TDO和TMS信号组成。其中TRST#为复位信号；TCK为时钟信号；TDI和TDO分别与数据输入和数据输出对应；而TMS信号为模式选择。

JTAG允许多个器件通过JTAG接口串联在一起，并形成一个JTAG链。目前FPGA和EPLD可以借用JTAG接口实现在线编程ISP(In-System Programming)功能。处理器也可以使用JTAG接口进行系统级调试工作，如设置断点、读取内部寄存器和存储器等一系列操作。除此之外JTAG接口也可用作“逆向工程”，分析一个产品的实现细节，因此在正式产品中，一般不保留JTAG接口。

#### 6 PRSNT1#和PRSNT2#信号

PRSNT1#和PRSNT2#信号与PCIe设备的热插拔相关。在基于PCIe总线的Add-in卡中，PRSNT1#和PRSNT2#信号直接相连，而在处理器主板中，PRSNT1#信号接地，而PRSNT2#信号通过上拉电阻接为高。PCIe设备的热插拔结构如图4‑3所示。



如上图所示，当Add-In卡没有插入时，处理器主板的PRSNT2#信号由上拉电阻接为高，而当Add-In卡插入时主板的PRSNT2#信号将与PRSNT1#信号通过Add-In卡连通，此时PRSNT2#信号为低。处理器主板的热插拔控制逻辑将捕获这个“低电平”，得知Add-In卡已经插入，从而触发系统软件进行相应地处理。

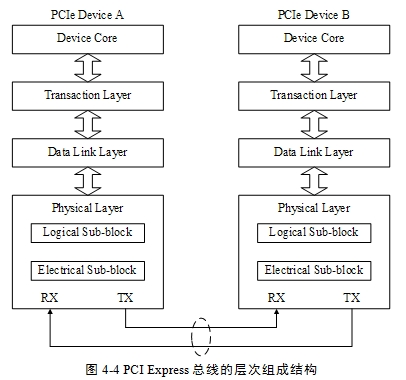
Add-In卡拔出的工作机制与插入类似。当Add-in卡连接在处理器主板时，处理器主板的PRSNT2#信号为低，当Add-In卡拔出后，处理器主板的PRSNT2#信号为高。处理器主板的热插拔控制逻辑将捕获这个“高电平”，得知Add-In卡已经被拔出，从而触发系统软件进行相应地处理。

不同的处理器系统处理PCIe设备热拔插的过程并不相同，在一个实际的处理器系统中，热拔插设备的实现也远比图4‑3中的示例复杂得多。值得注意的是，在实现热拔插功能时，Add-in Card需要使用“长短针”结构。

如图4‑3所示，PRSNT1#和PRSNT2#信号使用的金手指长度是其他信号的一半。因此当PCIe设备插入插槽时，PRSNT1#和PRSNT2#信号在其他金手指与PCIe插槽完全接触，并经过一段延时后，才能与插槽完全接触；当PCIe设备从PCIe插槽中拔出时，这两个信号首先与PCIe插槽断连，再经过一段延时后，其他信号才能与插槽断连。系统软件可以使用这段延时，进行一些热拔插处理。

### 4.1.3 PCIe总线的层次结构

PCIe总线采用了串行连接方式，并使用数据包(Packet)进行数据传输，采用这种结构有效去除了在PCI总线中存在的一些边带信号，如INTx和PME#等信号。在PCIe总线中，数据报文在接收和发送过程中，需要通过多个层次，包括事务层、数据链路层和物理层。PCIe总线的层次结构如图4‑4所示。



PCIe总线的层次组成结构与网络中的层次结构有类似之处，但是PCIe总线的各个层次都是使用硬件逻辑实现的。在PCIe体系结构中，数据报文首先在设备的核心层(Device Core)中产生，然后再经过该设备的事务层(Transaction Layer)、数据链路层(Data Link Layer)和物理层(Physical Layer)，最终发送出去。而接收端的数据也需要通过物理层、数据链路和事务层，并最终到达Device Core。

#### 1 事务层

事务层定义了PCIe总线使用总线事务，其中多数总线事务与PCI总线兼容。这些总线事务可以通过Switch等设备传送到其他PCIe设备或者RC。RC也可以使用这些总线事务访问PCIe设备。

事务层接收来自PCIe设备核心层的数据，并将其封装为TLP(Transaction Layer Packet)后，发向数据链路层。此外事务层还可以从数据链路层中接收数据报文，然后转发至PCIe设备的核心层。

事务层的一个重要工作是处理PCIe总线的“序”。在PCIe总线中，“序”的概念非常重要，也较难理解。在PCIe总线中，事务层传递报文时可以乱序，这为PCIe设备的设计制造了不小的麻烦。事务层还使用流量控制机制保证PCIe链路的使用效率。有关事务层的详细说明见第6章。

#### 2 数据链路层

数据链路层保证来自发送端事务层的报文可以可靠、完整地发送到接收端的数据链路层。来自事务层的报文在通过数据链路层时，将被添加Sequence Number前缀和CRC后缀。数据链路层使用ACK/NAK协议保证报文的可靠传递。

PCIe总线的数据链路层还定义了多种DLLP(Data Link Layer Packet)，DLLP产生于数据链路层，终止于数据链路层。值得注意的是，TLP与DLLP并不相同，DLLP并不是由TLP加上Sequence Number前缀和CRC后缀组成的。

#### 3 物理层

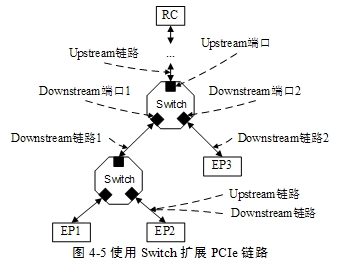
物理层是PCIe总线的最底层，将PCIe设备连接在一起。PCIe总线的物理电气特性决定了PCIe链路只能使用端到端的连接方式。PCIe总线的物理层为PCIe设备间的数据通信提供传送介质，为数据传送提供可靠的物理环境。

物理层是PCIe体系结构最重要，也是最难以实现的组成部分。PCIe总线的物理层定义了LTSSM(Link Training and Status State Machine)状态机，PCIe链路使用该状态机管理链路状态，并进行链路训练、链路恢复和电源管理。

PCIe总线的物理层还定义了一些专门的“序列”，有的书籍将物理层这些“序列”称为PLP(Phsical Layer Packer)，这些序列用于同步PCIe链路，并进行链路管理。值得注意的是PCIe设备发送PLP与发送TLP的过程有所不同。对于系统软件而言，物理层几乎不可见，但是系统程序员仍有必要较为深入地理解物理层的工作原理。

### 4.1.4 PCIe链路的扩展

PCIe链路使用端到端的数据传送方式。在一条PCIe链路中，这两个端口是完全对等的，分别连接发送与接收设备，而且一个PCIe链路的一端只能连接一个发送设备或者接收设备。因此PCIe链路必须使用Switch扩展PCIe链路后，才能连接多个设备。使用Switch进行链路扩展的实例如图4‑5所示。



在PCIe总线中，Switch[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012735.HTM#_ftn2)是一个特殊的设备，该设备由1个上游端口和2~n个下游端口组成。PCIe总线规定，在一个Switch中可以与RC直接或者间接相连[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3012735.HTM#_ftn3)的端口为上游端口，在PCIe总线中，RC的位置一般在上方，这也是上游端口这个称呼的由来。在Switch中除了上游端口外，其他所有端口都被称为下游端口。下游端口一般与EP相连，或者连接下一级Switch继续扩展PCIe链路。其中与上游端口相连的PCIe链路被称为上游链路，与下游端口相连的PCIe链路被称为下游链路。

上游链路和下游链路是一个相对的概念。如上图所示，Switch与EP2连接的PCIe链路，对于EP2而言是上游链路，而对Switch而言是下游链路。

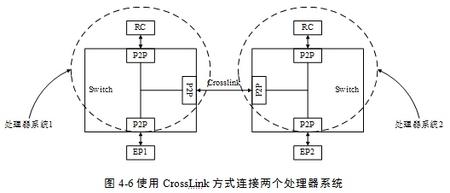
在上图所示的Switch中含有3个端口，其中一个是上游端口(Upstream Port)，而其他两个为下游端口(Downstream Port)。其中上游端口与RC或者其他Switch的下游端口相连，而下游端口与EP或者其他Switch的上游端口相连。

在Switch中，还有两个与端口相关的概念，分别是Egress端口和Ingress端口。这两个端口与通过Switch的数据流向有关。其中Egress端口指发送端口，即数据离开Switch使用的端口；Ingress端口指接收端口即数据进入Switch使用的端口。

Egress端口和Ingress端口与上下游端口没有对应关系。在Switch中，上下游端口可以作为Egress端口，也可以作为Ingress端口。如图4‑5所示，RC对EP3的内部寄存器进行写操作时，Switch的上游端口为Ingress端口，而下游端口为Egress端口；当EP3对主存储器进行DMA写操作时，该Switch的上游端口为Egress端口，而下游端口为Ingress端口。

PCIe总线还规定了一种特殊的Switch连接方式，即Crosslink连接模式。支持这种模式的Switch，其上游端口可以与其他Switch的上游端口连接，其下游端口可以与其他Switch的下游端口连接。

PCIe总线提供CrossLink连接模式的主要目的是为了解决不同处理器系统之间的互连，如图4‑6所示。使用CrossLink连接模式时，虽然从物理结构上看，一个Switch的上/下游端口与另一个Switch的上/下游端口直接相连，但是这个PCIe链路经过训练后，仍然是一个端口作为上游端口，而另一个作为下游端口。



处理器系统1与处理器系统2间的数据交换可以通过Crosslink进行。当处理器系统1(2)访问的PCI总线域的地址空间或者Requester ID不在处理器系统1(2)内时，这些数据将被Crosslink端口接收，并传递到对端处理器系统中。Crosslink对端接口的P2P桥将接收来自另一个处理器域的数据请求，并将其转换为本处理器域的数据请求。

使用Crosslink方式连接两个拓扑结构完全相同的处理器系统时，仍然有不足之处。假设图4‑6中的处理器系统1和2的RC使用的ID号都为0，而主存储器都是从0x0000-0000开始编址时。当处理器1读取EP2的某段PCI总线空间时，EP2将使用ID路由方式，将完成报文传送给ID号为0的PCI设备，此时是处理器2的RC而不是处理器1的RC收到EP2的数据。因为处理器1和2的RC使用的ID号都为0，EP2不能区分这两个RC。

由上所述，使用Crosslink方式并不能完全解决两个处理器系统的互连问题，因此在有些Switch中支持非透明桥结构。这种结构与PCI总线非透明桥的实现机制类似，本章对此不做进一步说明。

使用非透明桥仅解决了两个处理器间数据通路问题，但是不便于NUMA结构对外部设备的统一管理。PCIe总线对此问题的最终解决方法是使用MR-IOV技术，该技术要求Switch具有多个上游端口分别与不同的RC互连。目前PLX公司已经可以提供具有多个上游端口的Switch，但是尚未实现MR-IOV技术涉及的一些与虚拟化相关的技术。

即便MR-IOV技术可以合理解决多个处理器间的数据访问和对PCIe设备的配置管理，使用PCIe总线进行两个或者多个处理器系统间的数据传递仍然是一个不小问题。因为PCIe总线的传送延时仍然是制约其在大规模处理器系统互连中应用的重要因素。

|  |
| --- |
| **4.2 PCIe体系结构的组成部件** |
| 发布时间：2013-05-23 11:37:38 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

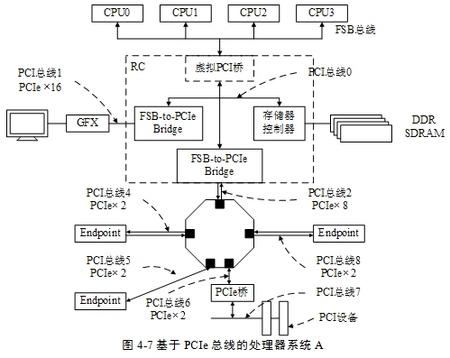
PCIe总线作为处理器系统的局部总线，其作用与PCI总线类似，主要目的是为了连接处理器系统中的外部设备，当然PCIe总线也可以连接其他处理器系统。在不同的处理器系统中，PCIe体系结构的实现方法略有不同。但是在大多数处理器系统中，都使用了RC、Switch和PCIe-to-PCI桥这些基本模块连接PCIe和PCI设备。在PCIe总线中，基于PCIe总线的设备，也被称为EP(Endpoint)。

### 4.2.1 基于PCIe架构的处理器系统

在不同的处理器系统中，PCIe体系结构的实现方式不尽相同。PCIe体系结构以Intel的x86处理器为蓝本实现，已被深深地烙下x86处理器的印记。在PCIe总线规范中，有许多内容是x86处理器独有的，也仅在x86处理器的Chipset中存在。在PCIe总线规范中，一些最新的功能也在Intel的Chipset中率先实现。本节将以一个虚拟的处理器系统A和PowerPC处理器为例简要介绍RC的实现，并简单归纳RC的通用实现机制。

#### 1 处理器系统A

在有些处理器系统中，没有直接提供PCI总线，此时需要使用PCIe桥，将PCIe链路转换为PCI总线之后，才能连接PCI设备。在PCIe体系结构中，也存在PCI总线号的概念，其编号方式与PCI总线兼容。一个基于PCIe架构的处理器系统A如图4‑7所示。



在上图的结构中，处理器系统首先使用一个虚拟的PCI桥分离处理器系统的存储器域与PCI总线域。FSB总线下的所有外部设备都属于PCI总线域。与这个虚拟PCI桥直接相连的总线为PCI总线0。这种架构与Intel的x86处理器系统较为类似。

在这种结构中，RC由两个FSB-to-PCIe桥和存储器控制器组成。值得注意的是在图4‑7中，虚拟PCI桥的作用只是分离存储器域与PCI总线域，但是并不会改变信号的电气特性。RC与处理器通过FSB连接，而从电气特性上看，PCI总线0与FSB兼容，因此在PCI总线0上挂接的是FSB-to-PCIe桥，而不是PCI-to-PCIe桥。

在PCI总线0上有一个存储器控制器和两个FSB-to-PCIe桥。这两个FSB-to-PCIe桥分别推出一个×16和×8的PCIe链路，其中×16的PCIe链路连接显卡控制器(GFX)，其编号为PCI总线1；×8的PCIe链路连接一个Switch进行PCIe链路扩展。而存储器控制器作为PCI总线0的一个Agent设备，连接DDR插槽或者颗粒。

此外在这个PCI总线上还可能连接了一些使用“PCI配置空间”管理的设备，这些设备的访问方法与PCI总线兼容，在x86处理器的Chipset中集成了一些内嵌的设备。这些内嵌的设备使用均使用“PCI配置空间”进行管理，包括存储器控制器。

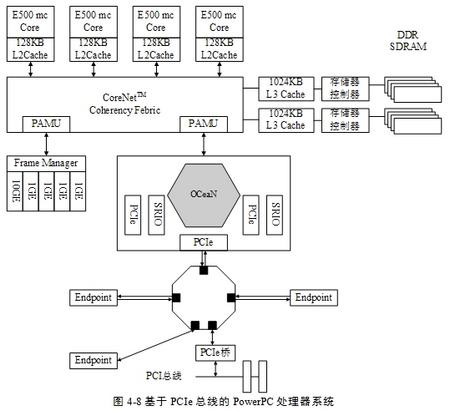
PCIe总线使用端到端的连接方式，因此只有使用Switch才能对PCIe链路进行扩展，而每扩展一条PCIe链路将产生一个新的PCI总线号。如图4‑7所示，Switch可以将1个×8的PCIe端口扩展为4个×2的PCIe端口，其中每一个PCIe端口都可以挂接EP。除此之外PCIe总线还可以使用PCIe桥，将PCIe总线转换为PCI总线或者PCI-X总线，之后挂接PCI/PCI-X设备。多数x86处理器系统使用这种结构连接PCIe或者PCI设备。

在PCIe总线规范中并没有明确提及PCIe主桥，而使用RC概括除了处理器之外的所有与PCIe总线相关的内容。在PCIe体系结构中，RC是一个很模糊也很混乱的概念。Intel使用PCI总线的概念管理所有外部设备，包括与这些外部设备相关的寄存器，因此在RC中包含一些实际上与PCIe总线无关的寄存器。使用这种方式有利于系统软件使用相同的平台管理所有外部设备，也利于平台软件的一致性，但是仍有其不足之处。

#### 2 PowerPC处理器

PowerPC处理器挂接外部设备使用的拓扑结构与x86处理器不同。在PowerPC处理器中，虽然也含有PCI/PCIe总线，但是仍然有许多外部设备并不是连接在PCI总线上的。在PowerPC处理器中，PCI/PCIe总线并没有在x86处理器中的地位。在PowerPC处理器中，还含有许多内部设备，如TSEC(Three Speed Ethenet Controller)和一些内部集成的快速设备，与SoC平台总线直接相连，而不与PCI/PCIe总线相连。在PowerPC处理器中，PCI/PCIe总线控制器连接在SoC平台总线的下方。

Freescale即将发布的P4080处理器，采用的互连结构与之前的PowerPC处理器有较大的不同。P4080处理器是Freescale第一颗基于E500 mc内核的处理器。E500 mc内核与之前的E500 V2和V1相比，从指令流水线结构、内存管理和中断处理上说并没有本质的不同。E500mc内核内置了一个128KB大小的L2 Cache，该Cache连接在BSB总线上；而E500 V1/V2内核中并不含有L2 Cache，而仅含有L1 Cache而且与FSB直接相连。在E500mc内核中，还引入了虚拟化的概念。在P4080处理器中，一些快速外部设备，如DDR控制器、以太网控制器和PCI/PCIe总线接口控制器也是直接或者间接地连接到CoreNet中，在P4080处理器，L3 Cache也是连接到CoreNet中。P4080处理器的拓扑结构如图4‑8所示。

  
目前Freescale并没有公开P4080处理器的L1、L2和L3 Cache如何进行Cache共享一致性。多数采用CoreNet架构互连的处理器系统使用目录表法进行Cache共享一致性，如Intel的Nehelem EX处理器。P4080处理器并不是一个追求峰值运算的SMP处理器系统，而针对Data Plane的应用，因此P4080处理器可能并没有使用基于目录表的Cache一致性协议。在基于全互连网络的处理器系统中如果使用“类总线监听法”进行Cache共享一致性，将不利于多个CPU共享同一个存储器系统，在Cache一致性的处理过程中容易形成瓶颈。

如图4‑8所示，P4080处理器的设计重点并不是E500mc内核，而是CoreNet。CoreNet内部由全互连网络组成，其中任意两个端口间的通信并不会影响其他端口间的通信。与MPC8548处理器相同，P4080处理器也使用OceaN[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftn1)结构连接PCIe与RapidIO接口。

在P4080处理器中不存在RC的概念，而仅存在PCIe总线控制器，当然也可以认为在P4080处理器中，PCIe总线控制器即为RC。P4080处理器内部含有3个PCIe总线控制器，如果该处理器需要连接更多的PCIe设备时，需要使用Switch扩展PCIe链路。

在P4080处理器中，所有外部设备与处理器内核都连接在CoreNet中，而不使用传统的SoC平台总线[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftn2)进行连接，从而在有效提高了处理器与外部设备间通信带宽的同时，极大降低了访问延时。此外P4080处理器系统使用PAMU(Peripheral Access Management Unit)分隔外设地址空间与CoreNet地址空间。在这种结构下，10GE/1GE接口使用的地址空间与PCI总线空间独立。

P4080处理器使用的PAMU是对MPC8548处理器ATMU的进一步升级。使用这种结构时，外部设备使用的地址空间、PCI总线域地址空间和存储器域地址空间的划分更加明晰。在P4080处理器中，存储器控制器和存储器都属于一个地址空间，即存储器域地址空间。此外这种结构还使用OCeaN连接SRIO[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftn3)和PCIe总线控制器，使得在OCeaN中的PCIe端口之间[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftn4)可以直接通信，而不需要通过CoreNet，从而减轻了CoreNet的负载。

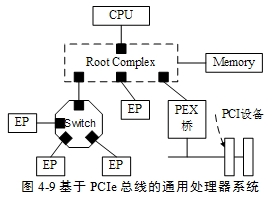
从内核互连和外部设备互连的结构上看，这种结构具有较大的优势。但是采用这种结构需要使用占用芯片更多的资源，CoreNet的设计也十分复杂。而最具挑战的问题是，在这种结构之下，Cache共享一致性模型的设计与实现。在Boxboro EX处理器系统中，可以使用QPI将多个处理器系统进行点到点连接，也可以组成一个全互连的处理器系统。这种结构与P4080处理器使用的结构类似，但是Boxboro EX处理器系统包含的CPU更多。

#### 3 基于PCIe总线的通用处理器结构

在不同的处理器系统中，RC的实现有较大差异。PCIe总线规范并没有规定RC的实现细则。在有些处理器系统中，RC相当于PCIe主桥，也有的处理器系统也将PCIe主桥称为PCIe总线控制器。而在x86处理器系统中，RC除了包含PCIe总线控制器之外，还包含一些其他组成部件，因此RC并不等同于PCIe总线控制器。

如果一个RC中可以提供多个PCIe端口，这种RC也被称为多端口RC。如MPC8572处理器的RC可以直接提供3条PCIe链路，因此可以直接连接3个EP。如果MPC8572处理器需要连接更多EP时，需要使用Switch进行链路扩展。

而在x86处理器系统中，RC并不是存在于一个芯片中，如在Montevina平台中，RC由MCH和ICH两个芯片组成。本节并不对x86和PowerPC处理器使用的PCIe总线结构做进一步讨论，而只介绍这两种结构的相同之处。一个通用的，基于PCIe总线的处理器系统如图4‑9所示。

  
上图所示的结构将PCIe总线端口、存储器控制器等一系列与外部设备有关的接口都集成在一起，并统称为RC。RC具有一个或者多个PCIe端口，可以连接各类PCIe设备。PCIe设备包括EP(如网卡、显卡等设备)、Switch和PCIe桥。PCIe总线采用端到端的连接方式，每一个PCIe端口只能连接一个EP，当然PCIe端口也可以连接Switch进行链路扩展。通过Switch扩展出的PCIe链路可以继续挂接EP或者其他Switch。

### 4.2.2 RC的组成结构

RC是PCIe体系结构的一个重要组成部件，也是一个较为混乱的概念。RC的提出与x86处理器系统密切相关。事实上，只有x86处理器才存在PCIe总线规范定义的“标准RC”，而在多数处理器系统，并不含有在PCIe总线规范中涉及的，与RC相关的全部概念。

不同处理器系统的RC设计并不相同，在图4‑7中的处理器系统中，RC包括存储器控制器、两个FSB-to-PCIe桥。而在图4‑8中的PowerPC处理器系统中，RC的概念并不明晰。在 PowerPC处理器中并不存在真正意义上的RC，而仅包含PCIe总线控制器。

在x86处理器系统中，RC内部集成了一些PCI设备、RCRB(RC Register Block)和Event Collector等组成部件。其中RCRB由一系列“管理存储器系统”的寄存器组成，而仅存在于x86处理器中；而Event Collector用来处理来自PCIe设备的错误消息报文和PME消息报文。RCRB寄存器组属于PCI总线域地址空间，x86处理器访问RCRB的方法与访问PCI设备的配置寄存器相同。在有些x86处理器系统中，RCRB在PCI总线0的设备0中。

RCRB是x86处理器所独有的，PowerPC处理器也含有一组“管理存储器系统”的寄存器，这组寄存器与RCRB所实现的功能类似。但是在PowerPC处理器中，该组寄存器以CCSRBAR寄存器为基地址，处理器采用存储器映像方式访问这组寄存器。

如果将RC中的RCRB、内置的PCI设备和Event Collector去除，该RC的主要功能与PCI总线中的HOST主桥类似，其主要作用是完成存储器域到PCI总线域的地址转换。但是随着虚拟化技术的引入，尤其是引入MR-IOV技术之后，RC的实现变得异常复杂。

但是RC与HOST主桥并不相同，RC除了完成地址空间的转换之外，还需要完成物理信号的转换。在PowerPC处理器的RC中，来自OCeaN或者FSB的信号协议与PCIe总线信号使用的电气特性并不兼容，使用的总线事务也并不相同，因此必须进行信号协议和总线事务的转换。

在P4080处理器中，RC的下游端口可以挂接Switch扩展更多的PCIe端口，也可以只挂接一个EP。在P4080处理器的RC中，设置了一组Inbound和Outbound寄存器组，用于存储器域与PCI总线域之间地址空间的转换；而P4080处理器的RC还可以使用Outbound寄存器组将PCI设备的配置空间直接映射到存储器域中。PowerPC处理器在处理PCI/PCIe接口时，都使用这组Inbound和Outbound寄存器组。

在P4080处理器中，RC可以使用PEX\_CONFIG\_ADDR与PEX\_CONFIG\_DATA寄存器对 EP进行配置读写，这两个寄存器与MPC8548处理器HOST主桥的PCI\_CONFIG\_ADDR和PCI\_CONFIG\_DATA寄存器类似，本章不再详细介绍这组寄存器。

而x86处理器的RC设计与PowerPC处理器有较大的不同，实际上和大多数处理器系统都不相同。x86处理器赋予了RC新的含义，PCIe总线规范中涉及的RC也以x86处理器为例进行说明，而且一些在PCIe总线规范中出现的最新功能也在Intel的x86处理器系统中率先实现。在x86处理器系统中的RC实现也比其他处理器系统复杂得多。深入理解x86处理器系统的RC对于理解PCIe体系结构非常重要。

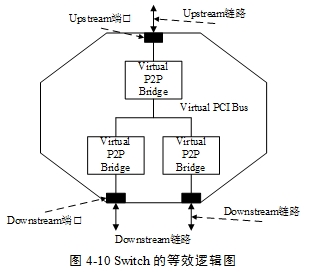
### 4.2.3 Switch

本篇在第4.1.4节，简单介绍了在PCIe总线中，如何使用Switch进行链路扩展，本节主要介绍Switch[[5]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM" \l "_ftn5" \o ")的内部结构。从系统软件的角度上看，每一个PCIe链路都占用一个PCI总线号，但是一条PCIe链路只能连接一个PCI设备，Switch、EP或者PCIe桥片。PCIe总线使用端到端的连接方式，一条PCIe链路只能连接一个设备。

一个PCIe链路需要挂接多个EP时，需要使用Switch进行链路扩展。一个标准Switch具有一个上游端口和多个下游端口。上游端口与RC或者其他Switch的下游端口相连，而下游端口可以与EP、PCIe-to-PCI-X/PCI桥或者下游Switch的上游端口相连。

PCIe总线规范还支持一种特殊的连接方式，即Crosslink连接方式。使用这种方式时，Switch的下游端口可以与其他Switch的下游端口直接连接，上游端口也可以其他Switch的上游端口直接连接。在PCIe总线规范中，Crosslink连接方式是可选的，并不要求PCIe设备一定支持这种连接方式。

在PCIe体系结构中，Switch的设计难度仅次于RC，Switch也是PCIe体系结构的核心所在。而从系统软件的角度上看，Switch内部由多个PCI-to-PCI桥组成，其中每一个上游和下游端口都对应一个虚拟PCI桥。在一个Switch中有多个端口，在其内部就有多少个虚拟PCI桥，就有多少个PCI桥配置空间。值得注意的是，在Switch内部还具有一条虚拟的PCI总线，用于连接各个虚拟PCI桥，系统软件在初始化Switch时，需要为这条虚拟PCI总线编号。Switch的组成结构如图4‑10所示。

  
Switch[[6]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftn6)需要处理PCIe总线传输过程中的QoS问题。PCIe总线的QoS要求PCIe总线区别对待优先权不同的数据报文，而且无论PCIe总线的某一个链路多么拥塞，优先级高的报文，如等时报文(Isochronous Packet)都可以获得额定的数据带宽。而且PCIe总线需要保证优先级较高的报文优先到达。PCIe总线采用虚拟多通路VC技术[[7]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM" \l "_ftn7" \o ")，并在这些数据报文设定一个TC(Traffic Class)标签，该标签由3位组成，将数据报文根据优先权分为8类，这8类数据报文可以根据需要选择不同的VC进行传递。

在PCIe总线中，每一条数据链路上最多可以支持8个独立的VC。每个VC可以设置独立的缓冲，用来接收和发送数据报文。在PCIe体系结构中，TC和VC紧密相连，TC与VC之间的关系是“多对一”。

TC可以由软件设置，系统软件可以选择某类TC由哪个VC进行传递。其中一个VC可以传递TC不相同的数据报文，而TC相同的数据报文在指定一个VC传递之后，不能再使用其他VC。在许多处理器系统中，Switch和RC仅支持一个VC，而x86处理器系统和PLX的Switch中可以支持两个VC。

下文将以一个简单的例子说明如何使用TC标签和多个VC，以保证数据传送的服务质量。我们将PCIe总线的端到端数据传递过程模拟为使用汽车将一批货物从A点运送到B点。如果我们不考虑服务质量，可以采用一辆汽车运送所有这些货物，经过多次往返就可以将所有货物从A点运到B点。但是这样做会耽误一些需要在指定时间内到达B点的货物。有些货物，如一些急救物资、EMS等其他优先级别较高的货物，必须要及时地从A点运送到B点。这些急救物资的运送应该有别于其他普通物资的运送。

为此我们首先将不同种类的货物进行分类，将急救物资定义为TC3类货物，EMS定义为TC2类货物，平信定义为TC1类货物，一般包裹定义为TC0类货物，我们最多可以提供8种TC类标签进行货物分类。

之后我们使用8辆汽车，分别是VC0~7运送这些货物，其中VC7的速度最快，而VC0的速度最慢。当发生堵车事件时，VC7优先行驶，VC0最后行驶。然后我们使用VC3运送急救物资，VC2运送EMS，VC1运送平信，VC0运送包裹，当然使用VC0同时运送平信和包裹也是可以的，但是平信或者包裹不能使用一种以上的汽车运送，如平信如果使用了VC1运输，就不能使用VC0。因为TC与VC的对应关系是“多对一”的关系。

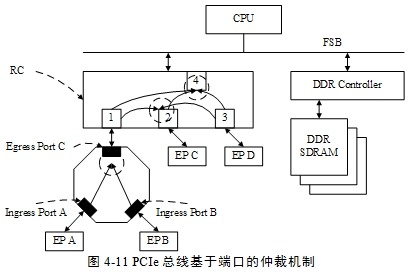
采用这种分类运输的方法，我们可以做到在A点到B点带宽有限的情况下，仍然可以保证急救物资和EMS可以及时到达B点，从而提高了服务质量。

PCIe总线除了解决数据传送的QoS问题之外，还进一步考虑如何在链路传递过程中，使用流量控制机制防止拥塞。

在PCIe体系结构中，Switch处于核心地位。PCIe总线使用Switch进行链路扩展，在Switch中，每一个端口对应一个虚拟PCI桥。深入理解PCI桥是理解Switch软件组成结构的基础。目前PCIe总线提出了MRA-Switch的概念，这种Switch与传统Switch有较大的区别。

### 4.2.4 VC和端口仲裁

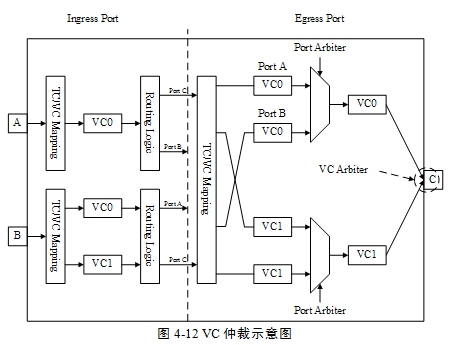
在Switch中存在多个端口，其中来自不同Ingress端口的报文可以发向同一个Egress端口，因此Switch必须要解决端口仲裁和路由选径的问题。所谓端口仲裁指来自不同Ingress端口的报文到达同一个Egress端口的报文通过顺序，端口仲裁机制如图4‑11所示。

  
在一个Switch中设有仲裁器，该仲裁器规定了数据报文通过Switch的规则。在PCIe总线中存在两种仲裁机制，分别是基于VC和基于端口的仲裁机制。端口仲裁机制主要针对RC和Switch，当多个Ingress端口需要向同一个Egress端口发送数据报文时需要进行端口仲裁。具体地讲，在PCIe体系结构中有三个端口，需要进行端口仲裁。

* Switch的Egress端口。当EP A和EP B同时访问EP C，D或者DDR-SDRAM时，需要通过Switch的Egress端口C。此时Switch需要进行端口仲裁确定是EP A的数据报文还是EP B的数据报文优先通过Egress端口C。
* 多端口RC的Egress端口。当RC的端口1和端口3同时访问Endpoint C时，RC的端口2需要进行端口仲裁，决定来自RC哪个端口的数据可以率先通过。
* RC通往主存储器的端口。当RC的端口1、端口2和端口3同时访问DDR控制器时，这些数据报文将通过RC的Egress端口4，此时需要进行端口仲裁。

在PCIe体系结构中，链路的端口仲裁需要根据每一个VC独立设置，而且可以使用不同的算法进行端口仲裁。

下文以图4‑11中，Switch的两个Ingress端口A和B向Egress端口C发送数据报文为例，简要说明端口仲裁和VC仲裁的使用方法，其过程如图4‑12所示。



基于VC的仲裁是指发向同一个端口的数据报文，根据使用的VC而进行仲裁的方式。如上图所示，当来自端口B和端口A数据报文(分别使用VC0和VC1通路)在到达端口C之前，需要首先进行端口仲裁后，才能进行VC仲裁。PCIe总线规定了3种VC仲裁方式, 分别为Strict Priority，RR(Round Robin)和WRR(Weighted Round Robin)算法。

当使用Strict Priority仲裁方式时，发向VC7的数据报文具有最高的优先级，而发向VC0的数据报文优先级最低。PCIe总线允许对Switch或者RC的部分VC采用Strict Priority方式进行仲裁，而对其他VC采用RR和WRR算法，如VC7~4采用Strict Priority方式，而采用其他方式处理VC3~0。

使用RR方式时，所有VC具有相同的优先级，所有VC轮流使用PCIe链路。WRR方式与RR算法类似，但是可以对每一个VC进行加权处理，采用这种方式可以适当提高VC7的优先权，而将VC0的优先权适当降低。

我们假定Ingress端口A和Ingress B向Egress端口C进行数据传递时，使用两个VC通路，分别是VC0和VC1。其中标签为TC0～3的数据报文使用VC0传送，而标签为TC4～7数据报文使用VC1传送。

而数据报文在离开Egress端口C时，需要首先进行端口仲裁，之后再通过VC仲裁，决定哪个报文优先传送。数据报文从Ingress A/B端口发送到Egress C端口时，将按照以下步骤进行处理。

(1) 首先到达Ingress A/B端口的数据报文，将根据该端口的TC/VC映射表[[8]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM" \l "_ftn8" \o ")决定使用该端口的哪个VC通道。如图4‑12所示，假设发向端口A的数据报文使用TC0～TC3，而发向端口B的数据报文使用TC0～TC7，这些数据报文在端口A中仅使用了VC0通道，而在端口B中使用了VC0和VC1两个通道。

(2) 数据报文在端口中传递时，将通过路由部件(Routing Logic)，将报文发送到合适的端口。如图4‑12所示，端口C可以接收来自端口A或者B的数据报文。

(3) 当数据报文到达端口C时，首先需要经过TC/VC映射表，确定在端口C中使用哪个VC通路接收不同类型的数据报文。

(4) 对于端口C，其VC0通道可能会被来自端口A的数据报文使用，也可能会被来自端口B的数据报文使用。因此在PCIe的Switch中必须设置一个端口仲裁器，决定来自不同数据端口的数据报文如何使用VC通路。

(5) 数据报文通过端口仲裁后，获得VC通路的使用权之后，还需要经过Switch中的VC仲裁器，将数据报文发送到实际的物理链路中。

PCIe总线规定，系统设计者可以使用以下三种方式进行端口仲裁。

(1) Hardware-fixed仲裁策略。如在系统设计时，采用固化的RR仲裁方法。这种方法的硬件实现原理较为简单，此时系统软件不能对端口仲裁器进行配置。

(2) WRR仲裁策略，即加权的RR仲裁策略。

(3) Time-Based WRR仲裁策略，基于时间片的WRR仲裁策略，PCIe总线可以将一个时间段分为若干个时间片(Phase)，每个端口占用其中的一个时间片，并根据端口使用这些时间片的多少对端口进行加权的一种方法。使用WRR和Time-Based WRR仲裁策略，可以在某种程度上提高PCIe总线的QoS。

PCIe设备的Capability寄存器规定了端口仲裁使用的算法。有些PCIe设备并没有提供多种端口仲裁算法，可能也并不含有Capability寄存器。此时该PCIe设备使用Hardware-fixed仲裁策略。

### 4.2.5 PCIe-to-PCI/PCI-X桥片

本篇将PCIe-to-PCI/PCI-X桥片简称为PCIe桥片。该桥片有两个作用。

* 将PCIe总线转换为PCI总线，以连接PCI设备。在一个没有提供PCI总线接口的处理器中，需要使用这类桥片连接PCI总线的外设。许多PowerPC处理器在提供PCIe总线的同时，也提供了PCI总线，因此PCIe-to-PCI桥片对基于PowerPC处理器系统并不是必须的。
* 将PCI总线转换为PCIe总线(这也被称为Reverse Bridge)，连接PCIe设备。一些低端的处理器并没有提供PCIe总线，此时需要使用PCIe桥将PCI总线转换为PCIe总线，才能与其他PCIe设备互连。这种用法初看比较奇怪，但是在实际应用中，确实有使用这一功能的可能。本节主要讲解PCIe桥的第一个作用。

PCIe桥的一端与PCIe总线相连，而另一端可以与一条或者多条PCI总线连接。其中可以连接多个PCI总线的PCIe桥也被称为多端口PCIe桥。

PCIe总线规范提供了两种多端口PCIe桥片的扩展方法。多端口PCIe桥片指具有一个上游端口和多个下游端口的桥片。其中上游端口连接PCIe链路，而下游端口推出PCI总线，连接PCI设备。

目前虽然PCIe总线非常普及，但是仍然有许多基于PCI总线的设计，这些基于PCI总线的设计可以通过PCIe桥，方便地接入到PCIe体系结构中。目前有多家半导体厂商可以提供PCIe桥片，如PLX、NXP、Tundra和Intel。就功能的完善和性能而言，Intel的PCIe桥无疑是最佳选择，而PLX和Tundra的PCIe桥在嵌入式系统中得到了广泛的应用。

[[1]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref1) OCeaN是一个基于交叉矩阵的总线结构，连接在OCeaN中的外部设备可以直接通信，而不相互干扰。

[[2]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref2) 这种方式也可以被认为是SoC平台总线从共享总线结构升级到Switch结构。

[[3]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref3) SRIO为串型RapidIO。

[[4]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref4) PCIe端口之间的直接通信过程也被称为Peer-to-Peer传送方式。

[[5]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref5) PCIe总线中的Switch与网络应用的Switch的功能并不相同，而与网络应用中的Route功能接近。

[[6]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref6) 在PCIe体系结构中，RC和EP也需要处理QoS。

[[7]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref7) 有关多通路VC的详细说明见第9章。

[[8]](http://bbs.ednchina.com/BLOG_ARTICLE_3012736.HTM#_ftnref8) 该表存在于PCI Express. Extended Capabilities结构中。

|  |
| --- |
| **4.3 PCIe设备的扩展配置空间** |
| 发布时间：2013-06-20 09:24:11 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

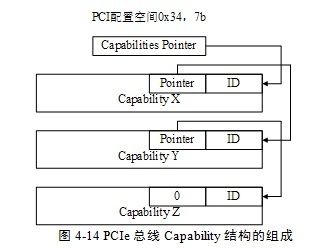
本书在第2.3.2节讲述了PCI设备使用的基本配置空间。这个基本配置空间共由64个字节组成，其地址范围为0x00~0x3F，这64个字节是所有PCI设备必须支持的。事实上，许多PCI设备也仅支持这64个配置寄存器。

此外PCI/PCI-X和PCIe设备还扩展了0x40~0xFF这段配置空间，在这段空间主要存放一些与MSI或者MSI-X中断机制和电源管理相关的Capability结构。其中所有能够提交中断请求的PCIe设备，必须支持MSI或者MSI-X Capability结构。

PCIe设备还支持0x100~0xFFF这段扩展配置空间。PCIe设备使用的扩展配置空间最大为4KB，在PCIe总线的扩展配置空间中，存放PCIe设备所独有的一些Capability结构，而PCI设备不能使用这段空间。

在x86处理器中，使用CONFIG\_ADDRESS寄存器与CONFIG\_DATA寄存器访问PCIe配置空间的0x00~0xFF，而使用ECAM方式访问0x000~0xFFF这段空间；而在PowerPC处理器中，可以使用CFG\_ADDR和CFG\_DATA寄存器访问0x000~0xFFF，详见第2.2节。

PCI-X和PCIe总线规范要求其设备必须支持Capabilities结构。在PCI总线的基本配置空间中，包含一个Capabilities Pointer寄存器，该寄存器存放Capabilities结构链表的头指针。在一个PCIe设备中，可能含有多个Capability结构，这些寄存器组成一个链表，其结构如图4‑14所示。

[](http://download.bbs.ednchina.com/images/attachments/201306/original/8957263754_TIME_1371691354082.jpg)

其中每一个Capability结构都有唯一的ID号，每一个Capability寄存器都有一个指针，这个指针指向下一个Capability结构，从而组成一个单向链表结构，这个链表的最后一个Capability结构的指针为0。

一个PCIe设备可以包含多个Capability结构，包括与电源管理相关、与PCIe总线相关的结构、与中断请求相关的Capability结构、PCIe Capability结构和PCIe扩展的Capability结构。在本书的其他章节也将讲述这些Capability结构，读者在继续其他章节的学习之前，需要简单了解这些Capability结构的寄存器组成和使用方法。

其中读者需要重点关注的是MSI/MSI-X Capability结构本篇在第6章详细将讨论MSI/MSI-X Capability结构。在PCIe总线规范中，定义了较多的Capability结构，这些结构适用于不同的应用场合，在一个指定的PCIe设备中，并不一定支持本篇中涉及的所有Capability结构。系统软件程序员也不需要完全掌握PCIe总线规范定义的这些Capability结构。

## 4.4 小结

   本章简要介绍了PCIe总线的各个组成部件，包括RC、Switch和EP等，并介绍了PCIe总线的层次组成结构，和PCIe设备使用的Capability结构。本章是读者了解PCIe体系结构的基础。本篇将在下文逐步对PCIe体系结构进行详细说明。

|  |
| --- |
| **第5章 PCIe总线的事务层** |
| 发布时间：2013-06-20 09:25:54 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

事务层是PCIe总线层次结构的最高层，该层次将接收PCIe设备核心层的数据请求，并将其转换为PCIe总线事务，PCIe总线使用的这些总线事务在TLP头中定义。PCIe总线继承了PCI/PCI-X总线的大多数总线事务，如存储器读写、I/O读写、配置读写总线事务，并增加了Message总线事务和原子操作等总线事务。

本节重点介绍与数据传送密切相关的总线事务，如存储器、I/O、配置读写总线事务。在PCIe总线中，Non-Posted总线事务分两部分进行，首先是发送端向接收端提交总线读写请求，之后接收端再向发送端发送完成(Completion)报文。PCIe总线使用Split传送方式处理所有Non-Posted总线事务，存储器读、I/O读写和配置读写这些Non-Posted总线事务都使用Split传送方式。PCIe的事务层还支持流量控制和虚通路管理等一系列特性，而PCI总线并不支持这些新的特性。

在PCIe总线中，不同的总线事务采用的路由方式不相同。PCIe总线继承了PCI总线的地址路由和ID路由方式，并添加了“隐式路由”方式。

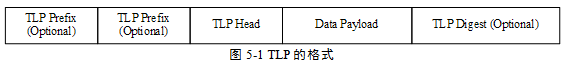
PCIe总线使用的数据报文首先在事务层中形成，这个数据报文也被称之为事务层数据报文，即TLP，TLP在经过数据链路层时被加上Sequence Number前缀和CRC后缀，然后发向物理层。

数据链路层还可以产生DLLP(Data Link Layer Packet)。DLLP和TLP没有直接关系，DLLP是产生于数据链路层，终止于数据链路层，并不会传递到事务层。DLLP不是TLP加上前缀和后缀形成的。数据链路层的报文DLLP通过物理层时，需要经过8/10b编码，然后再进行发送。数据的接收过程是发送过程的逆过程，但是在具体实现上，接收过程与发送过程并不完全相同。

|  |
| --- |
| **5.1 TLP的格式** |
| 发布时间：2013-06-25 16:10:26 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

当处理器或者其他PCIe设备访问PCIe设备时，所传送的数据报文首先通过事务层被封装为一个或者多个TLP，之后才能通过PCIe总线的各个层次发送出去。TLP的基本格式如图5‑1所示。

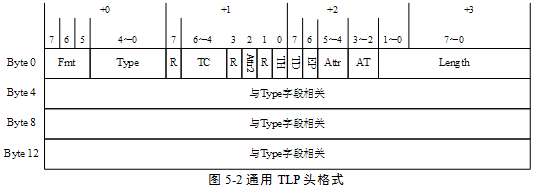
[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscl&url=http://s3.sinaimg.cn/orignal/6472c4ccgac8158de66e2)

一个完整的TLP由1个或者多个TLP Prefix、TLP头、Data Payload(数据有效负载)和TLP Digest组成。TLP头是TLP最重要的标志，不同的TLP其头的定义并不相同。TLP头包含了当前TLP的总线事务类型、路由信息等一系列信息。在一个TLP中，Data Payload的长度可变，最小为0，最大为1024DW。

TLP Digest是一个可选项, 一个TLP是否需要TLP Digest由TLP头决定。Data Payload也是一个可选项，有些TLP并不需要Data Payload，如存储器读请求、配置和I/O写完成TLP并不需要Data Payload。

TLP Prefix由PCIe V2.1总线规范引入，分为Local TLP Prefix和EP-EP TLP Prefix两类。其中Local TLP Prefix的主要作用是在PCIe链路的两端传递消息，而EP-EP TLP Prefix的主要作用是在发送设备和接收设备之间传递消息。设置TLP Prefix的主要目的是为了扩展TLP头，并以此支持PCIe V2.1规范的一些新的功能。

TLP头由3个或者4个双字(DW)组成。其中第一个双字中保存通用TLP头，其他字段与通用TLP头的Type字段相关。一个通用TLP头由Fmt、Type、TC、Length等字段组成，如图5‑2所示。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscl&url=http://s6.sinaimg.cn/orignal/6472c4ccgac815e9bdd85)

如果存储器读写TLP支持64位地址模式时，TLP头的长度为4DW，否则为3DW。而完成报文的TLP头不含有地址信息，使用的TLP头长度为3DW。其中Byte 4~Byte 15的格式与TLP相关，下文将结合具体的TLP介绍这些字段。

### 5.1.1 通用TLP头的Fmt字段和Type字段

Fmt和Type字段确认当前TLP使用的总线事务，TLP头的大小是由3个双字还是4个双字组成，当前TLP是否包含有效负载。其具体含义如表5‑1所示。

 表5‑1 Fmt[1:0]字段

| **Fmt[2:0]** | **TLP的格式** |
| --- | --- |
| 0b000 | TLP大小为3个双字，不带数据。 |
| 0b001 | TLP大小为4个双字，不带数据。 |
| 0b010 | TLP大小为3个双字，带数据。 |
| 0b011 | TLP大小为4个双字，带数据。 |
| 0b100 | TLP Prefix |
| 其他 | PCIe总线保留 |

其中所有读请求TLP都不带数据，而写请求TLP带数据，而其他TLP可能带数据也可能不带数据，如完成报文可能含有数据，也可能仅含有完成标志而并不携带数据。在TLP的Type字段中存放TLP的类型，即PCIe总线支持的总线事务。该字段共由5位组成，其含义如表5‑2所示。

表5‑2 Type[4:0]字段

| **TLP类型** | **Fmt[2:0]** | **Type[4:0]** | **描述** |
| --- | --- | --- | --- |
| MRd | 0b000  0b001 | 0b0 0000 | 存储器读请求；TLP头大小为3个或者4个双字，不带数据。 |
| MRdLk | 0b000  0b001 | 0b0 0001 | 带锁的存储器读请求；TLP头大小为3个或者4个双字，不带数据。 |
| MWr | 0b010  0b011 | 0b0 0000 | 存储器写请求；TLP头大小为3个或者4个双字，带数据。 |
| IORd | 0b000 | 0b0 0010 | IO读请求；TLP头大小为3个双字，不带数据。 |
| IOWr | 0b010 | 0b0 0010 | IO写请求；TLP头大小为3个双字，带数据。 |
| CfgRd0 | 0b000 | 0b0 0100 | 配置0读请求；TLP头大小为3个双字，不带数据。 |
| CfgWr0 | 0b010 | 0b0 0100 | 配置0写请求；TLP头大小为3个双字，带数据。 |
| CfgRd1 | 0b000 | 0b0 0101 | 配置1读请求；不带数据。 |
| CfgWr1 | 0b010 | 0b0 0101 | 配置1写请求；带数据。 |
| TCfgRd | 0b010 | 0b1 1011 | 本书对这两种总线事务不做介绍。 |
| TCfgWr | 0b001 | 0b1 1011 |
| Msg | 0b001 | 0b1 0r2r1r0 | 消息请求；TLP头大小为4个双字，不带数据。“rrr”字段是消息请求报文的Route字段，下文将详细介绍该字段。 |
| MsgD | 0b011 | 0b1 0r2r1r0 | 消息请求；TLP头大小为4个双字，带数据。 |
| Cpl | 0b000 | 0b0 1010 | 完成报文；TLP头大小为3个双字，不带数据。包括存储器、配置和I/O写完成。 |
| CplD | 0b010 | 0b0 1010 | 带数据的完成报文，TLP头大小为3个双字，包括存储器读、I/O读、配置读和原子操作读完成。 |
| CplLk | 0b000 | 0b0 1011 | 锁定的完成报文，TLP头大小为3个双字，不带数据。 |
| CplDLk | 0b010 | 0b0 1011 | 带数据的锁定完成报文，TLP头大小为3个双字，带数据。 |
| FetchAdd | 0b010  0b011 | 0b0 1100 | Fetch and Add原子操作。 |
| Swap | 0b010  0b011 | 0b0 1101 | Swap原子操作。 |
| CAS | 0b010  0b011 | 0b0 1110 | CAS原子操作。 |
| LPrfx | 0b100 | 0b0 L3L2L1L0 | Local TLP Prefix |
| EPrfx | 0b100 | 0b1 E3E2E1E0 | End-End TLP Prefix |

由上表所示，存储器读和写请求，IO读和写请求，及配置读和写请求的type字段相同，如存储器读和写请求的Type字段都为0b0 0000。此时PCIe总线规范使用Fmt字段区分读写请求，当Fmt字段是“带数据”的报文，一定是“写报文”；当Fmt字段是“不带数据”的报文，一定是“读报文”。

PCIe总线的数据报文传送方式与PCI总线数据传送有类似之处。其中存储器写TLP使用Posted方式进行传送，而其他总线事务使用Non-Posted方式。

PCIe总线规定所有Non-Posted存储器请求使用Split总线方式进行数据传递。当PCIe设备进行存储器读、I/O读写或者配置读写请求时，首先向目标设备发送数据读写请求TLP，当目标设备收到这些读写请求TLP后，将数据和完成信息通过完成报文(Cpl或者CplD)发送给源设备。

其中存储器读、I/O读和配置读需要使用CplD报文，因为目标设备需要将数据传递给源设备；而I/O写和配置写需要使用Cpl报文，因为目标设备不需要将任何数据传递给源设备，但是需要通知源设备，写操作已经完成，数据已经成功地传递给目标设备。

在PCIe总线中，进行存储器或者I/O写操作时，数据与数据包头一起传递；而进行存储器或者I/O读操作时，源设备首先向目标设备发送读请求TLP，而目标设备在准备好数据后，向源设备发出完成报文。

PCIe总线规范还定义了MRdLk报文，该报文的主要作用是与PCI总线的锁操作相兼容，但是PCIe总线规范并不建议用户使用这种功能，因为使用这种功能将极大影响PCIe总线的数据传送效率。

与PCI总线并不相同，PCIe总线规范定义了Msg报文，即消息报文。分别为Msg和MsgD，这两种报文的区别在于一个报文可以传递数据，一个不能传递数据。

PCIe V2.1总线规范还补充了一些总线事务，如FetchAdd、Swap、CAS、LPrfx和EPrfx。其中LPrfx和EPrfx总线事务分别与Local TLP Prefix和EP-EP TLP Prefix对应。在PCIe总线规范V2.0中，TLP头的大小为1DW，而使用LPrfx和EPrfx总线事务可以对TLP头进行扩展，本节不对这些TLP Prefix做进一步介绍。PCIe设备可以使用FetchAdd、Swap和CAS总线事务进行原子操作，本篇将在第5.3.5节详细介绍该类总线事务。

### 5.1.2 TC字段

TC字段表示当前TLP的传送类型，PCIe总线规定了8种传输类型，分别为TC0～TC7，缺省值为TC0，该字段与PCIe的QoS相关。PCIe设备使用TC区分不同类型的数据传递，而多数EP中只含有一个VC，因此这些EP在发送TLP时，也仅仅使用TC0，但是有些对实时性要求较高的EP中，含有可以设置TC字段的寄存器。

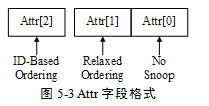
在Intel的高精度声卡控制器(High Definition Audio Controller)的扩展配置空间中含有一个TCSEL寄存器。系统软件可以设置该寄存器，使声卡控制器发出的TLP使用合适的TC。声卡控制器可以使用TC7传送一些对实时性要求较强的控制信息，而使用TC0传送一般的数据信息。在具体实现中，一个EP也可以将控制TC字段的寄存器放入到设备的BAR空间中，而不必和Intel的高精度声卡控制器相同，存放在PCI配置空间中。

目前许多处理器系统的RC仅支持一个VC通路，此时EP使用不同的TC进行传递数据的意义不大。x86处理器的MCH中一般支持两个VC通路，而多数PowerPC处理器仅支持一个VC通路。PLX公司的多数Switch也仅支持两个VC通路。

有些RC，如MPC8572处理器，也能决定其发出TLP使用的TC。在该处理器的PCIe Outbound窗口寄存器(PEXOWARn)中，含有一个TC字段，通过设置该字段可以确定RC发出的TLP使用的TC字段。不同的TC可以使用PCIe链路中的不同VC，而不同的VC的仲裁级别并不相同。EP或者RC通过调整其发出TLP的TC字段，可以调整TLP使用的VC，从而调整TLP的优先级。

### 5.1.3 Attr字段

Attr字段由3位组成，其中第2位表示该TLP是否支持PCIe总线的ID-based Ordering；第1位表示是否支持Relaxed Ordering；而第0位表示该TLP在经过RC到达存储器时，是否需要进行Cache共享一致性处理。Attr字段如图5‑3所示。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscl&url=http://s16.sinaimg.cn/orignal/6472c4ccgac8166b7d08f)

一个TLP可以同时支持ID-based Ordering和Relaxed Ordering两种位序。Relaxed Ordering最早在PCI-X总线规范中提出，用来提高PCI-X总线的数据传送效率；而ID-based Ordering由PCIe V2.1总线规范提出。TLP支持的序如表5‑3所示。

 表5‑3 TLP支持的序

| **Attr[2]** | **Attr[1]** | **类型** |
| --- | --- | --- |
| 0 | 0 | 缺省序，即强序模型 |
| 0 | 1 | PCI-X Relaxed Ordering模型 |
| 1 | 0 | ID-Based Ordering(IDO)模型 |
| 1 | 1 | 同时支持Relaxed Ordering和IDO模型 |

当使用标准的强序模型时，在数据的整个传送路径中，PCIe设备在处理相同类型的TLP时，如PCIe设备发送两个存储器写TLP时，后面的存储器写TLP必须等待前一个存储器写TLP完成后才能被处理，即便当前报文在传送过程中被阻塞，后一个报文也必须等待。

如果使用Relaxed Ordering模型，后一个存储器写TLP可以穿越前一个存储器写TLP，提前执行，从而提高了PCIe总线的利用率。有时一个PCIe设备发出的TLP，其目的地址并不相同，可能先进入发送队列的TLP，在某种情况下无法发送，但这并不影响后续TLP的发送，因为这两个TLP的目的地址并不相同，发送条件也并不相同。

值得注意的是，在使用PCI总线强序模型时，不同种类的TLP间也可以乱序通过同一条PCIe链路，比如存储器写TLP可以超越存储器读请求TLP提前进行。而PCIe总线支持Relaxed Ordering模型之后，在TLP的传递过程中出现乱序种类更多，但是这些乱序仍然是有条件限制的。在PCIe总线规范中为了避免死锁，还规定了不同报文的传送数据规则，即Ordering Rules。

PCIe V2.1总线规范引入了一种新的“序”模型，即IDO(ID-Based Ordering)模型，IDO模型与数据传送的数据流相关，是PCIe V2.1规范引入的序模型。

Attr字段的第0位是“No Snoop Attribute”位。当该位为0时表示当前TLP所传送的数据在通过FSB时，需要与Cache保持一致，这种一致性由FSB通过总线监听自动完成而不需要软件干预；如果为1，表示FSB并不会将TLP中的数据与Cache进行一致，在这种情况下，进行数据传送时，必须使用软件保证Cache的一致性。

在PCI总线中没有与这个“No Snoop Attribute”位对应的概念，因此一个PCI设备对存储器进行DMA操作时会进行Cache一致性操作(1) ,这种“自动的”Cache一致性行为在某些特殊情况下并不能带来更高的效率。

当一个PCIe设备对存储器进行DMA读操作时，如果传送的数据非常大，比如512MB，Cache的一致性操作不但不会提高DMA写的效率，反而会降低。因为这个DMA读访问的数据在绝大多数情况下，并不会在Cache中命中，但是FSB依然需要使用Snoop Phase进行总线监听。而处理器在进行Cache一致性操作时仍然需要占用一定的时钟周期，即在Snoop Phase中占用的时钟周期，Snoop Phase是FSB总线事务的一个阶段，如图3‑6所示。

对于这类情况，一个较好的做法是，首先使用软件指令保证Cache与主存储器的一致性，并置“No Snoop Attribute”位为1(2),然后再进行DMA读操作。同理使用这种方法对一段较大的数据区域进行DMA写时，也可以提高效率。

除此之外，当PCIe设备访问的存储器，不是“可Cache空间”时，也可以通过设置“No Snoop Attribute”位，避免FSB的Cache共享一致性操作，从而提高FSB的效率。“No Snoop Attribute”位是PCIe总线针对PCI总线的不足，所作出的重要改动。

### 5.1.4 通用TLP头中的其他字段

除了Fmt和Type字段外，通用TLP头还含有以下字段。

#### 1 TH位、TD位和EP位

TH位为1表示当前TLP中含有TPH(TLP Processing Hint)信息，TPH是PCIe V2.1总线规范引入的一个重要功能。TLP的发送端可以使用TPH信息，通知接收端即将访问数据的特性，以便接收端合理地预读和管理数据，TPH的详细介绍见第5.3.6节。

TD位表示TLP中的TLP Digest是否有效，为1表示有效，为0表示无效。而EP位表示当前TLP中的数据是否有效，为1表示无效，为0表示有效。

#### 2 AT字段

AT字段与PCIe总线的地址转换相关。在一些PCIe设备中设置了ATC(Address Translation Cache)部件，这个部件的主要功能是进行地址转换。只有在支持IOMMU技术的处理器系统中，PCIe设备才能使用该字段。

AT字段可以用作存储器域与PCI总线域之间的地址转换，但是设置这个字段的主要目的是为了方便多个虚拟主机共享同一个PCIe设备。对这个字段有兴趣的读者可以参考Address Translation Sevices规范，这个规范是PCI的IO Virtualization规范的重要组成部分。对虚拟化技术有兴趣的读者可以参考清华大学出版社的《系统虚拟化——原理与实现》，以获得基本的关于虚拟化的入门知识。

#### 3 Length字段

Length字段用来描述TLP的有效负载(Data Payload)大小(3).PCIe总线规范规定一个TLP的Data Payload的大小在1B～4096B之间。PCIe总线设置Length字段的目的是提高总线的传送效率。

当PCI设备在进行数据传送时，其目标设备并不知道实际的数据传送大小，这在一定程度上影响了PCI总线的数据传送效率。而在PCIe总线中，目标设备可以通过Length字段提前获知源设备需要发送或者请求的数据长度，从而合理地管理接收缓冲，并根据实际情况进行Cache一致性操作。

当PCI设备进行DMA写操作，将PCI设备中4KB大小的数据传送到主存储器时，这个PCI设备的DMA控制器将存放传送的目的地址和传送大小，然后启动DMA写操作，将数据写入到主存储器。由于PCI总线是一条共享总线，因此传送4KB大小的数据，可能会使用若干个PCI总线写事务才能完成(4),而每一个PCI总线写事务都不知道DMA控制器何时才能将数据传送完毕。

如果这些总线写事务还通过一系列PCI桥才能到达存储器，在这个路径上的每一个PCI桥也无法预知，何时这个DMA操作才能结束。这种“不可预知”将导致PCI总线的带宽不能被充分利用，而且极易造成PCI桥数据缓冲的浪费。

而PCIe总线通过TLP的Length字段，可以有效避免PCIe链路带宽的浪费。值得注意的是，Length字段以DW为单位，其最小单位为1个DW。如果PCIe主设备传送的单位小于1个DW或者传送的数据并不以DW对界时，需要使用字节使能字段，即“DW BE”字段。有关“DW BE”字段的详细说明见第5.3.1节。

(1)PowerPC处理器通过设置Inbound寄存器，也可以避免这个Cache一致性操作。

(2)FSB收到这类TLP后，不进行Cache一致性操作。

(3)存储器读请求TLP没有DataPayload字段，此时该TLP使用Length字段表示需要读取多少数据。

(4)当多个PCI设备共享一条PCI总线时，一个设备不会长时间占用PCI总线，这个设备在使用这条PCI总线一定的时间后，将让出PCI总线的使用权。

|  |
| --- |
| **5.2 TLP的路由** |
| 发布时间：2013-06-25 16:56:25 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

TLP的路由是指TLP通过Switch或者PCIe桥片时采用哪条路径，最终到达EP或者RC的方法。PCIe总线一共定义了三种路由方法，分别是基于地址(Address)的路由，基于ID的路由和隐式路由(Implicit)方式。

存储器和I/O读写请求TLP使用基于地址的路由方式，这种方式使用TLP中的Address字段进行路由选径，最终到达目的地。

而配置读写报文、“Vendor\_Defined Messages”报文、Cpl和CplD报文使用基于ID的路由方式，这种方式使用PCI总线号[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dscm.html#_ftn1)(Bus Number)进行路由选径。在Switch或者多端口RC的虚拟PCI-PCI桥配置空间中，包含如何使用PCI总线号进行路由选径的信息。

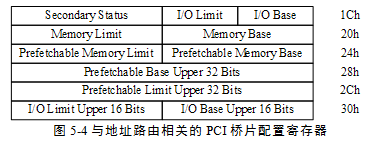
而隐式路由方式主要用于Message报文的传递。在PCIe总线中定义了一系列消息报文，包括“INTx Interrupt Signaling”，“Power Management Messages”和“Error Signal Messages”等报文。在这些报文中，除了“Vendor\_Defined Messages”报文，其他所有消息报文都使用隐式路由方式，隐式路由方式是指从下游端口到上游端口进行数据传递的使用路由方式，或者用于RC向EP发出广播报文。

### 5.2.1 基于地址的路由

在PCIe总线中，存储器读写和I/O读写TLP使用基于地址的路由方式。PCIe设备使用的地址路由方式与PCI设备使用的地址路由方式类似。只是PCIe设备使用TLP进行数据传送，而PCI设备使用总线周期进行数据传送。使用地址路由方式进行数据传递的TLP格式如第5.3.1节的图5‑8所示，在这类TLP中包含目的设备的地址。

当一个TLP进行数据传递时，可能会经过多级Switch，最终到达目的地。Switch将根据存储器读写和I/O读写请求TLP的目的地址将报文传递到合适的Egress端口上。如图4‑10所示，在一个Switch中包含了多个虚拟PCI-to-PCI桥。在Switch中有几个端口，就包含几个虚拟PCI-to-PCI桥。

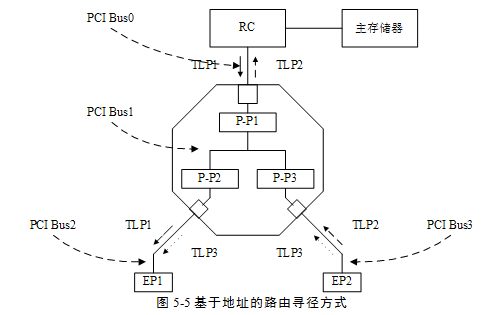
在虚拟PCI-to-PCI桥的配置寄存器空间中，包含一个桥片能够接收的物理地址范围。PCIe总线通过这个物理地址范围实现基于地址的路由。这段配置寄存器如图5‑4所示。当系统软件初始化PCI总线时，将合理地设置这些寄存器，之后当TLP通过这些Switch时将根据这些寄存器选择合适的路径。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscm&url=http://s9.sinaimg.cn/orignal/6472c4ccgac816c0e5258)

上图中的配置寄存器描述了该虚拟PCI-to-PCI桥下游PCI子树使用的三组空间范围，分别为I/O、存储器和可预取的存储器空间，分别用Base和Limit两类寄存器描述，其中Base寄存器表示可访问空间的基地址，Limit寄存器表示可访问空间的大小。TLP使用基于地址的路由时，一定要通过查询这组寄存器之后，再决定传送路径。这组寄存器的使用方法与PCI总线中的PCI桥兼容。

其中TLP从“上游端口发送到下游端口”与“下游端口发送到上游端口”的路由过程略有不同，如图5‑5所示。下文以TLP1~3的发送过程对地址路由过程进行说明。TLP1~3的描述如下。

* TLP1是一个存储器或者I/O请求TLP，由RC发出，并通过一个Switch发向EP1。存储器和I/O读写请求TLP使用这种地址路由方式。TLP1将从Switch的上游端口传送到下游端口。
* TLP2是一个存储器或者I/O请求TLP，由EP2发出，并通过一个Switch发向RC。当PCIe设备进行DMA读写操作时，将使用这种地址路由方式。TLP2将从Switch的下游端口传送到上游端口。
* TLP3是一个存储器或者I/O请求TLP，由一个EP2发出，并通过一个Switch后发送到另外一个EP。在x86处理器系统中，这种用法并不常见。但是在某些大规模处理器系统中，具有这种应用方式。此时TLP3将从Switch的下游端口传送到另外一个下游端口。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscm&url=http://s8.sinaimg.cn/orignal/6472c4ccgac81716aaca7)

#### 1 TLP1的传送过程

当TLP1从RC发向EP1时，这个TLP1为I/O或者存储器报文，其中TLP1目的地址在EP1的BAR空间中。当处理器访问EP的BAR空间时，需要使用该类TLP。值得注意的是这个数据报文在通过RC时需要进行地址转换。

TLP1首先通过PCI Bus0发向Switch，并通过Switch的Upstream端口到达P-P1桥片，P-P1桥片首先根据配置寄存器中的Limit和Base寄存器决定是否接收TLP1。如果Switch不接收TLP1，则将该TLP作为不支持的请求(Unsupported Request)处理，此时如果TLP1需要回应报文，Switch将发出完成报文，该报文的状态为UR(Unsupported Request)。

如果Switch接收TLP1，则表示TLP1所访问的地址在该Switch下游端口所连接的EP或者Switch中，此时Switch将TLP1从PCI Bus0推至PCI Bus1中，即穿越P-P1桥片。TLP1到达PCI Bus1后将同时查找P-P2和P-P3桥片配置寄存器中的Limit和Base寄存器，决定是P-P2还是P-P3桥片接收TLP1。本小节中的例子将使用P-P2桥片接收TLP1，并将TLP1推至PCI Bus2，而PCI Bus2上的EP1将接收TLP1，完成整个地址路由。

#### 2 TLP2的传送过程

当TLP2从EP2发向RC时，一般来说该TLP将访问处理器系统的主存储器。此时TLP2首先将请求发至P-P3桥片，在P-P3桥片配置寄存器的Limit和Base寄存器中当然不会包含TLP2所访问的地址，此时P-P3桥片将TLP2推至PCI Bus1。

TLP从“下游端口向上游端口”与“TLP从上游端口向下游端口”进行传递时，桥片的处理机制有所不同，从上游端口向下游端口传递时，如果桥片配置寄存器的Limit和Base寄存器包含该TLP的访问地址时，桥片将接收此TLP，否则不接收该TLP。而从下游端口向上游端口传递时，如果桥片配置寄存器的Limit和Base寄存器不包含该TLP的访问地址时，桥片将接收该TLP，并将其推至桥片的上游PCI总线。值得注意的是，这两种地址译码方式都属于PCI总线的正向译码。

当TLP2到达PCI Bus1时，首先检查在PCI Bus1总线上的P-P2桥片是否可以接收此TLP，如果不能接收则TLP2通过P-P1桥片传递到PCI Bus0，即到达RC。

在MPC8548处理器中，到达RC的TLP首先通过Inbound寄存器进行地址转换，将TLP的PCI总线地址转换为处理器的地址，然后访问处理器中相应的存储器空间；对于x86处理器而言，MCH也会完成PCI域地址空间到存储器域地址空间的转换，然后访问处理器中相应的存储器空间。

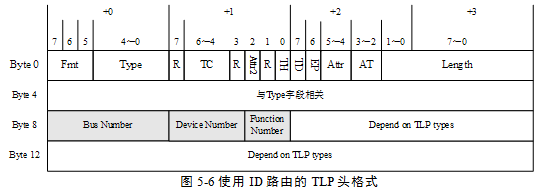
#### 3 TLP3的传送过程

TLP3的传递方式与TLP2的传递方式有些类似，当TLP3传递到PCI Bus1时，P-P2桥片将接收TLP3，并将TLP3传递到PCI Bus2上的EP1中。由以上叙述可以发现，PCIe总线中基于地址的路由方式与PCI总线上的基于地址的数据传递流程十分相近。TLP3在PCI总线域上进行数据传递，因此不需要进行PCI总线域到存储器域的地址转换。

### 5.2.2 基于ID的路由

在PCIe总线中，基于ID的路由方式主要用于配置读写请求TLP、Cpl和CplD报文，此外Vendor\_Defined消息报文也可以使用这种基于ID的路由方式。而在PCI总线中，只有配置读写周期才使用ID进行数据传递。

基于ID的路由方式与基于地址的路由方式有较大的不同，基于ID路由方式的TLP头格式也与基于地址路由方式的头格式不同，其报文格式如图5‑6所示。

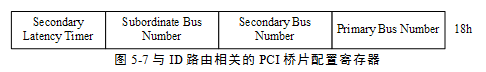
[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscm&url=http://s11.sinaimg.cn/orignal/6472c4ccgac817509a72a)

如上图所示，使用ID路由方式的TLP头，其Byte8～11字段与基于地址路由的TLP不同。基于ID路由的TLP，使用Bus Number、Device Number和Function Number进行路由寻址。从软件的角度上看，PCIe总线与PCI总线兼容，只是在PCIe总线中，每一个PCIe设备使用唯一的PCI设备号，但是每一个设备仍然可以有多个子设备(Function)。

PCIe总线规定，在一个PCI总线域空间中，最多只能有256条PCI总线，因此在一个TLP中，Bus Number由五位组成；而在每一条总线中最多包含32个设备，因此TLP中的Device Number由5位[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dscm.html#_ftn2)组成；而每一个设备中最多包含8个功能，因此一个TLP的Function Number由3位组成。

配置读写请求TLP是使用“基于ID路由”的一组重要报文，其主要作用是读写PCIe总线的EP、Switch及PCIe桥片的配置寄存器，以完成PCIe总线的配置。在处理器系统上电之后需要进行PCI总线系统的枚举，为PCI总线分配总线号，并设置Switch、PCIe桥片或者EP的配置寄存器，如Limit寄存器组、Base寄存器组、BAR寄存器、Subordinate Bus Number、Secondary Bus Number和Primary Bus Number等一系列配置寄存器。

在上文中我们简单介绍了Limit寄存器组和Base寄存器组的用法，下文将重点描述Subordinate Bus Number、Secondary Bus Number和Primary Bus Number寄存器。Subordinate Bus Number、Secondary Bus Number和Primary Bus Number寄存器在Type 01h配置寄存器中，用来描述PCI-to-PCI桥片的上游及下游总线号。这段寄存器在PCI配置寄存器中的位置如图5‑7所示。

[](http://photo.blog.sina.com.cn/showpic.html#blogid=6472c4cc0102dscm&url=http://s6.sinaimg.cn/orignal/6472c4ccgac81783cb5d5)

与PCI总线中的桥片类似，Primary Bus Number记录PCI-to-PCI桥上游的PCI总线号，Secondary Bus Number记录PCI-to-PCI桥下游的第一个PCI总线号，而Subordinate Bus Number记录PCI-to-PCI桥下游的最后一个PCI总线号。

如图5‑5所示，P-P1桥片的Primary Bus Number为0，Secondary Bus Number为1，而Subordinate Bus Number为3。这些总线号，在处理器系统对PCI总线进行枚举时由系统初始化程序设置，从系统初始化程序的角度上看，PCIe总线与PCI总线基本兼容，只是PCIe总线对配置空间进行了一些扩展。

如表5‑2所示，RC可以使用Type 00h和Type 01h读写请求TLP，对PCIe设备的配置寄存器进行读写访问，配置读写请求TLP只能由RC发出，配置读写请求TLP使用基于ID的路由方式。

如图5‑5所示的例子，RC首先使用Type 00h配置请求TLP访问在PCI Bus0总线上的设备，PCI Bus0上的所有设备，包括桥片都要监听PCI Bus 0上的配置请求，在本例中只有Switch挂接在PCI Bus0上，实际上是Switch的上游端口与PCI Bus0直接相连。因此Switch的上游端口将接收RC发出的Type 00h配置请求TLP，之后Switch将向RC发出完成报文，结束配置请求。与PCI总线相同，PCIe总线的Type 00h类型配置请求TLP不能够穿越桥片，在图5‑5中这类请求只能访问Switch上游端口的配置空间。

PCI总线是基于共享总线的数据传送方式，在一条PCI总线上可以连接多个PCI Agent设备，其中每一个PCI Agent都提供了一个IDSEL#信号，这个信号与PCI-to-PCI桥片或者HOST主桥的地址线直接相连，PCI总线根据与IDSEL#信号与地址线的连接关系决定相应设备的Device Number。

这与PCIe总线的使用方法不同，PCIe总线使用“端对端”的连接方式，在PCIe链路只能连接一个下游设备，而这个下游设备的Device Number只能为0。而只有在Switch的虚拟PCI总线上可以连接多个Device Number不同的端口。

当一个虚拟PCI总线上挂接PCI-to-PCI桥时，系统配置软件将使用Type 01h配置请求TLP访问PCI-PCI桥下游的PCI设备。如图5‑5所示，RC可以通过Type 01h配置请求TLP访问P-P2桥片、P-P3桥片，EP1和EP2。

当RC使用Type 01h配置请求TLP，直接访问P-P1桥的下游设备时，首先需要检查该TLP的Bus Number是否为1，如果为1表示该TLP的访问目标在PCI Bus 1总线上，此时PCI-to-PCI桥将这个Type 01h类型的TLP转换为Type 00h类型的TLP，然后推至PCI Bus 1总线，并访问其下的设备。

如果该TLP的Bus Number在P-P1桥片的Secondary Bus Number和Subordinate Bus Number寄存器之间，则P-P1桥片将该Type 01h类型的TLP直接转发到PCI Bus 1上，并不改变该TLP的类型，之后Type 01和类型的TLP将继续检查P-P2和P-P3桥片的配置空间，决定由P-P2还是P-P3接收该TLP。如果TLP的PCI Bus Number为2时，P-P2桥片将接收该TLP，并将该Type 01h类型TLP转换为Type 00h类型的TLP，然后发送给EP1，并由EP1处理该TLP。

上文简要讲述了配置请求TLP使用ID路由方式从上游端口向下游端口的传递规则，但是Vendor\_Defined消息报文和Cpl和CplD报文还可能从下游端口向上游端口进行传递。此时PCIe总线处理方法略有不同。下文仍以图5‑5为例说明这种情况。

当一个TLP从EP2传送到EP1或者RC时，首先检查P-P3桥片的配置空间，P-P3桥片发现该TLP不是发向自己时，将该TLP推至上游总线，即PCI Bus 1。如果PCI Bus1上P-P1桥片没有认领该TLP，该TLP将继续向P-P2桥片传递，并由这个桥片将TLP转发给合适的EP；如果P-P1桥片认领该TLP，该TLP将继续向上游总线传递，直至RC。

由以上描述可以发现，PCIe总线使用的基于ID的路由方式与PCI总线中配置读写总线事务通过PCI桥的方法较为类似。

### 5.2.3 隐式路由

PCIe总线规定消息请求报文使用隐式路由方式。在PCIe总线中，有许多消息是直接发向RC或者来自RC的广播报文，这些报文不使用地址或者ID进行路由，而是使用Msg和MsgD报文的Route字段进行路由，这种路由方式被称为隐式路由。

PCIe总线定义了一些用于中断请求、错误状态处理、锁定总线事务、热插拔信号处理和“Vendor\_Defined Messages”消息报文。这些消息报文需要使用隐式路由方式进行传递。消息报文的Route字段的含义如表5‑4所示。

表5‑4 Route[4:0]字段

|  |  |
| --- | --- |
| Route[2:0] | 描述 |
| 000 | 路由到RC |
| 001 | 使用地址路由 |
| 010 | 使用ID路由 |
| 011 | 来自RC的广播报文 |
| 100 | 本地消息，在接收端结束(Legacy中断消息使用这种报文格式，传递来自PCI总线的中断报文) |
| 101 | 用于PCIe电源管理(PME\_TO\_Ack报文使用) |
| 110~111 | 保留 |

由上表所示，使用隐式路由方式的TLP，其Route字段为“000”，“011”，“100”或者“101”。当一个报文使用隐式路由向EP发送时，EP将对Route字段进行检查，如果这个报文是“来自RC的广播报文”，或者是“本地报文”，EP将接收此报文。

如果Switch收到一条使用隐式路由的TLP时，将根据报文Route字段的不同而分别处理。如果Switch的上游端口接收了一条来自RC的广播消息，则将该报文发向所有的下游端口；如果Switch接收了一条来自下游端口发向RC的消息报文时，Switch将此报文直接转发到上游端口，直至RC；如果Switch接收了一条使用隐式路由方式的本地消息报文，则Switch接收并终结此报文，不再上传或下推。

如果RC收到一个使用隐式路由的TLP时，将根据报文Route字段而分别处理这些TLP。如果该Route字段为0b000和0b101，RC将接收该TLP，并作相应的处理；如果为0b100，RC将接收该TLP，并结束该TLP报文的传递。

[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dscm.html#_ftnref1) PCIe总线实际上使用Transaction ID进行ID路由，有关Transaction ID的详细说明见第6.3.1节。

[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dscm.html#_ftnref2) PCIe链路采用端到端的通信方式，每一个链路只能挂接一个设备，因此在多数情况下，使用3位描述Device Number是多余的，因此PCIe总线提出了ARI格式，该格式的详细描述见第6.3.1节。

|  |
| --- |
| **5.3 存储器、I/O和配置读写请求TLP** |
| 发布时间：2013-07-24 10:35:58 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

本节讲述PCIe总线定义的各类TLP，并详细介绍这些TLP的格式。在这些TLP中，有些格式对于初学者来说较难理解。读者需要建立PCIe总线中与TLP相关的一些基本概念，特别是存储器读写相关的报文格式。在PCIe总线中，存储器读写，I/O读写和配置读写请求TLP由以下几类报文组成。

1. 存储器读请求TLP和读完成TLP

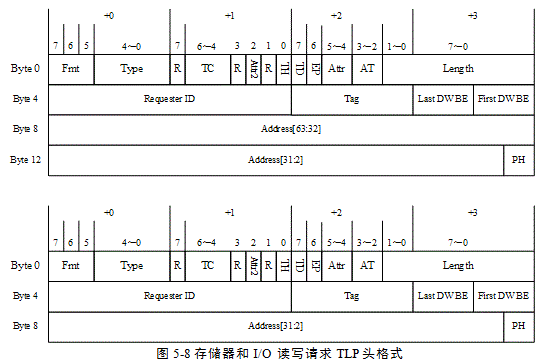
当PCIe主设备，RC或者EP，访问目标设备的存储器空间时，使用Non-Posted总线事务向目标设备发出存储器读请求TLP，目标设备收到这个存储器读请求TLP后，使用存储器读完成TLP，主动向主设备传递数据。当主设备收到目标设备的存储器读完成TLP后，将完成一次存储器读操作。

< >存储器写请求TLP原子操作请求和完成报文I/O读写请求TLP和读写完成TLP配置读写请求TLP和配置读写完成TLP消息报文

与PCI总线相比，PCIe总线增加了消息请求事务。PCIe总线使用基于报文的数据传送模式，所有总线事务都是通过报文实现的，PCIe总线取消了一些在PCI总线中存在的边带信号。在PCIe总线中，一些由PCI总线的边带信号完成的工作，如中断请求和电源管理等，在PCIe总线中由消息请求报文实现。

### 5.3.1 存储器读写请求TLP

存储器读写请求TLP的格式如图5‑8所示。

[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635234139.gif)

在PCIe总线中，存储器写请求TLP使用Posted数据传送方式。而其他与存储器和I/O相关的报文都使用Split方式进行数据传送，这些请求报文需要完成报文，通知发送端之前的数据请求报文已经被处理完毕，有关完成报文的详细说明见第5.3.2节。

存储器读写请求TLP使用地址路由方式进行数据传递，在这类TLP头中包含Address字段，Address字段具有两种地址格式，分别是32位和64位地址。在存储器读写和I/O读写请求的第3和第4个双字中，存放TLP的32或者64位地址。存储器、I/O和原子操作读写请求使用的TLP头较为类似。本节仅介绍存储器、I/O读写使用的TLP头，而在第5.3.5节详细介绍原子操作。

#### 1 Length字段

在存储器读请求TLP中并不包含Data Payload，在该报文中，Length字段表示需要从目标设备数据区域读取的数据长度；而在存储器写TLP中，Length字段表示当前报文的Data Payload长度。

Length字段的最小单位为DW。当该字段为n时，表示需要获得的数据长度或者当前报文的数据长度为n个DW，其中0£n£0x3FF。值得注意的是，当n等于0时，表示数据长度为1024个DW。

#### 2 DW BE字段

PCIe总线以字节为基本单位进行数据传递，但是Length字段以DW为最小单位。为此TLP使用Last DW BE和First DW BE这两个字段进行字节使能，使得在一个TLP中，有效数据以字节为单位。

这两个DW BE字段各由4位组成，其中Last DW BE字段的每一位对应数据Payload最后一个双字的字节使能位；而First DW BE字段的每一位对应数据Payload第一个双字的字节使能位。其对应关系如表5‑5所示。

表5‑5 First和Last DW BE字段

|  |  |  |
| --- | --- | --- |
| Last DW BE | 第3位 | 为1表示数据Payload的最后一个双字的字节3有效 |
| 第2位 | 为1表示数据Payload的最后一个双字的字节2有效 |
| 第1位 | 为1表示数据Payload的最后一个双字的字节1有效 |
| 第0位 | 为1表示数据Payload的最后一个双字的字节0有效 |
| First DW BE | 第3位 | 为1表示数据Payload的第一个双字的字节3有效 |
| 第2位 | 为1表示数据Payload的第一个双字的字节2有效 |
| 第1位 | 为1表示数据Payload的第一个双字的字节1有效 |
| 第0位 | 为1表示数据Payload的第一个双字的字节0有效 |

Last DW BE和First DW BE这两个字段的使用规则如下。

< >如果传送的数据长度在一个对界的双字(DW)之内，则Last DW BE字段为0b0000，而First DW BE的对应位置1；如果数据长度超过1DW，Last DW BE字段一定不能为0b0000。PCIe总线使用Last DW BE字段为0b0000表示所传送的数据在一个对界的DW之内。如果传送的数据长度超过1DW，则First DW BE字段至少有一个位使能。不能出现First DW BE为0b0000的情况。如果传送的数据长度大于等于3DW，则在First DW BE和Last DW BE字段中不能出现不连续的置1位。如果传送的数据长度在1DW之内时，在First DW BE字段中允许有不连续的置1位。此时PCIe总线允许在TLP中传送1个DW的第1，3字节或者第0，2字节。

* 如果传送的数据长度为2DW之内时，则First DW BE字段和Last DW BE字段允许有不连续的置1位。

值得注意的是，PCIe总线支持一种特殊的读操作，即“Zero-Length”读请求。此时Length字段的长度为1DW，而First DW BE字段和Last DW BE字段都为0b0000，即所有字节都不使能。此时与这个存储器读请求TLP对应的读完成TLP中不包含有效数据。再次提醒读者注意“Zero-Length”读请求使用的Length字段为1，而不是为0，为0表示需要获得的数据长度为1024个DW。

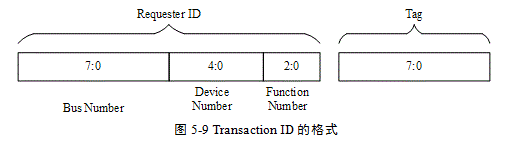
“Zero-Length”读请求的引入是为了实现“读刷新”操作，该操作的主要目的是为了确保之前使用Posted方式所传送的数据，到达最终的目的地，与“Zero-Length”读对应的读完成报文中不含有负载，从而提高了PCIe链路的利用率。

在PCIe总线中，使用Posted方式进行存储器写时，目标设备不需要向主设备发送回应报文，因此主设备并不知道这个存储器写是否已经达到目的地。而主设备可以使用“读刷新”操作，向目标设备进行读操作来保证存储器写最终到达目的地。

在PCIe总线中，标准的存储器读请求也可以完成同样的刷新操作。但是“Zero-Length”读请求与这种读请求相比，其完成报文不需要“Data Payload”，因此在一定程度上提高了PCIe总线的效率。如果一个存储器读请求TLP报文的TH位为1时，DW BE字段将被重新定义为ST[7:0]字段，有关ST字段的详细说明见第5.3.6节。

#### 3 Requester ID字段

Requester ID字段包含“生成这个TLP报文”的PCIe设备的总线号(Bus Number)、设备号(Device Number)和功能号(Function Number)，其格式如图5‑9所示。对于存储器写请求TLP，Requester ID字段并不是必须的，因为目标设备收到存储器写请求TLP后，不需要完成报文作为应答，因此Requester ID字段对于存储器写请求TLP并没有实际意义。

[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635260239.gif)

但是PCIe总线规范并没有明确说明存储器写请求TLP究竟需不需要Requester ID字段，为此IC设计者依然需要将存储器写TLP的Requester ID字段置为有效。值得注意的是，如果一个存储器写请求TLP报文的TH位为1时，Tag字段将被重新定义为ST[7:0]字段，有关ST字段的详细说明见第5.3.6节。

对于Non-Posted数据请求，目标设备需要使用完成报文做为回应。在这个完成报文中，需要使用源设备的Requester ID字段。因此在Non-Posted数据请求TLP中，如存储器读请求、I/O和配置读写请求TLP，必须使用Requester ID字段。

存储器，I/O读请求TLP中含有Requester ID和Tag字段。在PCIe总线中Requester ID和Tag字段合称为Transaction ID，Transaction ID字段的格式如图5‑9所示。存储器读，I/O和配置读写请求TLP使用Transaction字段的主要目的是使接收端通过分析报文的Transaction ID，确认完成报文的目的地。

在PCIe总线中，所有Non-Posted数据请求都需要完成报文作为应答，才能结束一次完整的数据传递。一个源设备在发送Non-Posted数据请求之后，如果并没有收到目标设备回送的完成报文，TLP报文的发送端需要保存这个Non-Posted数据请求，此时该设备使用的Transaction ID(Tag字段)不能被再次使用，直到一次数据传送结束，即数据发送端收齐与该TLP对应的所有完成报文。

PCIe设备发出的每一个Non-Posted数据请求TLP，在同一个时刻段内Transaction ID必须是唯一的。即在同一时间段内，在当前PCI总线域中不能存在两个或者两个以上的存储器读请求TLP，其Transaction ID完全相同。

源设备发送Non-Posted数据请求后，在没有获得全部完成报文之前，不能释放这个Transaction ID占用的资源。在同一个PCIe设备发送的TLP中，其Requester ID字段是相同的，因此PCIe设备的设计者所能管理的资源是Tag字段。PCIe设备的设计者需要合理地管理Tag资源，以保证数据传送的正确性。

PCIe设备在发送Non-Posted数据请求时，需要暂存这些Non-Posted数据请求。其中Tag字段的长度决定了发送端能够暂存多少个同类型的TLP，如果Tag字段长度为5，发送端能够暂存32个报文；如果PCIe设备使能了Extended Tag位，Tag字段可以由8位组成，此时发送端能够暂存256个报文。

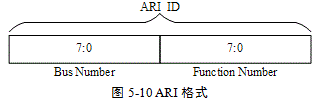
通过Tag字段的长度，可以发现每个PCIe设备最多可以暂存256个同类型的Non-Posted报文。但是在多数情况下，一个PCIe设备可能只包含1个Function。因此PCIe设备还可以使用Function号扩展Tag字段，从而扩展“暂存TLP报文”的数目。

PCIe设备在PCI Express Capability结构的Device Control寄存器中，设置了一个Phantom Functions Enable位，。当一个PCIe设备仅支持一个Function时，Phantom Functions Enable位可以被设置为1，此时PCIe设备可以使用Requester ID的Function Number字段对Tag字段进一步扩展，此时一个PCIe设备最多可以支持2048个同类型的数据请求。

由以上分析可以发现，一个PCIe设备最多可以支持2048个存储器读数据请求，基本上可以满足绝大多数需要。但是在某些特殊应用场合，PCIe设备即使可以暂存2048个存储器读请求，也并不足够。

与PCI总线相比，PCIe总线的数据传送延时较长，而为了弥补这个传送延时，PCIe设备通常使用流水线技术。此时PCIe设备必须能够连续发送多个存储器读请求报文，随后RC也将连续回送多个存储器读完成报文，在PCIe设备的实现中，需要保证能够源源不断地从RC接收这些报文，以充分利用报文接收流水线，。

PCIe V2.1总线规范还提出了另一种Requester ID格式，即ARI (Alternative Routing-ID Interpretation)格式，除了Requester ID外，在完成报文中使用的Completer ID也可以使用这种格式。ARI格式将ID号分为两个字段，分别为Bus号和Function号，而不使用Device号，ARI格式如图5‑10所示。

[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635283123.gif)

PCIe总线引入ARI格式的依据是在一个PCIe链路上仅可能存在一个PCIe设备，因而其Device号一定为0。在多数PCIe设备中，Requester ID和Completion ID包含的Device号是没有意义的。使用ARI格式时，一个PCIe设备最多可以支持256个Function，而传统的PCIe设备最多只能支持8个Function。

#### 4 I/O读写请求TLP的规则

I/O读写请求与存储器读写请求TLP的格式基本类似，只是I/O读写请求TLP只能使用32位地址模式和基于地址的路由方式，而且I/O读写请求TLP只能使用Non-Posted方式进行传递。PCIe总线并不建议PCIe设备支持I/O地址空间，但是Switch和RC需要具备接收和发送I/O请求报文的能力，因为许多老的PCI设备依然使用I/O地址空间，这些PCI设备可以通过PCIe桥连接到PCIe总线中。因此虽然支持I/O读写请求的PCIe设备极少，但是在PCIe体系结构中，依然需要支持PCI总线域的I/O地址空间。

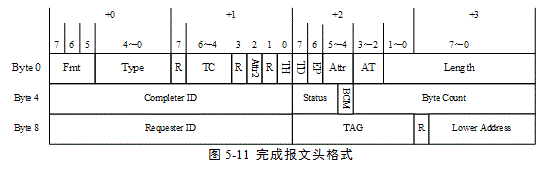
与存储器读写请求TLP不同，I/O读写请求TLP的某些字段必须为以下值。

* TC[2:0]必须为0，I/O请求报文使用的TC标签只能为0。
* TH和Attr2位保留，而Attr[1:0]必须为“0b00”，这表示I/O请求报文必须使用PCI总线的强序数据传送模式，而且在传送过程中，硬件保证其传送的数据与Cache保持一致，实际上I/O地址空间都是不可Cache的。
* AT[1:0]必须为“0b00”，表示不支持地址转换，因此在虚拟化技术中，并不处理PCI总线域中的I/O空间。
* Length[9:0]为“0b00 0000 0001”，表示I/O读写请求TLP最大的数据Payload为1DW，该类TLP不支持突发传送。
* Last DW[3:0]为“0b0000”。

### 5.3.2 完成报文

PCIe总线支持Split传送方式，目标设备使用完成报文向源设备主动发送数据。完成报文使用ID路由方式，由TLP Predix、报文头和Data Payload组成，但是在某些完成报文可以不含有Data Payload，如I/O或者配置写完成和Zero-Length读完成报文。在PCIe总线中，有一下几类数据请求需要收到完成报文之后，才能完成整个数据传送过程，完成报文格式如图5‑11所示。

* 所有的数据读请求，包括存储器、I/O读请求、配置读请求和原子操作请求。当一个PCIe设备发出这些数据请求报文后，必须收到目标设备的完成报文后，才能结束一次数据传送。这一类完成报文必须包含Data Payload。
* 所有的Non-Posted数据写请求，包括I/O和配置写请求。当一个PCIe设备发出这些数据请求报文后，必须收到目标设备的完成报文后，才能结束数据传送。但是这一类完成报文不包含数据，仅包含应答信息。
* 与ATS机制相关的一些报文。

[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635304988.gif)

完成报文“Byte 0”中的大部分字段与“存储器，I/O、配置请求报文”的对应字段的含义相同。完成报文一次最多能够传送的报文大小不能超过Max\_Payload\_Size参数。在多数处理器中，完成报文中包含的数据在一个Cache行之内，完成报文使用RCB参数来处理数据对界，RCB参数的大小与处理器系统的Cache行长度和DDR-SDRAM的一次突发传送长度相关，这些参数的详细描述见第5.4.3节。在x86和PowerPC处理器中，一个存储器读完成报文一般不超过RCB参数。

#### 1 Requester ID和Tag字段

完成报文使用ID路由方式，ID路由方式详见第5.2.2节。完成报文头的长度为3DW，完成报文头中包含Transaction ID信息，由Requester ID和Tag字段组成，这个ID必须与源设备发送的数据请求报文的Transaction ID对应，完成报文使用Transaction ID进行ID路由，并将数据发送给源设备。

当PCIe设备收到存储器读、I/O读写或者配置读写请求TLP时，需要首先保存这些报文的Transaction ID，之后当该设备准备好完成报文后，将完成报文的Requester ID和Tag字段赋值为之前保存的Transaction ID字段。

#### 2 Completer ID字段

Completer ID字段的含义与Requester ID字段较为相似，只是该字段存放“发送完成报文”的PCIe设备的ID号。PCIe设备进行数据请求时需要在TLP字段中包含Requester ID字段；而使用完成报文结束数据请求时，需要提供Completer ID字段。

#### 3 Status字段

Status字段保存当前完成报文的完成状态，表示当前TLP是正确地将数据传递给数据请求端；还是在数据传递过程中出现错误；或者要求数据请求方进行重试。PCIe总线规定了几类完成状态，如表5‑6所示。

表5‑6 Status字段

| **Status[2:0]** | **描述** |
| --- | --- |
| 0b000 | SC(Sucessful Completion)，正常结束 |
| 0b001 | UR(Unsupported Request)，不支持的数据请求 |
| 0b010 | CRS(Configuration Request Retry Status)，要求数据请求方进行重试。当RC对一个PCIe目标设备发起配置请求时，如果该目标设备没有准备好，可以向RC发出CRS完成报文，当RC收到这类报文时，不能结束本次配置请求，必须择时重新发送配置请求 |
| 0b100 | CA(Completion Abort)，数据夭折。表示目标设备无法完成本次数据请求 |
| 其他 | 保留 |

#### 4 BCM位与Byte Count字段

BCM(Byte Count Modified)字段由PCI-X设备设置。PCI-X设备也支持Split Transaction传送方式，当PCI-X设备进行存储器读请求时，目标设备不一定一次就能将所有数据传递给源设备。此时目标设备在进行第一次数据传送时，需要设置Byte Count字段和BCM位。

BCM位表示Byte Count字段是否被更改，该位仅对PCI-X设备有效，而PCIe设备不能操纵BCM位，只有PCI-X设备或者PCIe-to-PCI-X桥可以改变该位。本节对此位不做进一步介绍，对此位感兴趣的读者可以参考PCI-X Addendum to the PCI Local Bus Specification, Revision 2.0。

Byte Count字段记录源设备还需要从目标设备中，获得多少字节的数据就能完成全部数据传递，当前TLP中的有效负载也被Byte Count字段统计在内。该字段由12位组成。该字段为0b0000-0000-0001表示还剩一个字节，为0b1111-1111-1111表示还剩4095个字节，而为0b0000-0000-0000表示还剩4096个字节。除了存储器读请求的完成报文外，大多数完成报文的Byte Count字段为4。

如一个源设备向目标设备发送一个“读取128B的存储器读请求TLP”，而目标设备收到这个读请求TLP后，可能使用两个存储器读完成TLP传递数据。其中第1个存储器读完成TLP的有效数据为64B，而Byte Count字段为128；第2个存储器读完成TLP中的有效数据为64B，而Byte Count字段也为64。当数据请求端接收完毕第1个存储器读完成TLP后，发现还有64B的数据没有接收完毕，此时必须等待下一个存储器读完成TLP。等到数据请求端收齐所有数据后，才能结束整个存储器读请求。

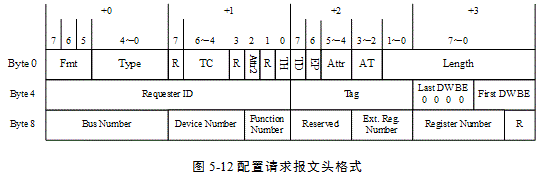
目标设备发出的第2个读完成TLP中的有效数据为64B，而Byte Count字段为64，当数据请求端接收完毕这个读完成TLP后，将完成一个完整的存储器读过程，从而可以释放这个存储器读过程使用的Tag资源。存储器读请求的完成报文的拆分方式较为复杂，Byte Count字段的设置也相对较为复杂。

#### 5 Lower Address字段

如果当前完成报文为存储器读完成TLP，该字段存放在存储器读完成TLP中第一个数据所对应地址的最低位。值得注意的是，在完成报文中，并不存在First DW BE和Last DW BE字段，因此接收端必须使用存储器读完成TLP的Low Address字段，识别一个TLP中包含数据的起始地址。

### 5.3.3 配置读写请求TLP

配置读写请求TLP由RC发起，用来访问PCIe设备的配置空间。配置请求报文使用基于ID的路由方式。PCIe总线也支持两种配置请求报文，分别为Type 00h和Type 01h配置请求。配置请求TLP的格式如图5‑12所示。

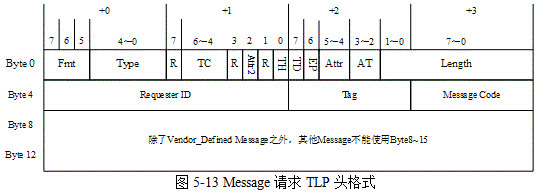
[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635433959.gif)

配置请求TLP的第4~7字节与存储器请求TLP类似。而第8~11字节的Bus、Device和Function Number中存放该TLP访问的目标设备的相应的号码，而Ext Register和Reigister Number存放寄存器号。配置请求报文的其他字段必须为以下值。

* TC[2:0]必须为0，I/O请求报文的传送类型(TC)只能为0。
* TH位为保留位；Attr2位为保留，而Attr[1:0]必须为“00b”，这表示I/O请求报文使用PCI总线的强序数据传送模式；AT[1:0]必须为“0b00”，表示不进行地址转换。
* Length[9:0]为“0b00 0000 0001”，表示配置读写请求最大Payload为1DW。
* Last DW BE字段为“0b0000”。而First DW BE字段根据配置读写请求的大小设置。

### 5.3.4 消息请求报文

在PCIe总线中，多数消息报文使用隐式路由方式，其格式如图5‑13所示。其中Byte 0字段为通用TLP头，而Byte 4的第3字节中存放Message Code字段。

[](http://download.bbs.ednchina.com/images/attachments/201307/original/8957263754_TIME_1374635451993.gif)

PCIe总线规定了以下几类消息报文。

* INTx中断消息报文(INTx Interrupt Signaling)。
* 电源管理消息报文(Power Management)。
* 错误消息报文(Error Signaling)。
* 锁定事务消息报文(Locked Transaction Support)。
* 插槽电源限制消息报文(Slot Power Limit Support)。
* Vendor-Defined Messages。

|  |
| --- |
| **5.4 TLP中与数据负载相关的参数** |
| 发布时间：2013-07-24 10:46:01 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

在PCIe总线中，有些TLP含有Data Payload，如存储器写请求、存储器读完成TLP等。在PCIe总线中，TLP含有的Data Payload大小与Max\_Payload\_Size、Max\_Read\_Request\_Size和RCB参数相关。下文将分别介绍这些参数的使用。

### 5.4.1 Max\_Payload\_Size参数

PCIe总线规定在TLP报文中，数据有效负载的最大值为4KB，但是PCIe设备并不一定能够发送这么大的数据报文。PCIe设备含有“Max\_Payload\_Size”和“Max\_Payload\_Size Supported”参数，这两个参数分别在Device Capability寄存器和Device Control寄存器中定义。

“Max\_Payload\_Size Supported”参数存放在一个PCIe设备中，TLP有效负载的最大值，该参数由PCIe设备的硬件逻辑确定，系统软件不能改写该参数。而Max\_Payload\_Size参数存放PCIe设备实际使用的，TLP有效负载的最大值。该参数由PCIe链路两端的设备协商决定，是PCIe设备进行数据传送时，实际使用的参数。

PCIe设备发送数据报文时，使用Max\_Payload\_Size参数决定TLP的最大有效负载。当PCIe设备的所传送的数据大小超过Max\_Payload\_Size参数时，这段数据将被分割为多个TLP进行发送。当PCIe设备接收TLP时，该TLP的最大有效负载也不能超过Max\_Payload\_Size参数，如果接收的TLP，其Length字段超过Max\_Payload\_Size参数，该PCIe设备将认为该TLP非法。

RC或者EP在发送存储器读完成TLP时，这个存储器读完成TLP的最大Payload也不能超过Max\_Payload\_Size参数，如果超过该参数，PCIe设备需要发送多个读完成报文。值得注意的是，这些读完成报文需要满足RCB参数的要求，有关RCB参数的详细说明见下文。

在实际应用中，尽管有些PCIe设备的Max\_Payload\_Size Supported参数可以为256B、512B、1024B或者更高，但是如果PCIe链路的对端设备可以支持的Max\_Payload\_Size参数为128B时，系统软件将使用对端设备的Max\_Payload\_Size Supported参数，初始化该设备的Max\_Payload\_Size参数，即选用PCIe链路两端最小的Max\_Payload\_Size Supported参数初始化Max\_Payload\_Size参数。

在多数x86处理器系统的MCH或者ICH中，Max\_Payload\_Size Supported参数为128B。这也意味着在x86处理器中，与MCH或者ICH直接相连的PCIe设备进行DMA读写时，数据的有效负载不能超过128B。而在PowerPC处理器系统中，该参数大多为256B。

目前在大多数EP中，Max\_Payload\_Size Supported参数不大于512B，因为在大多数处理器系统的RC中，Max\_Payload\_Size Supported参数也不大于512B。因此即便EP支持较大的Max\_Payload\_Size Supported参数，并不会提高数据传送效率。

而Max\_Payload\_Size参数的大小与PCIe链路的传送效率成正比，该参数越大，PCIe链路带宽的利用率越高，该参数越小，PCIe链路带宽的利用率越低。

PCIe总线规范规定，对于实时性要求较高的PCIe设备，Max\_Payload\_Size参数不应设置过大，因此这个参数有时会低于PCIe链路允许使用的最大值。

### 5.4.2 Max\_Read\_Request\_Size参数

Max\_Read\_Request\_Size参数由PCIe设备决定，该参数规定了PCIe设备一次能从目标设备读取多少数据。

Max\_Read\_Request\_Size参数在Device Control寄存器中定义。该参数与存储器读请求TLP的Length字段相关，其中Length字段不能大于Max\_Read\_Request\_Size参数。在存储器读请求TLP中，Length字段表示需要从目标设备读取多少数据。

值得注意的是，Max\_Read\_Request\_Size参数与Max\_Payload\_Size参数间没有直接联系，Max\_Payload\_Size参数仅与存储器写请求和存储器读完成报文相关。

PCIe总线规定存储器读请求，其读取的数据长度不能超过Max\_Read\_Request\_Size参数，即存储器读TLP中的Length字段不能大于这个参数。如果一次存储器读操作需要读取的数据范围大于Max\_Read\_Request\_Size参数时，该PCIe设备需要向目标设备发送多个存储器读请求TLP。

PCIe总线规定Max\_Read\_Request\_Size参数的最大值为4KB，但是系统软件需要根据硬件特性决定该参数的值。因为PCIe总线规定EP在进行存储器读请求时，需要具有足够大的缓冲接收来自目标设备的数据。

如果一个EP的Max\_Read\_Request\_Size参数被设置为4KB，而且这个EP每发出一个4KB大小存储器读请求时，EP都需要准备一个4KB大小的缓冲[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dse9.html#_ftn1)。这对于绝大多数EP，这都是一个相当苛刻的条件。为此在实际设计中，一个EP会对Max\_Read\_Request\_Size参数的大小进行限制。

### 5.4.3 RCB参数

RCB位在Link Control寄存器中定义。RCB位决定了RCB参数的值，在PCIe总线中，RCB参数的大小为64B或者128B，如果一个PCIe设备没有设置RCB的大小[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dse9.html#_ftn2)，则RC的RCB参数缺省值为64B，而其他PCIe设备的RCB参数的缺省值为128B。PCIe总线规定RC的RCB参数的值为64B或者128B，其他PCIe设备的RCB参数为128B。

在PCIe总线中，一个存储器读请求TLP可能收到目标设备发出的多个完成报文后，才能完成一次存储器读操作。因为在PCIe总线中，一个存储器读请求最多可以请求4KB大小的数据报文，而目标设备可能会使用多个存储器读完成TLP才能将数据传递完毕。

当一个EP向RC或者其他EP读取数据时，这个EP首先向RC或者其他EP发送存储器读请求TLP；之后由RC或者其他EP发送存储器读完成TLP，将数据传递给这个EP。

如果存储器读完成报文所传递数据的地址范围没有跨越RCB参数的边界，那么数据发送端只能使用一个存储器完成报文将数据传递给请求方，否则可以使用多个存储器读完成TLP。

假定一个EP向地址范围为0xFFFF-0000~0xFFFF-0010这段区域进行DMA读操作，RC收到这个存储器读请求TLP后，将组织存储器读完成TLP，由于这段区域并没有跨越RCB边界，因此RC只能使用一个存储器读完成TLP完成数据传递。

如果存储器读完成报文所传递数据的地址范围跨越了RCB边界，那么数据发送端(目标设备)可以使用一个或者多个完成报文进行数据传递。数据发送端使用多个存储器读完成报文完成数据传递时，需要遵循以下原则。

* 第一个完成报文所传送的数据，其起始地址与要求的起始地址相同。其结束地址或者为要求的结束地址(使用一个完成报文传递所有数据)，或者为RCB参数的整数倍(使用多个完成报文传递数据)。
* 最后一个完成报文的起始地址或者为要求的起始地址(使用一个完成报文传递所有数据)，或者为RCB参数的整数倍(使用多个完成报文传递数据)。其结束地址必须为要求的结束地址。
* 中间的完成报文的起始地址和结束地址必须为RCB参数的整数倍。

当RC或者EP需要使用多个存储器读完成报文将0xFFFE-FFF0~0xFFFF-00C7之间的数据发送给数据请求方时，可以将这些完成报文按照表5‑9方式组织。

 表5‑9 存储器读完成报文的拆分方法

|  |  |  |
| --- | --- | --- |
| 方式1 | 方式2 | 方式3 |
| 0xFFFE-FFF0~0xFFFE-FFFF | 0xFFFE-FFF0~0xFFFE-FFFF | 0xFFFE-FFF0~0xFFFE-FFFF |
| 0xFFFF-0000~0xFFFF-003F | 0xFFFF-0000~0xFFFF-007F | 0xFFFF-0000~0xFFFF-00C7 |
| 0xFFFF-0040~0xFFFF-007F | 0xFFFF-0080~0xFFFF-00C7 |  |
| 0xFFFF-0080~0xFFFF-00BF |  |  |
| 0xFFFF-00C0~0xFFFF-00C7 |  |  |

上表提供的方式仅供参考，目标设备还可以使用其他拆分方法发送存储器读完成TLP。PCIe总线使用多个完成报文实现一次数据读请求的主要原因是考虑Cache行长度和流量控制。在多数x86处理器系统中，存储器读完成报文的数据长度为一个Cache行，即一次传送64B。除此之外，较短的数据完成报文占用流量控制的资源较少，而且可以有效避免数据拥塞。

## 5.5 小结

   本章重点介绍PCIe总线的事务层。在PCIe总线层次结构中，事务层最易理解，同时也与系统软件直接相关。

[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dse9.html#_ftnref1) 这是流量控制Infinite FC Unit的要求，详见第9.3.2节。

[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dse9.html#_ftnref2) 有些PCIe设备可能没有Link Control寄存器。

|  |
| --- |
| **第6章 MSI和MSI-X中断机制** |
| 发布时间：2013-08-13 16:45:08 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

在PCI总线中，所有需要提交中断请求的设备，必须能够通过INTx引脚提交中断请求，而MSI机制是一个可选机制。而在PCIe总线中，PCIe设备必须支持MSI或者MSI-X中断请求机制，而可以不支持INTx中断消息。

在PCIe总线中，MSI和MSI-X中断机制使用存储器写请求TLP向处理器提交中断请求，下文为简便起见将传递MSI/MSI-X中断消息的存储器写报文简称为MSI/MSI-X报文。不同的处理器使用了不同的机制处理这些MSI/MSI-X中断请求，如PowerPC处理器使用MPIC中断控制器处理MSI/MSI-X中断请求，本章将在第6.2节中介绍这种处理情况；而x86处理器使用FSB Interrupt Message方式处理MSI/MSI-X中断请求。

不同的处理器对PCIe设备发出的MSI报文的解释并不相同。但是PCIe设备在提交MSI中断请求时，都是向MSI/MSI-X Capability结构中的Message Address的地址写Message Data数据，从而组成一个存储器写TLP，向处理器提交中断请求。

有些PCIe设备还可以支持Legacy中断方式[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dski.html#_ftn1)。但是PCIe总线并不鼓励其设备使用Legacy中断方式，在绝大多数情况下，PCIe设备使用MSI或者MSI/X方式进行中断请求。

PCIe总线提供Legacy中断方式的主要原因是，在PCIe体系结构中，存在许多PCI设备，而这些设备通过PCIe桥连接到PCIe总线中。这些PCI设备可能并不支持MSI/MSI-X中断机制，因此必须使用INTx信号进行中断请求。

当PCIe桥收到PCI设备的INTx信号后，并不能将其直接转换为MSI/MSI-X中断报文，因为PCI设备使用INTx信号进行中断请求的机制与电平触发方式类似，而MSI/MSI-X中断机制与边沿触发方式类似。这两种中断触发方式不能直接进行转换。因此当PCI设备的INTx信号有效时，PCIe桥将该信号转换为Assert\_INTx报文，当这些INTx信号无效时，PCIe桥将该信号转换为Deassert\_INTx报文。

与Legacy中断方式相比，PCIe设备使用MSI或者MSI-X中断机制，可以消除INTx这个边带信号，而且可以更加合理地处理PCIe总线的“序”。目前绝大多数PCIe设备使用MSI或者MSI-X中断机制提交中断请求。

MSI和MSI-X机制的基本原理相同，其中MSI中断机制最多只能支持32个中断请求，而且要求中断向量连续，而MSI-X中断机制可以支持更多的中断请求，而并不要求中断向量连续。与MSI中断机制相比，MSI-X中断机制更为合理。本章将首先介绍MSI/MSI-X Capability结构，之后分别以PowerPC处理器和x86处理器为例介绍MSI和MSI-X中断机制。

[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dski.html#_ftnref1) 通过发送Assert\_INTx和Deassert\_INTx消息报文进行中断请求，即虚拟中断线方式。

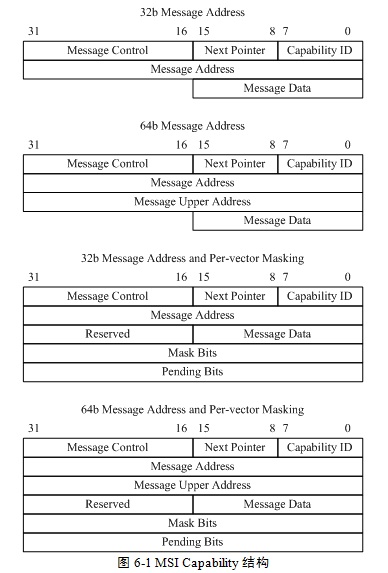
|  |
| --- |
| **6.1 MSI/MSI-X Capability结构** |
| 发布时间：2013-08-13 16:54:15 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PCIe设备可以使用MSI或者MSI-X报文向处理器提交中断请求，但是对于某个具体的PCIe设备，可能仅支持一种报文。在PCIe设备中含有两个Capability结构，一个是MSI Capability结构，另一个是MSI-X Capability结构。通常情况下一个PCIe设备仅包含一种结构，或者为MSI Capability结构，或者为MSI-X Capability结构。

### 6.1.1 MSI Capability结构

MSI Capability结构共有四种组成方式，分别是32和64位的Message结构，32位和64位带中断Masking的结构。MSI报文可以使用32位地址或者64位地址，而且可以使用Masking机制使能或者禁止某个中断源。MSI Capability寄存器的结构如图6‑1所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1376383677503.jpg)

* Capability ID字段记载MSI Capability结构的ID号，其值为0x05。在PCIe设备中，每一个Capability结构都有唯一的ID号。
* Next Pointer字段存放下一个Capability结构的地址。
* Message Control字段。该字段存放当前PCIe设备使用MSI机制进行中断请求的状态与控制信息，如表6‑1所示。

 表6‑1 MSI Cabalibities结构的Message Control字段

| **Bits** | **定义** | | **描述** |
| --- | --- | --- | --- |
| 15:9 | Reserved | | 保留位。系统软件读取该字段时将返回全零，对此字段写无意义。 |
| 8 | Per-vector Masking Capable | | 该位为1时，表示支持带中断Masking的结构；如果为0，表示不支持带中断Masking的结构。该位对系统软件只读，该位在PCIe设备初始化时设置。 |
| 7 | 64 bit Address Capable | | 该位为1时，表示支持64位地址结构；如果为0，表示只能支持带32位地址结构。该位对系统软件只读，该位在PCIe设备初始化时设置。 |
| 6:4 | Multiple Message Enable | | 该字段可读写，表示软件分配给当前PCIe设备的中断向量数目。系统软件根据Multiple Message Capable字段的大小确定该字段的值。在系统的中断向量资源并不紧张时，Multiple Message Capable字段和该字段的值相等；而资源紧张时，该字段的值可能小于Multiple Message Capable字段的值。 |
| 3:1 | Multiple Message Capable | 该字段对系统软件只读，表示当前PCIe设备可以使用几个中断向量号，在不同的PCIe设备中该字段的值并不不同。当该字段为0b000时，表示PCIe设备可以使用1个中断向量；为0b001、0b010、0b011、0b100和0b101时，表示使用4、8、16和32个中断向量；而0b110和0b111为保留位。 该字段与Multiple Message Enable字段的含义不同，该字段表示，当前PCIe设备支持的中断向量个数，而Multiple Message Enable字段是系统软件分配给PCIe设备实际使用的中断向量个数。 | |
| 0 | MSI Enable | | 该位可读写，是MSI中断机制的使能位。该位为1而且MSI-X Enable位为0时，当前PCIe设备可以使用MSI中断机制，此时Legacy中断机制被禁止。一个PCIe设备的MSI Enble和MSI-X Enable位都被禁止时，将使用INTx中断消息报文发出/结束中断请求[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskj.html#_ftn1)。 |
|  |  |  |  |

< >Message Address字段。当MSI Enable位有效时，该字段存放MSI存储器写事务的目的地址的低32位。该字段的31:2字段有效，系统软件可以对该字段进行读写操作；该字段的第1~0位为0。Message Upper Address字段。如果64 bit Address Capable位有效，该字段存放MSI存储器写事务的目的地址的高32位。

Message Data字段，该字段可读写。当MSI Enable位有效时，该字段存放MSI报文使用的数据。该字段保存的数值与处理器系统相关，在PCIe设备进行初始化时，处理器将初始化该字段，而且不同的处理器填写该字段的规则并不相同。如果Multiple Message Enable字段不为0b000时(即该设备支持多个中断请求时)，PCIe设备可以通过改变Message Data字段的低位数据发送不同的中断请求。Mask Bits字段。PCIe总线规定当一个设备使用MSI中断机制时，最多可以使用32个中断向量，从而一个设备最多可以发送32种中断请求。

Mask Bits字段由32位组成，其中每一位对应一种中断请求。当相应位为1时表示对应的中断请求被屏蔽，为0时表示允许该中断请求。系统软件可读写该字段，系统初始化时该字段为全0，表示允许所有中断请求。该字段和Pending Bits字段对于MSI中断机制是可选字段，但是PCIe总线规范强烈建议所有PCIe设备支持这两个字段。

Pending Bits字段。该字段对于系统软件是只读位，PCIe设备内部逻辑可以改变该字段的值。该字段由32位组成，并与PCIe设备使用的MSI中断一一对应。该字段需要与Mask Bits字段联合使用。

当Mask Bits字段的相应位为1时，如果PCIe设备需要发送对应的中断请求时，Pending Bits字段的对应位将被PCIe设备的内部逻辑置1，此时PCIe设备并不会使用MSI报文向中断控制器提交中断请求；当系统软件将Mask Bits字段的相应位从1改写为0时，PCIe设备将发送MSI报文向处理器提交中断请求，同时将Pending Bit字段的对应位清零。在设备驱动程序的开发中，有时需要联合使用Mask Bits和Pending Bits字段防止处理器丢弃中断请求[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskj.html#_ftn2)。

### 6.1.2 MSI-X Capability结构

MSI-X Capability中断机制与MSI Capability的中断机制类似。PCIe总线引出MSI-X机制的主要目的是为了扩展PCIe设备使用中断向量的个数，同时解决MSI中断机制要求使用中断向量号连续所带来的问题。

MSI中断机制最多只能使用32个中断向量，而MSI-X可以使用更多的中断向量。目前Intel的许多PCIe设备支持MSI-X中断机制。与MSI中断机制相比，MSI-X机制更为合理。首先MSI-X可以支持更多的中断请求，但是这并不是引入MSI-X中断机制最重要的原因。因为对于多数PCIe设备，32种中断请求已经足够了。而引入MSI-X中断机制的主要原因是，使用该机制不需要中断控制器分配给该设备的中断向量号连续。

如果一个PCIe设备需要使用8个中断请求时，如果使用MSI机制时，Message Data的[2:0]字段可以为0b000~0b111，因此可以发送8种中断请求，但是这8种中断请求的Message Data字段必须连续。在许多中断控制器中，Message Data字段连续也意味着中断控制器需要为这个PCIe设备分配8个连续的中断向量号。

有时在一个中断控制器中，虽然具有8个以上的中断向量号，但是很难保证这些中断向量号是连续的。因此中断控制器将无法为这些PCIe设备分配足够的中断请求，此时该设备的“Multiple Message Enable”字段将小于“Multiple Message Capable”。

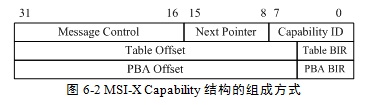
而使用MSI-X机制可以合理解决该问题。在MSI-X Capability结构中，每一个中断请求都使用独立的Message Address字段和Message Data字段，从而中断控制器可以更加合理地为该设备分配中断资源。

与MSI Capability寄存器相比，MSI-X Capability寄存器使用一个数组存放Message Address字段和Message Data字段，而不是将这两个字段放入Capability寄存器中，本篇将这个数组称为MSI-X Table。从而当PCIe设备使用MSI-X机制时，每一个中断请求可以使用独立的Message Address字段和Message Data字段。

除此之外MSI-X中断机制还使用了独立的Pending Table表，该表用来存放与每一个中断向量对应的Pending位。这个Pending位的定义与MSI Capability寄存器的Pending位类似。MSI-X Table和Pending Table存放在PCIe设备的BAR空间中。MSI-X机制必须支持这个Pending Table，而MSI机制的Pending Bits字段是可选的。

#### 1 MSI-X Capability结构

MSI-X Capability结构比MSI Capability结构略微复杂一些。在该结构中，使用MSI-X Table存放该设备使用的所有Message Address和Message Data字段，这个表格存放在该设备的BAR空间中，从而PCIe设备可以使用MSI-X机制时，中断向量号可以并不连续，也可以申请更多的中断向量号。MSI-X Capability结构的组成方式如图6‑2所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1376383833426.jpg)

上图中各字段的含义如下所示。

* Capability ID字段记载MSI-X Capability结构的ID号，其值为0x11。在PCIe设备中，每一个Capability都有唯一的一个ID号。
* Next Pointer字段存放下一个Capability结构的地址。
* Message Control字段，该字段存放当前PCIe设备使用MSI-X机制进行中断请求的状态与控制信息，如表6‑2所示。

 表6‑2 MSI-X Capability结构的Message Control字段

| **Bits** | **定义** | **描述** |
| --- | --- | --- |
| 15 | MSI-X Enable | 该位可读写，是MSI-X中断机制的使能位，复位值为0，表示不使能MSI-X中断机制。该位为1且MSI Enable位为0时，当前PCIe设备使用MSI-X中断机制，此时INTx和MSI中断机制被禁止。当PCIe设备的MSI Enble和MSI-X Enable位为0时，将使用INTx中断消息报文发出/结束中断请求。 |
| 14 | Function Mask | 该位可读写，是中断请求的全局Mask位，复位值为0。如果该位为1，该设备所有的中断请求都将被屏蔽；如果该位为0，则由Per Vector Mask位，决定是否屏蔽相应的中断请求。Per Vector Mask位在MSI-X Table中定义，详见下文。 |
| 10：0 | Table Size | MSI-X中断机制使用MSI-X Table存放Message Address字段和Message Data字段。该字段用来存放MSI-X Table的大小，该字段对系统软件只读。 |

< >Table BIR(BAR Indicator Register)。该字段存放MSI-X Table所在的位置，PCIe总线规范规定MSI-X Table存放在设备的BAR空间中。该字段表示设备使用BAR0~5寄存器中的哪个空间存放MSI-X table。该字段由三位组成，其中0b000~0b101与BAR0~5空间一一对应。Table Offset字段。该字段存放MSI-X Table在相应BAR空间中的偏移。PBA(Pending Bit Array) BIR字段。该字段存放Pending Table在PCIe设备的哪个BAR空间中。在通常情况下，Pending Table和MSI-X Table存放在PCIe设备的同一个BAR空间中。PBA Offset字段。该字段存放Pending Table在相应BAR空间中的偏移。

#### 2 MSI-X Table

MSI-X Table的组成结构如图6‑3所示。

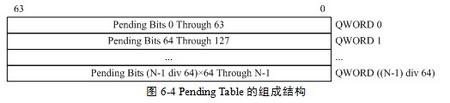
[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1376383939074.jpg)

由上图可见，MSI-X Table由多个Entry组成，其中每个Entry与一个中断请求对应。其中每一个Entry中有四个参数，其含义如下所示。

* Msg Addr。当MSI-X Enable位有效时，该字段存放MSI-X存储器写事务的目的地址的低32位。该双字的31:2字段有效，系统软件可读写；1:0字段复位时为0，PCIe设备可以根据需要将这个字段设为只读，或者可读写。不同的处理器填入该寄存器的数据并不相同。
* Msg Upper Addr，该字段可读写，存放MSI-X存储器写事务的目的地址的高32位。
* Msg Data，该字段可读写，存放MSI-X报文使用的数据。其定义与处理器系统使用的中断控制器和PCIe设备相关。
* Vector Control，该字段可读写。该字段只有第0位(即Per Vector Mask位)有效，其他位保留。当该位为1时，PCIe设备不能使用该Entry提交中断请求；为0时可以提交中断请求。该位在复位时为0。Per Vector Mask位的使用方法与MSI机制的Mask位类似。

#### 3 Pending Table

Pending Table的组成结构如图6‑4所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1376384030314.jpg)

如上图所示，在Pending Table中，一个Entry由64位组成，其中每一位与MSI-X Table中的一个Entry对应，即Pending Table中的每一个Entry与MSI-X Table的64个Entry对应。与MSI机制类似，Pending位需要与Per Vector Mask位配置使用。

当Per Vector Mask位为1时，PCIe设备不能立即发送MSI-X中断请求，而是将对应的Pending位置1；当系统软件将Per Vector Mask位清零时，PCIe设备需要提交MSI-X中断请求，同时将Pending位清零。

[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskj.html#_ftnref1) 此时PCI设备配置空间Command寄存器的“Interrupt Disable”位为1。

[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskj.html#_ftnref2) MSI机制提交中断请求的方式类似与边界触发方式，而使用边界触发方式时，处理器可能会丢失某些中断请求，因此在设备驱动程序的开发过程中，可能需要使用这两个字段。

|  |
| --- |
| **6.2 PowerPC处理器如何处理MSI中断请求** |
| 发布时间：2013-08-23 15:52:14 |
|  |

|  |  |
| --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) |  |
|  |

PowerPC处理器使用OpenPIC中断控制器或者MPIC中断控制器，处理外部中断请求。其中MPIC中断控制器基于OpenPIC中断控制器，但是作出了许多增强，目前Freescale新推出的PowerPC处理器，其中断控制器多与MPIC兼容。

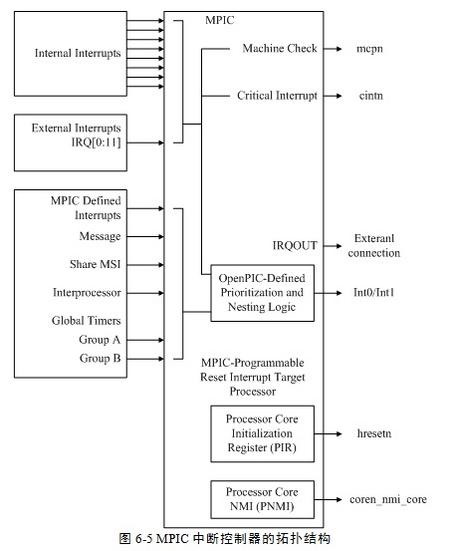
值得注意的是，PowerPC处理器和x86处理器处理MSI报文的方式有较大的不同。其中x86处理器使用的机制比PowerPC处理器更为合理，但是PowerPC处理器使用的方法使用的硬件资源相对较少。本节将MPC8572处理器为例说明MSI机制的处理过程，在第6.3节介绍x86处理器如何实现MSI机制。

MPIC中断控制器是Freescale的PowerPC处理器使用的通用中断控制器，目前基于E500内核的处理器，如MPC854x、8572等处理器使用这种中断控制器。目前Freescale使用QorIP架构，该架构使用的中断控制器与MPIC兼容。

使用MPIC中断控制器处理MSI中断时，PCIe设备的MSI报文，其目的地址为MPIC中断控制器的MSIIR寄存器。当该寄存器被PCIe设备写入后，MPIC中断控制器将向处理器内核提交中断请求，之后处理器再通过读取MPIC中断控制器的ACK寄存器获得中断向量号，并进行相应的中断处理。这种方式与x86处理器的FSB Interrupt Message机制相比，处理器需要读取ACK寄存器，从而中断处理的延时较大。

目前Freescale的P4080处理器对MPIC中断控制器进行了优化。在P4080处理器中，MPIC中断控制器向处理器提交中断请求的同时，也向处理器内核提交中断向量，处理器内核不必读取ACK寄存器获得中断向量，从而缩短了中断处理延时。使用这种方法的效率与x86处理器使用的FSB Interrupt Message机制相当。

目前Freescale并没有完全公开P4080处理器的实现细节，因此本节仍以MPC8572处理器为例介绍PCIe设备的MSI中断请求。在MPC8572处理器中，MPIC中断控制器的拓扑结构如图6‑5所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1377244313949.jpg)

由上图所示，MPIC中断控制器可以处理内部中断请求[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftn1)、外部中断请求，Message、处理器间中断请求和Share MSI中断请求等。而MPIC中断控制器使用Int0、Int1等中断线向处理器提交这些中断请求。其中Internal Interrupts和External Interrupts模块处理MPC8572内部和外部的中断请求，而Share MSI处理来自PCIe设备的MSI或者MSI-X中断请求。

当MPIC中断控制器收到MSI报文后，将使用中断线Int0、Int1或者cintn向处理器内核提交中断请求。处理器内核被中断后，将读取ACK寄存器获得中断向量，然后执行相应的中断服务例程。为此PowerPC处理器设置了一系列寄存器，如下文所示。

### 6.2.1 MSI中断机制使用的寄存器

PowerPC处理器设置了一系列寄存器，处理来自PCIe设备的MSI报文，其中最重要的寄存器是MSIIR寄存器。在PowerPC处理器系统中，PCIe设备Message Address寄存器中存放的值都为MSIIR寄存器的物理地址，而Message Data寄存器中存放的数据也与MSIIR寄存器相关。

在PowerPC处理器系统中，MSI机制的实现过程是PCIe设备向MSIIR寄存器写入指定的数据。MPIC中断控制器发现该寄存器被写入后，将向处理器提交中断请求。处理器收到这个中断请求后，将通过读取MPIC中断控制器的ACK寄存器确定中断向量，并依此确定中断源。为此PowerPC处理器还设置了其他寄存器实现MSI中断机制。

#### 1 MSIIR寄存器

在PowerPC处理器中，MSIIR(Shared Message Signaled Interrupt Index Register)寄存器是实现MSI机制的重要寄存器。

当PCIe设备对MSIIR寄存器进行写操作时，MPC8572处理器将使能MSIR0–MSIR7寄存器的相应位，从而向MPIC中断控制器提交中断请求，而中断控制器将转发这个中断请求，由处理器进一步处理。该寄存器各字段的详细描述如表6‑3所示。

 表6‑3 MSIIR寄存器

| **Bits** | **定义** | **描述** |
| --- | --- | --- |
| 27~31 | IBS | 该字段用来选择MSIR0~MSIR7寄存器的对应位。0b00000对应SH0；0b00001对应SH1；0b00010对应SH2；以此类推0b11111对应SH31； |
| 24~26 | SRS | 该字段用来选择MSIR0~MSIR7寄存器。0b000对应MSIR0；0b001对应MSIR1；0b010对应MSIR2；以此类推0b111对应MSIR7。 |
| 0~24 |  | 保留。 |

PCIe设备通过MSI机制，向此寄存器写入数据时，MSIR0~7寄存器的相应位SH0~31将有一位置1。例如PCIe设备向MSIIR寄存器写入0xFF00000时，MSIR7寄存器的SH31位将置1(SRS字段为0b111用来选择MSIR7，而IBS字段为0b11111用来选择SH31)。

#### 2 MSIR寄存器组

MSIR(Shared Message Signaled Interrupt Registers)寄存器组共由8个寄存器组成，分别为MSIR0~MSIR7。其中每一个MSIRx寄存器中有32个有效位，分别为SH0~31。当PCIe设备对MSIIR寄存器进行写操作时，某一个MSIIRx寄存器的某个SH位将被置为有效。系统软件通过读取该寄存器获得中断源，该寄存器读清除，对此寄存器进行写操作没有意义。

该寄存器组的大小决定了一个PowerPC处理器支持的MSI中断请求的个数。在MPC8572处理器中，有8个MSIRx寄存器，每个寄存器由32个有效位组成，因此MPC8572处理器最多能够处理256个MSI中断请求。该寄存器的结构如图6‑6所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957263754_TIME_1377244500665.jpg)

#### 3 MSISR寄存器

MSISR寄存器(Shared Message Signaled Interrupt Status Register)共由8个有效位组成，每一位对应一个MSIR寄存器。MPC8572处理器设置该寄存器的主要目的是方便系统软件定位究竟是哪个MSIR寄存器中存在有效的中断请求。首先系统软件通过MSISR寄存器判断是哪个MSIRx寄存器存在有效请求，之后再读取相应的MSIRx寄存器，该寄存器各字段的详细描述如表6‑4所示。

 表6‑4 MSISR寄存器

| **Bits** | **定义** | **描述** |
| --- | --- | --- |
| 0~23 |  | 保留。 |
| 24~31 | Sn | 该字段由8位组成，每一位与一个MSIR0~7寄存器对应。该位为0时表示在MSIRn寄存器中没有有效位，即没有中断请求；该位为1时表示MSIRn寄存器中至少有一个有效位，即存在中断请求。Sn位是MSIRn寄存器各个位的“与”，当MSIRn寄存器的相应位清除时，Sn也将被清除。 |

#### 4 MSIVPR寄存器组

MSIVPR(Shared Message Signaled Interrupt Vector/Priority Register)寄存器组由8个寄存器组成，分别为MSIVPR0~7寄存器。该组寄存器设置对应中断请求的优先级别和中断向量。其中每个MSIVPR寄存器对应一个MSIR寄存器，MSIVPR寄存器各字段的详细解释如表6‑5所示。

 表6‑5 MSIVPR寄存器

| **Bits** | **定义** | **描述** |
| --- | --- | --- |
| 0 | MSK | 该位为0，且MSIR寄存器的对应位为1时，则将向中断控制器提交中断请求；如果为1屏蔽该中断请求。 |
| 1 | A | 该位为0时，表示MPIC中断控制器没有处理该中断请求；该位为1时，表示MPIC中断控制器正在处理该中断请求，或者该中断控制器准备处理该中断请求，这个中断请求将在IPR(Interrupt Pending Regsiter)寄存器中排队等待处理，或者在ISR(Interrupt Service Register)寄存器中正在被处理。该位的详细描述见MPC8572的数据手册。 |
| 12~15 | PRIORITY | OpenPIC和MPIC中断控制器中为每一个中断请求设置了0~15，共16个优先级。其中1的优先权最低，15的优先权最高，0表示禁止中断请求。 |
| 16~31 | VECTOR | 该字段存放该中断的中断向量。当处理器读取IACK寄存器时，将获得对应中断请求的中断向量。 |

通过该组寄存器可以发现，在MPC8572处理器系统中，PCIe设备最多可以使用8个中断向量，并可以共享这些中断向量。

#### 5 MSIDR寄存器组

MSIDR(Shared Message Signaled Interrupt Destination Registers)寄存器组共由8个寄存器组成，分别为MSIDR0~7。其中每一个MSIDRn寄存器对应一个MSIR寄存器。

MPIC中断控制器支持Pass-through方式，在这种方式下，PowerPC处理器可以使用外部中断控制器处理中断请求(这种方法极少使用)，而不使用内部中断控制器。MPIC中断控制器可以使用cint#和int#信号提交中断请求，但是绝大多数系统软件都使用int#信号向处理器提交中断请求。

此外在MPC8572处理器中有两个CPU，分别为CPU0和CPU1，MSI机制提交的中断请求可以由CPU0或者CPU1处理。系统软件可以通过设置MSIDRn寄存器完成这些功能，该寄存器各字段的详细描述如表6‑6所示。

 表6‑6 MSIDRn寄存器

| **Bits** | **定义** | **描述** |
| --- | --- | --- |
| 0 | EP | 为1时，表示中断请求输出到IRQ\_OUT由外部中断控制器处理；为0时，表示由MPIC中断控制器处理。 |
| 1 | CI0 | 为1时，表示中断控制器使用cint#信号向CPU0提交中断请求。 |
| 2 | CI1 | 为1时，表示中断控制器使用cint#信号向CPU1提交中断请求。 |
| 30 | P1 | 为1时，表示中断控制器使用int#信号向CPU0提交中断请求。 |
| 31 | P0 | 为1时，表示中断控制器使用int#信号向CPU1提交中断请求。 |

### 6.2.2 系统软件如何初始化PCIe设备的MSI Capability结构

如果PCIe设备支持MSI机制，系统软件首先设置该设备MSI Capability结构的Message Address和Message Data字段。如果该PCIe设备支持64位地址空间，即MSI Capability寄存器的64 bit Address Capable位有效时，系统软件还需要设置Message Upper Address字段。系统软件完成这些设置后，将置MSI Cabalibities结构的MSI Enable位有效，使能该PCIe设备的MSI机制。

其中Message Address字段所填写的值是MSIIR寄存器在PCI总线域中的物理地址。在PowerPC处理器中，PCI总线域与存储器域地址空间独立，当PCIe设备访问存储器域的地址空间时，需要通过Inbound寄存器组将PCI总线域地址空间转换为存储器域地址空间。

在PowerPC处理器中，PCIe设备使用MSI机制访问MSIIR寄存器时，可以不使用Inbound寄存器组进行PCI总线地址到处理器地址的转换。在MPC8572处理器中，专门设置了一个PEXCSRBAR窗口[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftn2)，进行PCI总线域到存储器域的地址转换，使用这种方法可以节省Inbound寄存器窗口，Linux PowerPC使用了这种实现方式。

在MPC8572处理器中，MSIIR寄存器的基地址为CCSRBAR[[3]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftn3)(Configuration, Control, and Status Base Address Register)，其偏移为0x1740。为支持MSI中断机制，系统软件需要使用PEXCSRBAR窗口将MSIIR寄存器映射到PCI总线域地址空间，即将CCSRBAR寄存器空间映射到PCI总线域地址空间。之后PCIe设备就可以通过MSIIR寄存器在PCI总线域的地址访问MSIIR寄存器。

Linux PowerPC使用setup\_pci\_pcsrbar函数[[4]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftn4)设置PEXCSRBAR窗口，该函数的源代码在./arch/powerpc/sysdev/fsl\_pci.c文件中，如源代码6‑1所示，这段代码来自Linux 2.6.30.5。

 源代码6‑1 setup\_pci\_pcsrbar函数

|  |
| --- |
| static void \_\_init setup\_pci\_pcsrbar(struct pci\_controller \*hose)  {  #ifdef CONFIG\_PCI\_MSI      phys\_addr\_t immr\_base;        immr\_base = get\_immrbase();      early\_write\_config\_dword(hose, 0, 0, PCI\_BASE\_ADDRESS\_0, immr\_base);  #endif  } |

系统软件除了需要设置PCIe设备的Message Address字段和PEXCSRBAR窗口之外，还需要设置PCIe设备的Message Data字段。PCIe设备向MSIIR寄存器进行存储器写操作的数据存放在Message Data字段中。

系统软件在初始化Message Data字段之前，首先根据Multiple Message Capable字段预先存放的数据初始化Multiple Message Enable字段。一个PCIe设备最多可以申请32个中断请求，但是系统软件根据当前处理器系统的中断资源的使用情况，决定给这个PCIe设备提供多少个中断向量，并将这个结果存放到Multiple Message Enable字段。

MPC8572处理器最多可以为PCIe设备提供256个MSI中断请求。但是在某些极端的情况下，可能会出现PCIe设备需要的中断请求超过系统所能提供的中断请求。此时某些PCIe设备的Multiple Message Enable字段可能会小于Multiple Message Capable字段。

如果在PCIe设备中，使用了多个中断请求，那么Message Data字段存放的是一组中断向量号，而Message Data字段存放这组中断向量号的基地址。MSI机制要求“这组数据”连续，其范围在Message Data~Message Data+Multiple Message Enable-1之间。在多数情况下，MPC8572处理器系统仅为一个PCIe设备分配1个中断向量号。

由上所述，在MPC8572处理器系统中，PCIe设备使用存储器写TLP传送MSI中断报文，这个存储器写TLP使用的地址为PCIe设备Capability结构的Message Address字段，而数据为Message Data~Message Data＋Multiple Message Enable-1之间。其中Message Data字段与MSIIR寄存器要求的格式相同。

这个特殊的存储器写TLP报文通过若干Switch，并穿越RC后，最终将数据写入MSIIR寄存器中，并设置MSIIR寄存器的SRS和IBS字段，同时将使能MSIR0~MSIR7寄存器的相应位，从而向中断控制器提交中断请求(如果MSIVPR寄存器的MSK位为1)。MPIC中断控制器获得该中断请求后，向处理器系统转发这个中断请求，并由处理器系统执行相应的中断服务例程进行中断处理。MPC8572处理器也可以处理PCIe设备的MSI-X中断机制，本节对此不做进一步介绍。

[[1]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftnref1) PowerPC处理器中含有许多模块，如千兆以太网、ATM等，这些模块包含在芯片内部，由这些内部模块发起的中断请求，被称为内部中断请求。

[[2]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftnref2) 该窗口的大小为1MB，其基地址由PEXCSRBAR寄存器确定。

[[3]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftnref3) 在Linux PowerPC中使用immr\_base变量保存该寄存器。IMMR寄存器是PQ2处理器使用的寄存器，该寄存器在PQ3之后的处理器中升级为CCSRBAR。

[[4]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskk.html#_ftnref4) 该函数来自Linux 2.6.30.5内核。

|  |
| --- |
| **6.3 x86处理器如何处理MSI-X中断请求** |
| 发布时间：2013-08-23 16:01:49 |
|  |

|  |  |  |
| --- | --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) | |  |
|  |

PCIe设备发出MSI-X中断请求的方法与发出MSI中断请求的方法类似，都是向Message Address所在的地址写Message Data字段包含的数据。只是MSI-X中断机制为了支持更多的中断请求，在MSI-X Capablity结构中存放了一个指向一组Message Address和 Message Data字段的指针，从而一个PCIe设备可以支持的MSI-X中断请求数目大于32个，而且并不要求中断向量号连续。MSI-X机制使用的这组Message Address和 Message Data字段存放在PCIe设备的BAR空间中，而不是在PCIe设备的配置空间中，从而可以由用户决定使用MSI-X中断请求的数目。

当系统软件初始化PCIe设备时，如果该PCIe设备使用MSI-X机制传递中断请求，需要对MSI-X Capability结构指向的Message Address和Message Data字段进行设置，并使能MSI-X Enable位。x86处理器在此处的实现与PowerPC处理器有较大的不同。

### 6.3.1 Message Address字段和Message Data字段的格式

在x86处理器系统中，PCIe设备也是通过向Message Address写入Message Data指定的数值实现MSI/MSI-X机制。在x86处理器系统中，PCIe设备使用的Message Adress字段和Message Data字段与PowerPC处理器不同。

#### 1 PCIe设备使用Message Adress字段

在x86处理器系统中，PCIe设备使用的Message Address字段仍然保存PCI总线域的地址，其格式如图6‑7所示。

[3.jpg](http://download.bbs.ednchina.com/images/attachments/201308/original/8957264434_TIME_1377246948692.jpg)

其中第31~20位，存放FSB Interrupts存储器空间的基地址，其值为0xFEE。当PCIe设备对0xFEEX-XXXX这段“PCI总线域”的地址空间进行写操作时，MCH/ICH将会首先进行“PCI总线域”到“存储器域”的地址转换，之后将这个写操作翻译为FSB总线的Interrupt Message总线事务，从而向CPU内核提交中断请求。

x86处理器使用FSB Interrupt Message总线事务转发MSI/MSI-X中断请求。使用这种方法的优点是向CPU内核提交中断请求的同时，提交PCIe设备使用的中断向量，从而CPU不需要使用中断响应周期从寄存器中获得中断向量。FSB Interrupt Message总线事务的详细说明见下文。

Message Address字段其他位的含义如下所示。

* Destination ID字段保存目标CPU的ID号，目标CPU的ID与该字段相等时，目标CPU将接收这个Interrupt Message。FSB Interrupt Message总线事务可以向不同的CPU提交中断请求。
* RH(Redirection Hint Indication)位为0时，表示Interrupt Message将直接发向与Destination ID字段相同的目标CPU；如果RH为1时，将使能中断转发功能。
* DM(Destination Mode)位表示在传递优先权最低的中断请求时，Destination ID字段是否被翻译为Logical或者Physical APIC ID。在x86处理器中APIC ID有三种模式，分别为Physical、Logical和Cluster ID模式。
* 如果RH位为1且DM位为0时，Destination ID字段使用Physical模式；如果RH位为1且DM位为1，Destination ID字段使用Logical模式；如果RH位为0，DM位将被忽略。

以上这些字段的描述与x86处理器使用的APIC中断控制器相关。对APIC的详细说明超出了本书的范围，对此部分感兴趣的读者请参阅Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1。

#### 2 Message Data字段

Message Data字段的格式如图6‑8所示。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957264434_TIME_1377246972089.jpg)

Trigger Mode字段为0b0x时，PCIe设备使用边沿触发方式申请中断；为0b10时使用低电平触发方式；为0b11时使用高电平触发方式。MSI/MSI-X中断请求使用边沿触发方式，但是FSB Interrupt Message总线事务还支持Legacy INTx中断请求方式，因此在Message Data字段中仍然支持电平触发方式。但是对于PCIe设备而言，该字段为0b0x。

Vector字段表示这个中断请求使用的中断向量。FSB Interrupt Message总线事务在提交中断请求的同时，将中断向量也通知给处理器。因此使用FSB Interrupt Message总线事务时，处理器不需要使用中断响应周期通过读取中断控制器获得中断向量号。与PowerPC的传统方式相比，x86处理器的这种中断请求的效率较高[[①]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftn1)。

值得注意的是，在x86处理器中，MSI机制使用的Message Data字段与MSI-X机制相同。但是当一个PCIe设备支持多个MSI中断请求时，其Message Data字段必须是连续的，因而其使用的Vector字段也必须是连续的，这也是在x86处理器系统中，PCIe设备支持多个MSI中断请求的问题所在，而使用MSI-X机制有效避免了该问题。

Delivery Mode字段表示如何处理来自PCIe设备的中断请求。

* 该字段为0b000时，表示使用“Fixed Mode”方式。此时这个中断请求将被Destination ID字段指定的CPU处理。
* 该字段为0b001时，表示使用“Lowest Priority”方式。此时这个中断请求将被优先权最低的CPU处理。当使用“Fixed Mode”和“Lowest Priority”方式时，如果Vector字段有效，CPU接收到这个中断请求之后，将使用Vector字段指定的中断向量处理这些中断请求；而当Delivery Mode字段为其他值时，Message Data字段中所包含的Vector字段无效。
* 该字段为0b010时，表示使用SMI方式传递中断请求，而且必须使用边沿触发，此时Vector字段必须为0。这个中断请求将被Destination ID字段指定的CPU处理。
* 该字段为0b100时，表示使用NMI方式传递中断请求，而且必须使用边沿触发，此时Vector字段和Trigger字段的内容将被忽略。这个中断请求将被Destination ID字段指定的CPU处理。
* 该字段为0b101时，表示使用INIT方式传递中断请求，Vector字段和Trigger字段的内容将被忽略。这个中断请求将被Destination ID字段指定的CPU处理。
* 该字段为0b111时，表示使用INTR信号传递中断请求且使用边沿触发。此时MSI中断信息首先传递给中断控制器，然后中断控制器在通过INTR信号向CPU传递中断请求，之后CPU在通过中断响应周期获得中断向量。上文中PowerPC处理器使用的方法与此方法类似。而在x86处理器中多使用Interrupt Message总线事务进行MSI中断信息的传递，因此这种模式很少被使用。

边沿触发和电平触发是中断请求常用的两种方式。其中电平触发指外部设备使用逻辑电平1(高电平触发)或者0(低电平触发)，提交中断请求。使用电平或者边沿方式提交中断请求时，外部设备一般通过中断线(IRQ\_PIN#)与中断控制器相连，其中多个外部设备可能通过相同的中断线与中断控制器相连(线与或者与门)。

外部设备在使用低电平触发，提交中断请求的过程中，首先需要将IRQ\_PIN#信号驱动为低。当中断控制器将该中断请求提交给处理器，而且处理器将这个中断请求处理完毕后，处理器将通过写外部设备的某个寄存器来清除此中断源，此时外部设备将不再驱动IRQ\_PIN#信号线，从而结束整个中断请求。

IRQ\_PIN#信号线可以被多个外部设备共享，在这种情况之下，只有所有外部设备都不驱动IRQ\_PIN#信号线时，IRQ\_PIN#信号才为高电平。采用电平触发方式进行中断请求的优点是不会丢失中断请求，而缺点是一个优先权较高的中断请求有可能会长期占用中断资源，从而使其他优先权较低的中断不能被及时提交。因为优先级别较高的中断源有可能会持续不断地驱动IRQ\_PIN#信号。

而边沿触发使用上升沿(0到1)或者下降沿(1到0)作为触发条件，但是中断控制器并不是使用这个“边沿”作为触发条件。中断控制器使用内部时钟对IRQ\_PIN#信号进行采样，如果在前一个时钟周期，IRQ\_PIN#信号为0，而后一个时钟周期，IRQ\_PIN#信号为1，中断控制器认为外部设备提交了一个有效“上升沿”，中断控制器会锁定这个“上升沿”并向处理器发出中断请求。这也是外部设备至少需要将IRQ\_PIN#信号保持一个时钟采样周期的原因，否则中断控制器可能无法识别本次边沿触发的中断请求，从而产生Spurious中断请求。

外部设备使用“上升沿”进行中断申请时，不需要持续地将IRQ\_PIN#信号驱动为1，而只需要保证中断控制器可以进行正确采样这些中断信号即可。在处理边沿触发中断请求时，处理器不需要清除中断源。

使用边沿触发可以有效避免“优先级别”较高的中断源长期占用IRQ\_PIN#信号的情况，使用“下降沿”触发进行中断请求与“上升沿”触发类似。

但是外部设备使用边沿触发方式时，有可能会丢失一些中断请求。例如在一个处理器系统中，存在一个定时器，这个定时器使用上升沿触发方式向中断控制器定时提交中断。如果当处理器正在处理这个定时器的上一个中断请求时，将不会处理这个定时器发出的其他“边沿”中断请求，从而导致中断丢失。而使用电平触发方式不会出现这类问题，因为电平触发方式是一个“持续”过程，处理器只有处理完毕当前中断，并清除相应中断源之后，才会处理下一个中断源。

MSI中断请求实际上和边沿触发方式非常类似，MSI中断请求通过存储器写TLP实现，这个写动作是一个瞬间的动作，并不是一个持续请求，因此在x86处理器中MSI中断请求使用边沿触发进行中断请求。

还有一些外部设备可以通过I/O APIC进行中断请求[[②]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftn2)，这些I/O APIC接收的外部中断需要标明是使用边沿或者电平触发，I/O APIC使用FSB Interrupt Message总线事务将中断请求发向Local APIC，并由Local APIC向处理器提交中断请求。

### 6.3.2 FSB Interrupt Message总线事务

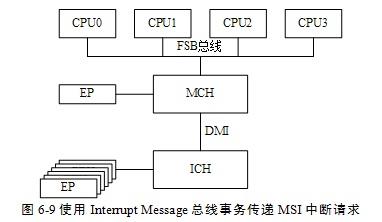
与MPC8572处理器处理MSI中断请求不同，x86处理器使用FSB的Interrupt Message总线事务，处理PCIe设备的MSI/MSI-X中断请求。由上文所示，MPC8572处理器处理MSI中断请求时，首先由MPIC中断控制器截获这个MSI中断请求，之后由MPIC中断控制器向CPU提交中断请求，而CPU通过中断响应周期从MPIC中断控制器的ACK寄存器中获得中断向量。

采用这种方式的主要问题是，当一个处理器中存在多个CPU时，这些CPU都需要通过中断响应周期从MPIC中断控制器的ACK寄存器中获得中断向量。在一个中断较为密集的应用中，ACK寄存器很可能会成为系统瓶颈。而采用Interrupt Message总线事务可以有效地避免这种系统瓶颈，因为使用这种方式中断信息和中断向量将同时到达指定的CPU，而不需要使用中断响应周期获得中断向量。

x86处理器也具有通过中断控制器提交MSI/MSI-X中断请求的方法，在I/O APIC具有一个 “The IRQ Pin Assertion Register”寄存器，该寄存器地址为0xFEC00020[[③]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftn3)，其第4~0位存放IRQ Number。系统软件可以将PCIe设备的Message Address寄存器设置为0xFEC00020，将Meaasge Data寄存器设置为相应的IRQ Number。

当PCIe设备需要提交MSI中断请求时，将向PCI总线域的0xFEC00020地址写入Message Data寄存器中的数据。此时这个存储器写请求将数据写入I/O APIC的The IRQ Pin Assertion Register中，并由I/O APIC将这个MSI中断请求最终发向Local APIC，之后再由Local APIC通过INTR#信号向CPU提交中断请求。

上述步骤与MPC8572处理器传递MSI中断的方法类似。在x86处理器中，这种方式基本上已被弃用。下文以图6‑9为例，说明x86处理器如何使用FSB总线的Interrupt Message总线事务，向CPU提交MSI/MSI-X中断请求。

[](http://download.bbs.ednchina.com/images/attachments/201308/original/8957264434_TIME_1377247064549.jpg)

PCIe设备在发送MSI/MSI-X中断请求之前，系统软件需要合理设置PCIe设备MSI/MSI-X Capability寄存器，使Message Address寄存器的值为0xFEExx00y[[④]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftn4)，同时合理地设置Message Data寄存器Vector字段。

PCIe设备提交MSI/MSI-X中断请求时，需要向0xFEExx00y地址写Message Data寄存器中包含的数据，并以存储器写TLP的形式发送到RC。如果ICH收到这个存储器写TLP时，将通过DMI接口将这个TLP提交到MCH。MCH收到这个TLP后，发现这个TLP的目的地址在FSB Interrupts存储器空间中，则将PCIe总线的存储器写请求转换为Interrupt Message总线事务，并在FSB总线上广播。

FSB总线上的CPU，根据APIC ID信息，选择是否接收这个Interrupt Message总线事务，并进入中断状态，之后该CPU将直接从这个总线事务中获得中断向量号，执行相应的中断服务例程，而不需要从APIC中断控制器获得中断向量。与PowerPC处理器的MPIC中断控制器相比，这种方法更具优势。

## 6.4 小结

本章详细描述了MSI/MSI-X中断机制的原理，并以PowerPC和x86两个处理器系统为例说明这两种中断机制实现机制。本章因为篇幅有限，并没有详细讲述这两个处理器使用的中断控制器。而理解这些中断控制器的实现机制是进一步理解MSI/MSI-X中断机制的要点。对此部分有兴趣的读者可以继续阅读MPIC中断控制器和APIC中断控制器的实现机制，以加深对MSI/MSI-X中断机制的理解。

设备的中断处理是局部总线的设计难点和重要组成部分，而中断处理的效率直接决定了局部总线的数据传送效率。在一个处理器系统的设计与实现中，中断处理的优化贯彻始终。

[[①]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftnref1) P4080处理器也提供了一种类似于FSB Interrupt Message总线事务的中断请求方法。

[[②]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftnref2) 与I/O APIC的IRQX#引脚链接的外部设备。

[[③]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftnref3) 该寄存器在存储器域和PCI总线域中的地址都为0xFEC00020。

[[④]](http://blog.sina.com.cn/s/blog_6472c4cc0102dskl.html#_ftnref4) 其中xx表示APIC ID，而y为RH+DM。

|  |
| --- |
| **结束语--浅谈PCIe体系结构** |
| 发布时间：2013-08-23 16:50:14 |
|  |

|  |  |  |
| --- | --- | --- |
| 技术类别：[接口电路](http://bbs.ednchina.com/BLOG_LIST_10054.HTM)     个人分类：[浅谈PCIe体系结构](http://bbs.ednchina.com/BLOG_PERSONALCAT_100363_2002055.HTM) | |  |
|  |

“浅谈PCIe体系结构”的更新到此告一段落。这些内容主要出自之前书写的《PCI Express体系结构导读》，因为与出版社的协议，无法在此处共享全文，但是也包含了与PCI与PCIe总线相关的最基础的内容。原书正在组织第二次印刷，我却很难有再版的想法。事实上如果我能决定一些事情，不会出现第二次印刷。

书中的错误令我不安，却鲜有读者指出。我意识到产生这些现象的原因是更多的入门者在阅读这些内容。与国外的知名教授十年磨一剑去书写书籍，剩下的除了反思还是反思。这些反思使我在今后相当长的时间内不会再次以纸质图书方式出版任何技术书籍。

也许很长时间 后，我可能会重新关注Local Bus。相对于SoC平台总线，PCIe总线并不复杂；相对于系统总线，SoC平台总线的理解也并不困难。如果考虑处理器系统中Cache与Cache的 互联总线，其下的所有总线几乎都是玩具。在有些细分的领域并不会再有书籍出现，只能看到持续挑战着自身极限的奋斗者的身影。

 始 于近代，华夏民族一次又一次与众多变革擦肩而过。十万万同胞，并非不够聪慧，并非不够勤劳。想起任公说过的，“造成今日之老大中国者，则中国老朽之冤业 也；制出将来之少年中国者，则中国少年之责任也。故今日之责任，不在他人，而全在我少年。少年智则国智，少年富则国富，少年强则国强，少年独立则国独立， 少年自由则国自由，少年进步则国进步，少年胜于欧洲，则国胜于欧洲，少年雄于地球，则国雄于地球”。

这些懵懂少年们本没有太多分辨能力，在年轻时接收的有些错误，他们需要用一辈子的时间偿还。最糟糕的情况还并不是出现在计算机领域。每念及此，泪如雨下。救救他们。这是一件虽千万难，也必须要做的事情。这是一个很严肃的话题。

经常回想少年时的读书时光，精力多在课外，聆听教诲时蒙头便睡，待到结业时，彻夜强记，试后所有知识还与老师。 怜我天朝，误人实多！曾记得那个年代，信息只能来自老师或者图书馆那些陈旧的不知对错的书籍。在那个年代，几乎每一个人都有出书的热情，莘莘学子认真研读的多是某些教授自编自演的很多体系结构入门书籍。回想起诺贝尔得主的费曼讲义，UT Austin的Y.N. Patt在认真地给本科生上课，心重如山。

有时不得不反思一个大学究竟怎样算是成功，是顶级文章的发布数量，重大的科研成果。可能这些都不是，大学出产的并不是老师，而是学生。毕业的学生在世界的影响力也许更为重要。有些问题是我等无法解决的，有些我们可以做到。因为每一个人都是群体，社会，国家，世界的组成部分。各自独立的个体组成的合力如浩荡江河，必会有所改变。上善若水。居善地，心善渊。