Chapter 2 PCIe Architecture Overview //PCIe体系结构概述

关于前一章

前一章为我们提供了 PCI 技术发展的历史，以此来建立更好地理解 PCIe 的基础。主要回顾了 PCI 与 PCI-X 1.0/2.0 的基础内容，目的是为了给接下来对 PCIe 的概述内容提供一些前因后果，以方便理解 PCIe。

关于本章

本章对 PCI Express 体系结构进行了全面的介绍，旨在将本章作为一个“执行层”概述，涵盖该体系结构在高层的所有基础知识。它介绍了 PCIe 协议规范中给出的分层的方法，并描述了每个层级的职责作用。介绍了各种数据包类型的同时也一起介绍了用于数据包通信和增强数据包传输可靠性的协议。

关于下一章

下一章节介绍了 PCI Express 环境中的配置部分。它包括用于实现功能的配置寄存器的空间，一个功能如何在总线上被发现，配置事务是如何被生成和路由到正确的位置，PCI 兼容空间与 PCIe 扩展空间之间的差异，以及软件是如何区分端点和桥。

2.1 PCI Express 简介 (Introduction to PCI express)

PCI Express 的出现代表了其前身并行总线的重大转变。作为一种串行总线，它与早期的串行设计（例如 InfiniBand 或者 Fibre Channel）有许多的共同点，但是它完全保持了在软件层面对 PCI 的后向兼容。

正如许多高速串行传输方法一样，PCIe 使用双向连接的方式，可以在同一时间进行信息的收发操作。这种模型被称为双单工连接，因为每个接口都有一个单工发送路径和一个单工接收路径，图2-1 展示了这种模型。因为数据流可以同时进行双向传输，因此在技术层面上来说两个设备间的通信其实是全双工的，但是 PCIe 协议规范依然使用双单工这个术语，这是因为这种称呼对实际通信信道也进行了一点描述。

Figure 2-1: Dual-Simplex Link

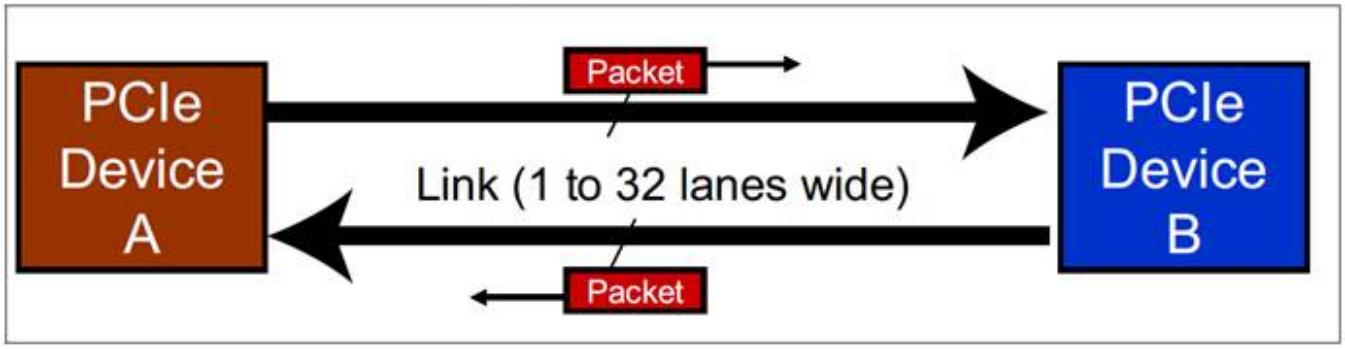

图2-1 双单工链路

用于描述设备之间信号传输路径的术语为“链路（Link）”，它由一个或以上的接收发送对组成。这样的一对接收和发送被称为一个“通道（Lane）”，协议规范允许一条链路内有 1、2、4、8、12、16 或 32 个通道。链路内通道的数量称为链路宽度，通常用 x1、x2、x4、x8、x16 以及 x32 来进行表示。用于权衡在实际设计中使用多少通道的思路其实很简单：使用更多的通道可以增加带宽，但是也会增加成本、增加空间占用以及增加功耗。更多关于这方面的信息，可以阅读“链路与通道”这一节。

Figure 2-2: One Lane

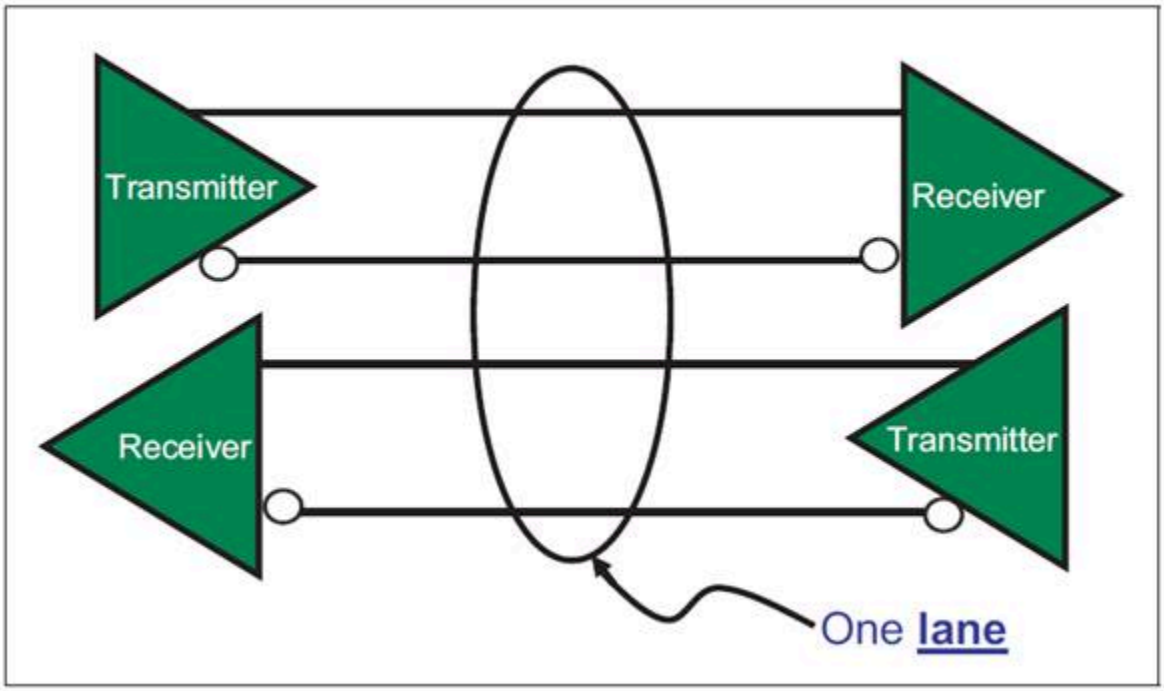

图2-2 一个通道

2.1.1 软件的后向兼容性 (Software Backward Compatibility)

PCIe 设计目标中极为重要的一点就是要保持对 PCI 软件的后向兼容性。如果一个设计使用的是现有的系统，而且这个设计已经能够在这个现有系统中正常工作，那么想要促使它转向使用另一种系统则需要满足两件事：第一，新技术需要一些改变但是有足够吸引人的性能提升，第二，尽可能减小技术迁移的成本、风险以及工作量。对于第二点来说，在计算机中通常的做法是让那些为旧模型所编写的软件在新模型中依然可用。为了在 PCIe 中做到这一点，PCI 中用到的所有的地址空间要么不做更改直接照搬，要么仅仅进行了了简单的扩展。内存、IO 和配置空间对于软件来说依然是可见的，而且连写入方式都和以前一样。因此就算是数年前为 PCI 而写的软件（BIOS 代码、设备驱动等等）将依然可以在现在的 PCIe 设备上使用。配置空间已经被大大的扩展，添加了许多新的寄存器来支持新的功能，但是老的寄存器依然存在且可以按照常规方式来访问他们（这部分的详细内容见“软件兼容性特性”）。

2.1.2 串行传输 (Serial Transport)

2.1.2.1 对传输速率的需求 (The need for Speed)

很明显，串行传输模型必须要比并行的设计跑的快很多才能达到相同的带宽，这是因为串行传输一次只发送 1 比特数据。然而，事实证明这并不困难，过去的 PCIe 在 2.5GT/s和 5.0GT/s 都能稳定的工作。之所以 PCIe 可以达到这样甚至更高的 8GT/s 的速率，是因为串行传输模型克服了并行模型的不足。

克服问题. 通过上一章对 PCI 历史的回顾，我们知道，并行总线的性能被一些问题所限制，图 2-3 展示了其中的三个问题。首先，回想一下，并行总线使用公共时钟；信号在一个时钟沿被输出，然后在下一个时钟沿被接收方接收。这个模型的第一个问题来自于信号从发送端传输到接收端所花费的时间，称为渡越时间。渡越时间必须小于一个时钟周期，否则将会出问题，这使得难以通过继续减小时钟周期来提升速度。因为若需要继续减小时钟周期，为了让信号渡越时间依然小于时钟周期，需要更短的布线并减少负载的设备数量，但是最终这样的做法都会到达极限并且越来越不现实。第二个造成并行模型性能受限的因素是使用公共时钟时，时钟到达发送方和接收方的时刻不一致，这称为时钟偏斜。电路板设计人员尽力去减小时钟偏斜的值，因为时钟偏斜将会降低信号传输时序预算，但是这种偏斜永远无法彻底消除。第三个因素是信号偏斜，它指的是多比特位宽数据的各个位到达接收端的时刻存在差异。显然，这样的多位宽数据在所有的比特都到达且稳定之前都不能被接收方采样，这使得我们必须去等待最慢的那一比特。

Figure 2-3: Parallel Bus Limitations

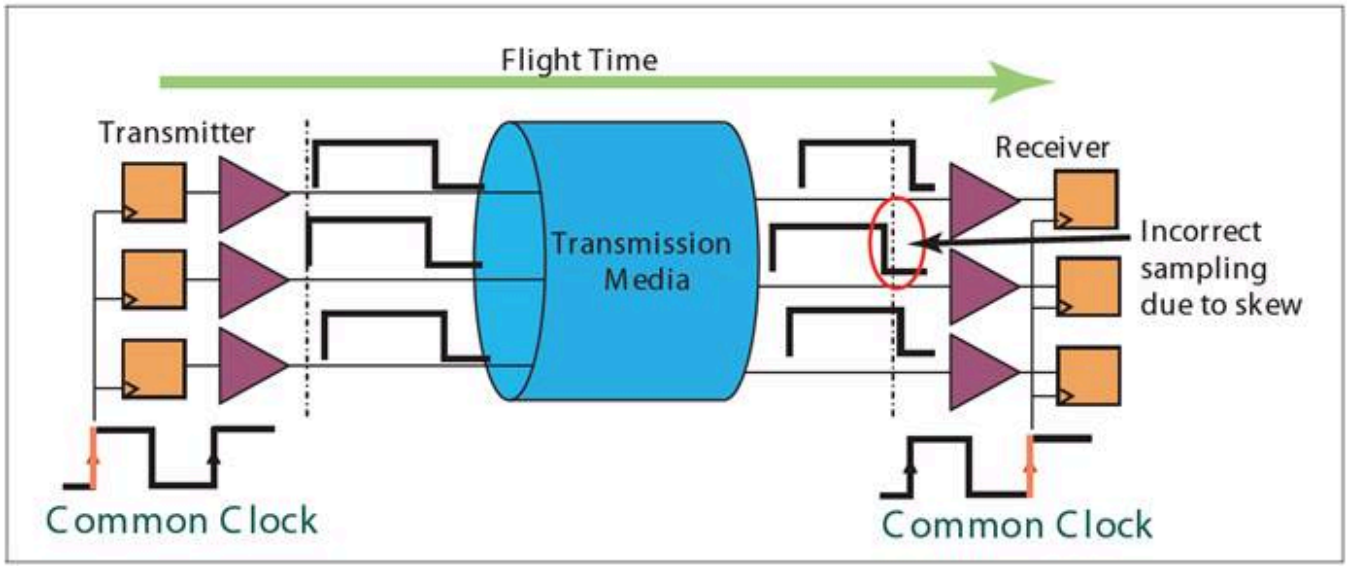

图2-3 并行总线的局限

PCIe 这样的串行传输方法是如何处理这些问题的呢？首先，信号渡越时间将不再是一个问题，因为用于指示接收端锁存数据的时钟现在已经被内置入数据流中，不在需要额外传输线来传输参考时钟。因此，无论再小的时钟周期，或是再长的信号传输时间都不会产生以前的问题了，这是因为内置在数据流中的时钟必然能与数据一起到达接收方。同样地，时钟偏斜的问题将不再存在，这还是因为时钟被内置入数据流中，接收方通过恢复出数据流中的时钟来进行数据采样，自然不存在时钟偏斜的问题。最后，信号偏斜的问题在一个通道内被消除了，因为一个通道一次只传输 1 比特数据。在多通道设计中，虽然信号偏斜的问题再次出现了，但是接收端将自动对其进行纠正，它可以很大程度上补偿偏斜。虽然串行设计克服了并行模型的许多问题，但是串行设计自身也有一系列的复杂问题。不过，我们稍后将看到，对这些问题的解决方法是易于控制的，并可以让我们做到高速的、可靠的通信。

带宽. PCIe 支持的高速，多通道的链路带来了傲人的带宽数值，如表 2-1 所示。这些数字的计算都来源于比特率与总线特性。其中一种特性与其他许多串行传输方法类似，那就是 PCIe 的前两代版本中使用了被称为 8b/10b 的编码过程，这种编码过程会根据 8 比特的输入而生成 10 比特 的输出。尽管这样做会引起一些开销，但是我们将会在后面讲到有几个很好的理由支持我们这样做。对于现在来说，只需要知道对于 8b/10b 编码来说，要发送 1 字节的数据实际需要传送 10 比特。

第一代 PCIe（称为 Gen1 或者 PCIe 协议规范版本 1.x）中，比特率为 2.5GT/s，将它除以 10 即可得知一个通道的速率将可以达到 0.25GB/s。因为链路可以在同一时刻进行发送和接收，因此聚合带宽可以达到这个数值的两倍，即每个通道达到 0.5GB/s。第二代 PCIe（称为 Gen2 或

者 PCIe 2.x）中将总线频率翻倍，这也使得它的带宽相较于 Gen1 翻倍。

第三代 PCIe（称为 Gen3 或者 PCIe 3.0）再次让带宽翻倍，但是这次协议的制定者们并没有选择将频率翻倍。相反，出于一些原因（我们稍后将会进行讨论），他们仅将频率提升到 8GT/s（Gen2中是 5GT/s，未翻倍），并不再采用 8b/10b 的编码方式，而是采用了另一种编码机制，即 128b/130b 编码（关于这个内容的更多信息，可以阅读“物理层-逻辑（Gen 3）”章节）。表 2-1 列出了当前几代 PCIe 的各种通道数量情况下的带宽，展示出了链路对应情况下的峰值吞吐量。

Table 2-1: PCIe Aggregate Gen1, Gen2 and Gen3 Bandwidth for Various Link Widths

Link Width	x1	x2	x4	x8	x12	x16	x32
Gen1 Bandwidth (GB /s)	0.5	1	2	4	6	8	16
Gen2 Bandwidth (GB/s)	1	2	4	8	12	16	32
Gen3 Bandwidth (GB/s)	2	4	8	16	24	32	64

表 2-1 PCIe Gen1,Gen2,Gen3 的各种链路宽度下的带宽汇总

2.1.2.2 PCIe 带宽计算方法 (PCIe Bandwidth Calculation)

要计算上述表格中的 PCIe 带宽大小，可以参照如下的计算方法。

- Gen1 PCIe 带宽 =（2.5Gb/s x 2 directions）/ 10bits per symbol = 0.5GB/s
- Gen2 PCIe 带宽 =（5.0Gb/s x 2 directions）/ 10bits per symbol = 1.0GB/s

需要注意，上述计算中，我们是除以 10bits 而不是 8bits，这是因为 Gen1 和 Gen2 的协议中要求将字节进行 8b/10b 编码后进行数据包的传输，因此原数据中的 1 字节在实际传输时其实是需要传输 10 比特。

- Gen3 PCIe 带宽 =（8.0Gb/s x 2 directions）/ 8bits per byte = 2.0GB/s

注意到在 Gen3 速率的计算中，我们除以的是 8bits 而不再是 10bits 了，这是因为 Gen3 中不再使用 8b/10b 编码方式，而是 128b/130b 编码方式。这种编码方式每 128位 引入 2 比特开销，这个开销非常小以至于我们暂且可以将它在我们的计算中忽略。

上述三种方法计算出来的带宽只需要再乘以链路宽度即可得到整个多通道链路的链路带宽。

2.1.2.3 PCIe 的差分信号 (Differential Signals)

每个通道都使用差分信号进行传输，差分信号是指每次传输一个信号时同时发送它的正信号和负信号（D+ 和 D-，这两种信号振幅相同相位相反），如图2-4 所示。当然，这样会将引脚增加一倍，但是相对于单端信号而言，差分信号在高速传输上的两个明显的优点足以抵消其引脚数方面的不足：它提高了噪声容限，并降低了信号电压。

差分信号的接收端将会接收这一对相位相反的信号，用正信号的电压减去其反相信号的电压，得到它们的差值，以此来判定这个比特的逻辑电平值。差分传输设计内置了抗噪声干扰的设计，因为它要求成对的差分信号必须位于每个设备的相邻的引脚上，它们的走线也必须彼此非常靠近，以保持合适的传输线阻抗。因此，任何因素在影响差分对中的一个信号的时候，都会同等程度且同样方向地影响到另一个信号。但是接收端所在意的是它们的差值，而这些噪声干扰并不会改变这个差值，所以带来的结果就是大多数情况下噪声对信号的影响并不会引起接收端对比特值的错误判别。

Figure 2-4: Differential Signaling

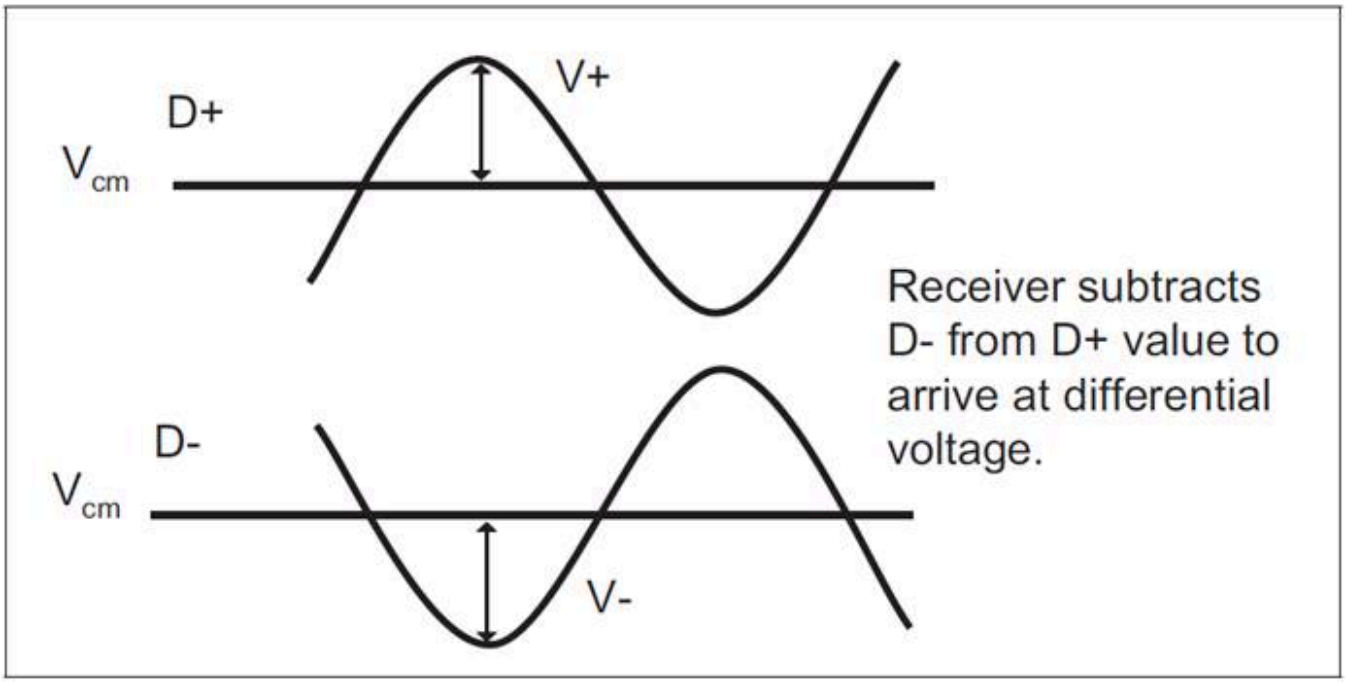

图2-4 差分信号示意图

2.1.2.4 不再使用公共时钟 (No Common Clock)

在先前的内容中有提到，PCIe 链路不再像 PCI 一样使用公共时钟，它使用了一个源同步模型，这意味着需要由发送端给接收端提供一个时钟来用于对输入数据进行锁存采样。对于 PCIe 链路来说，并不包括输出时钟信号。相反地，发送端会将时钟通过 8b/10b 编码来嵌入数据流中，

然后接收端将会从数据流中恢复出这个时钟，并用于对输入数据进行锁存。这一过程听起来可能非常神秘，但是其实很简单。在接收端中，PLL 电路（Phase-Locked Loop 锁相环，如图 2-5）将输入的比特流作为参考时钟，并将其时序或者相位与一个输出时钟相比较，这个输出时钟是 PLL 按照指定频率产生的时钟。也就是说 PLL 自身会产生一个指定频率的输出时钟，然后用比特流作为的参考时钟与自身产生的输出时钟相比较。基于比较的结果，PLL 将会升高或者降低输出时钟的频率，直到所比较的双方达到匹配。此时则可以称 PLL 已锁定，且输出时钟（恢复时钟）的频率已经精确地与发送数据的时钟相匹配。PLL 将会不断地调整恢复时钟，快速补偿修正由温度、电压因素对发送端时钟频率造成的影响。

关于时钟恢复，有一件需要注意的事情，PLL 需要输入端的信号跳变来完成相位比较。如果很长一段时间数据都没有任何跳变，那么 PLL 的恢复时钟可能会偏离正确的时钟频率。为了避免这种问题，8b/10b 编码中的设计目标之一就是要确保比特流中连续的 1 或者 0 的数量不能超过 5 个（想获得更多关于这部分的内容，请参考“8b/10b 编码”一节）

Figure 2-5: Simple PLL Block Diagram

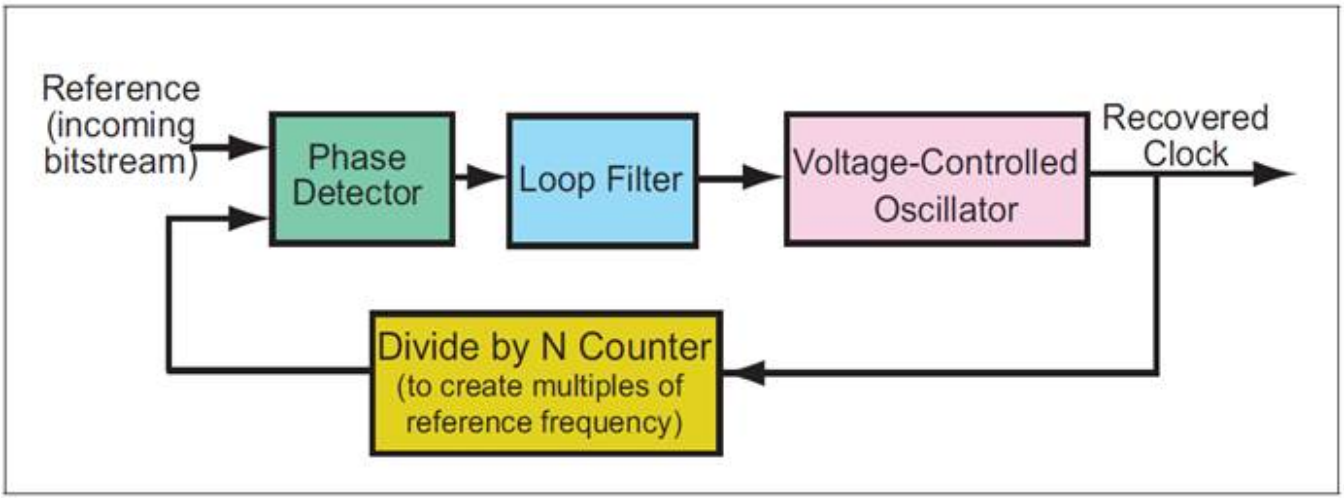

图2-5 简单的锁相环图示

一旦时钟被恢复出来，就可以使用它来锁存输入数据流的比特，并将锁存到的结果给到解串器。有时候学生们可能想知道这个恢复时钟能否作为接收端的所有逻辑所使用的工作时钟，但是这个问题的答案是“不行”。一个原因是，接收端不能指望用于恢复出时钟的比特流参考时钟一直存在且活跃，因为当链路的低功耗状态就包括了停止数据传输，此时必然也就无法继续恢复时钟。因此，接收端必须要有自己本地生成的内部时钟。

2.1.2.5 基于数据包的协议体系 (Packet-based Protocol)

将并行传输转变为串行传输可以极大的减少数据传输需要的引脚数量。如其他大多数串行传输协议一样，PCIe 通过消除了绝大部分原来并行总线中常用的边带控制信号来减少了引脚数量。然

而，如果没有控制信号来指示被接收的信息的类型，接收端如何知道输入的比特是什么信息呢？因此，在 PCIe 中，所有的事务在发送时都使用已经定义好的结构，称为数据包（packets）。接收端需要找到数据包的边界，并且知道这个数据包的被定义的结构是什么样子的（即数据包的模板），然后解析数据包结构来获知它需要执行什么操作。

关于数据包协议体系的详细信息将在“TLP 元素”这一章进行全面讲解，但在本章也可以找到对各种包格式的介绍以及他们各自用途的概述，请见“数据链路层”一节。

2.1.3 链路和通道 (Links and Lanes)

如先前的内容所提到，两个 PCIe 设备之间的物理连接被称为一条链路，这个链路由许多通道所构成。如图2-2，每个通道都含有一对发送差分信号对和一对接收差分信号对。这样的一个通道已经足够用于设备之间的通信，不需要其他额外的信号。

2.1.3.1 可扩展的性能 (Scalable Performance)

尽管一个通道就可以满足两个设备之间通信的需求，但是使用更多通道可以提升链路的传输性能，这个性能取决于链路的传输速率以及链路宽度。例如，使用多通道链路可以增加每个时钟传输的比特数量，因此这样就可以提升带宽。如表 2-1 所提到，PCIe 协议规范支持一条链路中含有的通道数目为 2 的幂，最多 32 通道。除了 2 的幂以外，x12 链路也是支持的，这可能是为了支持 InfiniBand 的 x12 链路，它是一种较早的串行设计。PCIe 这种允许多种链路宽度的特性，使得平台设计者可以在成本和性能之间做出适当的权衡，根据链路中通道的数量轻松地进行增减。

2.1.3.2 灵活的拓扑结构选择 (Flexible Topology Options)

一条 PCIe 链路必须是一个点对点的连接，而不是像 PCI 一样的共享总线，这是因为 PCIe 使用的链路速率非常高。由于一条链路只能连接两个接口，因此需要一种扩展连接的方法来构建一个不琐碎的系统，这里不琐碎的意思是指不过于细碎和冗杂，例如若直接对所有设备都采用直接的两两相连，那么会使得整个系统十分冗杂琐碎。这种需求在 PCIe 中通过交换机和桥接来实现，这两者可以灵活的构建系统拓扑——系统中元素之间的连接集。对系统中一个元素的定义以及一些拓扑结构的例子将在下面的小节中给出。

2.1.4 关于 PCIe 拓扑的一些定义 (Some Definitions)

如图2-6 展示了一个简单的 PCIe 拓扑示例，它将有助于我们对本节的一些定义内容进行讲解。

Figure 2-6: Example PCIe Topology

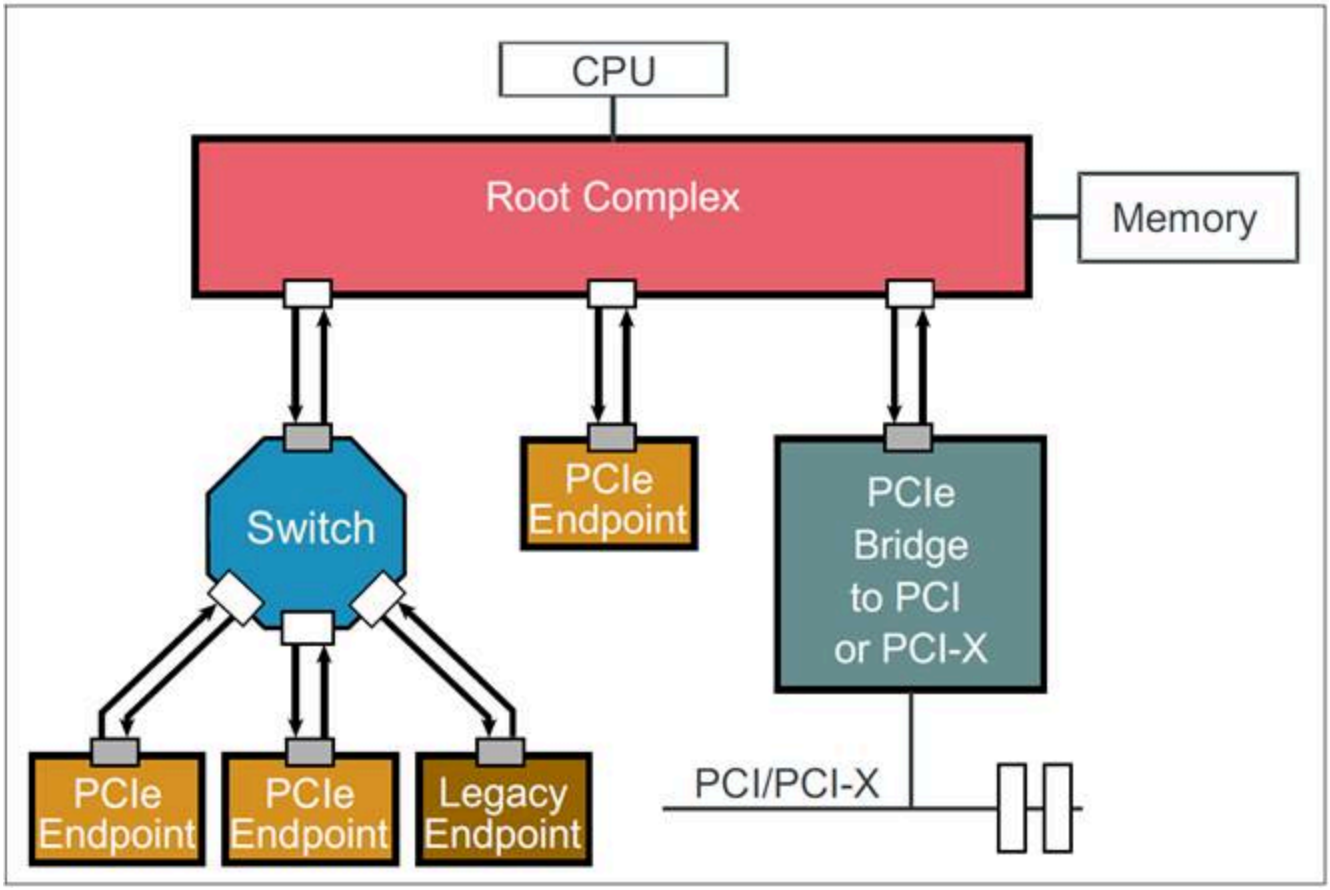

图2-6 PCIe 拓扑示例

2.1.4.1 拓扑特征 (Topology Characteristics)

在图的最上方是一个 CPU 。这里需要指出，CPU 被认为是 PCIe 层次结构的顶端。就像 PCI 一样，PCIe 只允许简单的树结构，这意味着不允许出现循环或者其他复杂的拓扑结构。这样做的原因是为了保持与 PCI 软件的向后兼容性，因为 PCI 软件使用简单的配置方法来记录拓扑结构，它并不支持复杂的环境。

为了保持这种向后兼容性，软件必须能够以与从前一样方式产生配置周期，并且总线拓扑结构也必须与以前一样。因此，软件希望找到的所有配置寄存器依然存在，而且它们的行为方式等也依然与从前相同。我们将稍晚些在回过头来讨论这一块的内容，因为我们要先定义一些术语概念。

2.1.4.2 根组件 (Root Complex)

CPU 与 PCIe 总线之间的接口可能包含一系列的组件（处理器接口，DRAM 接口等等），甚至是包含多个芯片。将这些组件合起来，称这一组组件为根组件（Root Complex, RC, Root）。RC 存在于 PCI 树状拓扑的“根部”，并代表 CPU 与系统的其余部分通信。但是，PCIe 协议规

范并没有很仔细的对 RC 进行定义，而是给出了一个 RC 必需功能与可选功能的列表。从广义上说，根组件可以被理解为系统 CPU 与 PCIe 拓扑之间的接口，这个 PCIe 端口，即 RC，在配置空间中被标记为“根端口”。

2.1.4.3 交换机与桥 (Switches and Bridges)

交换机提供了扇出以及聚合能力，使得单个 PCIe 端口上可以连接更多的设备。它扮演数据包路由器的角色，可以根据所给数据包的地址或者其他路由信息来识别这个数据包要走哪条路径。

桥提供了一个通往其他总线的接口，例如 PCI 或者 PCI-X，甚至也可以是其他的 PCIe 总线。图 2-6 中展示的桥有时被称为“前向桥”，它使得一个旧的 PCI 或者 PCI-X 板卡可以插入一个新系统中（PCIe 系统）。与前向桥相反类型的桥称为“反向桥”，它使得新的PCIe 板卡可以插入到旧的 PCI 系统中。

2.1.4.4 原生与传统端点 (Native PCIe Endpoints and Legacy PCIe Endpoints)

端点是 PCIe 拓扑中的既不是交换机也不是桥的设备，它们可以作为总线上事务的发起者也可以作为事务的完成者。端点存在于树状拓扑的分支的底部，仅实现一个上行端口（Upstream Port，与 RC 的连接方向），这里的上行指的是拓扑结构的向上，表示端点已经是树的分支的端点，不再产生下级分支。相比之下，一个交换机可以有几个下行端口（Downstream Port）。

用于老式总线（例如 PCI-X）的设备，如今拥有了可供它们使用的 PCIe 接口，这种 PCIe 接口将在配置寄存器中将自身标识为“传统 PCIe 端点”。它们使用了在 PCIe 设计中被禁止的东西，例如 IO 空间、支持 IO 事务以及支持锁定请求。与之相反，“原生 PCIe 端点”是从一开始就被设计用来在 PCIe 系统中使用的，这区别于在旧的 PCI 设备上添加 PCIe 接口。原生 PCIe 端点设备是内存映射设备。

2.1.4.5 软件兼容特性 (Software Compatibility Characteristics)

一种保持软件兼容性的方法是保持端点和桥的配置首部（Header）与 PCI 一致，如图2-7 所示。

Figure 2-7: Configuration Headers

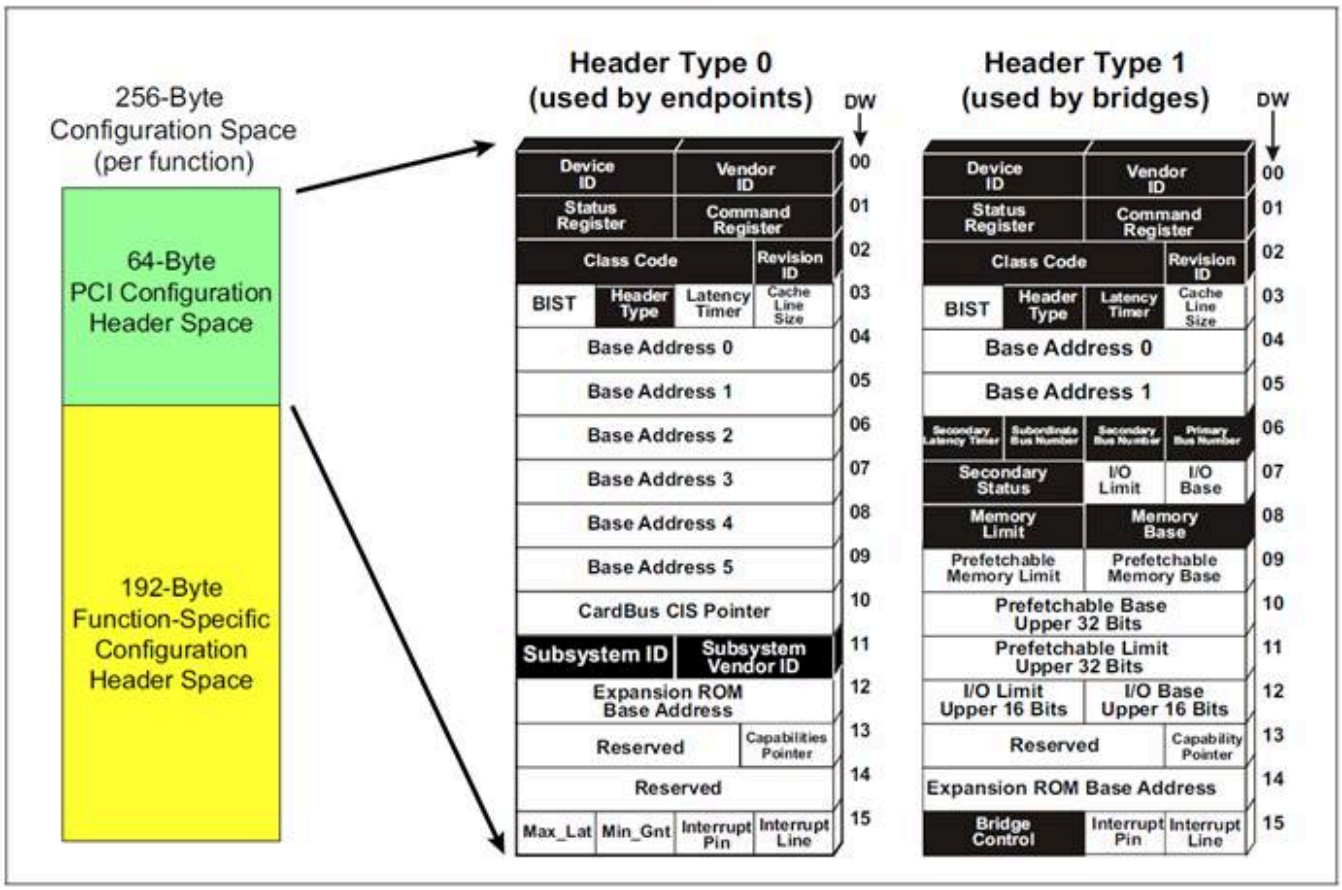

图2-7 配置首部

有一点不同，现在 PCIe 中的桥经常会被整合进交换机和根组件，但是遗留的 PCI 软件无需关注这种区别，它只把这些内含桥的设备简单的当做桥。在这里我们先对一些概念进行熟悉，所以我们不会在这里讨论这些寄存器的细节。配置是一个相当大的主题，关于它的具体介绍将在“配置综述”这一节进行。

为了举例说明 PCIe 系统在软件中的展现形式，我们可以参考图2-8 中所展示的拓扑结构的例子。像之前所说的一样，RC 位于整个层次结构的顶端。RC 自身的内部可以非常复杂，但是它通常会实现一个内部总线结构以及一些桥接结构，以便将拓扑扇出到几个端口。RC 的内部总线将被配置软件视作 PCI 总线 0，且 RC 上这几个 PCIe 端口将被视作是 PCI-to-PCI 桥。这种内部结构其实并不是一个实际的 PCI 总线，但是它在软件中看起来就是这种结构。因为这个总线是在 RC 内部的，所以它实际的逻辑设计并不需要遵循任何的标准，这里是可以由供应商来进行自定义的。

Figure 2-8: Topology Example

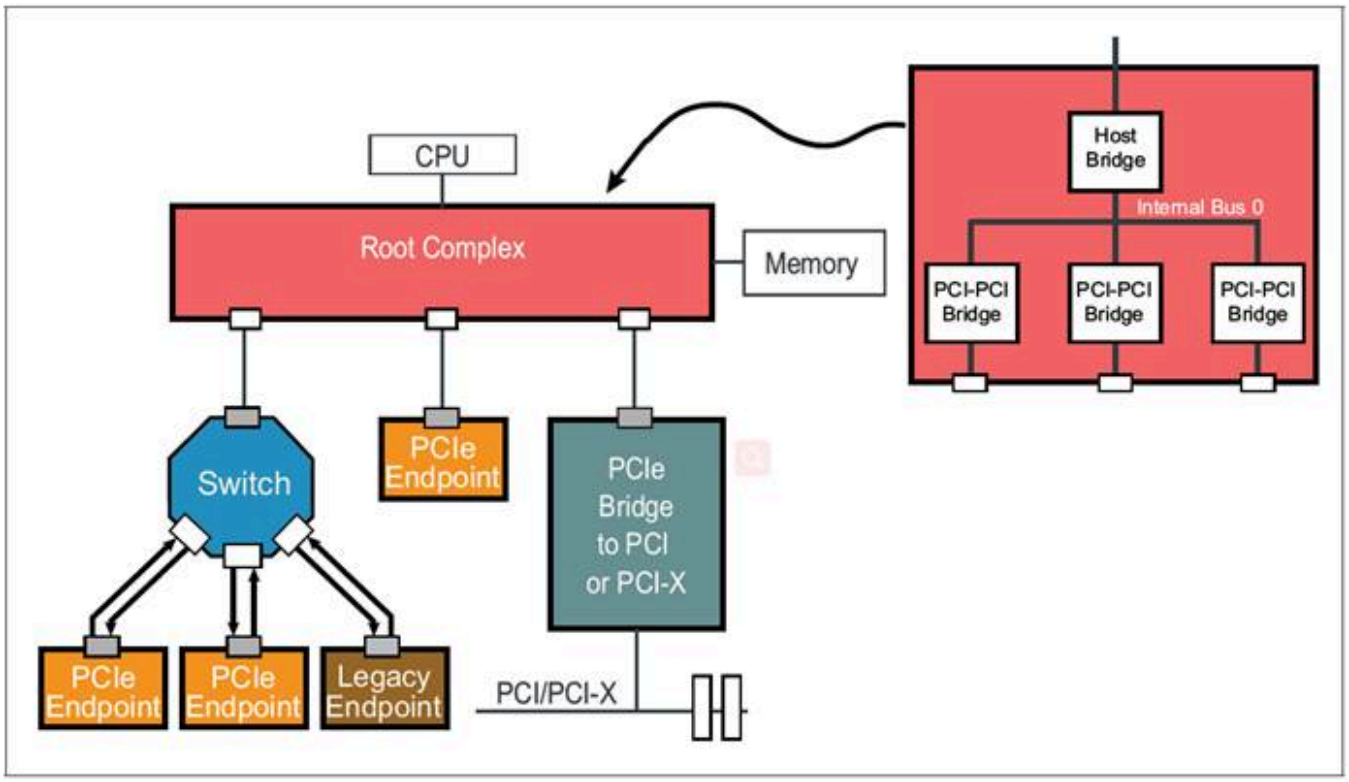

图2-8 拓扑结构示例

类似的，PCIe 交换机的内部结构对于软件来说，就是简单地几个桥共享一条公共总线，如图2-9所示。这样做的主要优点就是使得事务的路由方法能够与 PCI 一致。

枚举（Enumeration）是配置软件用来发现系统拓扑结构，并分配总线号和系统资源的过程，它的工作方式也与 PCI 中相同。在稍后的内容中我们将通过一些例子来讲述枚举是如何工作的。一旦枚举完成，系统中的总线号就将按照图2-9 所示的方式进行分配。

Figure 2-9: Example Results of System Enumeration

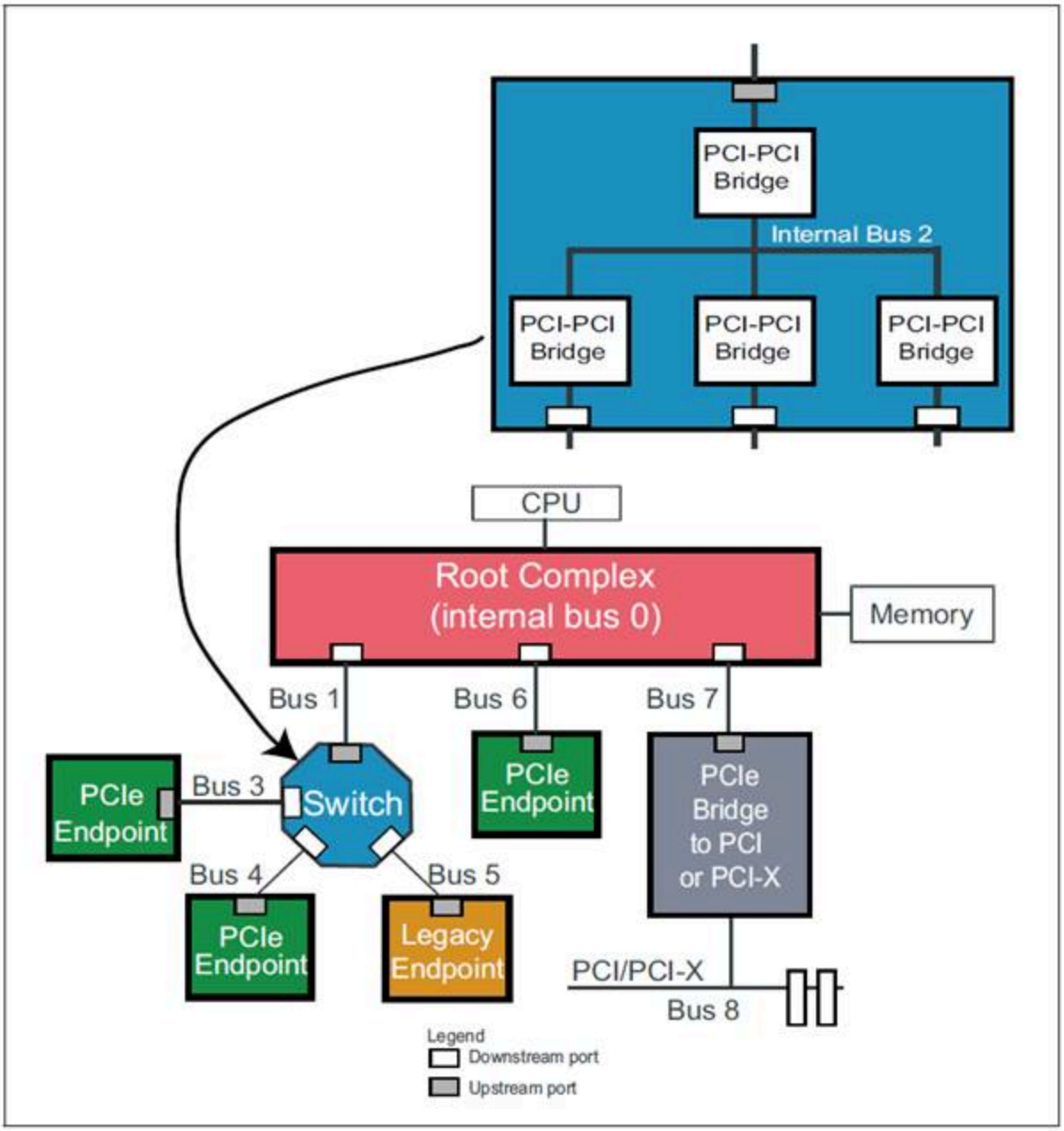

图2-9 系统枚举结果的示例

2.1.4.6 系统示例 (System Examples)

如图2-10 中举例说明了一个基于 PCIe 的系统，它被设计用于一些低成本应用比如消费级台式计算机。有一些 PCIe 端口实现时附带了一些板卡插槽，但是基本的框架与老式的 PCI 系统并没有太大区别。

相比之下，如图2-11 所示的高端的服务器系统展示了一些内置在系统的用于连接其他网络的接

口。在早期 PCIe 中，有人曾考虑将其作为一个网络来运行，用来取代那些旧的模型。毕竟，如果 PCIe 在大体上是其他网络协议的简化版本，或许它也能满足所有需求。由于各种各样的原因，这一概念从来没有真正得到大力发展，基于 PCIe 的系统通常仍然需要使用其他的协议来连接到外部网络。

这也给了我们一个机会来重新审视一个问题，RC 是由什么组成的。在这个例子中，被标识为“英特尔处理器”的方块包含了许多组件，大多数现代的 CPU 架构都是如此。这个处理器包含了一个用于访问图形设备（例如显卡）的 x16 PCIe 端口，以及两条 DRAM 通道，这两条 DRAM 通道意味着内存控制器以及一些路由逻辑已经被集成到 CPU 封装中。总的来说，这些资源通常被称为“非核心”资源，这样的称呼用于将他们与 CPU 封装中的几个 CPU 核心区分开来。此前，我们描述过 RC 是 CPU 与 PCIe 拓扑相连接的接口，这意味着在 CPU 封装中必须含有 RC 的一部分。正如图2-11 中虚线所框出的，RC 由多个组件的一部分共同组成。这种 RC 的组成方式可能将会是未来很长一段时间内的系统的设计方式。

Figure 2-10: Low-Cost PCIe System

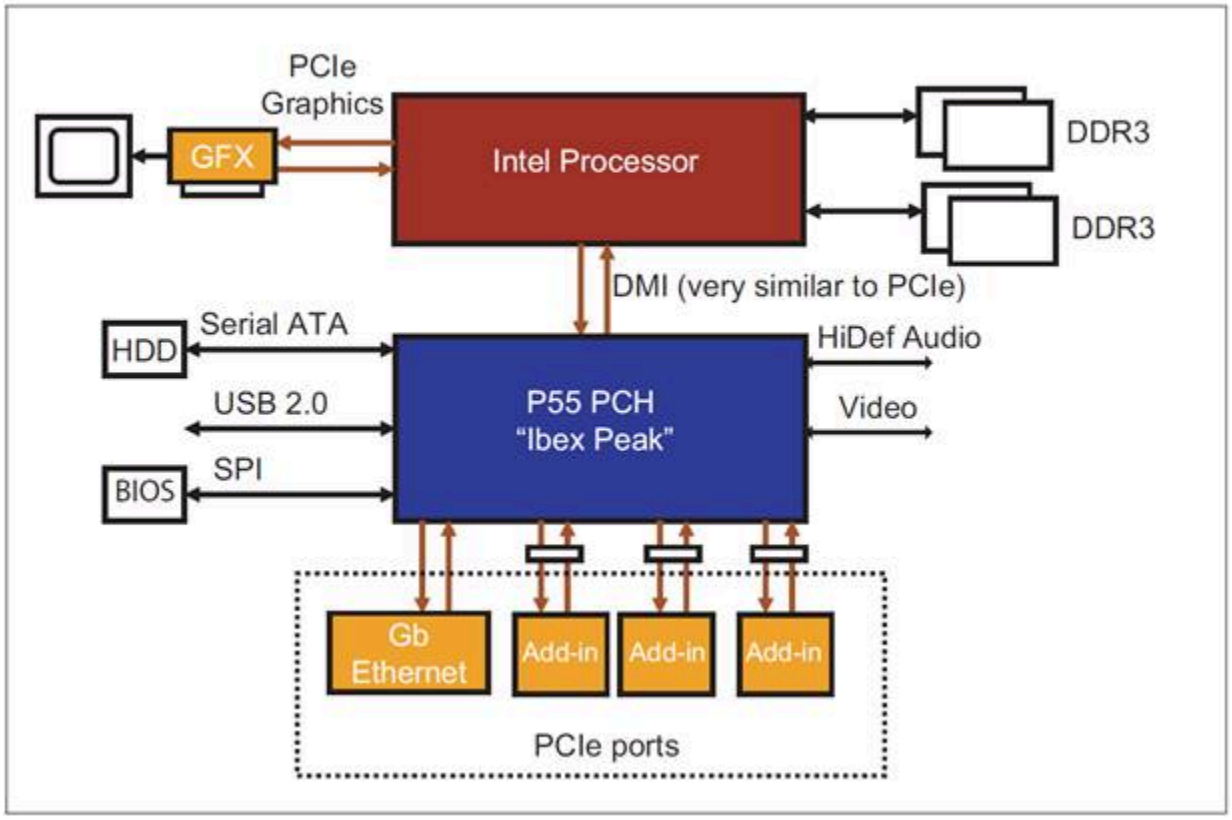

图2-10 低成本的 PCIe 系统

Figure 2-11: Server PCIe System

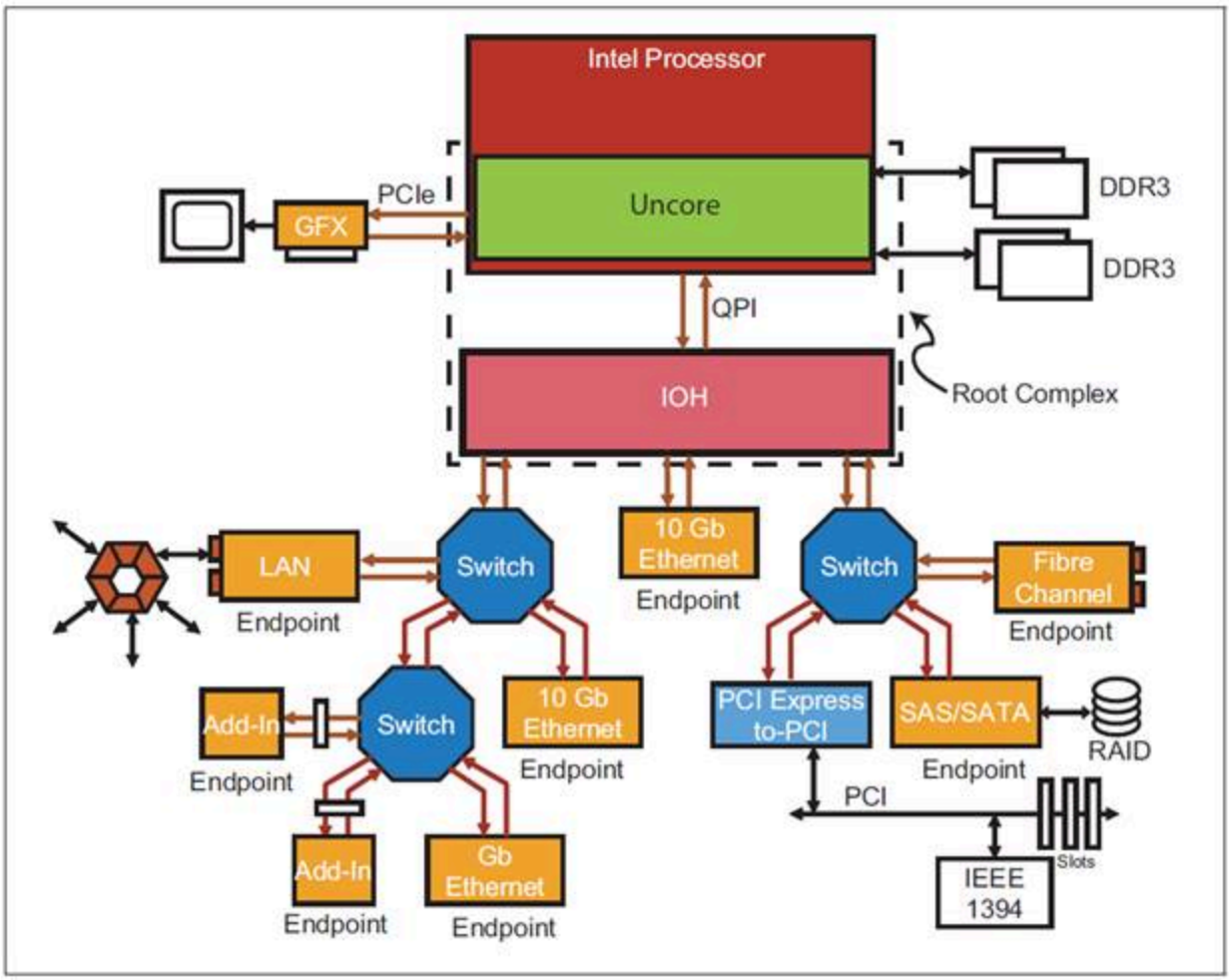

图2-11 服务器 PCIe 系统

2.2 设备层次介绍 (Introduction to Device Layers)

如图2-12 所示，PCIe 定义了一种分层的体系结构。可以认为这些层在逻辑上是相互独立的部分，因为他们各自都有一个用于发出信息流的发送端和一个接收信息流的接收端。这种分层的设计方法对硬件设计者来说有不少的优点，因为如果在设计中对逻辑进行了仔细的划分，那么就可以在以后升级到新的协议规范版本时仅改变原设计中的某一层即可，而不会影响或者变动其它层。虽然如此，但是需要注意的是，这些层只是定义了接口的工作职责，而实际设计并不要求为了符合规范而将设计严格按照层级来分成几个部分。本节的目的在于描述各个层的功能职责，以及描述完成一次数据传输时的事件流程。

Figure 2-12: PCI Express Device Layers

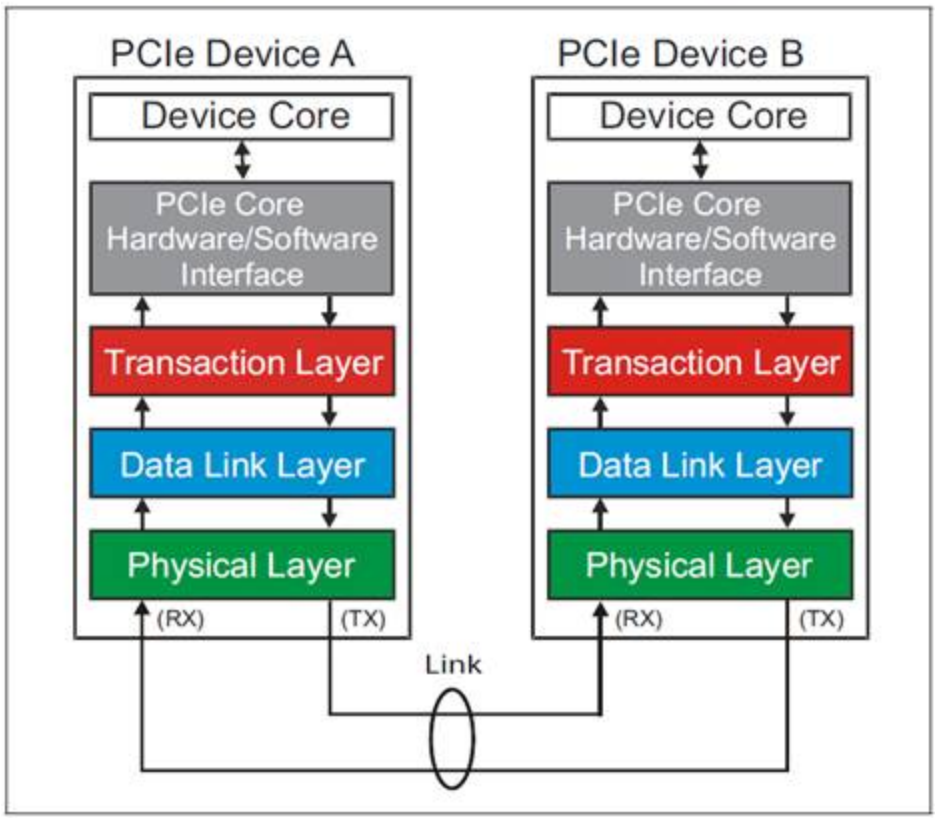

图2-12 PCI Express 设备层次示意图

图2-12 所展示的 PCIe 设备内部层次包括：

- 设备核心层以及它与事务层的接口。设备核心层实现设备的主要功能。如果设备是一个端点，那么它最多可以包含 8 个功能（function），每个功能实现自己的配置空间。如果设备是一个交换机，那么它的核心由数据包路由逻辑和为了实现路由的内部总线构成。如果设备是一个 RC，那么其核心会实现一个虚拟的 PCI 总线 0，在这个虚拟的 PCI 总线 0中存在着所有的芯片组嵌入式端点以及虚拟桥。
- 事务层。事务层负责在发送端产生 TLP（Transaction Layer Packet，事务层包），在接收端对 TLP 进行译码。这一层也负责 QoS（Quality of Service，服务质量）、流量控制以及事务排序。所有的这四个事务层的功能将在本书的第二部分进行讲解。
- 数据链路层。数据链路层负责在发送端产生 DLLP（Data Link Layer Packet，数据链路层包），在接收端对 DLLP 进行译码。这一层也负责链路错误检测以及修正，这个数据链路层功能被称为 Ack/Nak 协议。这两个数据链路层功能会在本书的第三部分进行讲解。
- 物理层。物理层负责在发送端产生字符序列包，在接收端对字符序列包进行译码。

这一层将处理上述三种类型的包（TLP、DLLP、字符序列包）在物理链路上的发送与接收。数据包在发送端要经过字节条带化逻辑、扰码器、8b/10b 编码器（对于 Gen1/Gen2）或是 128b/130b 编码器（对于 Gen3）以及数据包并串转换模块的处理。最终数据包以训练后的链路速率在所有通道上按照时钟以差分形式输出。在物理层的接收端，数据包处理包括串行地接收差分形式的比特信号，将其转换为数字信号形式，然后将输入比特流做串并转换。这个操作基于来源于 CDR（Clock and Data Recovery，时钟数据恢复）电路所提供的恢复时钟。接收下来的数据包要经过弹性缓存、8b/10b 解码器（对于 Gen1/Gen2）或者 128b/130b 解码器（对于 Gen3）、解扰器以及字节交换恢复逻辑。最终，物理层的 LTSSM（Link Training and Status State Machine，链路训练状态机）负责进行链路初始化以及训练。所有这些物理层功能将在本书的第四部分进行讲解。

每个 PCIe 接口都支持这些层的功能，包括交换机端口，如图2-13 所示。在早期大家经常会产生一个疑问，那就是一个交换机端口是否还需要实现所有的层次呢，毕竟它通常只用于转发数据包。答案是需要，因为对包的内容进行解析来确定它们的路由是需要查看数据包的内部细节的，而这件事情是在事务层中完成的。

Figure 2-13: Switch Port Layers

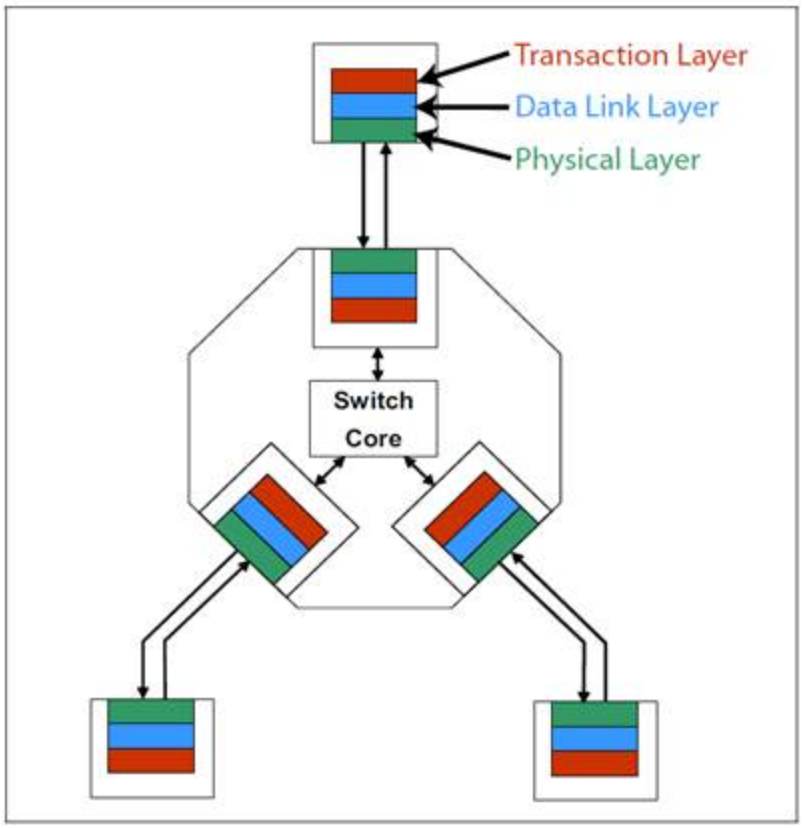

图2-13 交换机端口的层次结构

原则上，本设备上的每个层都与链路对端设备的各自对应层进行通信，例如本设备的事务层与对端的事务层通信、本设备的数据链路层与对端的数据链路层通信。上面的两层通过在数据包内添加一串特定的比特信息，产生一个接收端的对应层可以识别的字段样式来实现对应层之间的通信。数据包通过其他层的转发，从而做到到达或者离开链路。物理层也直接与另一个设备中的物理层通信，但是方式与此不同。

在我们讲的更深入之前，先大致了解一下这些层是如何交互的。从广义上说，设备所发出的请求包或者完成包是在事务层进行组包的，组包所用到的信息是由设备核心层所提供的，有时我们将设备核心层称为软件层（尽管协议规范中并没有使用这一术语）。它提供的组包信息通常包括期望的命令类型、目标设备的地址、请求的属性特征等等。刚组好的数据包会被存入一个被称为虚拟通道缓存，直到这个包可以被发往下一个层级。当这个包被向下发给数据链路层后，数据链路层会在数据包中加入额外的信息以供对端的接收方进行错误检查，而且这个数据包会在本地储存下来，这样我们就可以在对端检测到传输出错时重新发送这个数据包。当数据包到达物理层后，它被编码，并使用链路上的所有可用的通道、以差分信号的形式进行传输。

Figure 2-14: Detailed Block Diagram of PCI Express Device's Layers

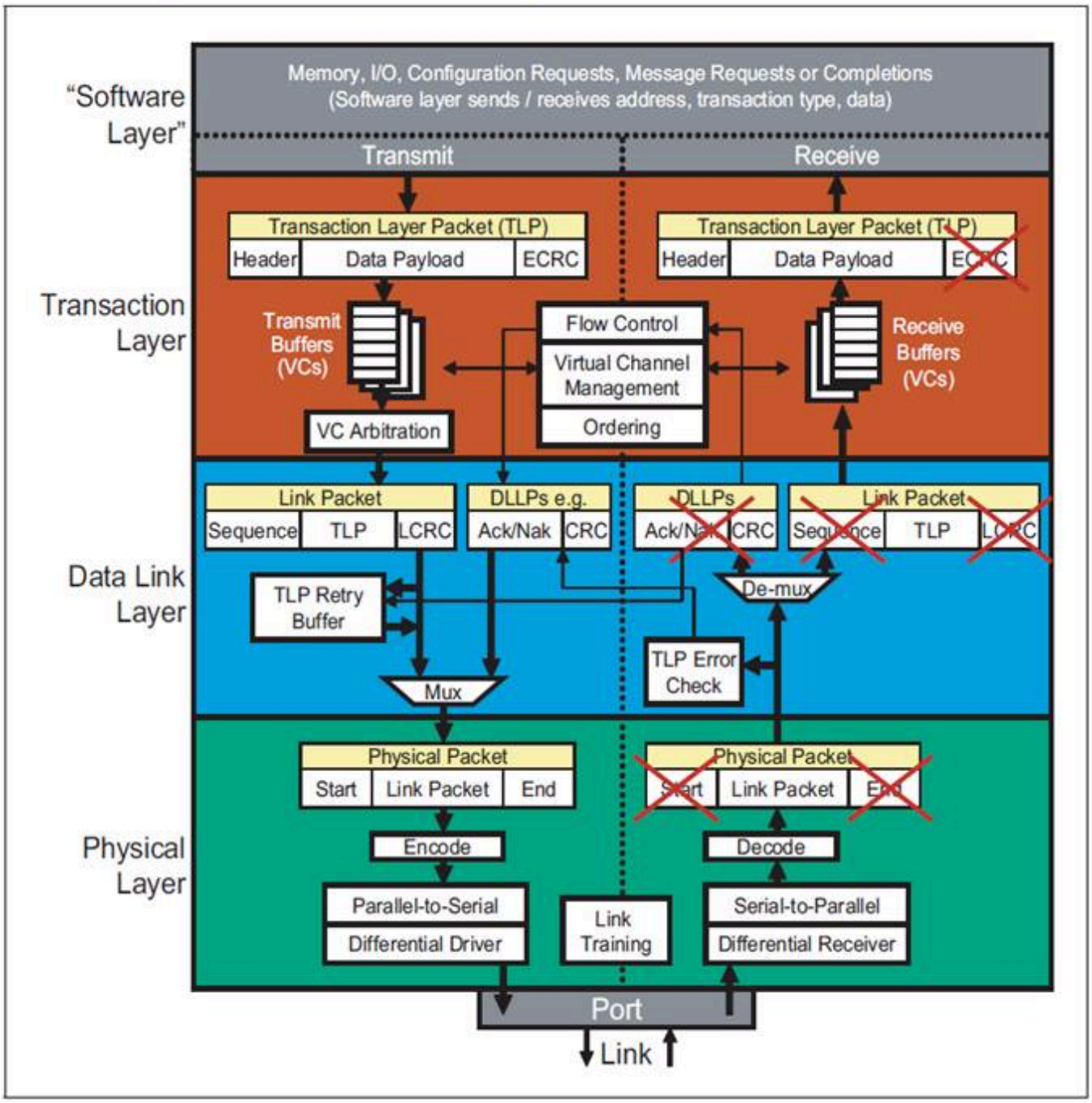

图2-14 PCIe 设备层次结构的详细框图

接收端对物理层的输入比特进行解码，检查本层级所能发现的错误，如果没有检查到错误，那么就将接收到的数据包向上转发到数据链路层。在数据链路层，数据包进行不同于物理层的错误检查，如果没有发现错误就向上转发给事务层。在事务层数据包被缓存，检查错误，并拆解成原始信息（命令、属性等），以便将这些数据传输到接收设备的核心层。接下来，让我们更深入的探索每一个层必须要做什么才能让上述过程正常工作，这个操作过程在图2-14 进行了展示。我们

从最顶端开始。

2.2.1 设备核心层/软件层（Device Core/Software Layer)

设备核心层是一个设备的核心功能，例如网络接口或是硬盘驱动控制器。它并不是 PCIe 协议规范中所定义的一个层级，但是我们可以把它当做一个 PCIe 的层级，这是因为它位于事务层的上方，而且它是所有请求的源头或是目的地。它为事务层提供了需要发送的请求信息，其中的信息包括事务类型、地址、需要传输的数据量等等。当事务层接收到输入数据包时，它也是事务层向上转发输入数据包信息的目的地。

2.2.2 事务层（Transaction Layer)

为了响应来自软件层的请求，事务层生成出站数据包（outbound packet）。它也会检查入站数据包（inbound packet），并将入站数据包内包含的信息向上转发给软件层。事务层支持非报告式请求（non-posted transaction）的拆分事务协议，并将入站完成包（inbound Completion）与先前传输的出站非报告式请求包关联起来，即知道这个完成包是对应到哪个非报告式请求包。事务层所处理的事务使用的数据包种类为 TLP，TLP 可以分为四个请求种类：

1. 内存（Memory）
2. IO
3. 配置（Configuration）
4. 消息（ Messages）

前三种在 PCI 和 PCI-X 中就已经得到支持，但是消息是 PCIe 中的一个新的请求种类。一个请求包向目标设备传送命令，目标设备作为响应请求而发回的一个或多个完成包，这二者组合起来就是对一个事务的定义，即一个事务由一个请求包以及所有返回的完成包共同组成。如表2-2 列出了 PCIe 请求的类型。

Table 2-2: PCI Express Request Types

Request Type	Non-Posted or Posted
Memory Read	Non-Posted
Memory Write	Posted
Memory Read Lock	Non-Posted
IO Read	Non-Posted
IO Write	Non-Posted
Configuration Read (Type 0 and Type 1)	Non-Posted
Configuration Write (Type 0 and Type 1)	Non-Posted
Message	Posted

表2-2 PCIe请求类型

这些请求还可以被归为两类，如上表的右边一列：非报告和报告（**non-posted** and **posted**）。对于非报告式请求，发起方首先向完成方发送一个请求数据包，完成方应该产生一个完成包作为响应。读者们应该认出来了，这就是从 PCI-X 那里继承来的拆分事务协议。例如，所有的读请求都是非报告式的，因为这个读请求所请求的数据需要通过完成包来返回给发起方。可能令你感到出乎意料，IO 写请求和配置写请求也是非报告式的。尽管它们在发送给目标设备的命令中就已经包含了要写入目标设备的数据，但是这两种请求依然需要目标设备在写入完成后返回完成包，以此来让发起方确认数据被正确无误的写入到目的设备中。

与上述的请求相反地，内存写请求和消息请求都是报告式的，这意味着完成方在完成这些请求后不需要向发起方返回完成包 TLP。报告式事务对整体性能提升是有好处的，因为发起方不需要等待响应，也不需要承担对完成包进行处理的额外开销。这里做出的取舍就是发起方无法得到写请求是否被正确无误的完成的反馈信息。报告式这种操作行为继承自 PCI，它依然被认为是一个不错的操作方法，虽然无法得到反馈信息，但是发生错误的可能性比较小并且使用报告式得到的性能提升也比较明显。需要注意的是，尽管它们**不要求**返回完成包，报告式写操作仍然要参与数据链路层的 Ack/Nak 协议，以此来保证较为可靠的数据包传输。关于这一点的更多内容，请见第十章“Ack/Nak 协议”。

2.2.2.1 TLP 基础内容

PCIe 请求包和完成包的包类型被罗列在表2-3 中。

Table 2-3: PCI Express TLP Types

TLP Packet Types	Abbreviated Name
Memory Read Request	MRd
Memory Read Request - Locked access	MRdLk
Memory Write Request	MWr
IO Read	IORd
IO Write	IOWr
Configuration Read (Type 0 and Type 1)	CfgRd0, CfgRd1
Configuration Write (Type 0 and Type 1)	CfgWr0, CfgWr1
Message Request without Data	Msg
Message Request with Data	MsgD
Completion without Data	Cpl
Completion with Data	CplD
Completion without Data - associated with Locked Memory Read Requests	CplLk
Completion with Data - associated with Locked Memory Read Requests	CplDLk

表2-3 PCIe 的 TLP 类型

TLPs 起始于发送方的事务层，终止于接收方的事务层，如图2-15 所示。当 TLP 途经发送方的数据链路层以及物理层时，这两层分别会向数据包中添加一些信息，接收方的数据链路层和物理层会分别根据发送方对应层所添加的信息来进行校验，以此确认数据包是否在链路传输中依旧保持正确没有出错。

Figure 2-15: TLP Origin and Destination

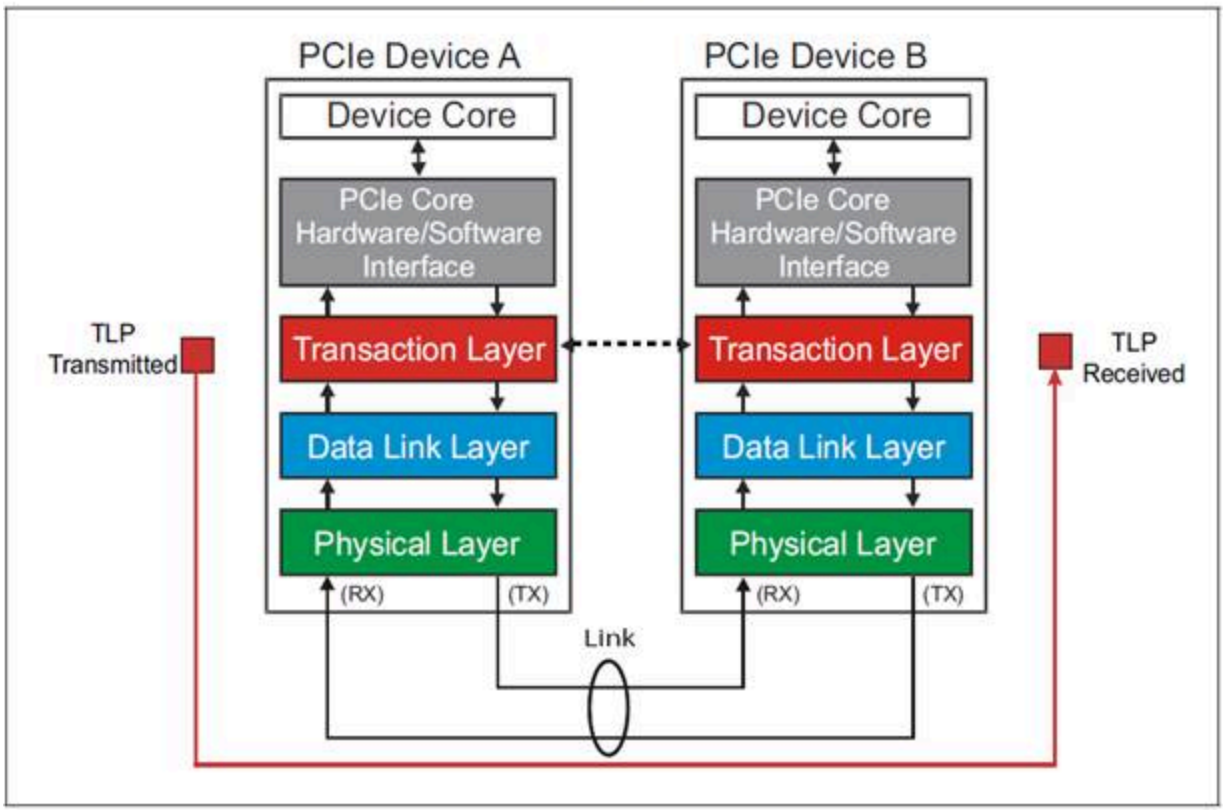

图2-15 TLP 的起点与目的地

TLP 组包. 如图2-16 展示了一个封装完成的 TLP 的各个部分是如何在链路上进行传输的，我们可以从中发现这个数据包中的不同部分是分别由不同的层来添加的。为了更容易看出这个数据包是怎么构成的，我们将 TLP 的不同部分用不同的颜色进行标识，以此来表示对应的部分是由哪一层添加的：红色代表事务层，蓝色代表数据链路层，绿色代表物理层。

Figure 2-16: TLP Assembly

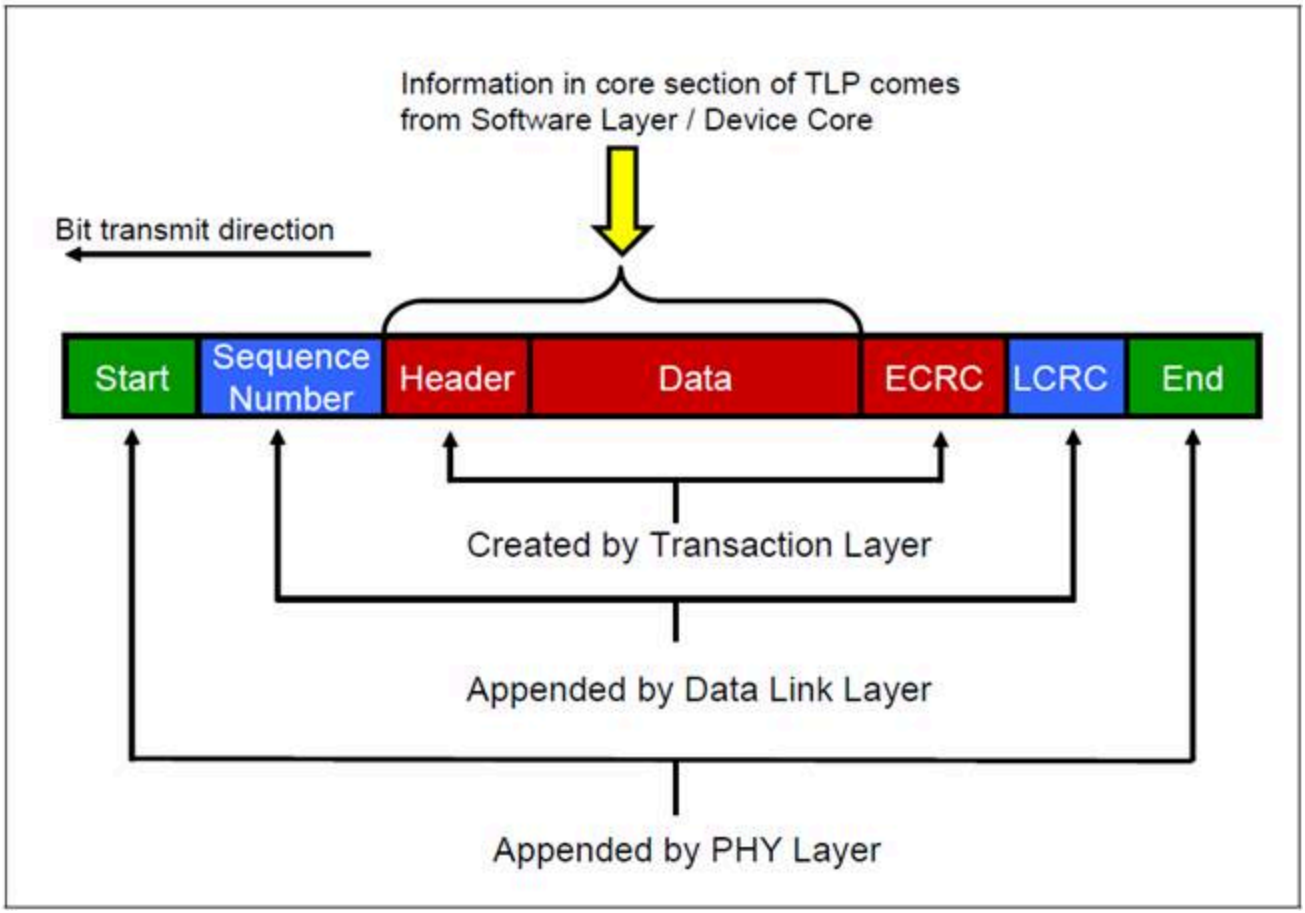

图2-16 TLP 组包形式

设备核心层负责将组成 TLP 的核心部分的信息发送给事务层，即上图中的首部和数据。每个 TLP 都会有一个首部，而有些 TLP 会没有数据部分，例如读请求。

事务层还可以选择添加 ECRC（End-to-End CRC， 端到端 CRC）字段，这个 ECRC 由事务层进行计算并附加在数据包的后面。CRC 的意思是循环冗余校验码，它被几乎所有的串行传输架构所使用，原因很简单，它实现起来比较简单同时又能提供很强的错误检测功能。CRC 还可以检测“突发错误”，即一串重复的错误比特，这一串比特的长度取决于 CRC 的长度（对于 PCIe 来说是 32比特）。因为这种类型的错误在发送一长串比特时会遇到，因此 CRC 的这种特性非常适用于串行传输。ECRC 字段区域在通过发送方和接收方之间的任何服务点（服务点（service point），通常指的是交换机或者根组件的端口这些有 TLP 路由功能的地方）时都不改变，这使得目的端可以用它来验证在整个传输过程中都没有发生错误。

对于 TLP 的传输，TLP 的核心部分由事务层转发至数据链路层，数据链路层负责在 TLP 中添加一个序列号和另外一个被称为 LCRC（Link CRC, 链路CRC） CRC 字段区域。LCRC 被对端接收

方用来进行错误检查，并将链路上传输的每个数据包的检查结果都汇报给发送方。善于思考的读者可能会有疑问，如果 LCRC 已经证明了这次链路传输是无差错的，而 LCRC 又是数据包必需含有的字段，那么 ECRC 还有什么作用呢？这个疑问的回答是，还使用 ECRC 是因为还有一个地方的传输错误没有被检查，那就是负责路由数据包的设备内部。当一个数据包到达一个端口，并进行错误和路由检查，然后当它被从另一个端口发出时，设备会计算出一个新的 LCRC 值并添加在数据包中，新的LCRC 会取代老的 LCRC 。内部端口之间的转发可能会遇到 PCIe 协议没有检查到的错误，这时候就需要 ECRC 来发挥它的作用了。这里可以理解为，当 TLP 到达交换机的一个接收端口时，交换机会在接收端口对其 LCRC 进行校验，然后根据路由信息对数据包进行转发，但是这个转发过程中是不对 LCRC 进行校验的，直到转发到对应的输出端口，然后在输出端口给数据包加上新的 LCRC 后发送出去，不难发现因为在内部转发过程中不对 LCRC 进行校验，因此若转发过程中出现了错误则就需要通过更内层封装的 ECRC 来进行校验了，否则内部转发过程是否出错将无从可知。

最后，数据链路层封装好的数据包被转发给物理层，物理层将其他一些字符添加到数据包中，这些字符可以让接收方知道接下来将会接收什么（例如标识一个数据包的开始或结尾）。对于前两代的 PCIe，物理层将在数据包的头和尾添加控制字符。而在第三代 PCIe 中不再使用控制字符，而是在数据包中添加一些额外的比特来提供关于数据包的信息。经过这些处理后，数据包被编码，然后在链路上所有可用通道中以差分形式传输。

TLP 拆包. 当对端的接收方看到了输入的比特流时，它需要对此前组包时添加的那些部分进行识别和剥除，这样就能恢复出发送方设备的设备核心层的原始请求信息。如图2-17 所示，物理层将会确认当前比特流中是否存在正确的“起始”、“结束”或者其他字符，并将它们剥除，然后将剥除了这些字符后的 TLP 转发给数据链路层。数据链路层将首先进行 LCRC 以及序列号的错误校验。如果并未发现错误，那么数据链路层将会把 LCRC 和序列号从 TLP 剥除，并将 TLP 转发给事务层。如果接收方是一个交换机，那么将在事务层对这个数据包进行解析评估，从它的数据包头中找到路由信息来确定这个数据包要被转发到哪一个端口。即使这个交换机并不是 TLP 最终的目的地，它也可以对这个 TLP 进行 ECRC 校验以及在发现错误时进行 ECRC 错误汇报。但是，交换机不能更改这个 TLP 中的 ECRC，这是希望最终的目标设备也可以检测到这个 ECRC 错误。

如果目标设备有能力并且启用了 ECRC 校验的功能，那么它将对 ECRC 进行错误校验。若 TLP 到达了最终的目的设备，而且校验无错，那么在事务层中将把这个 ECRC 字段剥除，使得整个数据包只剩下首部和数据部分，并将这些剩余部分转发给软件层。

Figure 2-17: TLP Disassembly

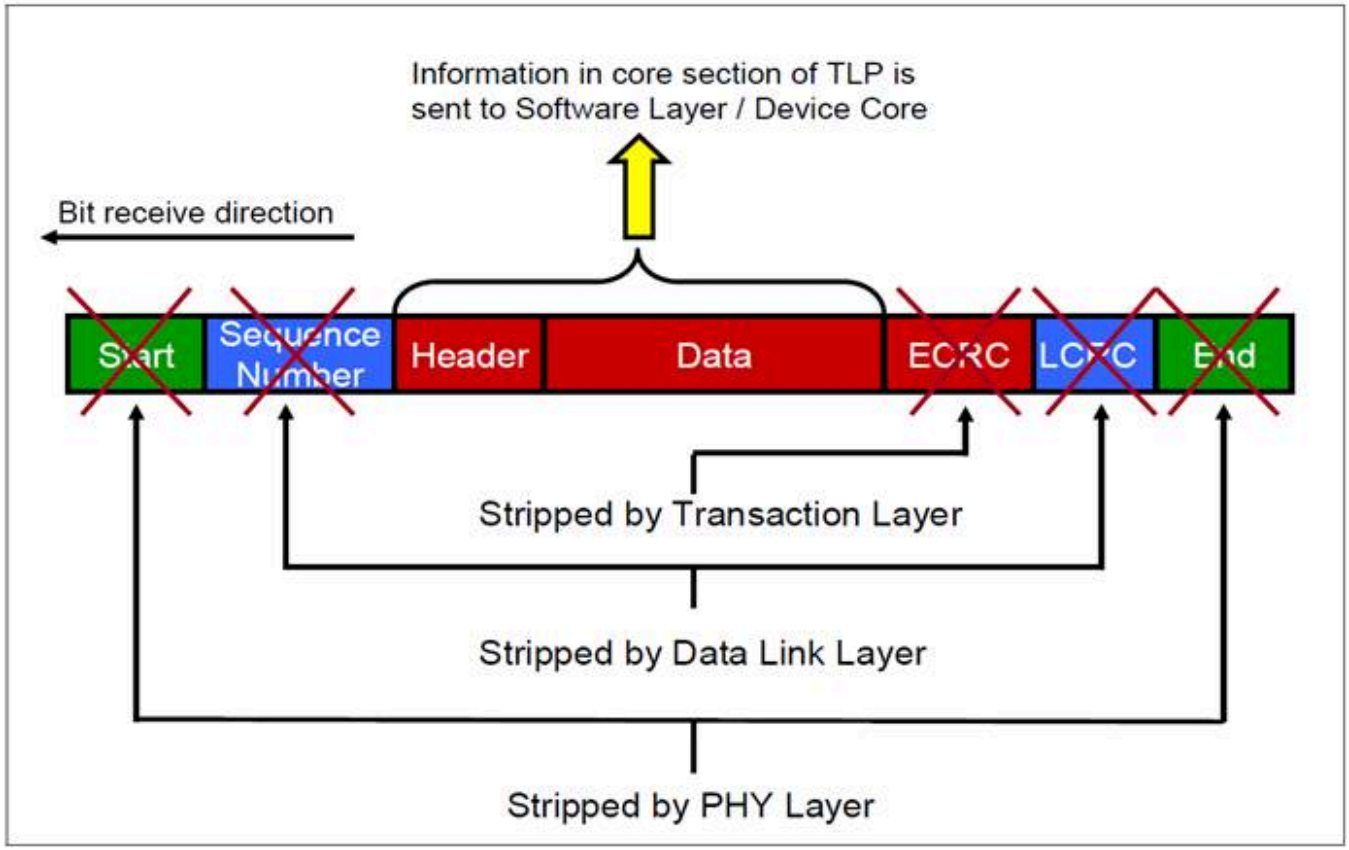

图2-17 TLP 的拆包

2.2.2.2 非报告式事务 (Non-Posted Transactions)

普通读（Ordinary Reads）. 如图2-18 展示了一个内存读请求，这个请求由端点发往系统内存。在内存读请求 TLP 中有一个很重要的部分，那就是目标地址（关于更多有关 TLP 内容的详细讲解，请见第五章“TLP 元素”）。一个内存请求的地址可以是 32 位或者 64 位的，这也决定了数据包的路由方式。在这个例子中，这个请求事务通过两个交换机的路由后向上转发给了目标设备，例子中的目标设备为根组件。当根组件对请求包进行译码，并识别出了数据包中的地址指向了系统内存，它将从那段系统内存中取出端点所请求的数据。为了将这些从内存中取出来的数据返回给发起方，根组件端口的事务层将产生足够多的完成包，这些数量的完成包足以将发起方所请求的全部数据都返回回去。PCIe 中规定一个数据包中数据荷载最大为 4KB，但是实际设计的设备中一般会使用的数据荷载会比 4KB 小，因此需要好几个完成包才能足够将比较大量的数据返回给发起方。

这些完成包内也包含了**路由信息**，用于将它们指引回到发起方，这是因为发起方在原先的请求包中就附带了需要返回的地址的信息，所以完成方就可以将这个地址用来放在完成包中作为完成包的路由信息。这个“返回地址”其实很简单，它就是 PCI 中定义的设备 ID，这个设备 ID 由三

个东西组成：发起方所属 PCI 总线在系统中的 PCI 总线号、发起方在所属 PCI 总线上的设备号、发起方在所属设备中的功能号。这样的总线号、设备号、功能号组合起来的信息（有时被缩写为 BDF，Bus、Device、Function）就是完成包用来返回到发起方的路由信息。

正如 PCI-X 所做的那样，发起方可以同时有多个正在进行的**拆分事务**，它必须能够将输入的完成包和正确的请求关联起来。为了便于实现这一点，发起方在原先的请求包中加入了一个值称为 Tag，这个 Tag 对于每一个请求而言都是独一无二的，也就是说每一个未完成的请求都有一个与其他未完成的请求不同的 Tag 号。完成方会将这个事务的 Tag 拷贝进完成包中，这样发起方就可以快速地通过完成包中的 Tag 来将这个完成包与正确的请求关联起来，也就是找到了这个完成包是用来服务哪个请求的。

最后还要说一点，完成方还可以设置完成包中的完成状态字段中的比特，以此来表示出**事务的错误情况**。这至少可以让发起方对哪里可能出错了有一个大致的了解。发起方如何去处理大多数的错误，这个事情一般是由软件来决定，这并不在 PCIe 协议规范的范围内。

Figure 2-18: Non-Posted Read Example

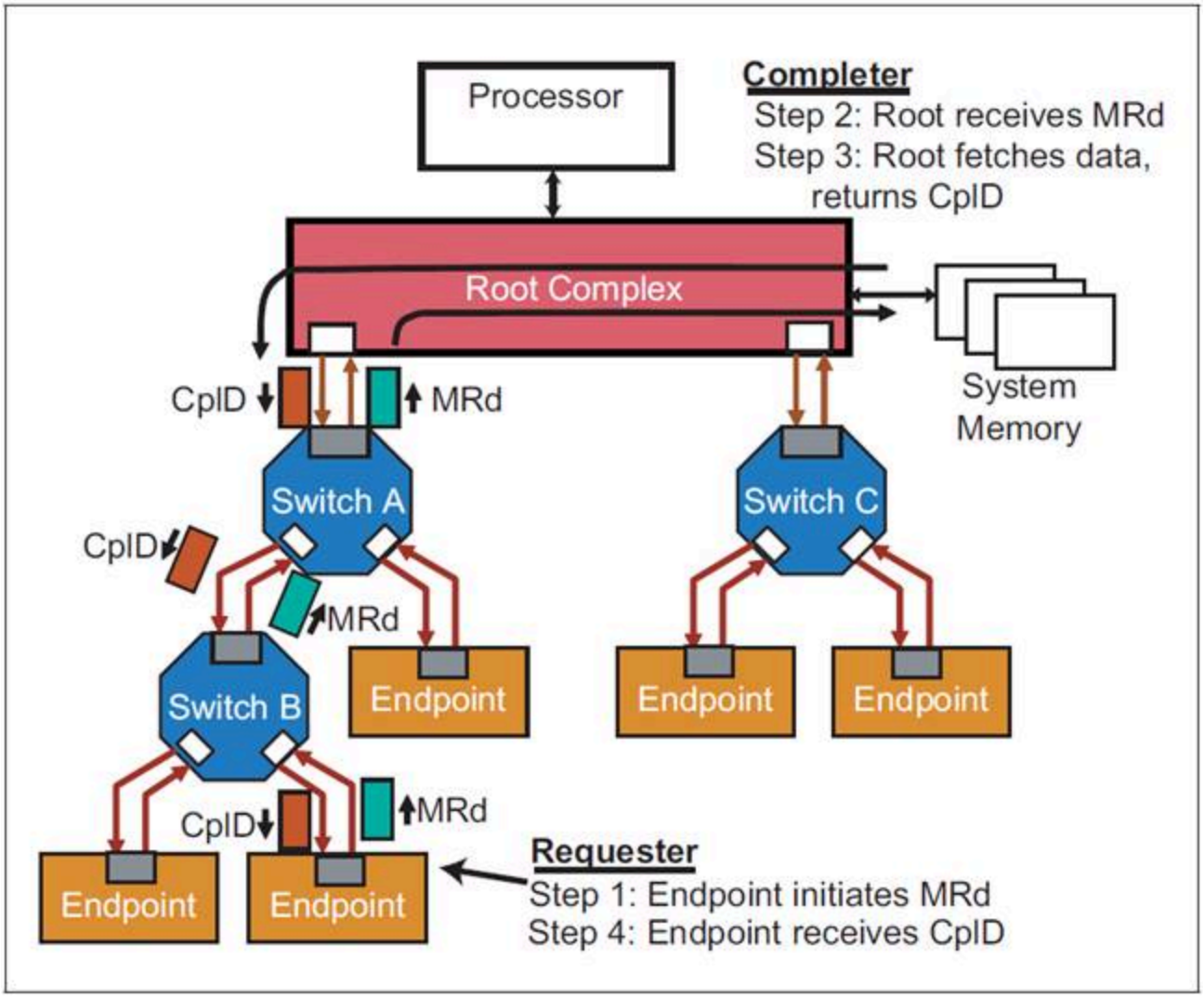

图2-18 非报告式读的示例

锁定读（Locked Reads）. 锁定的内存读这种读请求是为了支持原子读-修改-写操作，所谓原子操作就是一种不可中断的事务，处理器使用这种事务来进行测试以及设置信号标志等任务。当测试和设置信号标正在进行中时，不允许其他对信号标的访问发生，不允许发生竞争的情况。为了避免发生竞争情况，处理器使用一个锁定指示符（例如并行总线前段的一个单独的 pin），来阻止总线上的其他事务，直到这个锁定的事务完成。接下来是对这个主题内容的一个高层次介绍。有关被锁定事务的更多信息，请参阅附录 D，“锁定事务”。

回顾一下历史，在 PCI 的早期，协议规范的制定者们预期 PCI 将会实际取代处理器总线。因此，PCI 协议规范中要求总线要能支持处理器要完成的事情，比如锁定事务。然而，PCI 很少以这种方式使用，最终，这种处理器总线支持的东西大部分都被丢弃了。不过，锁定周期依然存在，用以支持一些特殊情况，在更进一步的 PCIe 的发展中也保留了它，以实现对一些遗留事务

的支持。也许是为了加速向 PCIe 的迁移，在新的 PCIe 设备中禁止接收锁定请求，**只有那些传统设备才能够使用它**。在图2-19 所示的例子中，一个发起方发出了一个 MRdLk（锁定读）来发起事务。根据定义这样的请求仅允许来自 CPU，因此在 PCIe 中仅有根组件的端口可以发起这种事务。

这个锁定请求使用目标内存地址作为路由信息，并最终到达了传统设备的端点。当数据包经过沿途的每个路由设备时（称为服务点），数据包的出口端口被锁定，这意味着在被解锁之前这条路径都不能通过其他的数据包。

Figure 2-19: Non-Posted Locked Read Transaction Protocol

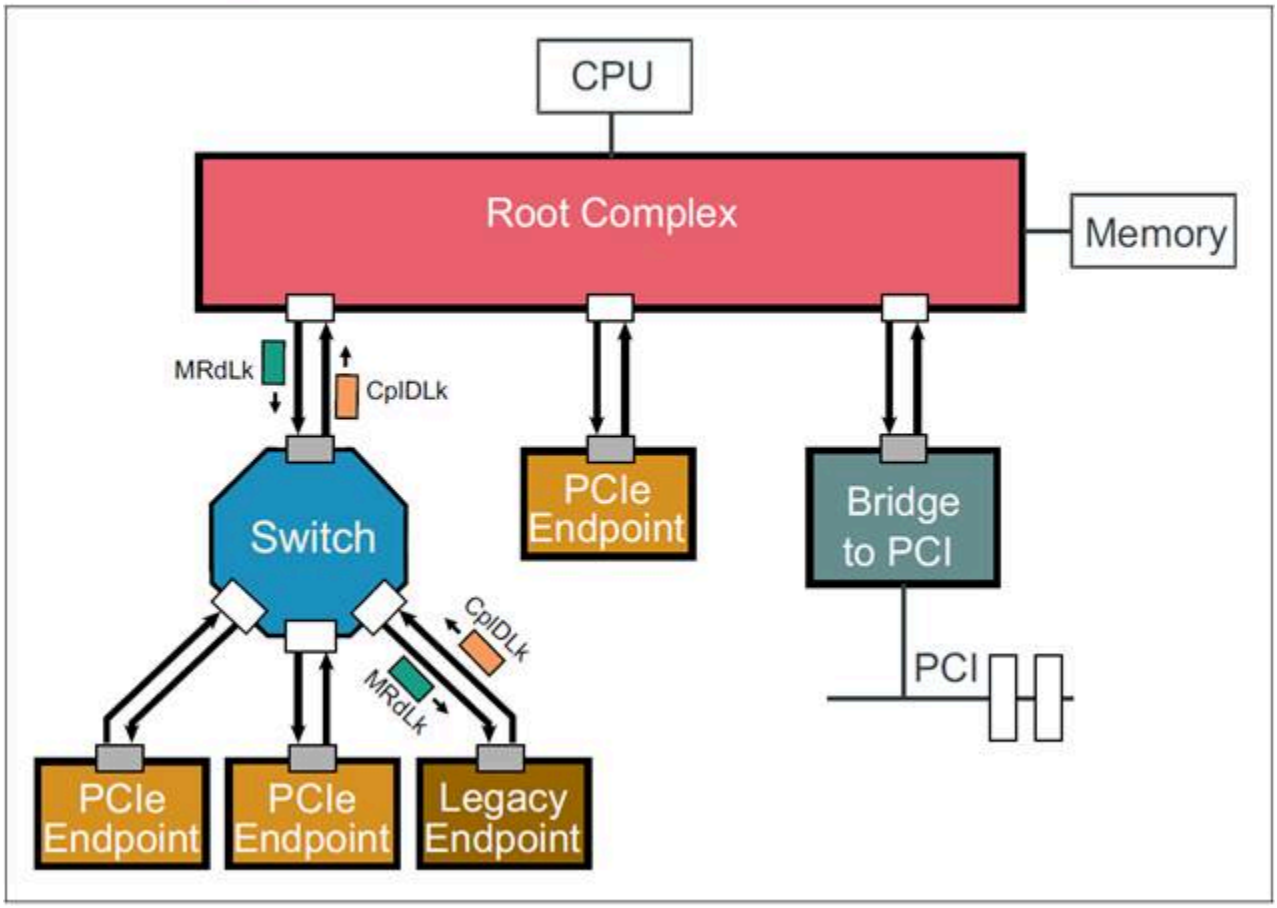

图2-19 非报告式锁定读事务协议

当完成方接收到这个请求包并将其内容进行译码，它将收集请求所需要的数据，并使用这些数据产生一个或多个锁定完成包。这些完成包将会被路由回到发起方，使用的路由信息是发起方 ID，完成包路径上的出口端口也会被锁定。

如果完成方遇到了一些问题，无法正常响应请求，那么它将向发起方返回一个不含有数据的锁定完成包（正常的读请求完成包应该含有要返回的数据，而这里不包含数据那么我们就知道出现了

一些问题），并使用完成包的状态指示区域来标识发生的错误的一些信息。发起方接收到这样的完成包就明白了这个锁定请求失败了，它就会取消这个事务的执行，然后由软件来决定接下来要做什么。

IO和配置写（IO and Configuration Writes）. 如图2-20 中展示了一个非报告式 IO 写事务。与锁定请求相似，这样的一套 IO 操作流程的目的设备只能是一个传统端点（Legacy Endpoint）。请求包使用 IO 地址作为路由信息在交换机中被路由转发，直到它到达目标端点。当完成方接收到这个写请求包，它接收请求包内的数据并返回一个单独的不含有数据的完成包，以此来确认自己收到了这个写请求包。完成包中的状态标识区域将会指示出是否发生了错误，如果发生了错误，发起方的软件需要对错误进行处理。

如果完成包中显示并未出现错误，那么发起方就认为写数据被成功的送达目标设备并成功写入，可以允许针对这一个完成方的指令序列继续向下执行。也就是说，在发起方中有一系列的针对完成方的指令，当知道这一个 IO 写请求完成了之后，允许执行指令序列中的下一个指令，即想表达的是需要确认这个写请求成功了才允许继续执行下一条指令。这很好的总结了非报告式写的目的：不同于一个内存写（MWr 是报告式的），非报告式写不能仅仅知道数据被送往了目的方，还必须要知道数据真的到达了目的方才行，否则在逻辑上不能继续执行下一步。又如同锁定操作一样，非报告式写请求只能由处理器发出。

Figure 2-20: Non-Posted Write Transaction Protocol

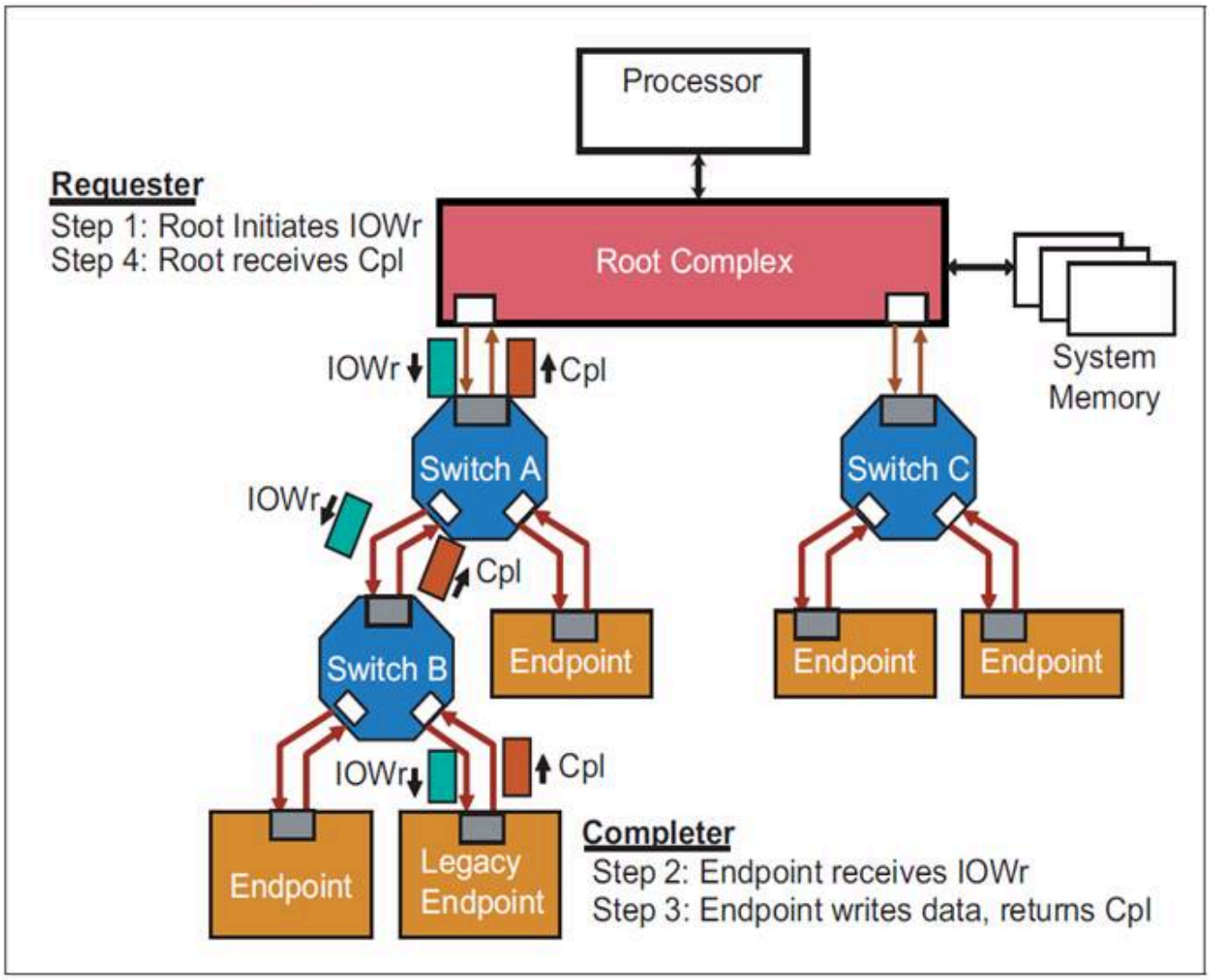

图2-20 非报告式写事务协议

2.2.2.3 报告式写 (Posted Writes)

内存写（Memory Writes）. 内存写请求必然是报告式类型的，它不需要返回完成包。一旦这种写请求包被发出，发起方不需要等到任何反馈就可以继续去进行下一条请求，不需要在返回完成包这件事情上花费时间和带宽。因此，报告式写会比非报告式写更加快速且高效，并很好的提升了系统的性能。如图2-21 所示，请求包使用内存地址作为路由信息在系统中进行路由转发，并最终到达完成方。一旦链路成功的将这个请求发送过去（这里的成功是指把数据包传输了过去，而不需要得到完成方的成功确认），这个事务在链路上就已经结束了，链路上就可以继续传输其他的数据包了。最终，完成方接收写请求包中的数据，这个事务才真正完成。当然，这种事务执行方法在提升效率的同时也舍弃了一些东西，因为完成方不需要发送完成包，所以这也意味着它无法将错误报告给发起方。如果完成方发生了错误，那么它可以记录下这个错误，并向 RC

发一个消息来通知系统软件这些情况，但是发起方对这些是完全不知情的。

Figure 2-21: Posted Memory Write Transaction Protocol

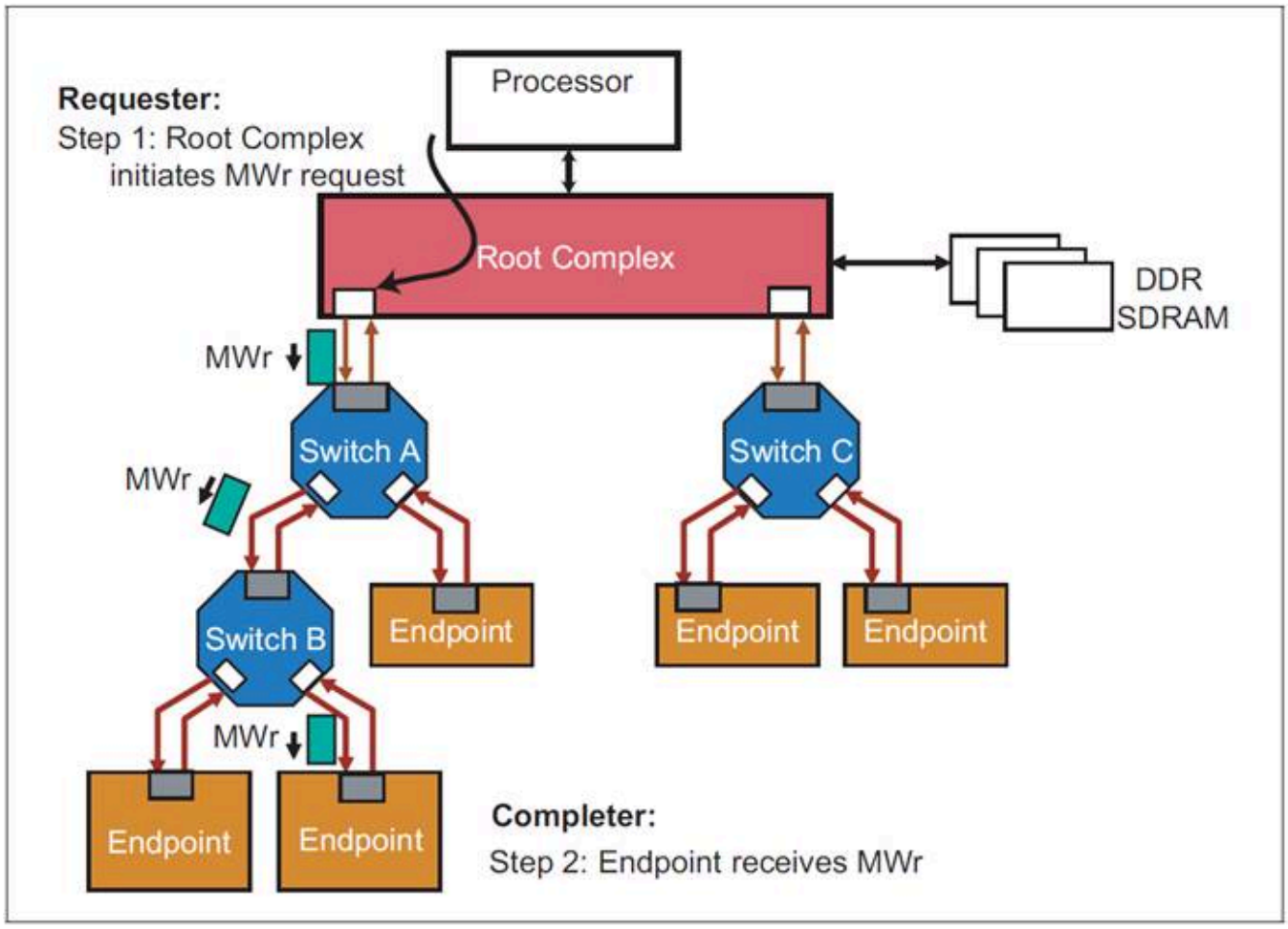

图2-21 报告式内存写事务协议

消息写（Message Writes）. 十分有趣的是，消息不同于我们前面所说的那些请求事务，它有好几种路由方法（前面的都是通过内存地址、IO 地址中的一种），在消息内部专门有一个区域来标识使用的是哪一种路由方式。例如，有一些消息是报告式写请求，其目的方为特定的完成方；有一些是根组件向所有端点广播的请求；还有一些是端点发出的要自动路由到根组件的请求。想要学习更多关于不同的路由类型的内容，请参阅第四章“地址空间和事务路由”。

消息在 PCIe 中很有用，它可以使得 PCIe 达到减少引脚数量的设计目标。它使得 PCIe 不再需要边带信号，而在 PCI 中则是需要用边带信号来报告中断、功耗管理事项以及错误信息，而在 PCIe 中使用消息则可以将这些信息使用数据包来进行报告，数据包是直接在一般数据路径上传输的，因此不再需要额外的边带信号。

2.2.2.4 QoS服务质量（Quality of Service)

PCIe 从一开始就被设计为能够支持时间敏感（time-sensitive）事务，例如流视频、音频应用程序，对于这些应用程序来说数据必须被及时传输才能保证数据有效。这被称为提供了 QoS，它是通过添加一些东西来实现的。首先，每个数据包都被软件分配了一个优先级，这个优先级是通过设置数据包内的一个 3 比特的字段区域来进行标识的，称之为 TC (Traffic class, 流量类型)。一般来说，给一个数据包分配一个编号较大的 TC 表示希望给与这个数据包一个更高的优先级。第二，使用多缓冲区，称为 VC (Virtual Channels, 虚拟通道)，将其构建在硬件的每一个端口中，数据包会根据其 TC 值，来被放入相应的 VC 中（相应的缓存区中）。第三，由于现在对于一个端口来说，在一个时刻将会有多个缓冲区都存在可以进行传输的数据包，因此需要有对 VC 进行选择的仲裁逻辑。最后，交换机必须在相竞争的各个输入端口间做出选择，以便访问相应端口的 VC 。这一步被称为端口仲裁，它可以由硬件来进行分配或是由软件来进行编程配置。（译注：这与第三点的不同之处在于，第三点是在一个端口内对多个 VC 进行仲裁选择，而端口仲裁是指各个端口已经选择好了此次访问的 VC，需要由交换机仲裁选择访问哪一个端口。）所有的这些硬件部件都必须要能够支持系统对数据包进行优先级排序。如果一个这样的系统被正确的编程配置，它可以为给定的路径提供有保障的服务。

图2-22阐述了 QoS 的概念，其中的视频摄像头以及 SCSI 设备都需要向系统 DRAM 发送数据。这二者的不同之处在于，摄像头是对时效性要求严格的，如果传输路径无法满足摄像头的带宽，那么将出现丢帧的现象。因此系统需要保障摄像头所需要的最小带宽，否则它捕捉的视频画面将会出现不稳定。与此同时，SCSI 数据需要正确无误的进行传输，但是对于它来说传输需要的时间长短并不是那么重要。显然，当视频数据和SCSI数据同时需要被发送时，视频数据流应当具有更高的优先级。QoS 指的是系统的一种能力，是系统为数据包分配不同优先级并将这些数据包按照确定性的延时、带宽在系统拓扑中进行路由的能力。想了解更多关于 QoS 的细节，请参阅第七章“服务质量”。

Figure 2-22: QoS Example

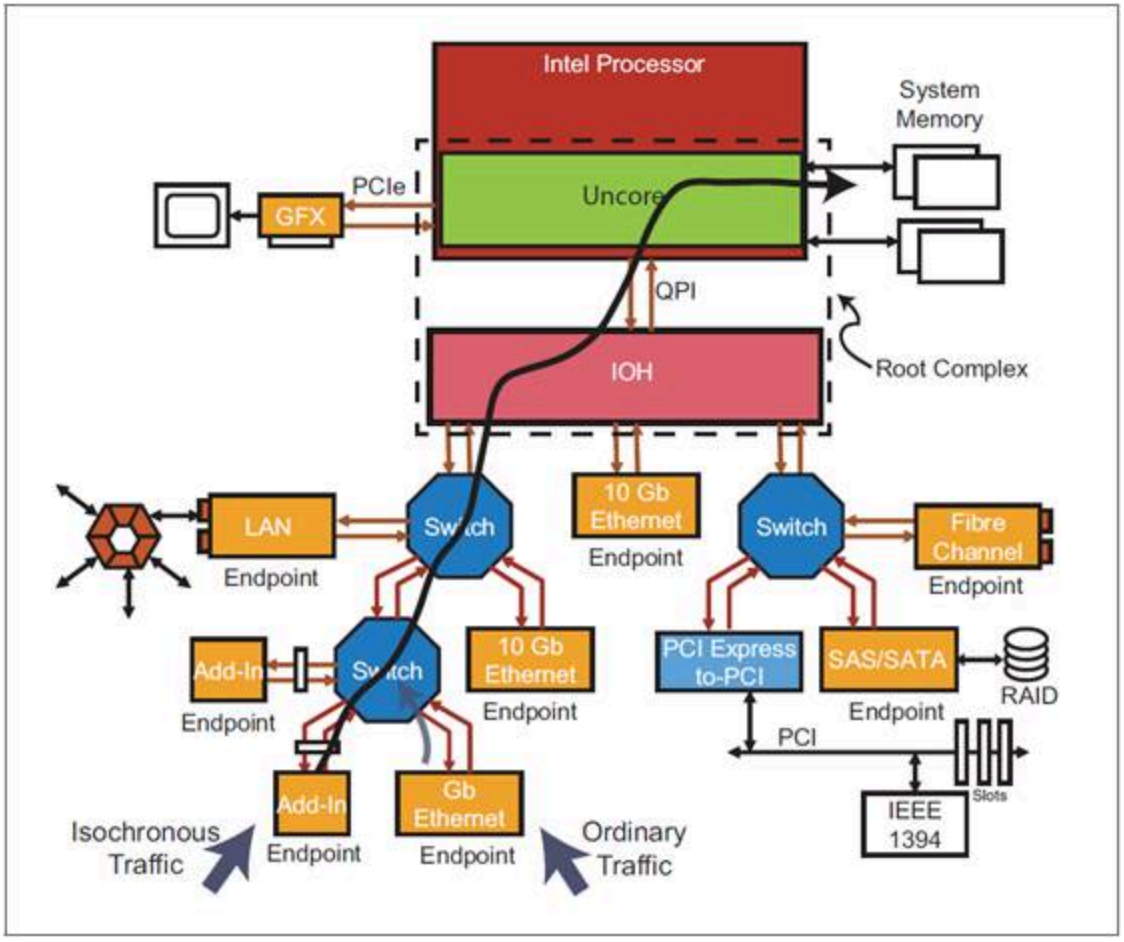

图2-22 QoS 示例

2.2.2.5 事务排序（Transaction Ordering）

在一个 VC (Virtual Channels) 内，数据包一般全都按照它们到达的先后顺序进行排序和移动，但是也有例外情况。PCIe 协议继承了 PCI 的事务排序模型，包括支持 PCI-X 中加入的宽松排序。这些排序规则保证了使用相同 TC 的数据包都会按照正确的顺序在拓扑结构中被路由转发，防止潜在的死锁或者活锁情况。需要注意的一点是，由于排序规则仅应用于 VC，且使用不同 TC 的数据包一般不会被划分进同一个 VC，因此使用不同的 TC 的数据包在软件看来就不存在排序关系。这种排序在事务层中的 VC 中维护。

2.2.2.6 流量控制（Flow Control）

串行传输所使用的一个典型协议是，要求发送方仅在对端有足够的缓冲区接收时才发送数据包。这样的规定删去了总线上浪费性能的操作事件，比如 PCI 中允许进行的断开与重试(disconnects and retries)，这使得这类问题在传输中得到消除。这样做的代价是，接收方必须足够频繁的报告它的可用缓冲区空间来避免不必要的传输停顿，而且这样的报告也需要占用接收

方自己的一点带宽。在 PCIe 中，这个可用缓冲区空间的报告是由 DLLP（Data Link Layer Packet）来完成的，我们将在下一节讲到 DLLP。这里不使用 TLP 的原因是为了避免可能出现的死锁现象，如果使用TLP则可能出现，例如一个设备 A 作为发送方需要对接收方 B 的可用缓冲区空间进行更新，但是当A的接收缓冲区满导致它又无法接收来自B的可用缓冲区报告，这样就出现了一个死循环。而 DLLP 可以不管缓冲区状态就进行收发，这样就避免了死锁的问题。这样的流量控制协议由硬件级进行自动管理，对于软件来说是透明的。

Figure 2-23: Flow Control Basics

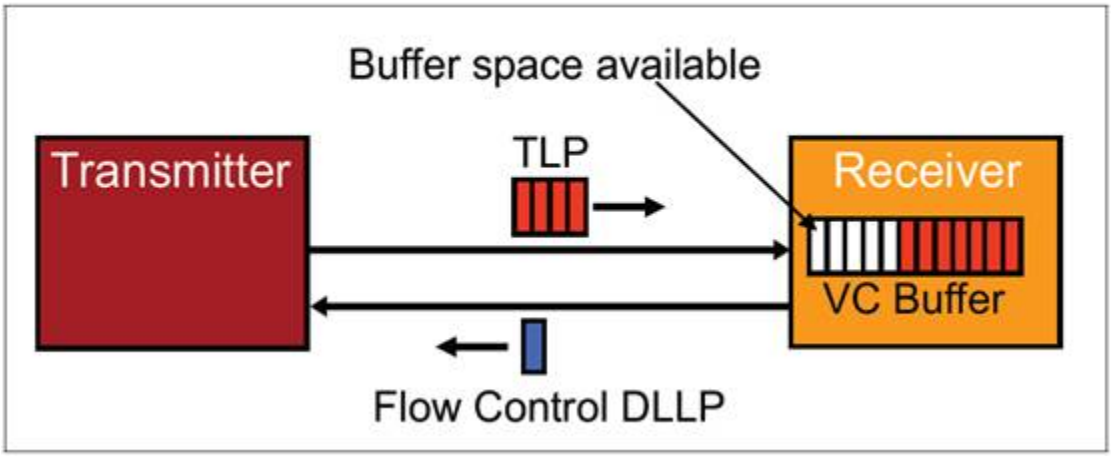

图2-23 流量控制基础操作

如图2-23 所示，接收方内的 VC 缓冲区内缓存了接收到的 TLP。接收方将自己的缓冲区大小通过流量控制 DLLP 来告知发送方。发送方将会跟踪接收方的可用缓冲区空间的值，并且不允许发出大于这个可用空间的数据包。当接收方对缓冲区中的 TLP 进行了处理，并将这个 TLP 移出了缓冲区，此时缓冲区中就空闲出了新的可用空间，那么接收方将会定期的发送流量控制更新 DLLP ，保持发送方能够获取到最新的可用空间的值。想了解更多关于这方面的内容，请见第六章“流量控制”。

2.2.3 数据链路层（Data Link Layer)

数据链路层这一层的逻辑是用来负责链路管理的，它主要表现为3个功能：TLP 错误纠正、流量控制以及一些链路电源管理。它是通过如图2-24 所示的 DLLP 来完成这些功能的。

2.2.3.1 DLLPs数据链路层包（Data Link Layer Packet)

DLLP 是一种在位于同一条链路上的本端设备的数据链路层与对端设备的数据链路层之间传输的数据包。事务层对这些数据包并无感知，它们只在两个相临近的设备之间传输，不会被路由到任何其他地方。DLLP通常要小于TLP，大部分情况下DLLP只有8Bytes，这是一件好事，因为DLLP的大小也影响着维护链路所需要的开销。

Figure 2-24: DLLP Origin and Destination

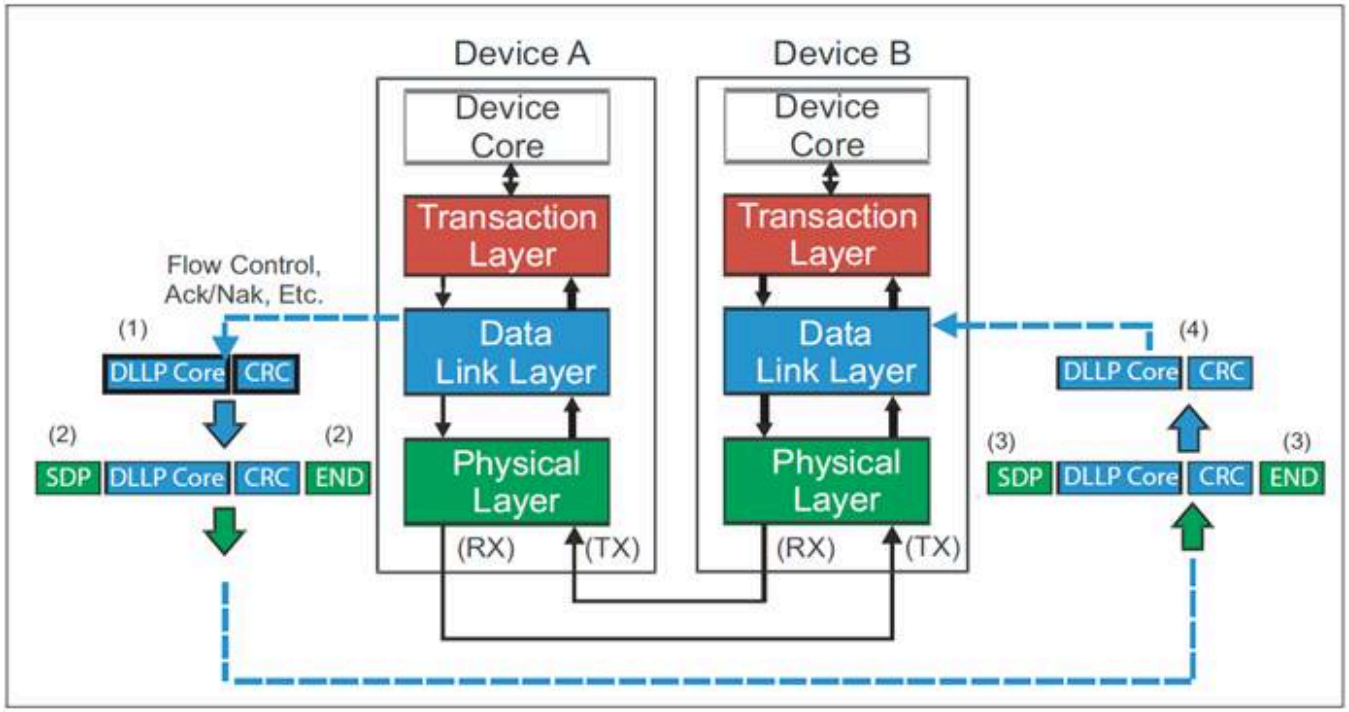

图2-24 DLLP的起始地和目的地

l DLLP 组包（DLLP Assembly）

如图2-24 所示，一个 DLLP 起源于发送方的数据链路层，并结束于接收方的数据链路层。在这个 DLLP 中由发送方给 DLLP Core 加上了一个 16 比特 CRC，用来供接收方进行错误校验。在加完 CRC 之后，这个 DLLP 被转发至物理层，在这一层里给 DLLP 加上包起始字符和包结束字符（这种方法是 Gen1 和 Gen2 的方法，Gen3 不再使用），并对加完两种字符的 DLLP 进行编码，然后通过链路上的所有通道进行差分传输。

l DLLP 拆包（DLLP Disassembly）

当接收方的物理层接收到了一个 DLLP ，这个 DLLP 的比特流将被译码并剥去包起始字符与包结束字符。剩余的 DLLP 部分被转发至数据链路层，在这里进行 CRC 错误校验，并根据包的内容执行相应的操作。接收方的数据链路层就已经是 DLLP 的最终目的地，它不会被继续向上呈交给事务层。

2.2.3.2 Ack/Nak 协议（Ack/Nak Protocol）

如图2-25 所示是一种基于硬件的自动重试机制。如图2-26 所示，每一个被发送方发出的 TLP 都被加上了 LCRC 与序列号，在接收方会对这两个信息进行检查。发送方的重传缓存保存着每个被发送的 TLP 的副本，直到对端确认成功收到了这个 TLP。这个确认成功收到的过程是通过接收方发送 Ack DLLP 来实现的，在这个 Ack DLLP 中包含有接收方成功接收的上一个 TLP 的

序列号。当发送方收到这个 Ack DLLP，它将会把里面序列号所对应的 TLP、以及这个 TLP 之前的所有 TLP，都从重传缓存内清除。

如果接收方检测到了一个 TLP 错误，它将会把这个 TLP 丢弃，并向发送方返回一个 Nak DLLP，以期望发送方能对未确认成功接收的 TLP 进行重传，并通过重传获得一个完好的 TLP。由于错误检测通常是十分迅速的操作，因此从错误检测开始到发起重传的时间也比较短，这样就可以在较短的时间内纠正发生的错误。这一操作过程被称为 Ack/Nak（Acknowledge/ NotAcknowledge） 协议。

Figure 2-25: Data Link Layer Replay Mechanism

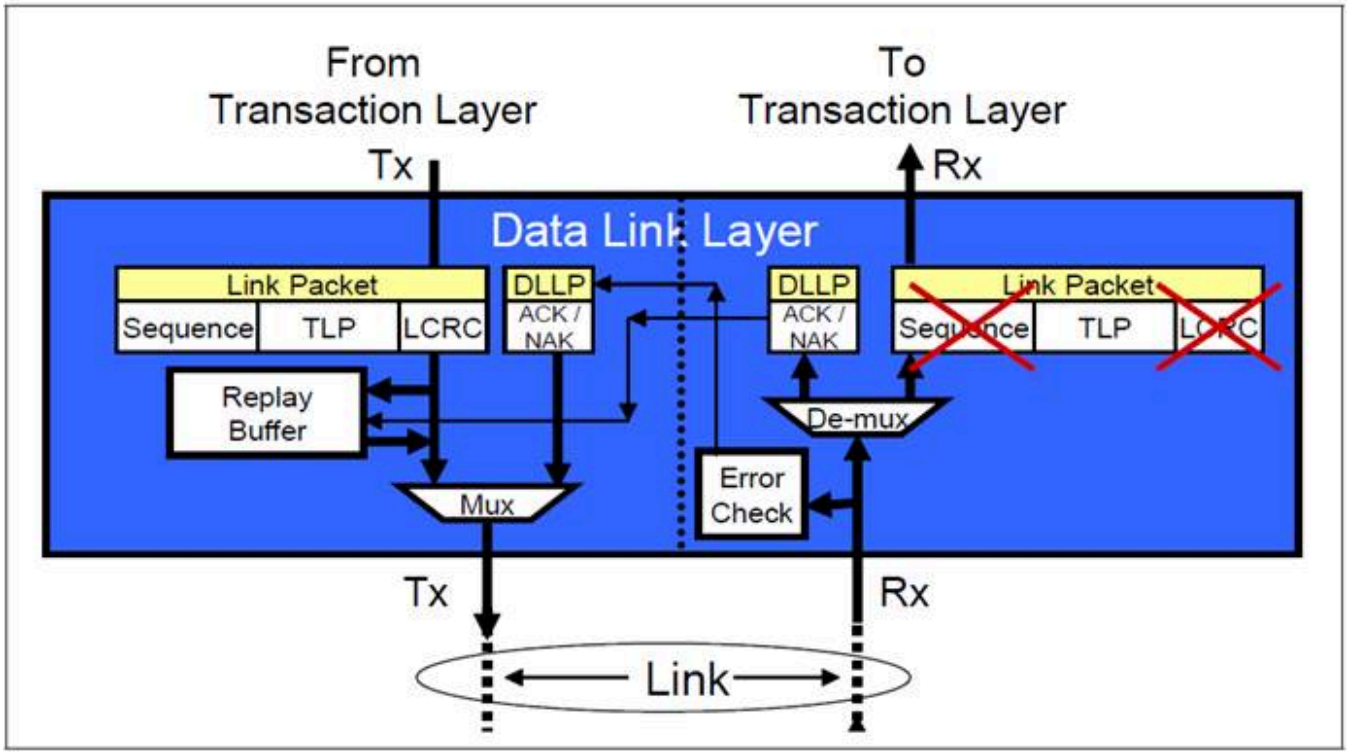

图2-25 数据链路层重传机制

Figure 2-26: TLP and DLLP Structure at the Data Link Layer

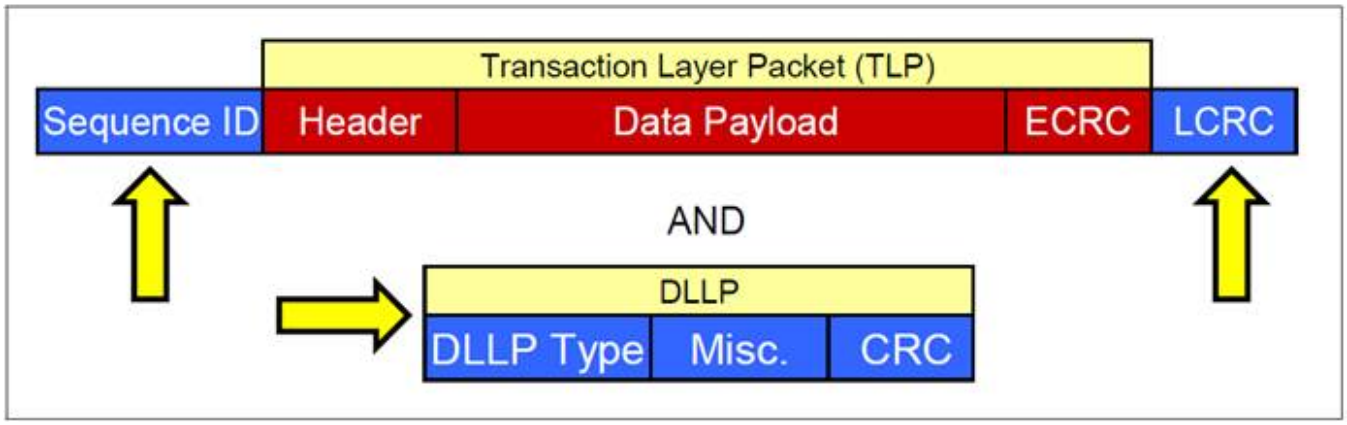

图2-26 在数据链路层中的 TLP 与 DLLP 的包结构

一个 DLLP 的基础形式如图2-26 所示，它由 4 字节的 DLLP 类型域以及 2 字节 CRC 组成，其中 DLLP 类型域中包含了一些其他的信息。

如图2-27 所示，它展示了一个内存读 TLP 通过交换机的示例。一般来说，这个过程的步骤如下：

1. **步骤1a**：发起方发送一个内存读请求，并在自身的重传缓存中也保存一个请求包的副本。交换机接收这个 MRd TLP（Memory Read TLP，以后简写为 MRd TLP），并校验 LCRC 以及序列号。
   步骤1b：校验未发现出错，因此交换机向发起方返回一个 Ack DLLP。作为响应，发起方将其重传缓存中保存的 MRd TLP 副本丢弃。
2. **步骤2a**：交换机将这个 MRd TLP 转发到正确的输出端口，使用的路由信息是这个 MRd TLP 的内存地址，在转发出去的同时，交换机会在这个输出端口的重传缓存内保存一份这个 TLP 的副本。完成方接收到这个 MRd TLP 并对其进行错误校验。
   步骤2b：校验未发现出错，因此完成方向交换机返回一个 Ack DLLP。作为响应，交换机之前输出这个 TLP 的输出端口会将其重传缓存中保存的这个 MRd TLP 的副本丢弃。
3. **步骤3a**：作为这个请求的最终目的地，完成方将会校验 MRd TLP 中的 ECRC 字段域，这个字段域是可选的不是必需的。若校验未出错，则将这个请求呈交给设备核心层。然后根据这个请求命令中的相关信息，设备将会收集被请求的数据，发起方返回一个 CplD（Completion with Data TLP，带有数据的完成包），同时它也将在自己的重传缓存中保存这个 CplD TLP 的副本。当交换机接收到这个 CplD TLP 时将会对其进行错误校验。
   步骤3b：校验未发现出错，因此交换机向完成方返回一个 Ack DLLP。作为响应，

完成方将其重传缓存中保存的这个 CplD TLP 的副本清除。

4. **步骤4a**：交换机对这个 CplD TLP 中的发起方 ID 进行译码，并通过这个信息将 CplD TLP 路由至正确的输出端口，输出的同时在这个输出端口的重传缓存内保存 CplD TLP 的副本。当发起方接收到这个 CplD TLP 时将会对其进行错误校验。
   步骤4b：校验未发现出错，因此发起方向交换机返回一个 Ack DLLP。作为响应，交换机之前输出这个 CplD TLP 的输出端口会将其重传缓存中保存的这个 CplD TLP 的副本丢弃。发起方将校验 ECRC 字段域（这个字段是可选的，不是必需的），校验未出错则将完成包里的数据向上呈交给设备核心层逻辑。

Figure 2-27: Non-Posted Transaction with Ack/Nak Protocol

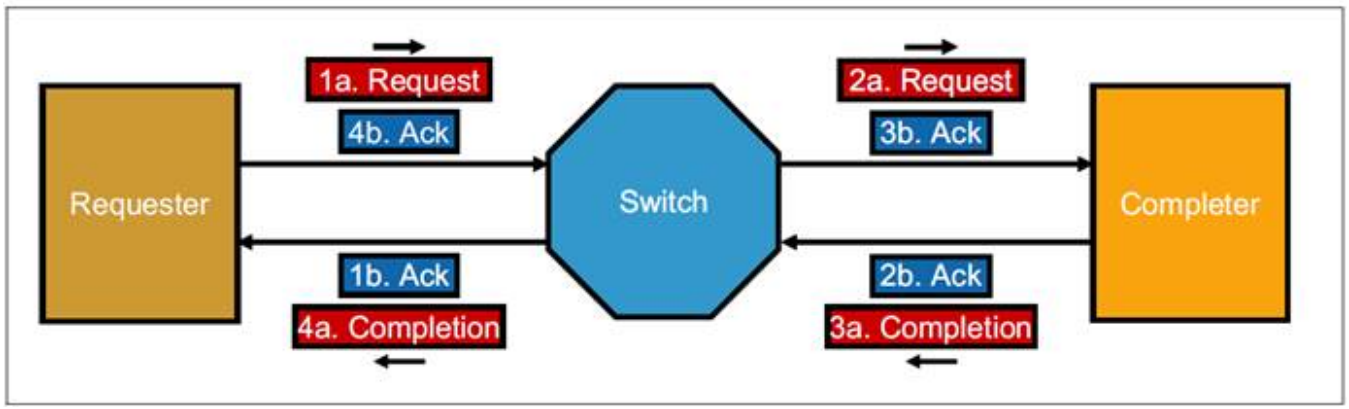

图2-27 使用 Ack/Nak 协议的非报告式事务

2.2.3.3 流量控制（Flow Control）

数据链路层的第二个主要功能就是流量控制。在系统上电和复位之后，流量控制机制被数据链路层的硬件自动初始化，并在整个运行过程中更新。关于这些内容的概述已经在 TLP 的那一节给出，所以这里不再赘述。想要了解更多关于这一主题的内容，请参阅第六章“流量控制”。

2.2.3.4 电源管理（Power Management）

最后要介绍的一点，数据链路层也参与了电源管理，因为链路与系统电源状态相关的请求和握手也可以通过 DLLP 来进行。想了解关于这一主题的更详细的内容，请参阅第十六章“电源管理”。

2.2.4 物理层（Physical Layer）

2.2.4.1 整体说明（General）

物理层是 PCIe 协议层次里的最底层，如图2-14。TLP 和 DLLP 这两种类型的数据包都需要从数据链路层向下转发至物理层，这样才能通过链路传输至对端接收方设备，并从接收方的物理层向

上转发至它的数据链路层。协议规范中将关于物理层的讨论分为两部分：逻辑部分和电气部分。我们这里也将保留这种划分方式。逻辑物理层包含了一系列的数字逻辑，这些数字逻辑是关于准备将数据包在链路上进行串行传输的逻辑以及相反的输入数据包的处理逻辑。电气物理层是物理层的模拟电路接口，它与链路直接相连，它为每个通道提供差分驱动器以及差分接收器。

2.2.4.2 逻辑物理层（Physical Layer-Logical）

由数据链路层转发来的 TLP 与 DLLP 被物理层中的缓存所记录，在这个缓存中会对这些数据包加上包起始字符和包结束字符，这两个字符将有助于接收端用来检测数据包的边界。由于包起始字符和包结束字符分别出现在数据包的两端（头和尾），因此他们也被称作“组帧”字符。如图 2-28 展示了组帧字符被添加在 TLP 和 DLLP 上，它也展示了这两种字符的字段长度。

Figure 2-28: TLP and DLLP Structure at the Physical Layer

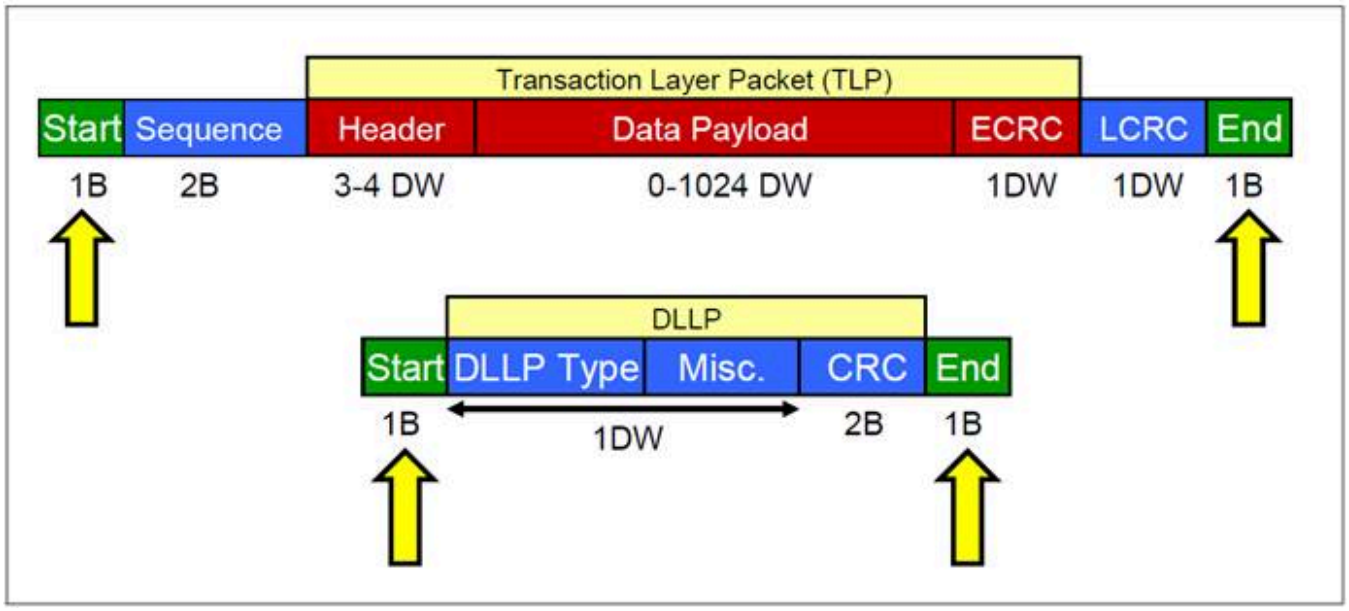

图2-28 物理层中的 TLP 与 DLLP 的结构

在物理层中，数据包中的每一个字节都将被分割到链路所使用的所有通道，这一过程被称为字节条带化（byte striping）。实际上，每个通道在链路上都扮演着一个独立的串行传输路径，这些通道们各自传输的数据会在接收端被聚合。每个字节都进行了扰码，以此来减少在传输线上传输连续重复的“0”或“1”，这有助于减少链路上的 EMI（electro-magnetic interference，电磁干扰）。

对于前两代的 PCIe（Gen1 和 Gen2），8 比特的字符将被编码为10比特“符号（symbol）”，所使用的编码方式被称为 8b/10b 编码。这种编码增加了输出数据流的额外开销，但是也带来了不少有用的特性（关于这里的更多内容请见“8b/10b 编码”一节）。PCIe Gen3 的物理层逻辑在使用 Gen3 速率进行传输时（因为也可以向下兼容 Gen1、Gen2 速

率），不再使用 8b/10b 编码方式，而是采用了另一种被称为 128b/130b 的编码，它会将数据包的字节扰码后进行传输。每个通道的10b 符号（对于Gen1 Gen2）或者每个通道的数据包的字节（对于 Gen3）随后就会在链路的每个通道上以串行差分的形式被传输，对应的速率为 Gen1 2.5GT/s，Gen2 5GT/s，Gen3 8GT/s。

当数据包的比特到达接收端时，接收端以训练得到的时钟速率对这些数据包的比特进行接收。如果使用的是 8b/10b 编码，那么输入的比特流将被串并转换器转换成 10 比特符号，然后准备用来做 8b/10b 解码。然而，在解码之前，10 比特符号将会经过一个弹性缓存，这是一个聪明（clever）的设备，它可以对两个相连接设备各自内部时钟之间的微小频率差异进行补偿。接下来，10 比特符号被 8b/10b 解码器解码，正确地恢复成 8 比特字符。

PCIe Gen3 与前面所述的这个过程不同，对于 Gen3 的物理层逻辑来说，当接收到以 Gen3 速率传输的数据包串行比特流时，将使用串并转换器将这个比特流转换为字节流，这个串并转换器已经建立了块锁定（Block Lock，这是链路训练中的一个要素）。随后字节流会通过一个弹性缓存进行时钟容忍度补偿。在 Gen3 速率下数据包数据并没有使用 8b/10b 编码，因此可以跳过 8b/10b 解码这一步。最终，每个通道中的 8 比特字符都将经过解扰，然后经过反条带化（de-scrambled）之后将所有通道内的字节聚合重新恢复成单符号流，这样就在接收端恢复出了发送端所发出的数据流。

2.2.4.3 链路训练和初始化（Link Training and Initialization)

物理层的另一个任务就是负责链路的初始化以及训练过程。在这个全自动化的过程中，为了让链路准备好进行正常工作，我们采用了几个步骤，主要为确定几个可选条件的状态。例如，链路宽度可以从 1 通道到 32 通道，并且可以提供多种速度。链路训练过程将会发现这些可选项，并通过一个状态机来寻求一个建立最佳连接的组合，也就是说链路训练将会确认这些可选项具体的值（链路宽度、速率等）。在这个过程中，为了确保执行正确以及最优的操作，我们需要检查或者建立一些东西，例如：

- 链路宽度（Link width）
- 链路数据率（Link data rate）
- 通道调转（Lane reversal）—— 交换多个通道间的连接顺序
- 极性反转（Polarity inversion）—— 通道极性连接相反，TX_P 应当连接对端的 RX_P，若相反则变成了 TX_P 连接了对端的 RX_N。
- 每个通道的位锁定（bit lock）—— 恢复出发送方的时钟。
- 每个通道的符号锁定（symbol lock）—— 在比特流中找到一个可辨认的位置。
- 多通道链路中的 Lane-to-Lane 歪斜去除。

2.2.4.4 电气物理层（Physical Layer-Electrical）

物理的发送器和接收器是通过一个 AC 耦合链路相连接，如图2-29 所示。所谓“AC 耦合（交流耦合）”，意思是两个设备相连接的物理路径上放置有电容，它用于让信号的高频部分（AC 交流）通过，而阻塞低频（DC 直流）部分。许多串行传输方式中都使用了这种方法，因为它允许发送端和接收端的共模电压不同（共模电压指的是0、1电平交界处的电压），这意味着发送端和接收端可以使用不同的参考电压。当两个设备的物理连接距离很近时，这一特性优势并不明显，而若它们距离较远，例如位于不同的大楼里，那么它们将很难使用一个完全相同的公共的参考电压。

Figure 2-29: Physical Layer Electrical

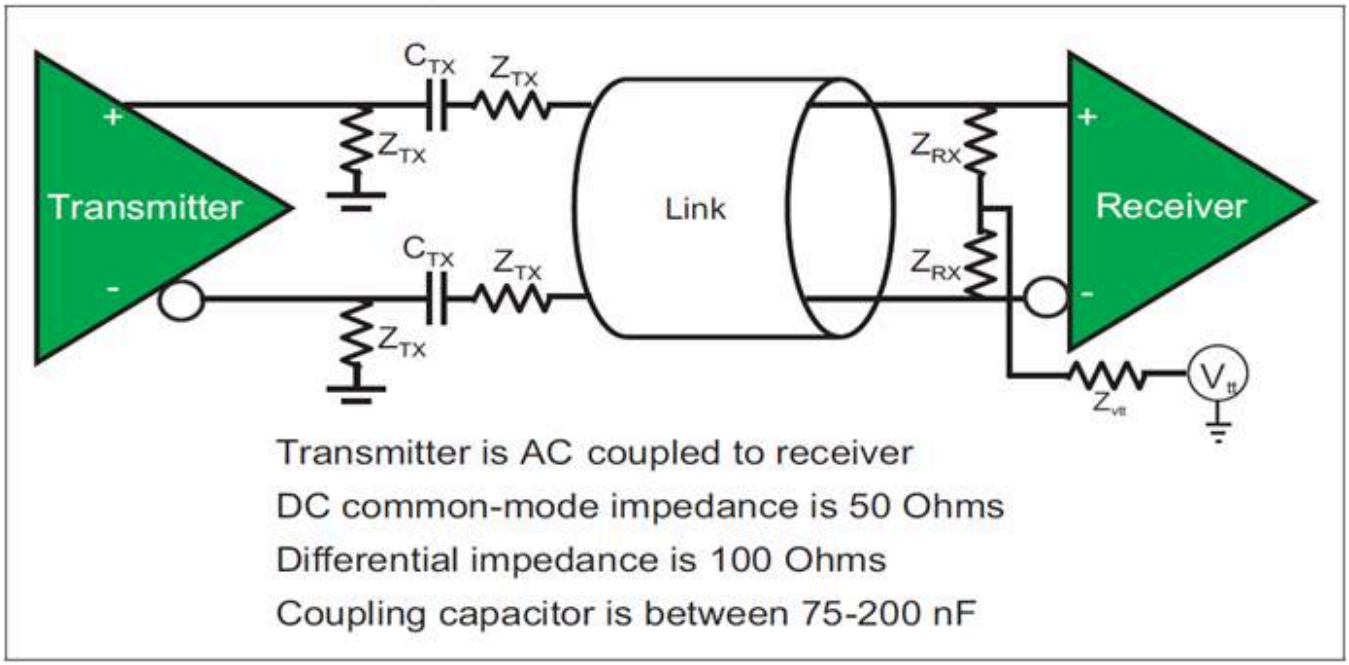

图2-29 电气物理层

2.2.4.5 字符集（Ordered Sets）

最后要介绍的一种只使用物理层在设备之间发送的流。尽管这种信息对于接收端来说很容易进行识别，但是它并没有被做成数据包的形式，原因是比如它没有包起始字符和包结束字符。因此作为一种替代方法，这种信息被组织成了一种被叫做“字符集（Ordered Sets）”的东西，它起始于发送端的物理层，终止于接收端的物理层，如图2-30 所示那样。对于 Gen1 和 Gen2 的数据率下，一个字符集使用一个单独的 COM 字符作为起始，然后后面接着 3 个或以上的其他字符用于定义要发送的信息。关于这些不同类型字符在 PCIe 中的命名的更细节的讨论请见“字符符号（Character Notation）”一节，对于现阶段来说只要知道 COM 字符的特性能让我们达到想要的设计目标即可。字符集的大小总是 4 字节的整数倍，图2-31 展示了一个字符集的例

子。在 Gen3 操作模式中，字符集的格式就不同于上述的 Gen1/Gen2 格式了。更多详细内容请见第十四章“链路初始化和训练”。字符集永远只会终止于链路的对端设备，而不会被交换机路由转发至 PCIe 网络中，也就是说对端设备如果是交换机那么它的最终目的地就是交换机。

Figure 2-30: Ordered Sets Origin and Destination

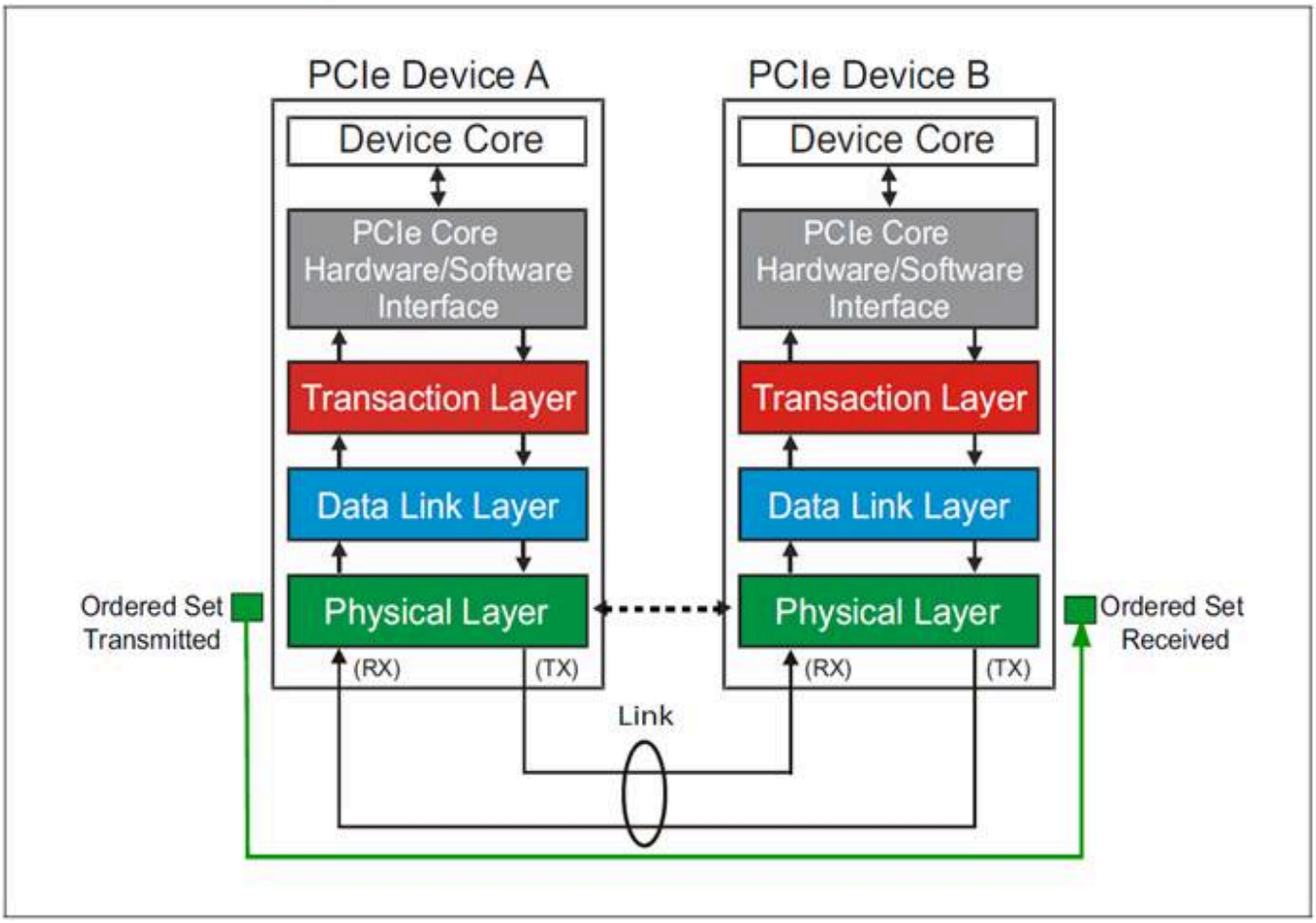

图2-30 字符集的出发地和目的地

字符集在链路训练过程中同样有应用，请参阅第十四章“链路初始化和训练”。它们也被用于补偿发送端和接收端内部时钟的轻微差异，这一过程被称为时钟容忍度补偿。最后一点，字符集还可以用于指示链路进入或者退出低功耗状态。

Figure 2-31: Ordered-Set Structure

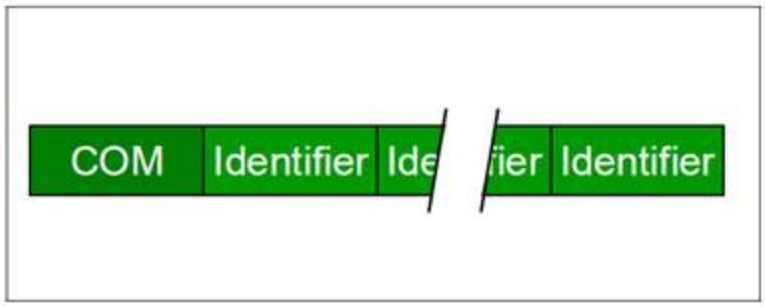

图2-31 字符集结构

2.3 协议回顾（Protocol Review Example)

现在，让我们通过一个示例来回顾整个链路协议，这个例子将从发起方发起一个内存读请求（MRd）开始，直到发起方从完成方获得所请求的数据为止，讲解这过程中所发生的操作步骤。

2.3.1 内存读请求（Memory Read Request)

关于我们开始讨论的第一部分，请参考图2-32。发起方的设备核心层或者说是软件层将会向事务层发起一个请求，这个请求中包含了这些信息：32 比特或 64 比特的内存地址、事务类型、需要读取的数量总量（以dw为单位计数）、流量类型、字节使能、属性等等。

Figure 2-32: Memory Read Request Phase

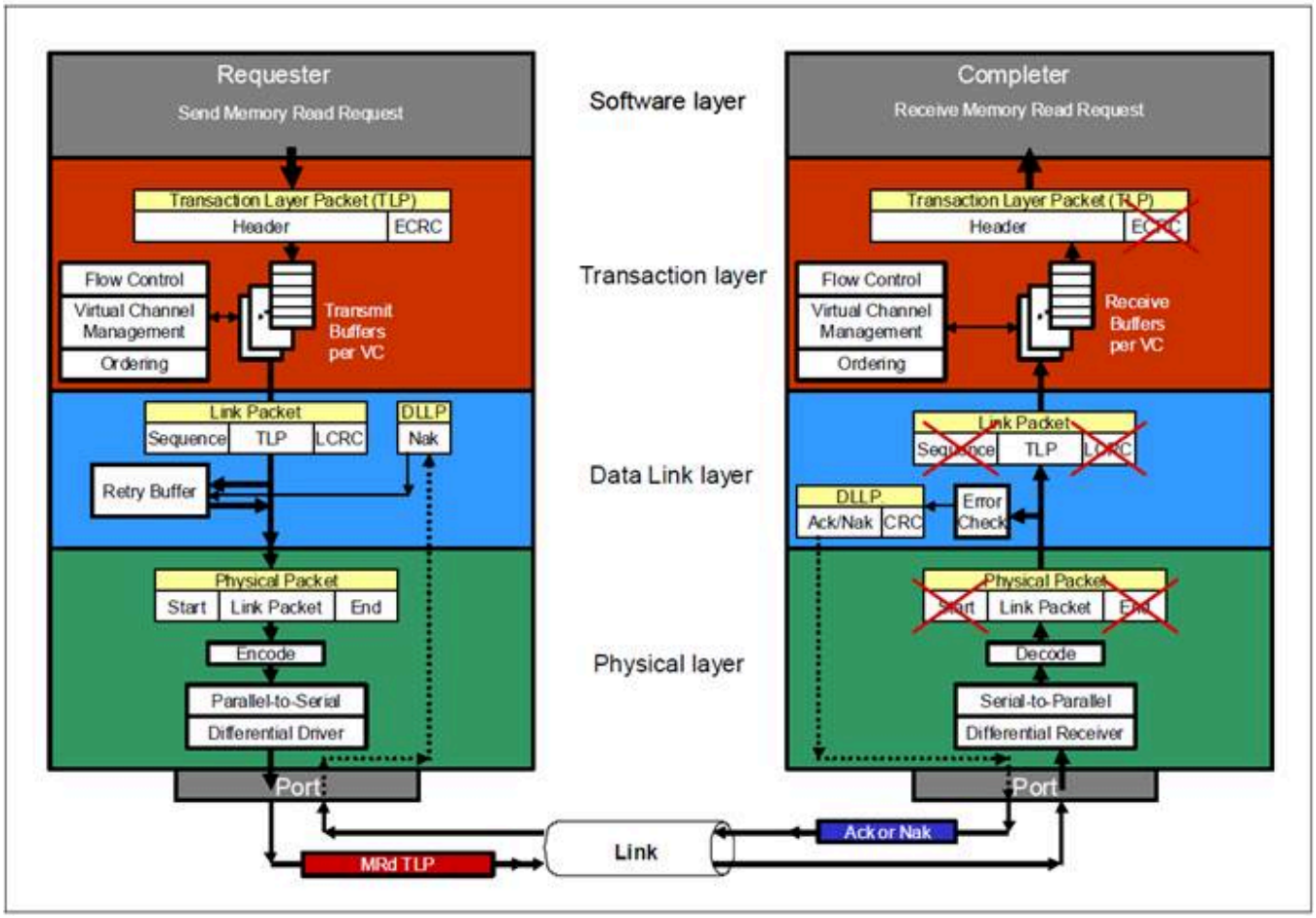

图2-32 内存读请求的各阶段

事务层使用上面所述的那些信息来构建一个 MRd TLP。关于 TLP 内部格式的细节我们稍后再进行描述，目前我们只需要知道 TLP 的 Header 大小为 3DW 或 4DW，这取决于它的地址字段的大小（32 位地址对应 3DW Header，64 位地址对应 4DW Header）。此外，在 Header 中还有事务层加入的发起方 ID字段（bus#，device#，function#），完成方可以通过这个发起方 ID 字段来返回完成包。TLP 随后被置入相应优先级的虚拟通道缓存中，等待轮到它被发送。一旦这个 TLP 被选中，流量控制逻辑将会确认对端设备的接收缓存（虚拟通道）有足够的可用空间来接收这个 TLP，然后 MRd TLP 就被向下发送至数据链路层。

在数据链路层中，数据包被加上 12 比特的序列号以及32 比特的 LCRC。并在重传缓存中保存这个TLP的一个副本，然后这个数据包就被向下转发至物理层。

在物理层中，数据包被加上包起始字符以及包结束字符，然后在所有可用的通道上进行字节条带化，再进行扰码、8b/10b 编码。最终这个数据包的比特在链路上的每个通道内以串行差分的形式传输到对端。

完成方接收到输入的比特流，它将这个比特流先进行串并转换将比特流恢复成 10 比特符号，然后让它们通过一个弹性缓存。随后 10 比特符号被解码后恢复成字节，然后每个通道上的字节都被解扰以及反条带化。接下来完成方物理层检测到包起始字符和包结束字符，并将它们从 TLP 中剥除。剩余的 TLP 被向上转发至数据链路层。

完成方的数据链路层将对接收到的 TLP 进行 LCRC 错误校验，并检查 TLP 序列号以确定是否存在 TLP 丢失或 TLP 失序。若这些步骤都无错，数据链路层将生成一个 Ack 信息，里面包含了与这个 MRd TLP 中相同的序列号。然后再给这个 Ack 信息加上 16 比特的 CRC，这样就组成了一个 Ack DLLP，将这个 Ack DLLP 送回物理层，由物理层加上相应的组帧符号，并把 Ack DLLP 传输给发起方。

发起方的物理层接收到 Ack DLLP 后，对其组帧符号进行检查和剥除，然后向上转发至数据链路层。如果数据链路层对其 CRC 校验无错，它将会用 Ack DLLP 内的序列号与重传缓存中保存的 TLP 副本的序列号进行比较，并将与 Ack DLLP 中序列号相匹配的 TLP 副本从重传缓存中清除。相反地，若发起方接收到的是一个 Nak DLLP，那么它将把序列号匹配的这个 MRd TLP 进行重传。由于 DLLP 仅对数据链路层有意义，所以在这些 DLLP 的操作中将不会有东西向上转发给事务层。

除了上述的产生 Ack DLLP 之外，完成方的数据链路层还将把 TLP 向上转发给事务层。在完成方的事务层中，TLP 被置于与其优先级相对应的虚拟通道接收缓存中，等待被处理。然后可以对 TLP 进行 ECRC 校验（ECRC 为可选项），若校验无错，那么 TLP Header 的内容（地址、发起方 ID、MRd 事务类型、请求数据总量、流量类型等）将被向上转发至完成方的软件层。

2.3.2 CpId（Completion with Data, 带有数据的完成包）

下面开始介绍此次举例回顾 PCIe 协议的另一半内容，请参考图2-33。为了服务 MRd 请求，完成方的设备核心层/软件层将会向它的事务层发送一个 CplD（Completion with Data，带有数据的完成包）请求，这个完成包请求中包含了与 MRd 请求中相同的发起方 ID、Tag，还包含了事务类型以及另外的完成包头内容，并且完成包中还包含了被请求的数据。

Figure 2-33: Completion with Data Phase

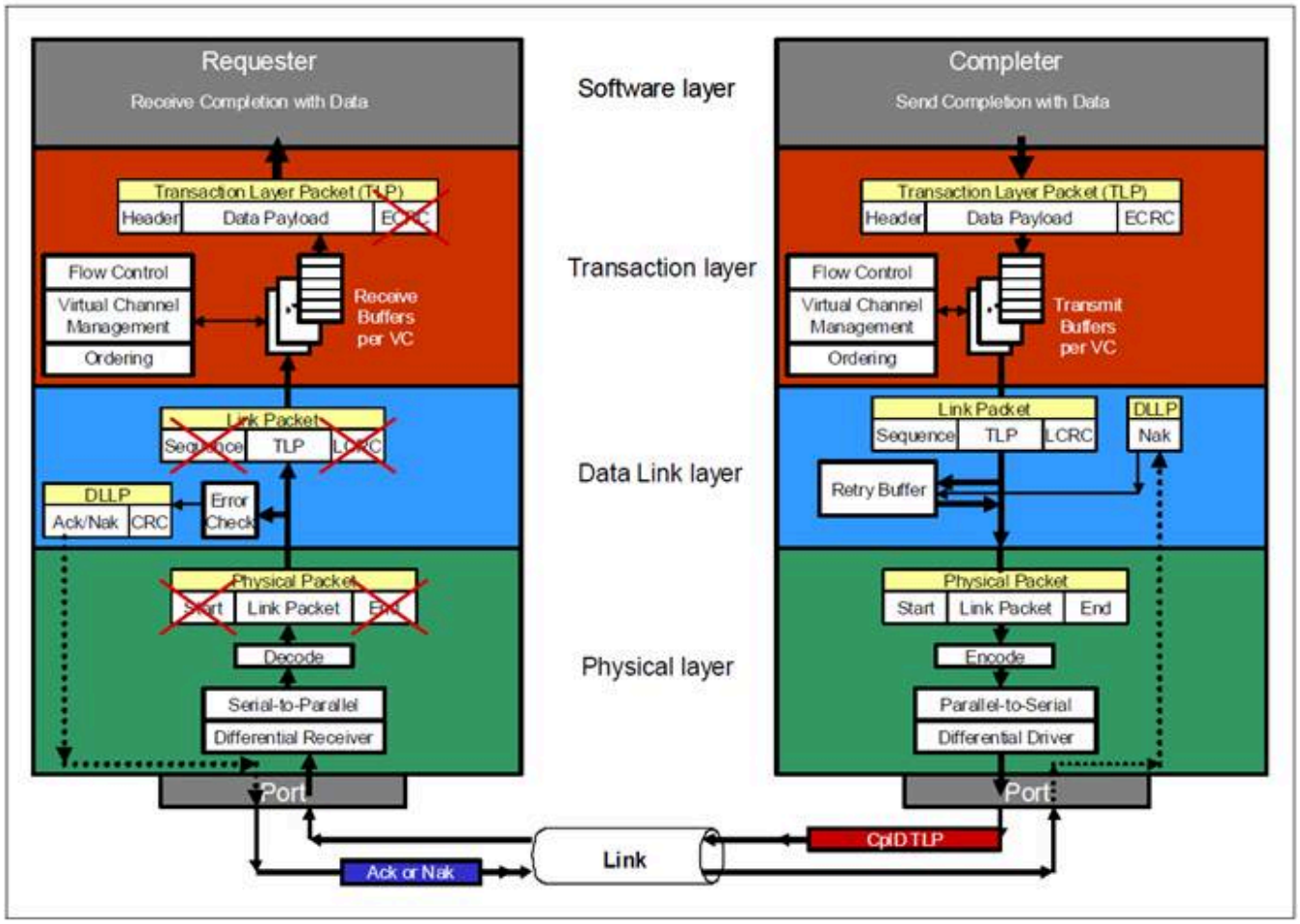

图2-33 带有数据的完成包的各阶段

事务层使用 CplD 请求中的这些信息来构建一个 CplD TLP，这种 TLP 的 Header 固定为 3DW（这是由于 CplD TLP 使用发起方 ID作为路由信息，因此不会需要用到 64 比特的地址）。事务层也会将自身的完成方 ID添加到 CplD TLP Header 中。这个数据包随后被放置入一个合适的虚拟通道的发送缓存中，一旦这个虚拟通道被仲裁选中并要发送这个 CplD TLP，流量控制逻辑将会确认对端设备有足够的可用缓冲区空间来接收它，当确认足够后则将这个 CplD TLP 向下转发至数据链路层。

如前文所述那样，数据链路层给数据包加上 12 比特的序列号和 32 比特的 LCRC。然后将添加完这些信息的 TLP 保存一份副本在重传缓存中，之后变将数据包向下转发至物理层。

依然如前文所述那样，物理层给数据包加上包起始字符和包结束字符，并将其在所有可用通道上进行字节条带化，再进行扰码、8b/10b 编码。最终，这个 CplD TLP 数据包在链路的所有通道上以串行差分的形式被传输至对端。

当发起方接收到这个 CplD TLP 的输入比特流时，它将比特流转换恢复成 10 比特符号（10bit symbol），并让 10bit 符号流通过一个弹性缓存。随后 10bit 符号被解码恢复为字节，并进行解扰和字节反条带化。然后物理层就能检测到这个 CplD TLP 的包起始字符和包结束字符，并将这两个字符剥去。之后便将剩余的 CplD TLP 向上转发至数据链路层。

和之前一样，数据链路层对 CplD TLP 进行 LCRC 校验，并检查序列号以确定是否存在 TLP 丢失或出现 TLP 失序。如果并未出现错误，数据链路层将产生一个 Ack DLLP，其中包含了与 CplD TLP 中相同的序列号，并给这个 Ack DLLP 加上 16 比特 CRC，然后将其送回给物理层加上相应的组帧字符并将这个 Ack DLLP 传输给完成方。

完成方的物理层将会检测并剥除 Ack DLLP 的组帧字符，并将剩余的部分向上转发给数据链路层，在这里对 Ack DLLP 的 CRC 进行校验。若校验无错，完成方的数据链路层将 Ack DLLP 中的序列号与重传缓存中的 TLP 的序列号进行对比。与 Ack DLLP 序列号相匹配的 TLP 将被从重传缓存中清除。反之，若完成方接收到的是一个 Nak DLLP，那么它将使用重传缓存中存储的 CplD TLP 副本来进行重传。

与此同时，发起方的事务层在某个虚拟通道的缓存中接收到了这个 CplD TLP。作为一个可选操作，事务层可以校验这个 TLP 的 ECRC。若校验无错，事务层将这个 CplD TLP 的 Header 内容、数据荷载以及请求完成状态信息向上呈交给发起方的软件层。至此，大功告成。

术语翻译

英文	翻译
Function	功能
Endpoint	端点
Bridge	桥
Dual-simplex	双单工
Link	链路
Lane	通道
bit	比特
Common Clock	公共时钟

英文	翻译
skew	偏斜
Differential Signaling	差分信号
forwarded clock	随路时钟
Recovered Clock	恢复时钟
deserializer	解串器
side-band control signal	边带控制信号
Switches	交换机
Root Complex	根组件
forward bridge	前向桥
reverse bridge	反向桥
legacy	传统
configuration header	配置首部
Device Core	设备核心层
Transaction Level	事务层
Data Link Level	数据链路层
Physical Level	物理层
Ordered set Packet	字符序列包
byte striping logic	字节条带化逻辑
Scrambler	扰码器
elastic buffers	弹性缓存
de-scrambler	解扰器
byte unstriping logic	字节反条带化逻辑

英文	翻译
Link Training and Status State Machine	链路训练状态机
Virtual Channel	虚拟通道
inbound packet	入站数据包
outbound packet	出站数据包
non-posted	非报告式
Split Transaction Protocol	拆分事务协议
requester	发起方
completer	完成方
framing	组帧
clock tolerance compensation	时钟容忍度补偿
Replay Buffer	重传缓存

原文： Mindshare

译者： Michael ZZY

校对： XTang

欢迎参与《Mindshare PCI Express Technology 3.0 一书的中文翻译计划》

https://gitee.com/ljgibbs/chinese-translation-of-pci-express-technology