



NVMe 科普教程

古猫 著

2017.7



目录

1.0 NVMe 概述	4
2.0 队列(Queue)管理	8
3.0 命令仲裁机制	15
4.0 寻址模型 PRP 和 SGL	19
5.0 中断机制	28
6.0 电源管理	34
7.0 E2E 数据保护	38



微信公众平台：NVMe 专题文章列表(点击即可跳转)

NVMe 系列专题之一：NVMe 技术概述

NVMe 系列专题之二：队列(Queue)管理

NVMe 系列专题之三：命令仲裁机制

NVMe 系列专题之四：寻址模型 PRP 和 SGL 解析

NVMe 系列专题之五：中断机制

NVMe 系列专题之六：电源管理

NVMe 系列专题之七：E2E 数据保护



1.0 NVMe 概述

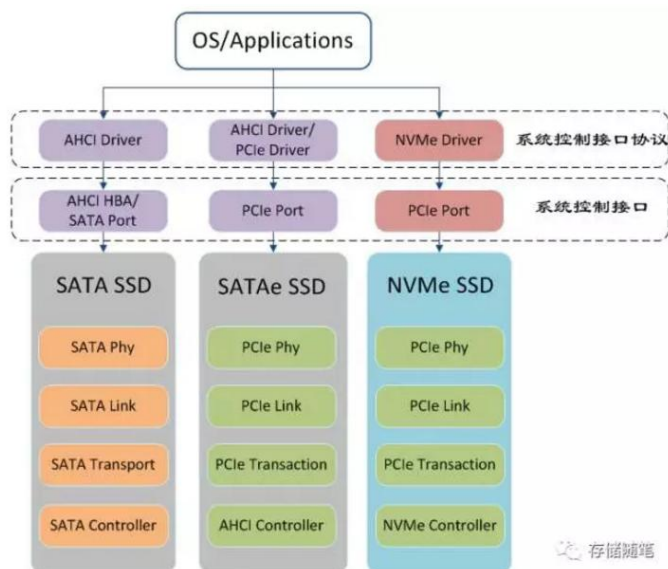
(1) NVMe 的诞生

在 NVMe 横空出世之前，硬盘的世界还是 AHCI 的天下。那么问题来了，AHCI 又是什么？AHCI，英文全名是 Serial ATA Advanced Host Controller Interface，中文名是“串行 ATA 高级主控接口”或者“高级主机控制器接口”，是 Intel 联合多家公司研发的系统接口标准。

在信息爆炸时代之前，SATA 在 AHCI 的领导下可谓是风生水起，青史留名。但是，SATA/AHCI 其实是为机械硬盘 HDD 而生的，其致命的缺陷就是传输速度有瓶颈，最大不超过 600MB/s。后来，AHCI 也意识到了自身的危机，并高薪挖来了另一位得力干将 PCIe。PCIe 呢，光芒万丈，AHCI 根本就驾驭不了，PCIe 对 AHCI 也是心生抱怨，AHCI 不能为 PCIe 提供施展才华的平台。在与 AHCI 搭档了很短的时间之后，PCIe 就萌生退意。

正当 PCIe 对这个世界开始失望的时候，PCIe 遇到了 NVMe。郁郁不得志之后，PCIe 终于等到了自己的伯乐。PCIe/NVMe 这对搭档在结合之后，展现了前所未有的能量，正在用他们的实力征服这个世界。

上面的一段“废话”叙述了 AHCI，NVMe，SATA，PCIe 相互之间的关系，画了张图，方便大家理解：



NVMe 最初的名字叫 NVMHCI，是在 2007 年的 Intel 开发者论坛 (IDF) 上被提出来的，并在同年 Intel 牵头成立 NVMe 开发工作组。在这个工作组中有 13 个主导公司 (如下图) 和近百个成员。





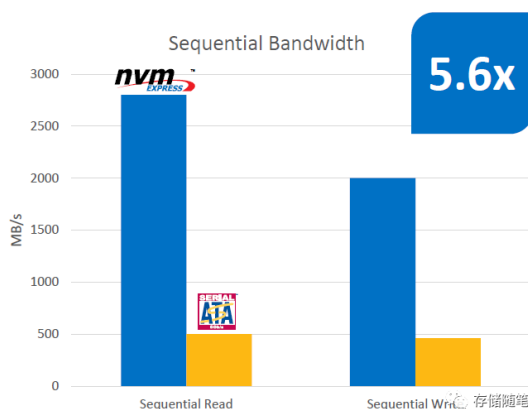
(2) NVMe 的特点与优势

NVMe 主要是面向 PCIe SSD 开发的一套接口标准(注意，也不仅仅局限在 SSD，对其他 NVM 存储依然很有前景，本文还是主要针对 PCIe SSD)。NVMe 定义了系统接口和命令集。其目的就是性能更好，延迟更低，功耗更低。我们先对比一下 AHCI 与 NVMe 特点，看图(来自 Intel FMS 2012)。

	AHCI	nvm EXPRESS
Uncacheable Register Reads Each consumes 2000 CPU cycles	4 per command 8000 cycles, ~ 2.5 μ s	0 per command
MSI-X and Interrupt Steering Ensures one core not IOPs bottleneck	No	Yes
Parallelism & Multiple Threads Ensures one core not IOPs bottleneck	Requires synchronization lock to issue command	No locking, doorbell register per Queue
Maximum Queue Depth Ensures one core not IOPs bottleneck	1 Queue 32 Commands per Q	64K Queues 64K Commands per Q
Efficiency for 4KB Commands 4KB critical in Client and Enterprise	Command parameters require two serialized host DRAM fetches	Command parameters in one C.B fetch

NVMe 的速度优势

对用户来说，SSD 的数据传输速度体验是第一感官。
先来看一下 NVMe/PCIe 与 AHCI/SATA 的对比数据(来源: Intel)。下面是 Intel NVMe SSD P3700 系列与 Intel SATA SSD S3700 系列的 128K 顺序读写速度对比，可以看到，NVMe SSD 的读写性能有近 6 倍的提升。



我们再来看另外一组 NVMe/PCIe 与 AHCI/PCIe 的速度对比。数据同样来自 Intel。从下图中可以看出，NVMe SSD 相对 AHCI/PCIe SSD 在读写性能上有 2~3 倍的提升。



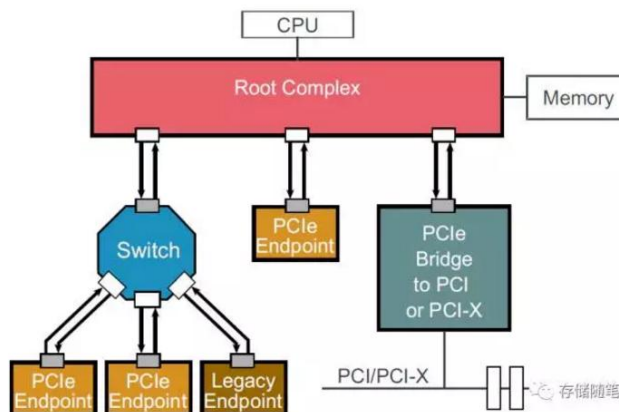
经过两组数据的对比，可以证明 NVMe/PCIe 是目前最强的组合。



NVMe 的低延迟优势

与 AHCI 相比，NVMe 在两个方面的作出了努力降低存储系统延迟：

如前面所讲，NVMe 主要服务 PCIe SSD，而 **PCIe 通道不需要像 SATA 一样连接到南桥中转，可以直接通往 CPU**(这里的 CPU 确切的说是 Root Complex)，高铁直达，延迟降低了一部分；



NVMe 如总理一样坚持“简政放权”，不给群众添麻烦。所以 **NVMe 执行命令过程被彻底简化，直接执行，不用再读取寄存器**。但在之前的 AHCI 中，执行命令时则要读取 4 次寄存器，这个过程会造成 2.5μs 的延迟。

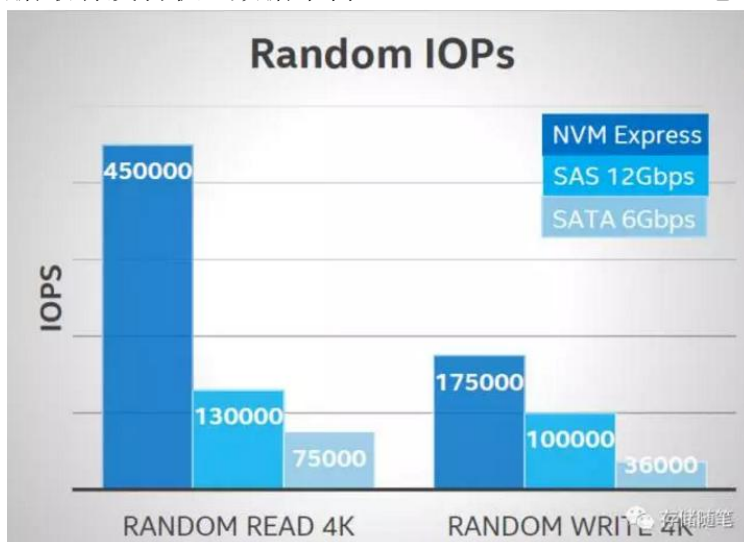
NVMe 的 IOPS 优势

理论上 IOPS 与队列深度 (Queue Depth) 和 IO 延迟有关，用数学表达式是：

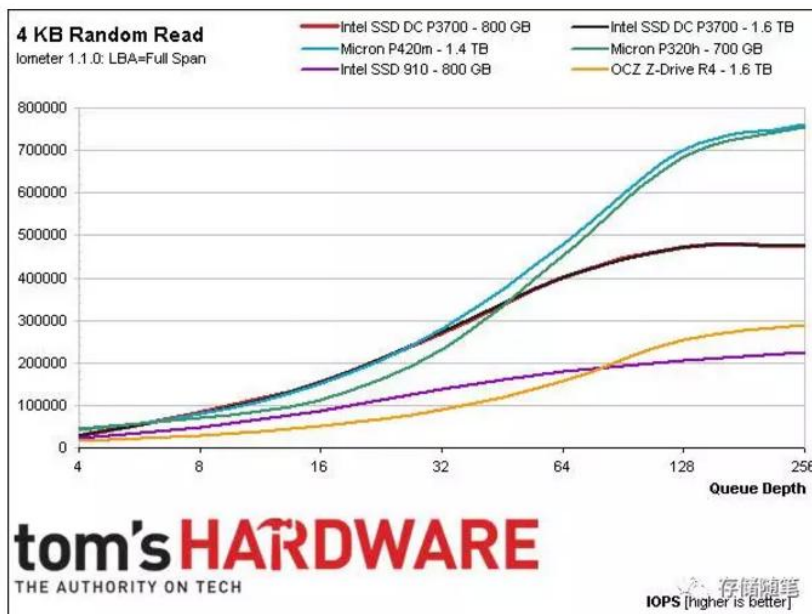
$$IOPS = \text{队列深度} / \text{IO 延迟}$$

从上述的表达式中，我们可以看出 IOPS 与队列深度有很大的关系。但是有一点需要注意：实际应用中，随着队列深度的增加，IO 延迟也会相应的变大。AHCI 中，只支持一个队列，并且队列深度只有 32。而 NVMe 支持 64K 个队列，每个队列的深度可达 64K。这样一对比，AHCI 真是弱爆了~不怪 PCIe 郁郁不得志呢~

再来张图，数据最有发言权（数据来自 Intel IDF 2015）。NVMe 绝对碾压 SATA。



其实，如果要完全释放 NVMe SSD 的 IOPS 性能，设计人员要先做好评估呢，因为当队列深度达到一定程度后，NVMe SSD 的 IOPS 才会达到最佳。如下图(来自 Tom' Hardware)，六块 NVMe SSD 均在队列深度 128 以上才达到最佳的性能。



NVMe 白皮书中对企业级 SSD 和消费级 SSD 设计时需要的队列深度的建议是：

企业级 SSD: 16~128 Queues;

消费级 SSD: 2~8 Queues.

Feature	Enterprise Recommendation	Client Recommendation
Number of Queues	16 to 128	2 to 8
Physically Discontiguous Queues	Implementation choice	No
Logical Block Size	4KB	4KB
Interrupt Support	MSI-X	MSI-X
Arbitration Mechanism	Weighted Round Robin with Urgent Priority Class or Round Robin	Round Robin
PCIe Advanced Error Reporting	Yes	Yes
Firmware Update	Required	Required
End-to-end Data Protection	Yes	No
SR-IOV Support	Yes	No
Security Send & Receive	Yes	Yes

NVMe 的低功耗优势

NVMe 中加入了自动电源状态转换和动态电源管理机制。NVMe Spec 支持 1-32 电源状态 (PS, Power State)。在 Host 开启自动电源状态转换功能时，可以根据自己喜好设置 Idle 多长时间后自动转换其他电源状态。比如，我们定义了 6 个 PS，并且定义在 Idle (PS0) 10ms 之后进入 PS1，Idle 50ms 进入 PS2，Idle 100ms 进入 PS3...这些都是自定义的。如果 Host 未开启自动电源状态转换功能，也可以通过下发 Set feature 命令进行电源状态的转换。

Power State	Maximum Power (MP)	Entry Latency (ENTLAT)	Exit Latency (EXLAT)	Relative Read Throughput (RRT)	Relative Read Latency (RRL)	Relative Write Throughput (RWT)	Relative Write Latency (RWL)
0	25 W	5 μ s	5 μ s	0	0	0	0
1	18 W	5 μ s	7 μ s	0	0	1	0
2	18 W	5 μ s	8 μ s	1	0	0	0
3	15 W	20 μ s	15 μ s	2	0	2	0
4	10 W	20 μ s	30 μ s	1	1	3	0
5	8 W	50 μ s	50 μ s	2	2	4	0
6	5 W	20 μ s	5000 μ s	4	3	5	1

不过，自动电源状态切换功能一般只用在消费级 SSD 上，对笔记本电脑的续航问题有很大的帮助。在企业级 SSD 中数据的安全性还是第一位的，不大会考虑功耗的问题。



2.0 队列(Queue)管理

在上一篇文章([NVMe 系列专题之一：NVMe 技术概述](#))中，我们提到了 NVMe 有一个很大的优势就是队列深度达到了 64K，并且支持队列个数最大可达 64K。所以呢，这里我们就先聊聊 NVMe 中队列相关的一些知识点。

队列，在 NVMe 协议中，是专门为 NVMe 命令服务的。介绍队列之前，我们还是先看看 NVMe 定义的命令种类：

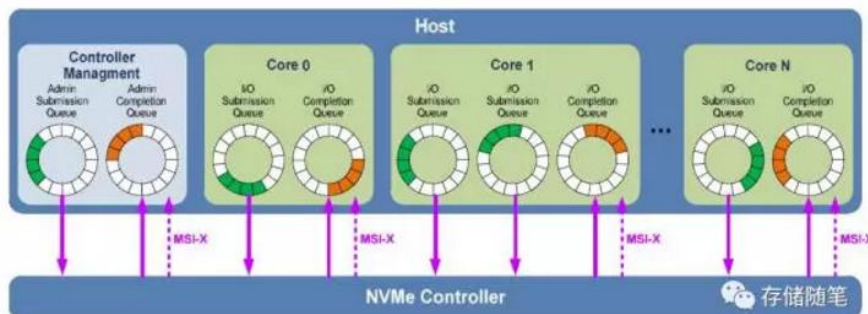
NVMe 定义的命令很简单，只有两种：**Admin Command** 和 **IO Command**。加起来总共只要求 13 个(10Admin Commands + 3 IO Commands)，相比于 ATA Commands，真是一下子清爽了很多~

Admin Commands	NVM I/O Commands
Create I/O Submission Queue	Read
Delete I/O Submission Queue	Write
Create I/O Completion Queue	Flush
Delete I/O Completion Queue	Write Uncorrectable (optional)
Get Log Page	Compare (optional)
Identify	Dataset Management (optional)
Abort	Write Zeros (optional)
Set Features	Reservation Register (optional)
Get Features	Reservation Report (optional)
Asynchronous Event Request	Reservation Acquire (optional)
Firmware Activate (optional)	Reservation Release (optional)
Firmware Image Download (optional)	
Format NVM (optional)	
Security Send (optional)	
Security Receive (optional)	

存储随笔

- ✧ 当 Host 要下发 Admin command 时，需要一个放置 Admin command 的队列，这个队列就叫做 Admin Submission Queue，简称 **Admin SQ**。
- ✧ Device 执行完成 Admin command 时，会生成一个对应的 Completion 回应，此时也需要一个放置 Completion 的队列，这个队列就叫做 Admin Completion Queue，简称 **Admin CQ**。

同样，执行 IO Command 时，也会有对应的两个队列，分别是 **IO SQ** 和 **IO CQ**。



NVMe Spec 对 Admin SQ/CQ 和 IO SQ/CQ 有不同的约定：

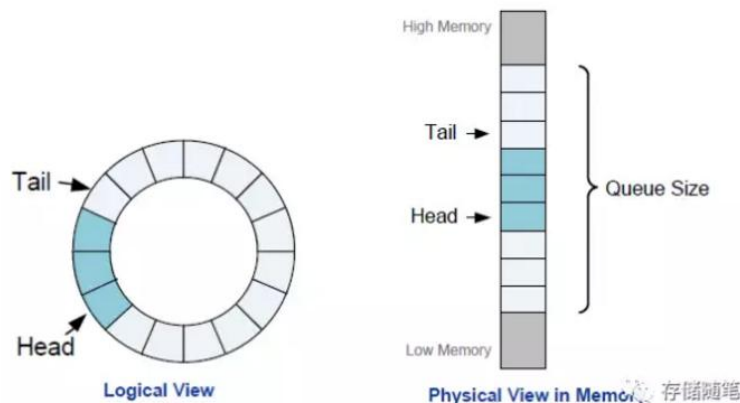
- ✧ 系统中只有一对 Admin SQ/CQ，则可以有最多 64K 对 IO SQ/CQ；
- ✧ Admin SQ/CQ 的队列深度是 2~4K；而 IO SQ/CQ 的队列深度是 2~64K；

注：Admin/IO command 大小为 64B，对应的 Completion 大小为 16B。



- ✧ Admin SQ 和 CQ 是一对的，而 IO SQ 和 CQ 可以一对一，也可以多对一。多个 SQ 可以支持多线程工作，不同 SQ 之间可以赋予不同的优先级；
- ✧ Admin 和 IO 的 SQ/CQ 均放在 Host 端 Memory 中；
- ✧ SQ 由 Host 来更新，CQ 则由 NVMe Controller 更新。

SQ/CQ 是队列，那就应该有队列的头(Head)和尾(Tail)。在 NVMe Spec 中定义 SQ/CQ 均是循环队列，可以理解为一个圆形(如下图左)，但是内存中实际的长条状的(如下图右)，其实，队列可以是连续的物理空间，也可以不连续。



Tail: 指向队列中的下一个空位；

Head: 指向下一个将要被执行的命令所在的位置。

- ✧ 当队列为空时，Head==Tail；
- ✧ 当队列为满时，Head==Tail+1；

有了 SQ/CQ 是不是就可以执行 Command 了呢？

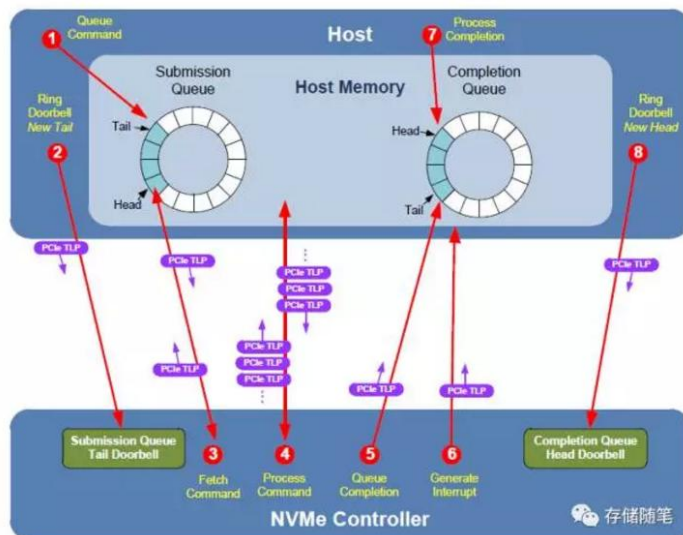
我们先看一下 NVMe Spec 中 Command 执行的整个流程再回过来审视一下这个问题。

在 NVMe Spec 中 Command 执行的流程有八步，Host 与 Controller 之间用 PCIe TLP 传递信息。

- 1) **Host 提交新的 Command**。Host 下发一个新 Command 时，将其放入 Host 内存中 SQ；
- 2) **Host 通知 Controller 提取 Command**。Host 把 Command 写入 SQ 之后，此时 Device 并不知道这件事。所以，Host 此时需要给 Controller 发信息，通知 NVMe Controller：“我提交了新的命令请求，麻烦尽快帮忙处理！”。这个过程通过更新在 Controller 内部的寄存器 **SQ Tail Doorbell** 来完成。
- 3) **NVMe Controller 从 SQ 提取 Command**。取走 Command 之后，需要在 Controller 内部的 **SQ Head Pointer** 寄存器中更新 Head 所在的位置。NVMe 没有规定 Command 存入队列的执行顺序，Controller 可以一次取出多个 Command 进行批量处理。
- 4) **NVMe Controller 执行从 SQ 提取的 Commands**。一个队列中的 Command 执行顺序是不固定的（可能导致先提交的请求后处理），这涉及到 NVMe Spec 定义的命令仲裁机制，在后续文章中介绍。执行 Read/Wirte Command 时，这个过程也会与 Host Memory 进行数据传递。
- 5) **NVMe Controller 将 Commands 的完成状态写入 CQ**。此时，Controller 需要更新 **CQ Tail Pointer** 寄存器。



- 6) **NVMe Controller 通知 Host 检查 Commands 的完成状态。** Controller 通过发送一个中断信息告知 Host:“您提交的 Commands, 我已经执行完毕了, 请您检查结果!”。
- 7) **Host 检查 CQ 中的 Completion 信息。**
- 8) **Host 告知 Controller 已处理完成 Completion 信息。**此时, Host 更新 Controller 内部的 **CQ Head Doorbell**。告知 Controller:“您发送回来的 Command 执行结果, 我已处理完毕, 非常感谢!”。



看完上面 NVMe Command 执行流程之后,我们再回过头来看一下刚才我们的问题:“**有了 SQ/CQ 是不是就可以放心的执行 Command 了呢?**”。

答案是否定的。从上述 Command 执行流程中,我们发现除了 SQ/CQ 之外,还有两个关键的“人物”: **PCIe TLP** 和 **寄存器**。

在之前的 PCIe 专题 ([PCIe 系列专题之二: 2.2 TLP 事务处理方式解析](#)) 中,我们有介绍过 PCIe TLP 的类型有很多,如下图,不过, NVMe 只挑选了 **Memory Read/Wirte** 传递信息。

Transaction 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 1)	Non-posted
Configuration Write (Type 0 and 1)	Non-posted
Message	Posted
AtomicOp	Non-Posted

注: Non-Posted: 需要 completion 返回响应包; Posted: 不需要 completion 返回响应包



NVMe Command 执行过程中所需的寄存器有两种：**Doorbell Register** 和 **Pointer Register**。Doorbell, 可以简称 DB, 是 NVMe Spec 定义的寄存器; Pointer register 是主控厂商自定义的寄存器，归纳一下：

寄存器	用途	位置	Update By	Defined By
SQ Tail Doorbell	记录 SQ Tail 的位置	Controller Memory	Host	NVMe Spec
SQ Head Pointer	记录 SQ Head 的位置	Controller Memory	Controller	Vendor Specific
CQ Tail Pointer	记录 CQ Tail 的位置	Controller Memory	Controller	Vendor Specific
CQ Head Doorbell	记录 CQ Head 的位置	Controller Memory	Host	NVMe Spec

从上表的信息中，不知道聪慧的你，有没有发现：

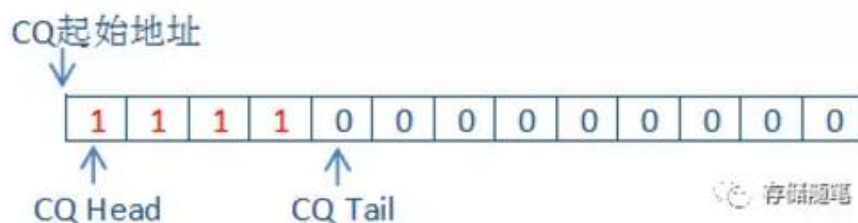
- 1) 这个四个寄存器全部放在 Controller 内存中。也就是说 Controller 知道这 SQ Tail/Head 和 CQ Tail/Head 的全部信息。
- 2) 而 Host 仅仅知道自己更新的两个信息 SQ Tail 和 CQ Head。

很显然，Host 与 Controller 之间的信息是不对等的。不过，还好 Controller 是个乐于分享的人。Controller 会与 Host 共享自己所知的信息。那 Controller 怎么把 SQ Head 和 CQ Tail 的信息告知 Host 呢？

不怕，聪明如你！Controller 把 SQ Head 和 CQ Tail 的信息写入了 Completion 报文中，如下图：

	31	23	15	7	0
DW0	Command Specific				
DW1	Reserved				
DW2	SQ Identifier		SQ Head Pointer		
DW3	Status Field		P	Command Identifier	

- ✧ SQ Head Pointer 就是 SQ Head 的信息。
- ✧ P 代表着 Phase bit。CQ 队列中所有的位置会被初始为 0，当有新的 Completion 信息写入时，Phase bit 被置为 1。Host 通过读取 Host 内存中 CQ 的 Phase bit 信息就能判断中 CQ Tail 的位置。



前面说了理论，我们还是通过真实的 NVMe/PCIe Trace 再回顾一下 CMD 执行流程，以 Admin Command: **Set Feature** 和 IO Command: **Read** 为例：

Set Feature

首先，看全局，如下图，



Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBel		MW; ASubmQBel; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MW; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP	ACpiQBel		MW; ACpiQBel; ACQHead = 0x0000001A;	24	0000001E

结合前面介绍的 Command 执行流程的八步分解 Trace:

(1) 第一步是 Host 向 Host 内存中的 SQ 写入新的 Command。因为这部分是在 Host 内部执行的，所以在 PCIe Trace 中没有体现。

(2) 第二步是 Host 更新 Controller 内存中的 SQ Tail DB，告知 Controller 过来提取 Command。

PCIe 通过 Memory Write TLP(Posted)完成 Host 对 SQ Tail DB 的更新。从下图的 Trace 中，我们可以看到 SQ Tail=0x1E (先记住这个值哈，后面有用)。SQ Tail DB 在 Controller 内存中的位置=0xFB301000。

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBel		MW; ASubmQBel; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MW; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP	ACpiQBel		MW; ACpiQBel; ACQHead = 0x0000001A;	24	0000001E
PCIe Expert Target	XPT		NVMe Exchange	NVMe Exchange; Set Features; Good Completion;	72	00000000

Transaction Layer Packet (TLP)

STP = 0x6F802995

TLP Header

Fmt/Type = 0x40 MW

Flags = 0x000001

Source ID = 0x0000

Tag = 0x00

Byte Enables = 0x0F

Address = 0xFB301000 ASubmQBel

PH = 0x0 Off

NVMe Admin Submission Queue Doorbell

Admin Submission Queue Tail = 0x0000001E Entry

End of Packet

LCRC = 0x6E064853 (Correct)

(3) 第三步是 Controller 去 Host 内存中的 SQ 中取回 Command。Controller 通过发送 Memory Read TLP (Non-Posted) 从 Host 内存提取 Command。SQ 在 Host 内存中的位置=0x10040C740。读出数据的长度=0x40 (64)，这个也是 NVMe 规定 Command 的长度 64B。

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBel		MW; ASubmQBel; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MW; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP	ACpiQBel		MW; ACpiQBel; ACQHead = 0x0000001A;	24	0000001E
PCIe Expert Target	XPT		NVMe Exchange	NVMe Exchange; Set Features; Good Completion;	72	00000000

Transaction Layer Packet (TLP)

STP = 0x6F802F41

TLP Header

Fmt/Type = 0x20 MRd

Flags = 0x000010

Source ID = 0x0200 Micron:5402

Tag = 0x00

Byte Enables = 0xFF

Address = 0x10040C740 ASubmQ

PH = 0x0 Off

End of Packet

LCRC = 0x882E55F2 (Correct)



因为 Memory Read 是 Non-Posted TLP，所以 Host 会发回一个 Memory Read 对应的 Completion TLP。（需要注意的是在 Xgig PCIe 设备中，此时 MRd 的 CpID 被叫做 ASubmQ，实际就是 CpID）

从 CpID 中包含取回 Command 的一些信息，比如这个 Set feature 命令是为了改变 NVMe 设备的 Power state.

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBell		MWw; ASubmQBell; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MWw; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP	ACpiQBell		MWw; ACpiQBell; ACQHead = 0x0000001A;	24	0000001E
PCIe Expert Target	XPT		NVMe Exchange	NVMe Exchange; Set Features; Good Completion;	72	00000000

Transaction Layer Packet (TLP)

STP = 0x5F011996

TLP Header

Fmt/Type = 0x4A CpID

Flags = 0x000010

Source ID = 0x0000

Completion Status = 0x0 Cpl Success

BCM = 0x0 Off

Byte Count = 0x040 Bytes

Destination ID = 0x0200 Micron:5402

Tag = 0x00

Lower Address = 0x40

NVMe Admin Submission Queue Entry

Opcode = 0x09 Set Features (Interesting Event Found)

PRP or SGL = 0x0 PRP

FUSE = 0x0 No

CID = 0x001D

NSID = 0x00000000 Not Used

Metadata Pointer = 0x0000000000000000

PRP Entry 1 = 0x0000000000000000

PRP Entry 2 = 0x0000000000000000

Feature Identifier (FID) = 0x02 Power Mgmt

Save = 0x0 Off

Power State = 0

End of Packet

LCRC = 0xCB51E9F6 (Correct)

(4) 第四步是在 Controller 内部开始执行上一步取回的 Set feature 命令。此时在 PCIe 链路中没有交互，所以在 PCIe Trace 中看不到执行过程的 Trace。

(5) 第五步是 Controller 更新 CQ，反馈 Set feature 命令执行结果。

Controller 发送 Memory Write TLP 将 set feature 命令执行结果写入 CQ。此时 CQ 在 Host 内存中的位置=0x10041090。同时在告知 Host SQ Head 的位置=0x1E。还记得第二步中 SQ Tail=0x1E 这个信息吗？Head==Tail 就说明了 SQ 现在空了。此外，还有一个信息就是 Phase Tag=1，代表 Host CQ 中有新增 Completion 信息。

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBell		MWw; ASubmQBell; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MWw; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP	ACpiQBell		MWw; ACpiQBell; ACQHead = 0x0000001A;	24	0000001E
PCIe Expert Target	XPT		NVMe Exchange	NVMe Exchange; Set Features; Good Completion;	72	00000000

Transaction Layer Packet (TLP)

STP = 0xAF003F42

TLP Header

Fmt/Type = 0x60 MWw

Flags = 0x000004

Source ID = 0x0200 Micron:5402

Tag = 0x00

Byte Enables = 0xFF

Address = 0x100410190 ACpiQ

PH = 0x0 Off

NVMe Admin Completion Queue Entry

SQHD = 0x001E

SQID = 0x0000

CID = 0x001D

StatusType/Code = 0x0000 Success

Phase Tag = 0x1 On

Do Not Retry = 0x0 Off

More = 0x0 Off

End of Packet

LCRC = 0x0FB3E76F (Correct)



(6) 第六步是 Controller 通知 Host 检查 Set feature 命令执行结果。这个过程中，Controller 通过发送 MSI-X 中断告知 Host 去检查 CQ 中的返回结果。

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBell		MWrr; ASubmQBell; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MWrr; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP		ACpiQBell	MWrr; ACpiQBell; ACQHead = 0x0000001A;	24	0000001E
PCIe Expert Target	XPT		NVMe Exchange	NVMe Exchange; Set Features; Good Completion;	72	00000000

Transaction Layer Packet (TLP)

STP = 0x6F802F43

TLP Header

Fmt/Type = 0x40 MWrr

Flags = 0x000001

Source ID = 0x0200 Micron:5402

Tag = 0x00

Byte Enables = 0x0F

Address = 0xFEE0100C MSI-X Interrupt

PH = 0x0 Off

MSI-X Interrupt

Message Data = 0x00004990

End of Packet

LCRC = 0x7A33BEB2 (Correct)

(7) 第七步是 Host 检查 CQ 中的 Set feature 命令执行结果。这个过程时在 Host 内部实现的，在 PCIe Trace 中也没有体现。

(8) 第八步是 Host 更新 Controller 内存中的 CQ Head DB，告知 Controller：“您的完成报告我已经处理完了，非常感谢您！”从 Trace 中可以看到，CQ Head 的位置=0x1A。CQ Head DB 在 Controller 内部的位置=0xFB301004。

Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	ASubmQBell		MWrr; ASubmQBell; ASQTail = 0x0000001E;	24	0000001E
PCIe Target	TLP		MRd	MRd; ASubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	ASubmQ		Set Features; Power Mgmt; Power State = 0;	84	0000001E
PCIe Target	NVMe		ACpiQ	Success;	40	0000001E
PCIe Target	TLP		MSI-X Interrupt	MWrr; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP		ACpiQBell	MWrr; ACpiQBell; ACQHead = 0x0000001A;	24	0000001E

Transaction Layer Packet (TLP)

STP = 0x6F802997

TLP Header

Fmt/Type = 0x40 MWrr

Flags = 0x000001

Source ID = 0x0000

Tag = 0x01

Byte Enables = 0x0F

Address = 0xFB301004 ACpiQBell

PH = 0x0 Off

NVMe Admin Completion Queue Doorbell

Admin Completion Queue Head = 0x0000001A Entry

End of Packet

LCRC = 0x00CC5412 (Correct)

Read

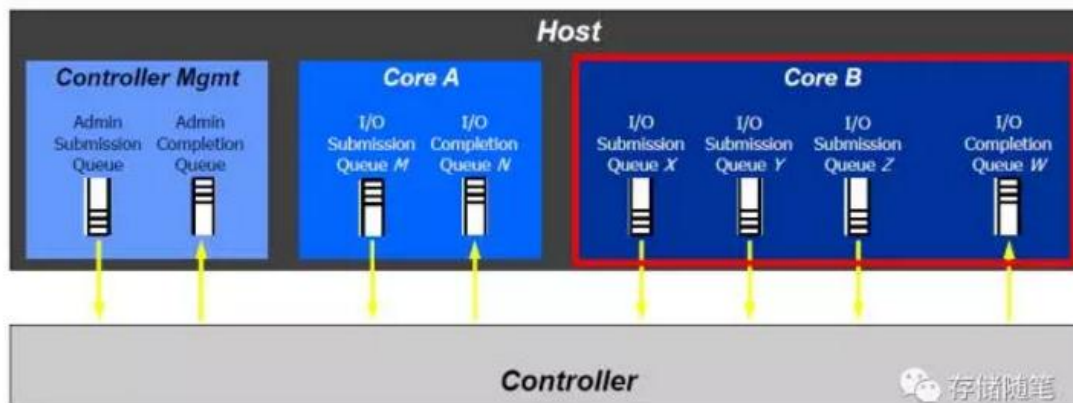
Port	Protocol	Down	Up	Summary	Bytes	Tag
PCIe Host	TLP	SubmQBell		MWrr; SubmQBell; SQTail = 0x00000001;	24	00010001
PCIe Target	TLP		MRd	MRd; SubmQ; Len = 0x0040;	24	00
PCIe Host	NVMe	SubmQ		Read; NSID = 0x00000001; LBA = 0x0000000000000000;	84	00010001
PCIe Target	NVMe		Data	Data; Len = 0x0100;	280	000100011
PCIe Target	NVMe		Data	Data(0x100); Len = 0x0100;	280	000100011
PCIe Target	NVMe		CpiQ	Success;	40	000100011
PCIe Target	TLP		MSI-X Interrupt	MWrr; MSI-X Interrupt;	24	0000FFFF
PCIe Host	TLP		CpiQBell	MWrr; CpiQBell; CQHead = 0x00000001;	24	00010001

I/O command 是与 Admin command 处理的流程基本一致，有一点不同的是：I/O Command 处理过程中会涉及到数据的传输。在这里就不展开解析咯~



3.0 命令仲裁机制

在 NVMe Spec 没有规定 Command 存入 SQ 队列的执行顺序，Controller 可以一次取出多个 Command 进行批量处理。一个 SQ 队列中的 Command 执行顺序是不固定，同时在多个 SQ 队列之间的 Command 执行顺序也不固定，这就涉及到了 NVMe Spec 定义的命令仲裁机制。



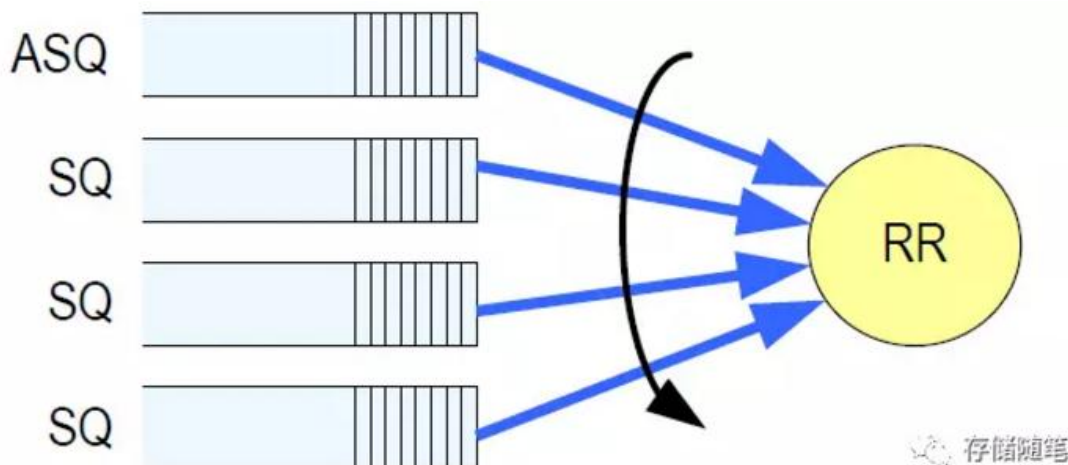
NVMe Spec 中主要定义了两种命令仲裁机制：[循环仲裁\(Round Robin Arbitration\)](#)和[加权循环仲裁\(Weighted Round Robin Arbitration\)](#)。

- ✧ 对于循环仲裁，所有主控必须支持。
- ✧ 对于加权循环仲裁，主控可以选择性支持。
- ✧ 除了这两个仲裁机制以外，主控设计人员还可以自行定义仲裁机制。

(1). 循环仲裁(Round Robin Arbitration)

当 NVMe 设备选择 RR 仲裁机制时，所有 SQs(包括 Admin Command SQ 和 IO Command SQ)都会执行 RR 仲裁。此时，所有 SQs 的级别一样高，按照顺序依次从所有 SQs 中分别取出一定数目的 Commands(如下图)。

这里的“一定数目”用参数 **Arbitration Burst** 定义，代表了一次从 SQ 中取 Commands 的数目。Arbitration Burst 的值可以通过 Set feature 定义。



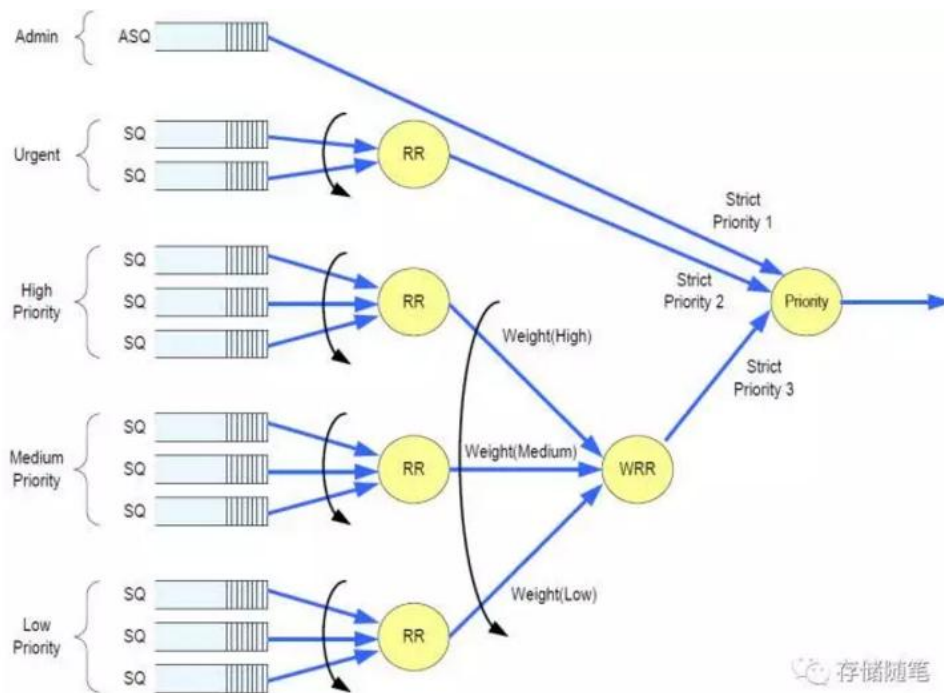


2. 加权循环仲裁(Weighted Round Robin Arbitration)

加权循环仲裁(WRR)机制定义了 **3 个绝对优先级**和 **3 个加权优先级**。

- ✧ **Admin Class:** 只有 Admin SQ 具有这一最高优先级。也就是说，Admin Command 必须最先被执行。
- ✧ **Urgent Class:** 一人之下，万人之上。优先级仅次于 Admin Class。被赋予 Urgent Class 优先级的 IO SQ 在 Admin SQ 中的命令执行后，紧接着执行。
- ✧ **WRR Class:** 最低绝对优先级。包含了三个加权优先级: **High, Medium, Low**.

用户可以用个 `set feature` 命令控制加权优先级中每个优先级的权重，也即每次执行 Command 的数目。每个加权优先级内部执行 RR 仲裁机制。



前面的讲述中，我们提到 Arbitration Burst 和加权优先级均可以通过 `set feature` 设定，NVMe Spec 的定义如下：

Figure 105: Set Features – Feature Identifiers

Feature Identifier	O/M	Persistent Across Power Cycle and Reset ²	Uses Memory Buffer for Attributes	Description
00h				Reserved
01h	M	No	No	Arbitration
02h	M	No	No	Power Management
03h	O	Yes	Yes	LBA Range Type
04h	M	No	No	Temperature Threshold
05h	M	No	No	Error Recovery
06h	O	No	No	Volatile Write Cache
07h	M	No	No	Number of Queues
08h	NOTE 5	No	No	Interrupt Coalescing
09h	NOTE 5	No	No	Interrupt Vector Configuration
0Ah	M	No	No	Write Atomicity Normal
0Bh	M	No	No	Asynchronous Event Configuration
0Ch	O	No	Yes	Autonomous Power State Transition
0Dh	O	No ³	No ⁴	Host Memory Buffer
0Eh				Reserved
0Fh	O	No	No	Keep Alive Timer
10h – 77h				Reserved
78h – 7Fh		Refer to the NVMe Management Interface Specification for definition.		
80h – BFh				Command Set Specific ¹
C0h – FFh				Vendor Specific ¹



5.15.1.1 Arbitration (Feature Identifier 01h)

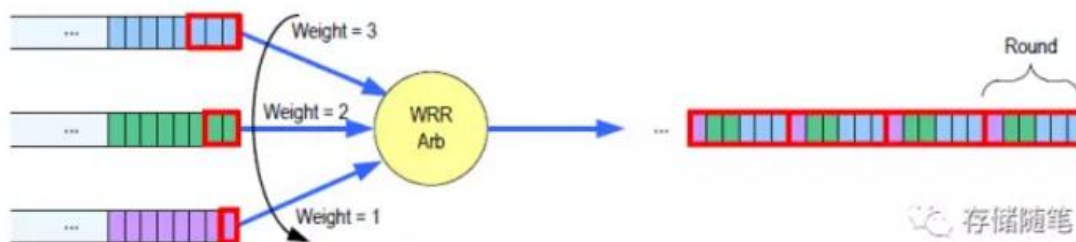
This Feature controls command arbitration. Refer to section 4.11 for command arbitration details. The attributes are indicated in Command Dword 11.

If a Get Features command is submitted for this Feature, the attributes specified in Figure 107 are returned in Dword 0 of the completion queue entry for that command.

Figure 107: Arbitration & Command Processing – Command Dword 11

Bit	Description
31:24	High Priority Weight (HPW): This field defines the number of commands that may be executed from the high priority service class in each arbitration round. This is a 0's based value.
23:16	Medium Priority Weight (MPW): This field defines the number of commands that may be executed from the medium priority service class in each arbitration round. This is a 0's based value.
15:08	Low Priority Weight (LPW): This field defines the number of commands that may be executed from the low priority service class in each arbitration round. This is a 0's based value.
07:03	Reserved
02:00	Arbitration Burst (AB): Indicates the maximum number of commands that the controller may launch at one time from a particular Submission Queue. This value is specified as a 0's based value. A value of 111b indicates no limit. Thus, the possible settings are 1, 2, 4, 8, 16, 32, 64, or no limit.

举个关于加权优先级的例子：



如上图，HPW=3, MPW=2, LPM=1，那么，在命令执行时，相应的 SQ 队列每次分别执行 3,2,1 个命令作为一个循环。

如何判断一个 NVMe 设备是否支持 Arbitration 机制呢？

可以查看该设备的 register。与 Arbitration 相关的有两个 Register: **Controller Capabilities** 和 **Controller Configuration**，如下图。

3.1 Register Definition

The following table describes the register map for the controller.

The Vendor Specific address range starts after the last doorbell supported by the controller and continues to the end of the BAR0/1 supported range. The start of the Vendor Specific address range starts at the same location and is not dependent on the number of allocated doorbells.

Start	End	Symbol	Description
00h	07h	CAP	Controller Capabilities
08h	0Bh	VS	Version
0Ch	0Fh	INTMS	Interrupt Mask Set
10h	13h	INTMC	Interrupt Mask Clear
14h	17h	CC	Controller Configuration
18h	1Bh	Reserved	Reserved
1Ch	1Fh	CSTS	Controller Status
20h	23h	NSSR	NVM Subsystem Reset (Optional)
24h	27h	AQA	Admin Queue Attributes



Controller Capabilities Register 的 Bit[18:17]代表 NVMe 主控可选性支持的仲裁机制。

Bit	Type	Reset	Description					
18:17	RO	Impl Spec	Arbitration Mechanism Supported (AMS): This field is bit significant and indicates the optional arbitration mechanisms supported by the controller. If a bit is set to '1', then the corresponding arbitration mechanism is supported by the controller. Refer to section 4.11 for arbitration details.					
			Bit	Definition	17	Weighted Round Robin with Urgent Priority Class	18	Vendor Specific
			Bit	Definition				
17	Weighted Round Robin with Urgent Priority Class							
18	Vendor Specific							
The round robin arbitration mechanism is not listed since all controllers shall support this arbitration mechanism.								

Controller Configuration Register 的 Bit[13:11]代表了 NVMe 主控正在使用的仲裁机制。

13:11	RW	0h	Arbitration Mechanism Selected (AMS): This field selects the arbitration mechanism to be used. This value shall only be changed when EN is cleared to '0'. Host software shall only set this field to supported arbitration mechanisms indicated in CAP.AMS. If this field is set to an unsupported value, the behavior is undefined.	
			Value	Definition
			000b	Round Robin
			001b	Weighted Round Robin with Urgent Priority Class
			010b – 110b	Reserved
			111b	Vendor Specific



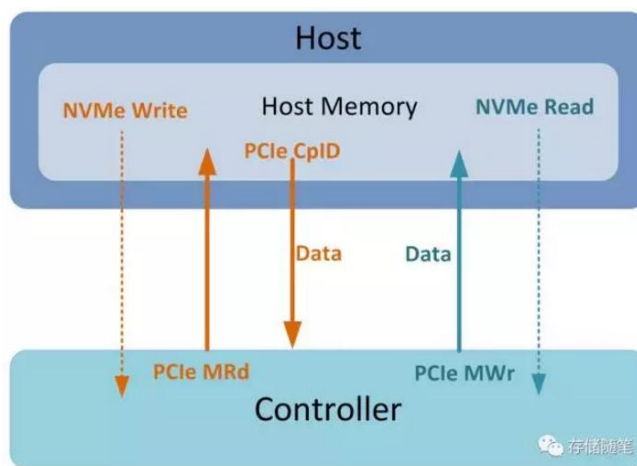
4.0 寻址模型 PRP 和 SGL

我们现在讲的 NVMe，主要是基于 NVMe over PCIe 展开的。在之前的文章中，我们提到了 NVMe over PCIe 系统中 Host 与 Controller 主要依靠 PCIe TLPs (Memory Read/Write) 进行信息的传递。

在之前的 PCIe 专题 ([PCIe 系列专题之二：2.2 TLP 事务处理方式解析](#)) 中，我们详细介绍 Memory Read/Write TLP 的处理方式。在这里就略过咯~

在 Host 与 Controller 之间有数据交互时，Controller 会多次访问 Host 内存。比如执行 NVMe Read/Write:

- ✧ 当 Host 下发 NVMe Write 命令时，Host 会先放数据放在 Host 内存中，然后通知 Controller 过来取数据。Controller 接到信息后，会通过 PCIe Memory Read TLP 读取相应的数据，接着 Host 返回的 PCIe Completion 报文中会携带数据给 Controller，最后再写入 NAND 中。
- ✧ 当 Host 下发 NVMe Read 命令时，Controller 先从 NAND 中读出相应数据，然后通过 PCIe Memory Write TLP 将数据写入 Host 内存中。



上述 Read/Write 均有访问 Host 内存进行数据交互，那么，问题来了：

- ✧ NVMe Write 时，Controller 怎么知道数据在 Host 内存的具体位置？
- ✧ NVMe Read 时，Controller 怎么知道要把数据写到 Host 内存中哪个位置？

不怕，NVMe 给 Host 配备了两大“法宝”：PRP 和 SGL。这两个模型均可以帮助 Host 告知 Controller 数据在 Host 内存中的具体地址。

(1). PRP

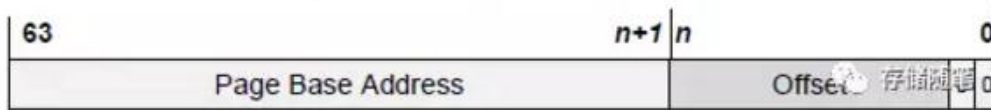
PRP 的全称是 Physical Region Page，记录的是 Host 内存中物理页的位置。Host 可以通过配置 Controller 的 CC.MPS 设定物理页的大小，范围是 4KB~128MB。

10:07	RW	0h	Memory Page Size (MPS): This field indicates the host memory page size. The memory page size is $2^{(12 + MPS)}$. Thus, the minimum host memory page size is 4KB and the maximum host memory page size is 128MB. The value set by host software shall be a supported value as indicated by the CAP.MPSMAX and CAP.MPSMIN fields. This field describes the value used for PRP entry size. This field shall only be modified when EN is cleared to '0'.
-------	----	----	---



物理页对应的地址记录在 PRP Entry 中，如下图。

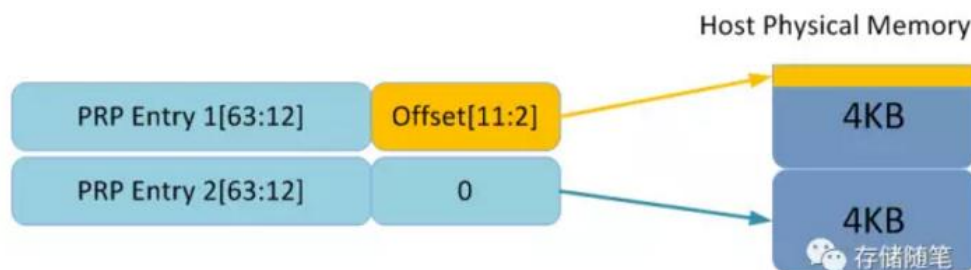
Figure 13: PRP Entry Layout



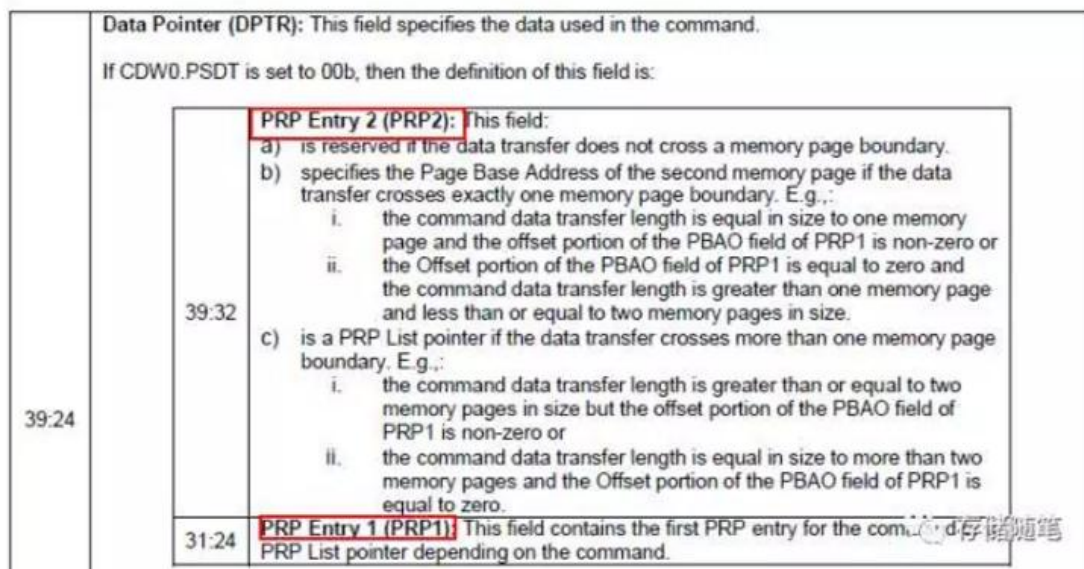
PRP Entry 是一个 64-bit 物理地址，分为两个部分：Page Base Address 和 Offset。

- ✧ Page Base Address: Bit[63:n]记录物理页的基地址；
- ✧ Offset: Bit[n:2]记录在物理页中的偏移量；

n 的取值有 CC.MPS 中定义的物理页大小决定。比如，物理页大小为 4KB，那么， $n=11$ ，也即 Bit[11:2]代表 offset。如果一个 Command 中包含两个 PRP，那么第二个 PRP Entry 的 offset 为 0。

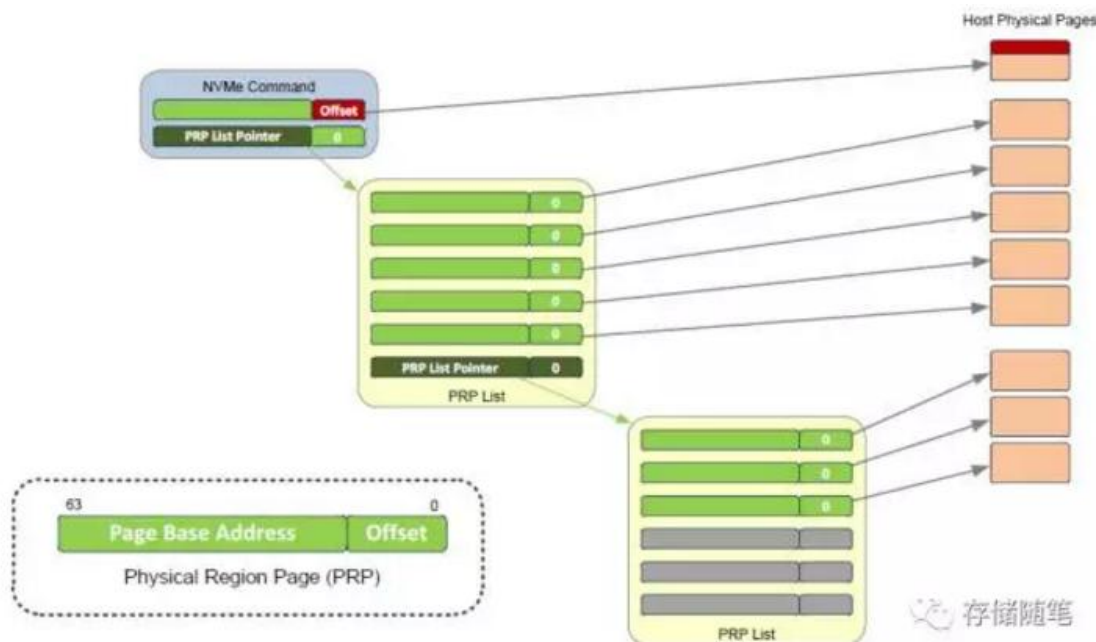


每个 PRP Entry 只能指向一个物理页。在 NVMe Command 的定义中，只定义了两个 PRP Entry，如下图，



假设物理页大小为 4KB，一个 Command 需要传输 12KB 数据。此时，两个 PRP Entry 可以最大指向 8KB 的空间，不满足要求，怎么办？

NVMe 想好了办法。PRP Entry 2 不指向物理页，而是指向由若干个 PRP 组成的 PRP list。如果需要定义更大的空间，PRP list 的最后一个 PRP 还可以指向另外一个 PRP list，依次类推，比如下图，



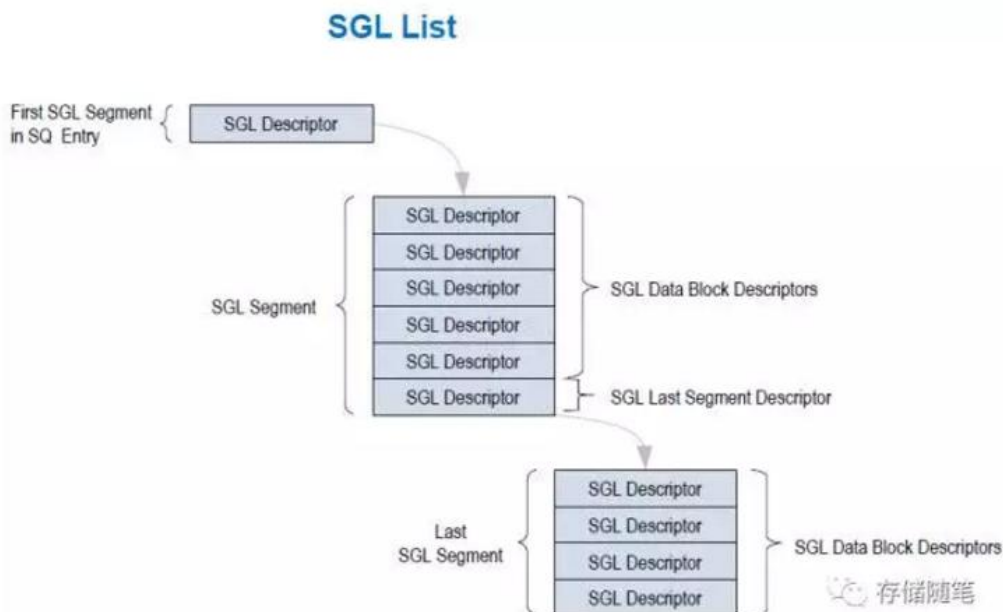
上图中每个 PRP list 包含 6 个 PRP。NVMe Command 传输数据时需要指向 9 个物理页，最后是用两个 PRP list 搞定了。需要注意的是，PRP list 中的 offset 必须为 0。

扩展: 文章底部解析了一个"PRP list" PCIe trace，有兴趣的话，欢迎翻阅并留言讨论。

(2). SGL

SGL 全称是 Scatter Gather List, 是一种用于描述物理地址的数据结构。在 NVMe over PCIe 系统中，SGL 可以用于 NVMe IO Command，但是绝不可以用于 NVMe Admin Command。其实，SGL 主要还是用在 NVMe over Fabric 系统中。我们这里就简单介绍一下。

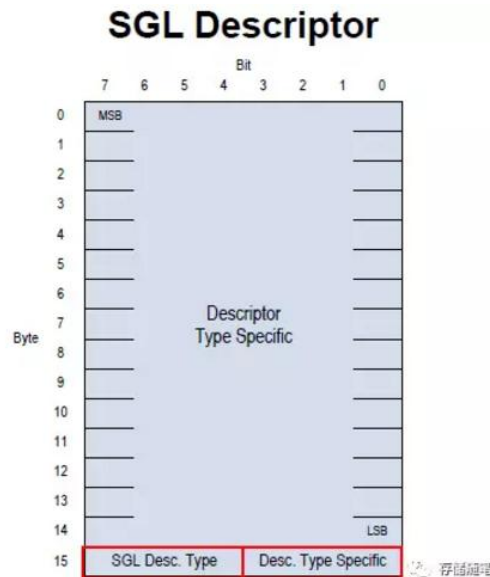
一个 SGL 包含多个 SGL Segment。一个 SGL Segment 包含多个 SGL Descriptor。其中，SGL Segment 要求 Qword 对齐。另外，倒数第二个 SGL Segment 中的最后一个 Descriptor 有一个特殊的名字，叫做 SGL Last Segment Descriptor。如下图，





SGL Descriptor 是 SGL 的最小的组成单位，结构如下图，

- ✧ Bit[3:0]代表 SGL Descriptor Sub Type, 在 NVMe over PCIe 中是保留的，只用在 NVMe over Fabric。
- ✧ Bit[7:4]代表 SGL Descriptor Type.



最新 NVMe Spec 1.3(发布于 2017.5.1)中定义了五种 SGL Descriptor Types:

Figure 18: SGL Descriptor Type

Code	Descriptor
0h	SGL Data Block descriptor
1h	SGL Bit Bucket descriptor
2h	SGL Segment descriptor
3h	SGL Last Segment descriptor
4h	Keyed SGL Data Block descriptor
5h – Eh	Reserved
Fh	Vendor specific

上表中的 0,2,3 这三种描述类型的含义比较简单，我们主要说一下 1 和 4 两种描述类型。

- ✧ **SGL Bit Bucket Descriptor:** 如 NVMe Spec 定义，这个描述类型仅为 Host 服务，告知 Controller 哪些数据是不必要写入 Host 内存的。

Figure 21: SGL Bit Bucket descriptor

Bytes	Description						
7:0	Reserved						
11:8	<p>Length: The Length field specifies the amount of source data that is discarded. An SGL Bit Bucket descriptor specifying that no source data be discarded (i.e., the length field set to 00000000h) is a valid SGL Bit Bucket descriptor.</p> <p>If the SGL Bit Bucket Descriptor describes a destination data buffer (e.g., a read from the controller to memory), then the Length field specifies the number of bytes of the source data which the controller shall discard (i.e., not transfer to the destination data buffer).</p> <p>If the SGL Bit Bucket Descriptor describes a source data buffer (e.g., a write from memory to the controller), then the Bit Bucket Descriptor shall be treated as if the Length field were set to 00000000h (i.e., the Bit Bucket Descriptor has no effect).</p> <p>If SGL Bit Bucket descriptors are supported, their length in a destination data buffer shall be included in the Number of Logical Blocks (NLB) parameter specified in NVM Command Set data transfer commands. Their length in a source data buffer is not included in the NLB parameter.</p>						
14:12	Reserved						
15	<p>SGL Identifier: The definition of this field is described in the table below.</p> <table border="1"> <tr> <th>Bits</th><th>Description</th></tr> <tr> <td>03:00</td><td>SGL Descriptor Sub Type field. Valid values are specified in Figure 19.</td></tr> <tr> <td>07:04</td><td>SGL Descriptor Type: 1h as specified in Figure 18.</td></tr> </table>	Bits	Description	03:00	SGL Descriptor Sub Type field. Valid values are specified in Figure 19.	07:04	SGL Descriptor Type: 1h as specified in Figure 18.
Bits	Description						
03:00	SGL Descriptor Sub Type field. Valid values are specified in Figure 19.						
07:04	SGL Descriptor Type: 1h as specified in Figure 18.						



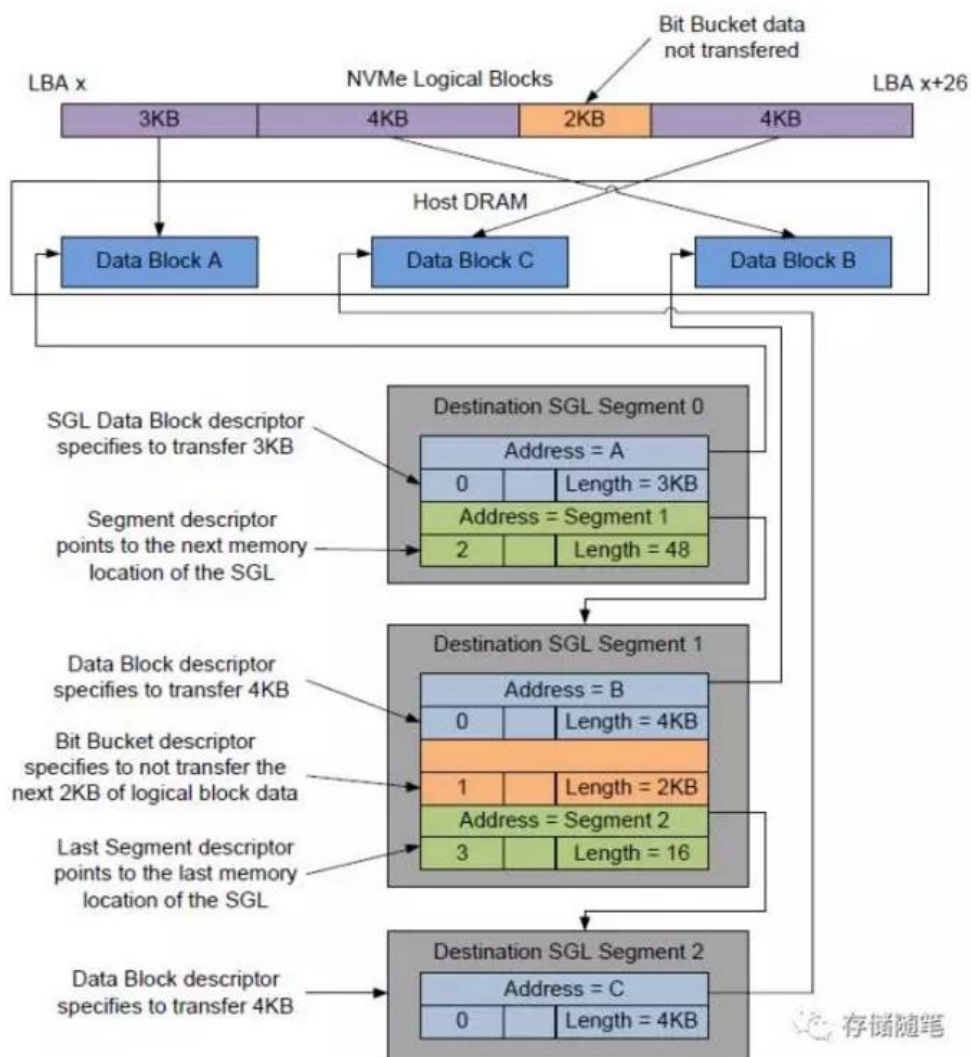
- ✧ **Keyed SGL Data Block Descriptor:** 这个描述类型代表了数据传输中带有密钥。此功能在 NVMe Spec 1.2.1 之后新增的。

Figure 24: Keyed SGL Data Block descriptor

Bytes	Description						
7:0	Address: The Address field specifies the starting 64-bit memory byte address of the data block.						
10:8	Length: The Length field specifies the length in bytes of the data block. A Length field set to 000000h specifies that no data is transferred. An SGL Data Block descriptor specifying that no data is transferred is a valid SGL Data Block descriptor. If the value in the Address field plus the value in the Length field is greater than 1_00000000_00000000h then the SGL Data Block descriptor shall be processed as having a Data SGL Length Invalid or Metadata SGL Length Invalid error.						
14:11	Key: Specifies a 32-bit key that is associated with the data block.						
15	SGL Identifier: The definition of this field is described in the table below.						
<table border="1"> <thead> <tr> <th>Bits</th><th>Description</th></tr> </thead> <tbody> <tr> <td>03:00</td><td>SGL Descriptor Sub Type field. Valid values are specified in Figure 19.</td></tr> <tr> <td>07:04</td><td>SGL Descriptor Type: 4h as specified in Figure 18.</td></tr> </tbody> </table>		Bits	Description	03:00	SGL Descriptor Sub Type field. Valid values are specified in Figure 19.	07:04	SGL Descriptor Type: 4h as specified in Figure 18.
Bits	Description						
03:00	SGL Descriptor Sub Type field. Valid values are specified in Figure 19.						
07:04	SGL Descriptor Type: 4h as specified in Figure 18.						

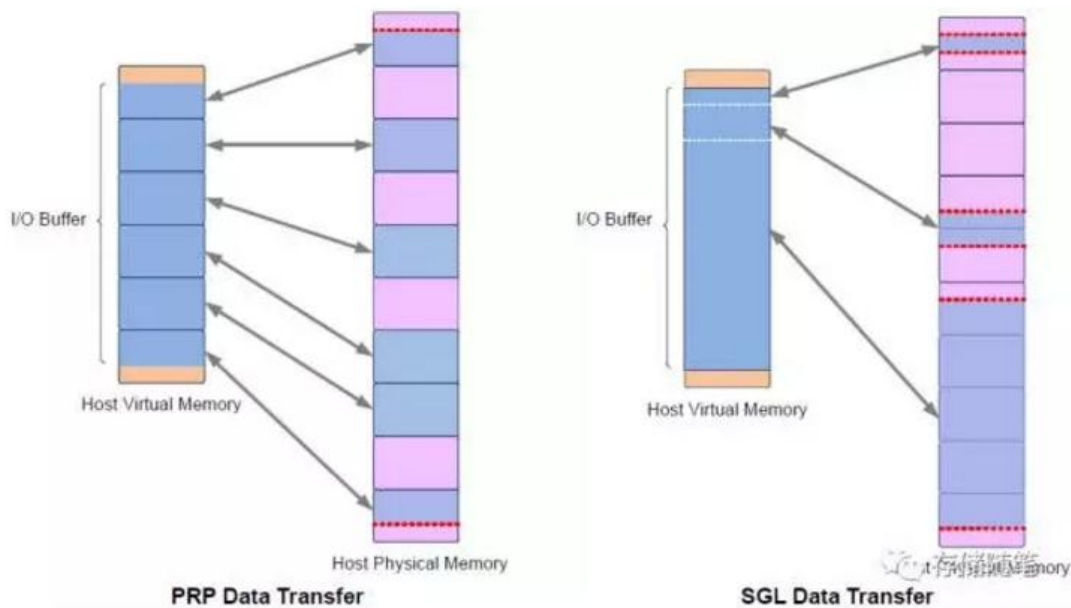
接下来看一个 SGL Bit Bucket Descriptor 的例子：

- ✧ 在这个例子中，Logical Block 大小为 512B. Host 要读 13KB 的数据，其中有 2KB 的数据不要 Controller 写入 Host 内存。所以，最终写入 Host 内存的数据只有 11KB。
- ✧ 总共有 3 个 SGL 描述块。对应的 Data Block Descriptor 的大小分别是 3KB,4KB,4KB。





PRP 和 SGL 是描述 Host 内存物理空间的两种方式，本质的不同是：PRP 必须是物理页对齐的，而 SGL 则可以表述任意的物理空间。如下图。



这里是华丽的分割线

【扩展】解析 PCIe Trace 中的 PRP list.

先看一下完整的 Trace:



Port	Protocol	Down	Up	Summary
PCIe Host	TLP	SubmQbErr		MWr: SubmQbErr, SQTid = 0x00000001;
PCIe Target	TLP		MRd	MRd: SubmQ; Len = 0x0040;
PCIe Host	NVMe	SubmQ		Read: NSID = 0x00000001; LBA = 0x0000000000000002; NbBlocks = 0x0020;
PCIe Target	NVMe		Data	Data; Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x100); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x200); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x300); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x400); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x500); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x600); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x700); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x800); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x900); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xA00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xB00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xC00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xD00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xE00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xF00); Len = 0x0100;
PCIe Target	TLP		MRd	MRd: PRPLst; Len = 0x0010;
PCIe Host	NVMe	PRPLst		PRPLst;
PCIe Target	NVMe		Data	Data(0x1000); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1100); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1200); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1300); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1400); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1500); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1600); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1700); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1800); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1900); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1A00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1B00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1C00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1D00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1E00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x1F00); Len = 0x0100;
PCIe Target	TLP		MRd	MRd: PRPLst;
PCIe Target	NVMe		Data	Data(0x2000); Len = 0x0100;
PCIe Host	NVMe	PRPLst		PRPLst;
PCIe Target	NVMe		Data	Data(0x2100); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2200); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2300); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2400); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2500); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2600); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2700); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2800); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2900); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2A00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2B00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2C00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2D00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2E00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x2F00); Len = 0x0100;
PCIe Target	TLP		MRd	MRd; Address = 0x1000E8018;
PCIe Target	NVMe		Data	Data(0x3000); Len = 0x0100;
PCIe Host	TLP	CplD		Pld = 0x00000000 00000000;
PCIe Target	NVMe		Data	Data(0x3100); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3200); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3300); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3400); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3500); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3600); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3700); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3800); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3900); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3A00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3B00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3C00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3D00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3E00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x3F00); Len = 0x0100;
PCIe Target	NVMe		CplQ	Success;
PCIe Target	TLP		MSI-X interrupt	MWr; MSI-X interrupt;
PCIe Host	TLP	CplQbErr		MWr; CplQbErr; CQHead = 0x00000001;



第一步，我们先解析一个 NVMe Read Command 中的 PRP Entry:

从下图中，我们可以了解到两个内容:

- ✧ Host 需要读取的数据大小= $0x20 \times 512B = 32 \times 512B = 16KB$;
- ✧ 数据要放在 Host 内存中的位置:

PRP1=0x0000000101104000;

PRP2=0x00000001000E8000;

Port	Protocol	Down	Up	Summary
PCIe Host	TLP	SubmQBel		MWrr; SubmQBel; SQTail = 0x00000001;
PCIe Host	NVMe	SubmQ		Read; NSID = 0x00000001; LBA = 0x0000000000000002; NbBlocks = 0x0020
PCIe Target	NVMe		Data	Data; Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x100); Len = 0x0100;

Transaction Layer Packet (TLP)

NVMe Submission Queue Entry

Opcode = 0x02 Read

PRP or SGL = 0x0 PRP

FUSE = 0x0 No

CID = 0x0000

NSID = 0x00000001

Metadata Pointer = 0x0000000000000000

PRP Entry 1 = 0x0000000101104000

PRP Entry 2 = 0x00000001000E8000

Starting LBA = 0x0000000000000002

Number of Logical Blocks = 0x0020

Limited Retry = 0x0 Off

Force Unit Access = 0x0 Off

第二步，查看 PRP1 和 PRP2 分别对应的内容:物理页或 PRP list.

(1)PRP1=0x0000000101104000: 根据下图内容，可以判断 PRP1 对应大小为 4KB 的物理页。

Port	Protocol	Down	Up	Summary
PCIe Host	TLP	SubmQBel		MWrr; SubmQBel; SQTail = 0x00000001;
PCIe Host	NVMe	SubmQ		Read; NSID = 0x00000001; LBA = 0x0000000000000002; NbBlocks = 0x0020;
PCIe Target	NVMe		Data	Data; Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x100); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x200); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x300); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x400); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x500); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x600); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x700); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x800); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0x900); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xA00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xB00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xC00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xD00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xE00); Len = 0x0100;
PCIe Target	NVMe		Data	Data(0xF00); Len = 0x0100;

Transaction Layer Packet (TLP)

STP = 0x6F047F79

TLP Header

Fmt/Type = 0x60 MWrr

Flags = 0x000040

Source ID = 0x0200 Micron.5402

Tag = 0x00

Byte Enables = 0xFF

Address = 0x101104000 Data

PH = 0x0 Off

NVMe Data

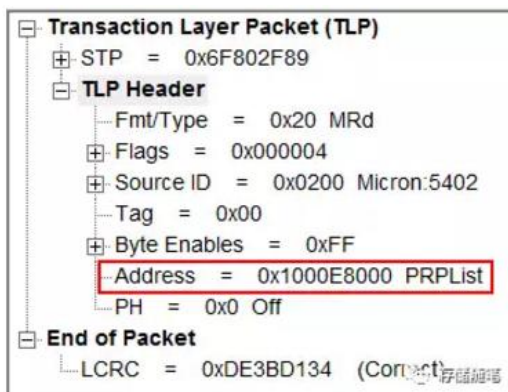
Payload = 28732AC1 1FF8D211 BA4B00A0 C93EC93B C33CB544 D4E24C40 B7B2523B 3689BBF

End of Packet

LCRC = 0x45007215 (Correct)



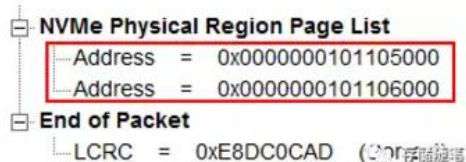
(2) PRP2=0x00000001000E8000, 查看 PCIe trace 发现, PRP2 对应 PRP list, 如下图。



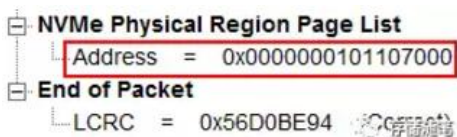
第三步, 查看 PRP2 list 包含的 PRP Entry.

查看 PCIe Trace, 发现:

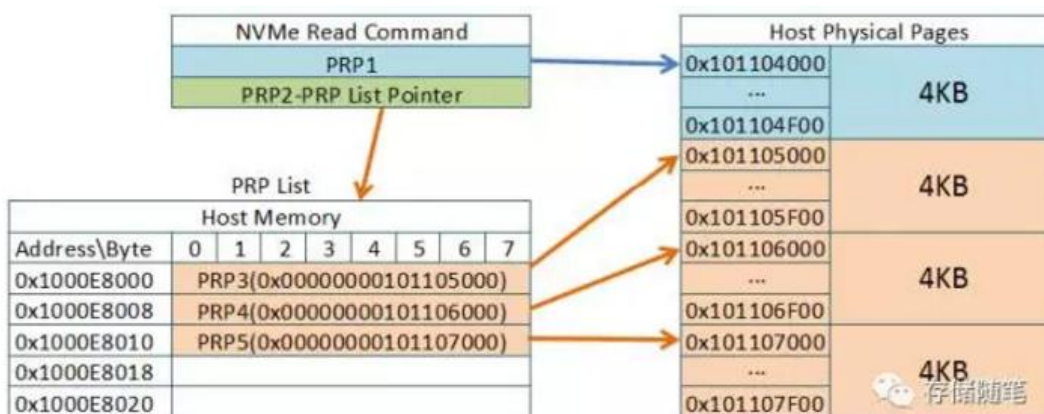
Address=0x1000E8000 的位置有两个 PRP, 我们暂且称之为 PRP3 和 PRP4,



Address=0x1000E8010 的位置有一个 PRP, 我们暂且称之为 PRP5,



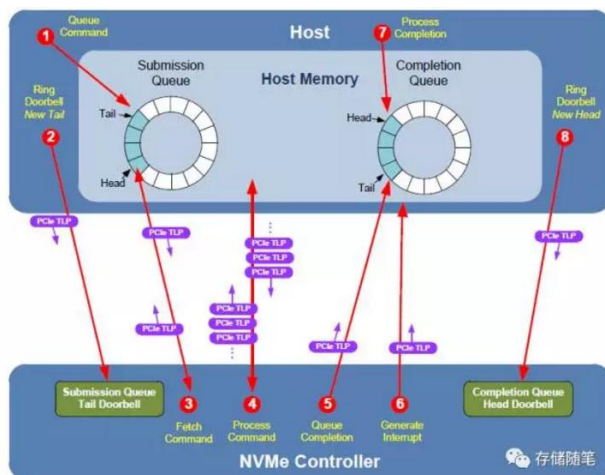
至此, 这个 NVMe Read 命令从 SSD 中获取的 data 在 Host 内存中的分布就比较明朗了, 画了示意图, 如下:





5.0 中断机制

在之前的文章中，我们有介绍 NVMe Command 的处理流程，其中第六步是 NVMe Controller 通过发送一个中断信息通知 Host 检查 CQ 中 Commands 的完成状态，如下图。



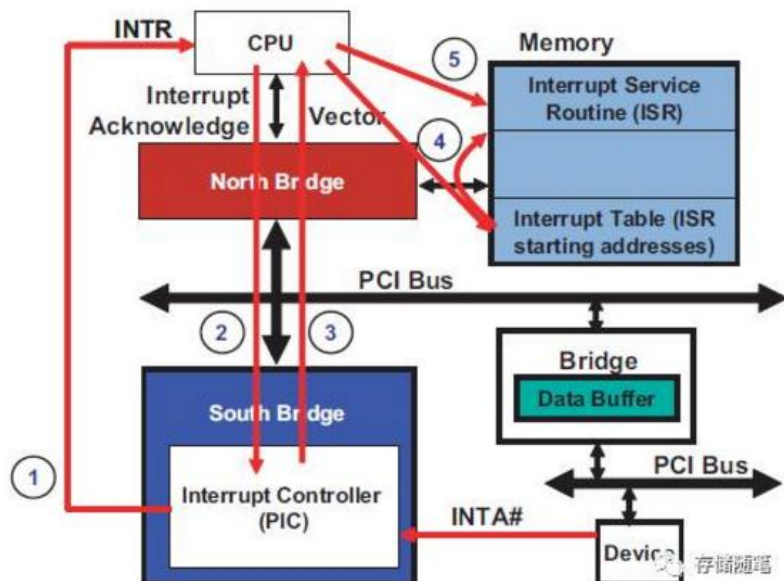
在 NVMe Spec 中规定，有四种中断类型：Pin-based Interrupt, Single Message MSI, Multiple Message MSI, MSI-X。我们一一介绍一下。

注：MSI=Message Signaled Interrupts.

(1). Pin-based Interrupt

Pin-based interrupt 主要针对 PCI 总线设计的中断方式。PCI 总线会为每一个设备分配 4 个 $INTx$ 引脚($INTA\#$, $INTB\#$, $INTC\#$, $INTD\#$)。 $INTx$ 引脚中断采用的是电平触发方式。

我们看个示例了解一下 $INTx$ 引脚中断的工作机理：

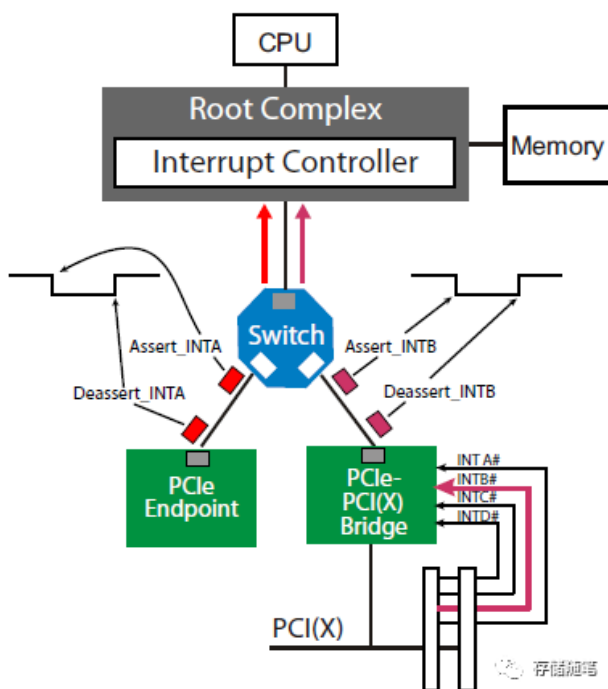




- ✧ PCI 设备通过触发 INTx 引脚将中断告知中断处理器(PIC)，之后 PIC 通过输出信号 INTR 通知 CPU 处理中断信息。
- ✧ CPU 收到 INTR 信号后，给 PIC 发送一个特殊的命令，目的是向 PIC 询问哪些中断需要处理。
- ✧ PIC 接到 CPU 询问的命令后，传送需要处理的中断信息。
- ✧ CPU 获得信息后，去中断表(Interrupt Table)中读取中断的起始位置。
- ✧ CPU 找到目标中断后开始执行。

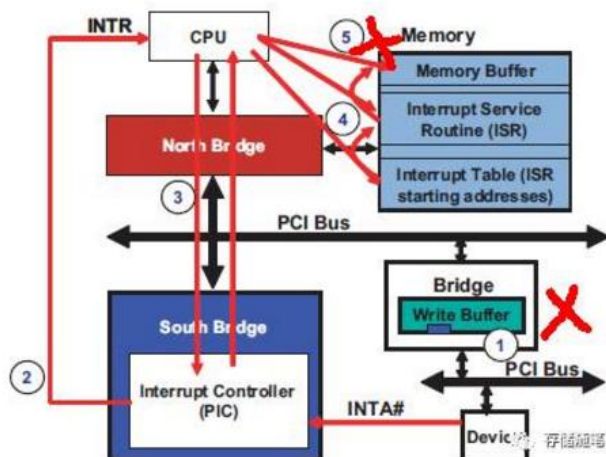
NVMe/PCIe 系统为了保持向下的兼容性，虽然保留了 INTx 中断机制，但是，此 INTx 非彼 INTx 也。PCIe 总线中并没有多根独立的中断 PIN，通过发送 INTx Message Packet 模拟中断 PIN 的置位和复位。在 NVMe/PCIe 系统中用到虚拟 INTx 信号情况有两种：

- ✧ **PCIe-PCI 桥连器**：PCI 设备通过触发 INTB#引脚将中断传递到 PCIe-PCI 桥连器，之后以 INTx 信息的形式发送至中断处理器。
- ✧ **启动设备**：在 PC 系统启动过程中需要 INTx 信息中断方式。因为 MSI/MSI-X 一般都需要 OS-level 的初始化。用当 PCIe 设备通过发送 INTx 模拟信息至中断处理器。

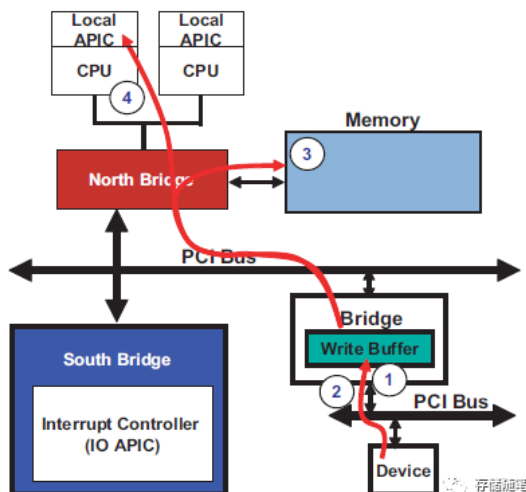


(2). Single/Multiple Message MSI

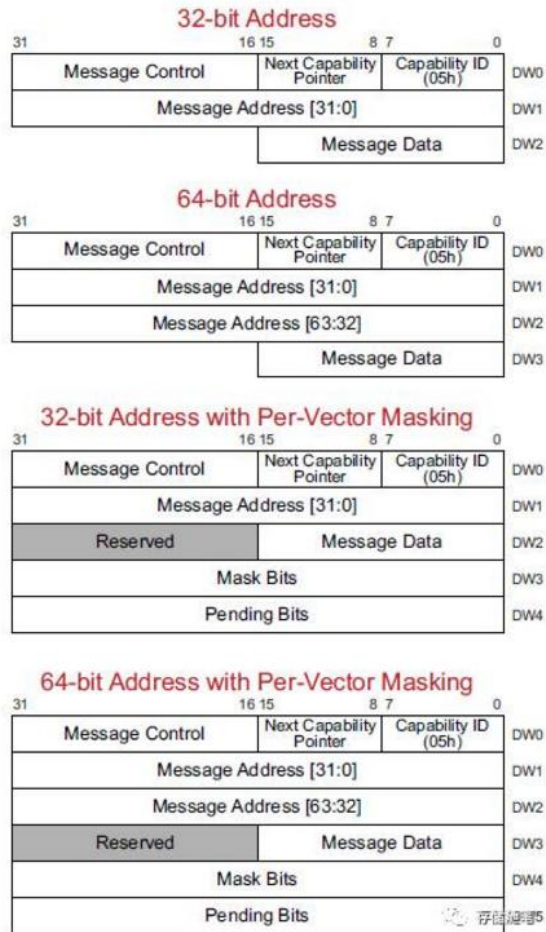
INTx 引脚中断有一个存储同步的问题：比如，PCI 设备写了一笔数据到 Host 内存中，不过，在传到 PCI 桥时因为某个原因产生了延时，但是中断已经发送给了 CPU，CPU 收到中断后去 Host 内存中读数据，而此时，数据还未到达 Host 内存，那么，这就会产生传输错误。



所以，为了解决这个 Bug，从 PCI 总线开始就引入了另外一种中断机制，那就是 MSI。之前的文章([PCIe 系列专题之二：2.8 事务排序机制](#))中我们说到 PCI 总线执行 Strong-Ordering 事务排序机制(先进先出)。数据和 MSI 中断报文通过同一路径传输(数据在前，MSI 中断报文在后)，只有把数据成功写入 Host 内存之后，MSI 中断报文才会发送到 CPU，这样就不会出现 INTx 存储同步的问题咯。



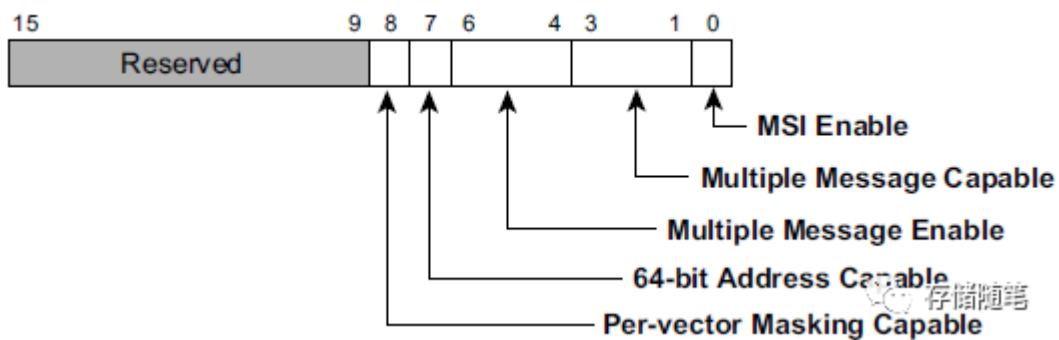
在 PCIe 总线中，MSI 中断机制使用 Memory Write TLP 向处理提交中断请求。PCIe 设备提交 MSI 中断请求时，需要向 MSI Capability 结构中的 Message Address 的地址写入 Message Data 数据。根据 Address 的长度(32 位或者 64 位)以及是否 Mask 中断向量，MSI Capability 结构分为四类如下图：



Capability ID: 05h 代表是 MSI Capability，这个参数只读不能写。

Next Capability Pointer: 一般情况下为 0，有时候需要多个 Capability 结构时，指向下一个 Capability 结构，也是只能读不能写。

Message Control: MSI 相关的具体设定。当 Enable MSI 中断时，MSI-X 和 INTx 中断不能用。



Bit[3:1] Multiple Message Capable 字段可以控制是 Single 或者 Multiple(最大支持 32 个) Message MSI。

Message Address Register: 中断信息发送的目的地。低 32 位[31:0]中的 bit[1:0]为 0，所以地址是 DWORD 对齐。在 x86 系统中，中断信息的地址基本是固定区域，0xFFEx_xxxxh。

Message Data Register: 中断时发送的数据。

Mask bit & Pending bit: MSI 最大支持 32 中断向量。所以 Mask bit 和 Pending bit 长度均有 32bits，每个 bit 对应一个中断向量。当某个中断向量对应的 Mask bit=1 时，这个中断向量不能发送，此时其对应的 Pending bit 置为 1。当 Mask bit 以及对应的 Pending bit 被清除时，此中断立刻发送。

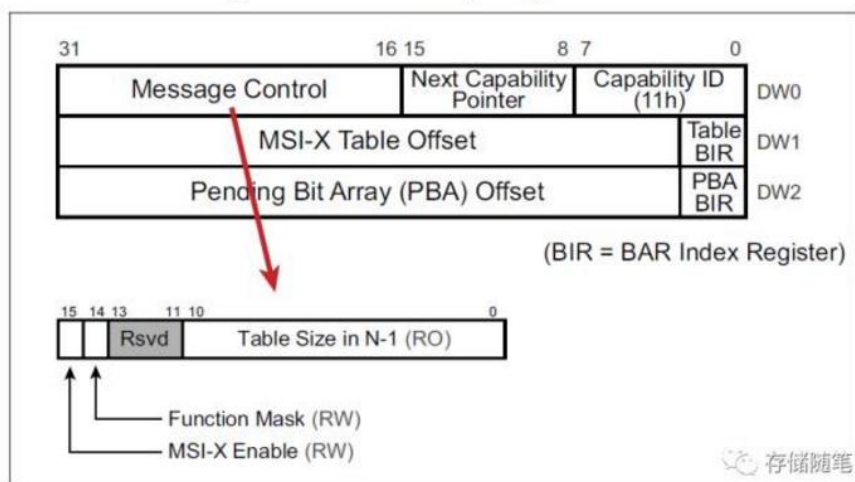


(3). MSI-X

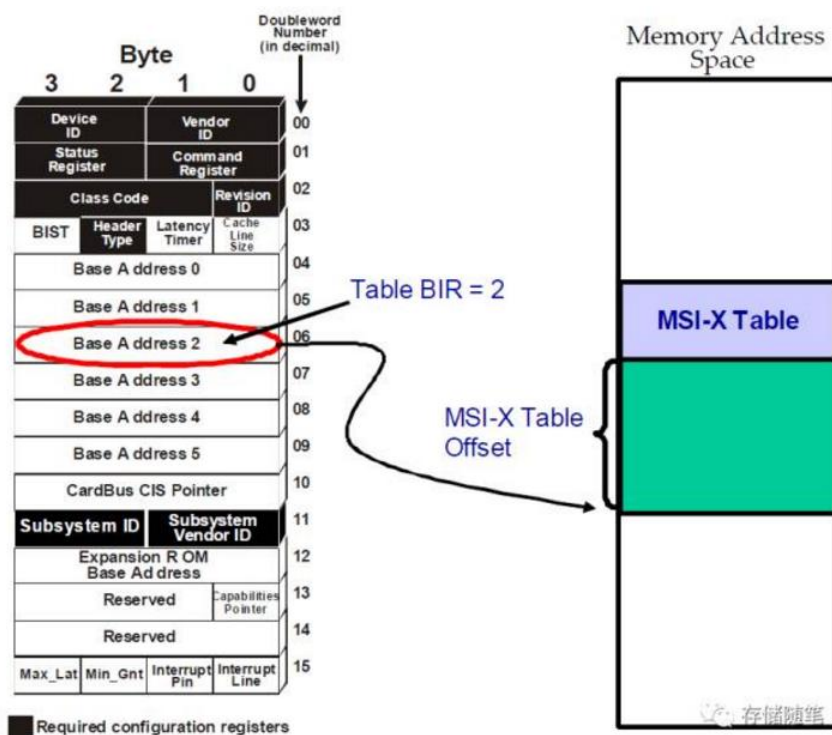
MSI-X 与 MSI 的机制基本相同，不过，MSI 中断机制最多支持 32 个中断请求，并且要求中断向量连续。而 MSI-X 中断机制可以支持 2048 个中断请求，并且不要求中断向量连续。

与 MSI Capability 寄存器相比，MSI-X Capability 寄存器使用了一个数组存放 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，这个数组称为 MSI-X Table。所以，MSI-X Capability 寄存器比 MSI Capability 寄存器要小，具体格式如下图，

Figure 17-17: MSI-X Capability Structure



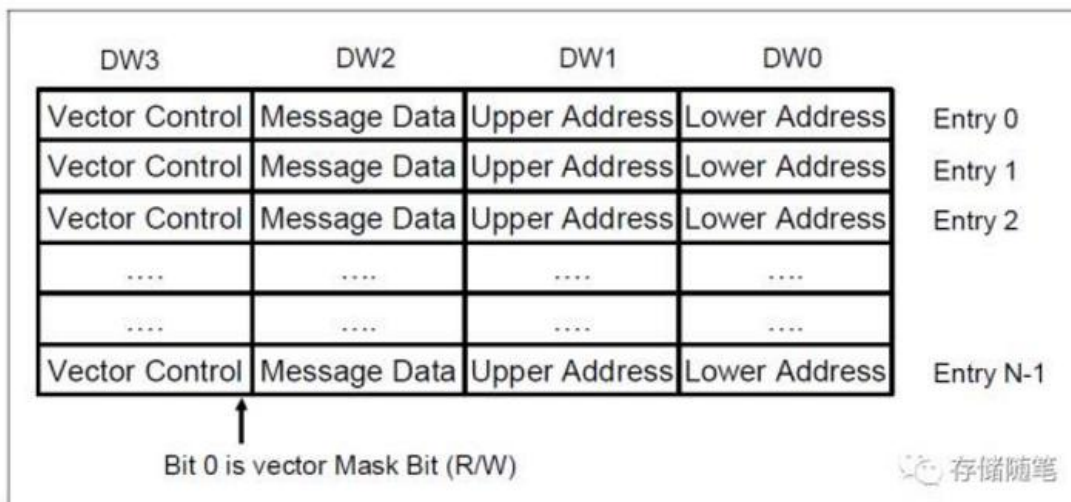
结合 MSI-X Table offset 和 Table BIR 可以在内存中定位到 MSI-X Table 的位置，如下图。





MSI-X Table 中每一条 Entry 代表了一条中断信息，大小为 4 DWords，如下图，

Figure 17-19: MSI-X Table Entries



DW0-DW1: 提供 64 位的地址空间，

DW2: 代表了 32 位的数据信息，

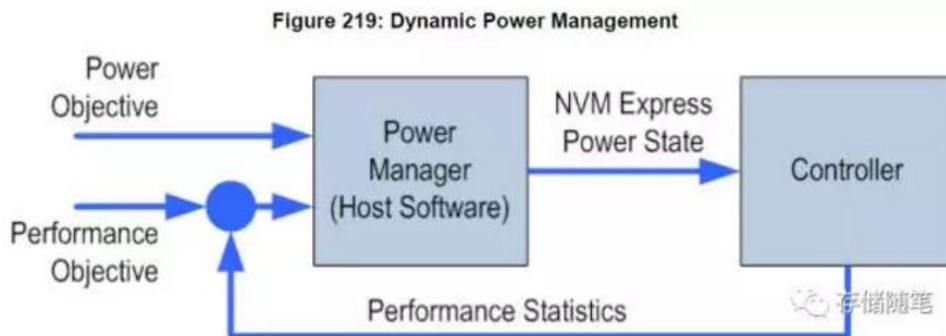
DW3: 目前只有 Bit0 有效，是 Mask bit，允许每一条中断可以被独立 Mask.



6.0 电源管理

NVMe 协议其中有一项优势，就是低功耗！为了达成这个目标，NVMe 中加入了自动电源状态转换和动态电源管理机制。

先来看一下 NVMe Spec 中对动态电源管理的描述图：



- ✧ Host 设定性能和功耗：Power Objective 和 Performance Objective。
- ✧ Host 通知 Controller 更改设备的 power state。

NVMe 最多支持 32 个电源状态 (PS, Power State)。NVMe 设备可支持的 Power State 数目可以在 Identify Controller Data Structure Byte[263]中查看。

263	M	<p>Number of Power States Support (NPSS): This field indicates the number of NVM Express power states supported by the controller. This is a 0's based value. Refer to section 8.4.</p> <p>Power states are numbered sequentially starting at power state 0. A controller shall support at least one power state (i.e., power state 0) and may support up to 31 additional power states (i.e., up to 32 total).</p>
-----	---	--

每个 Power State 对应有一个长度为 32Bytes 的 Power State Descriptor，里面会描述每个 Power State 对应的最大功耗 (MP)，进入延迟，退出延迟等等。比如，下表中，定义了 7 个 power states，

Power State	Maximum Power (MP)	Entry Latency (ENTLAT)	Exit Latency (EXLAT)	Relative Read Throughput (RRT)	Relative Read Latency (RRL)	Relative Write Throughput (RWT)	Relative Write Latency (RWL)
0	25 W	5 μ s	5 μ s	0	0	0	0
1	18 W	5 μ s	7 μ s	0	0	1	0
2	18 W	5 μ s	8 μ s	1	0	0	0
3	15 W	20 μ s	15 μ s	2	0	2	0
4	10 W	20 μ s	30 μ s	1	1	3	0
5	8 W	50 μ s	50 μ s	2	2	4	0
6	5 W	20 μ s	5000 μ s	4	3	5	1

目前，动态电源管理一般只用在消费级 SSD 上，对笔记本电脑的续航问题有很大的帮助。在企业级 SSD 中数据的安全性还是第一位的，不大会考虑功耗的问题。NVMe 白皮书对消费级 NVMe SSD 的 Power State 建议如下：



Power State	Description	Performance	Power	Resume Latency
PS0	Full Performance	100% Performance	No restriction	n/a
PS1	Thermal Throttle, Light	< 100% Performance	E.g., for M.2 do not exceed 2.4 W	n/a
PS2	Thermal Throttle, Heavy	< 100% Performance	E.g., for M.2 do not exceed 1.9 W	n/a
PS3	Non-operational, good power with fast recovery	n/a	Idle < 50 - 100 mW	< 1-10 ms
PS4	Non-operational, lowest power state	n/a	Idle < 5 mW	< 50-100 ms

上表中，PS0 是全速状态。PS1、PS2 是过热保护时降频需要过渡的状态。PS3、PS4 是非操作状态，具有较低的功耗，并且退出延迟很短。

大家知道，目前 NVMe SSD 都需要 PCIe 总线的配合，所以，NVMe 电源状态也必须与 PCIe 链路电源状态 (PCIe 系列专题之五: PCIe 总线电源管理) 相互映衬。

NVMe Power State	PCIe link Power State
PS0	L0/L0s/L1
PS1	L0/L0s/L1
PS2	L0/L0s/L1
PS3	L1/L1.1/L1.2
PS4	L1.2

知道了 NVMe 设备的 Power State 定义，Host 如何更改 NVMe 设备的 Power State 呢？手段主要有两种：

- ✧ Host 启动自动电源状态切换功能；
- ✧ Host 发送 Set Feature 命令更改 Power State；

Host 可以访问 Identify Controller Data Structure Byte[265] 查看 NVMe 设备是否支持自动电源状态切换功能。

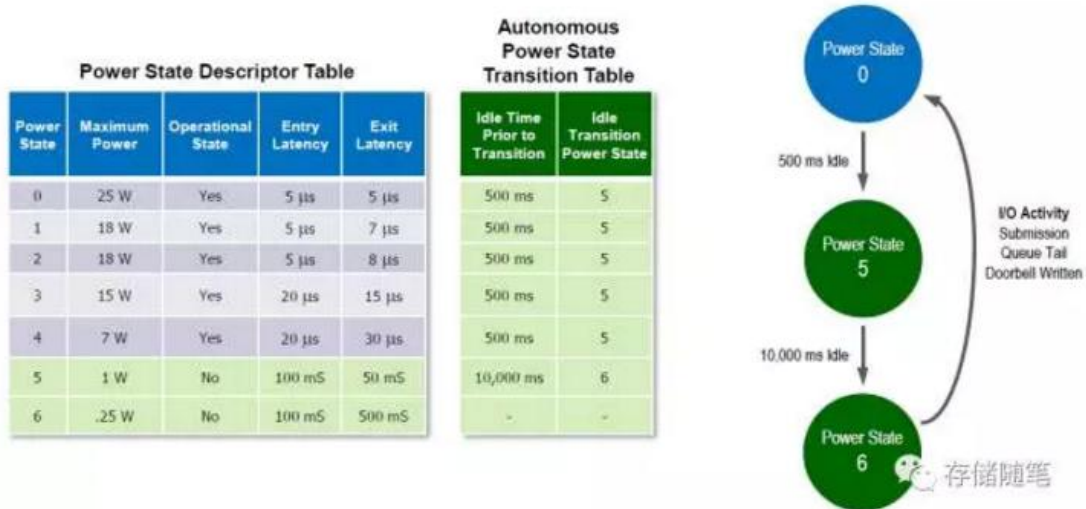
265	0	<p>Autonomous Power State Transition Attributes (APSTA): This field indicates the attributes of the autonomous power state transition feature. Refer to section 8.4.2.</p> <p>Bits 7:1 are reserved.</p> <p>Bit 0 if set to '1' then the controller supports autonomous power state transitions. If cleared to '0' then the controller does not support autonomous power state transitions.</p>
-----	---	--

比如，下面这个例子：

NVMe 设备中定义了 7 个 power state，并且定义了电源状态自动切换的计划。

- ✧ 当设备 Idle 时间超过 500ms 后，自动切换进入 Power State 5；
- ✧ 当设备 Idle 时间超过 1000ms 后，自动切换进入 Power State 6；

注意，如果处在非操作状态时(e.g. Power State 6)，SQ 中收到新的 Command，那么电源状态立刻切回全速状态(Power State 0)。



另外一种切回电源状态的方式：Set Feature。很简单，在 Set feature 中写下要切换的目标状态，然后发送 Command 给 NVMe 设备即可。Set Feature 定义如下：

Figure 105: Set Features – Feature Identifiers

Feature Identifier	O/M	Persistent Across Power Cycle and Reset ²	Uses Memory Buffer for Attributes	Description
00h				Reserved
01h	M	No	No	Arbitration
02h	M	No	No	Power Management
03h	O	Yes	Yes	LBA Range Type
04h	M	No	No	Temperature Threshold
05h	M	No	No	Error Recovery
06h	O	No	No	Volatile Write Cache
07h	M	No	No	Number of Queues
08h	NOTE 5	No	No	Interrupt Coalescing
09h	NOTE 5	No	No	Interrupt Vector Configuration
0Ah	M	No	No	Write Atomicity Normal
0Bh	M	No	No	Asynchronous Event Configuration
0Ch	O	No	Yes	Autonomous Power State Transition
0Dh	O	No ³	No ⁴	Host Memory Buffer
0Eh				Reserved
0Fh	O	No	No	Keep Alive Timer
10h – 77h				Reserved
78h – 7Fh		Refer to the NVMe Management Interface Specification for definition.		
80h – BFh				Command Set Specific ¹
C0h – FFh				Vendor Specific ¹

Figure 108: Power Management – Command Dword 11

Bit	Description
31:08	Reserved
07:05	Workload Hint (WH): This field indicates the type of workload expected. This hint may be used by the NVM subsystem to optimize performance. Refer to section 8.4.3 for more details.
04:00	Power State (PS): This field indicates the new power state into which the controller should transition. This power state shall be one supported by the controller as indicated in the Number of Power States Supported (NPSS) field in the Identify Controller data structure. If the power state specified is not supported, the controller shall abort the command and should return an Invalid Field in Command.



PCIe Tracer 也可以抓到 Set Feature 修改 Power State 的过程，如下图。但是，是否真的进入 Power State 3 可能需要直接量测 NVMe 设备的功耗来判断。

Delta Time	Port	Protocol	Down	Up	Summary
		PCIe Host	TLP	ASubmQBel	MW; ASubmQBel; ASQTail = 0x0000001D;
0.294_8	PCIe Host	NVMe	ASubmQ		Set Features, Power Mgmt, Power State = 3.
473.638_1	PCIe Target	NVMe		ACpIQ	Success;
3.772_9	PCIe Host	TLP	ACpIQBel		MW; ACpIQBel; ACQHead = 0x00000019;

Transaction Layer Packet (TLP)

NVMe Admin Submission Queue Entry

Opcode = 0x09 Set Features (Interesting Event Found)

PRP or SGL = 0x0 PRP

FUSE = 0x0 No

CID = 0x001C

NSID = 0x00000000 Not Used

Metadata Pointer = 0x0000000000000000

PRP Entry 1 = 0x0000000000000000

PRP Entry 2 = 0x0000000000000000


Feature Identifier (FID) = 0x02 Power Mgmt

Save = 0x0 Off

Power State = 3

End of Packet

Errors/Warnings = Interesting Event Found

 存储随笔



7.0 E2E 数据保护

Host 与 SSD 之间的数据交互是通过逻辑块 (LBA, Logical Block Address) 进行的, 大小为 512B, 1024B, 2048B, 4096B 等, 不过, 为了 SSD 性能的最大化, 现在 Host 一般采用 4KB 访问 SSD, 这也就是我们经常提到的“4K 对齐” ([SSD 性能优化之 4K 对齐](#))。

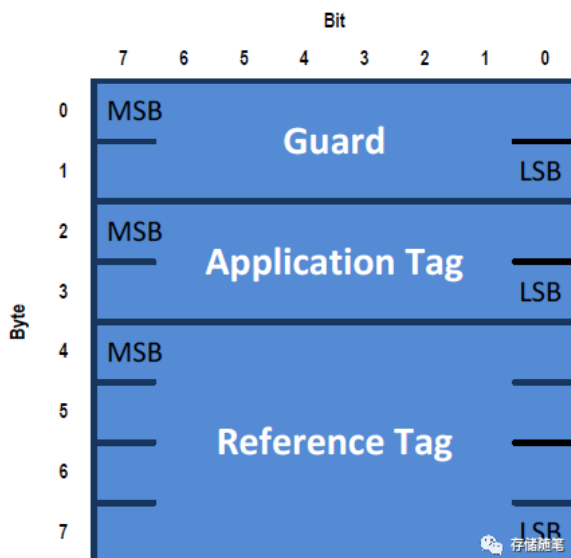


有些情况下, Host 与 SSD 之间的数据交互, 除了用户数据, 还有携带了一些元数据 (Meta Data), 大小为 $0 \sim N$ Bytes. 至于 Meta data 区域的用户, NVMe spec 并没有特别的要求。Metadata 一个最重要的角色就是传递端对端 (E2E, End to End) 数据保护信息 (PI, Protection Information)。

E2E 数据保护的必要性: 数据在存储系统传输中, 经过了多个部件、多种传输通道和复杂的软件处理过程, 其中任意一个环节发生错误都可能会导致数据错误。但是这种错误一般无法被立即检测出来, 而是后续通过应用在访问数据过程中, 才发现数据已经出错, 这种数据很难在数据发生错误那一刻被检查出来的错误, 我们称为静默数据破坏, 即 Silent Data Corruption。

E2E 数据保护机制主要用在企业级 SSD 上 (具体可以参考 Mamblaze 以及 Shannon 白皮书), 因为企业级数据服务器必须采用一切可能的措施来保护客户数据。不过, 目前的趋势是 E2E 也多用在消费级 SSD 上面了 (比如 SMI 主控 SM2262, [慧荣科技发布最新 SSD 主控, 支持 NVMe 1.3 标准](#)), 因为消费级客户的数据也得好好保护, 不能厚此薄彼~

PI 大小为 8 字节, 包括 2 字节的 Logical Block Guard, 2 字节的 Logical Block Application Tag 和 4 字节的 Logical Block Reference Tag, 如下图。





Logical Block Guard: 这块区域包含 16-bit CRC，保护数据的一致性。

Logical Block Application Tag: 这块区域对 Controller 不可见，为 Host 所用。

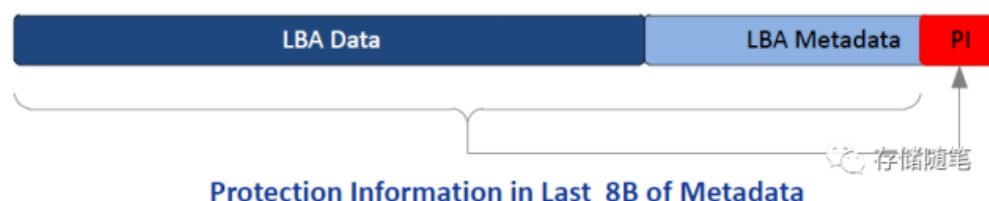
Logical Block Reference Tag: 将用户数据和地址相关联，防止数据错乱。

PI 在 Metadata 区域的位置有两种：

✧ Metadata 前 8 个字节，



✧ Metadata 后 8 个字节。



假设 $PI > 8$ Bytes:

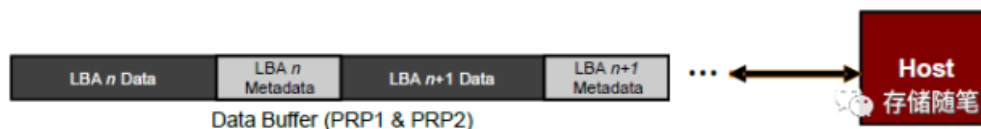
- (1) 如果 PI 是放在前 8byte，那么，CRC 只包括 LBA data;
- (2) 如果 PI 放在后 8byte，那么 CRC 包括除 PI 之外的所有数据和 Metadata.

同样，Metadata 与数据之间的相处方式也有两种：

✧ 一种是“相偎相依”；

Metadata 放在 LBA Data 之后，寸步不离，时刻守护数据的安全。这种形式类似于 SCSI 协议中的“DIF”（Data Integrity Field），也称作 T10 DIF .

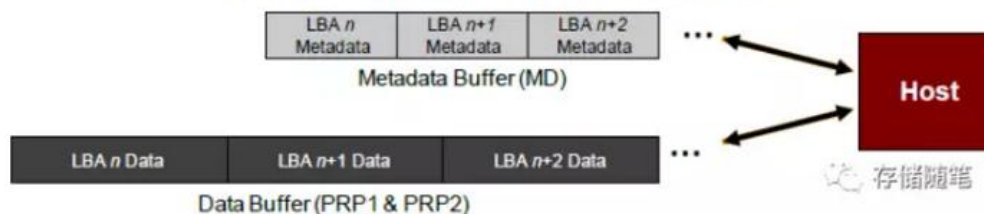
Figure 213: Metadata – Contiguous with LBA Data, Forming Extended LBA



✧ 另外一种“距离产生美”。

Metadata 放在单独的一块空间，虽然不能时刻陪伴，但是守护数据安全的心是一样的。这种形式类似于 SCSI 协议中的“DIX”（Data Integrity Extension）。

Figure 214: Metadata – Transferred as Separate Buffer





NVMe Spec 定义的 E2E 保护机制兼容 DIF 和 DIX 两种方式。但在格式化时对 PI 方式的三种选择都是基于 DIF 定义的，如下图，

Format NVM – Command Dword 10, Bit[7:5]:

07:05	Protection Information (PI): This field specifies whether end-to-end data protection is enabled and the type of protection information. The values for this field have the following meanings:	
	Value	Definition
	000b	Protection information is not enabled
	001b	Protection information is enabled, Type 1
	010b	Protection information is enabled, Type 2
	011b	Protection information is enabled, Type 3
	100b – 111b	Reserved
When end-to-end data protected is enabled, the host shall specify the appropriate protection information in the Read, Write, or Compare commands.		

也可以通过查看 Identify Data Structure 来获取 PI 类型，如下图：

Identify – Identify Namespace Data Structure, NVM Command Set Specific, Byte[29]:

29	M	End-to-end Data Protection Type Settings (DPS): This field indicates the Type settings for the end-to-end data protection feature. Refer to section 8.3.										
		Bits 7:4 are reserved.										
		Bit 3 if set to '1' indicates that the protection information, if enabled, is transferred as the first eight bytes of metadata. Bit 3 if cleared to '0' indicates that the protection information, if enabled, is transferred as the last eight bytes of metadata.										
		Bits 2:0 indicate whether Protection Information is enabled and the type of Protection Information enabled. The values for this field have the following meanings:										
		<table border="1"><thead><tr><th>Value</th><th>Definition</th></tr></thead><tbody><tr><td>000b</td><td>Protection information is not enabled</td></tr><tr><td>001b</td><td>Protection information is enabled, Type 1</td></tr><tr><td>010b</td><td>Protection information is enabled, Type 2</td></tr><tr><td>011b</td><td>Protection information is enabled, Type 3</td></tr><tr><td>100b – 111b</td><td>Reserved</td></tr></tbody></table>	Value	Definition	000b	Protection information is not enabled	001b	Protection information is enabled, Type 1	010b	Protection information is enabled, Type 2	011b	Protection information is enabled, Type 3
Value	Definition											
000b	Protection information is not enabled											
001b	Protection information is enabled, Type 1											
010b	Protection information is enabled, Type 2											
011b	Protection information is enabled, Type 3											
100b – 111b	Reserved											

扩展- PI 类型主要分为 4 类：

- ✧ Type 0: 不支持数据保护；
- ✧ Type 1: 支持数据保护，对 32-Byte 命令无效；
- ✧ Type 2: 支持数据保护，只对 32-Byte 命令有效；
- ✧ Type 3: 支持数据保护，对 32-Byte 命令无效。同时，对 Reference Tag 没有定义，可能将 Reference Tag 所在区域定义为 Application Tag 区域的扩展。

详细内容，请见 SCSI Spec SBC-3 “Protection information model”章节。

如果对 SCSI 与 NVMe 命令之间的转换感兴趣，请参考[“SCSI 命令下发方式<续>:对 NVMe 硬盘如何实现 SCSI 命令转换？”](#)。



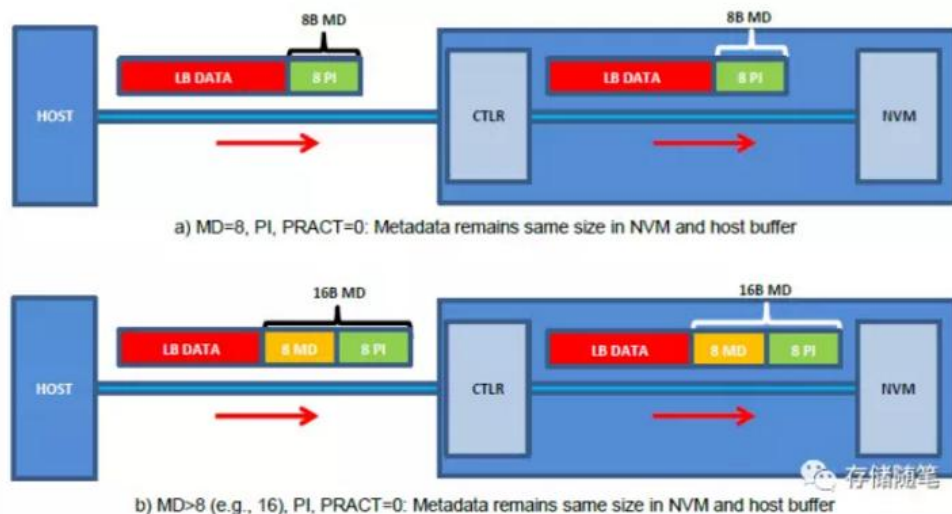
在 Enable PI 的情况下，有一个 bit，名叫“PI 操作位”(PRACT, Protection Information Action)对 PI 在 Host 与 SSD 之间执行过程影响很大。

Figure 154: Protection Information Field Definition

Bit	Description												
03	<p>Protection Information Action (PRACT): The protection information action field indicates the action to take for the protection information. This field is only used if the namespace is formatted to use end-to-end protection information. Refer to section 8.3.</p> <table><thead><tr><th>PRACT Value</th><th>Metadata Size</th><th>Description</th></tr></thead><tbody><tr><td>1b</td><td>8 Bytes</td><td>The protection information is stripped (read) or inserted (write).</td></tr><tr><td>1b</td><td>> 8 Bytes</td><td>The protection information is passed (read) or replaces the first or last 8 bytes of the metadata (write).</td></tr><tr><td>0b</td><td>any</td><td>The protection information is passed (read and write).</td></tr></tbody></table>	PRACT Value	Metadata Size	Description	1b	8 Bytes	The protection information is stripped (read) or inserted (write).	1b	> 8 Bytes	The protection information is passed (read) or replaces the first or last 8 bytes of the metadata (write).	0b	any	The protection information is passed (read and write).
PRACT Value	Metadata Size	Description											
1b	8 Bytes	The protection information is stripped (read) or inserted (write).											
1b	> 8 Bytes	The protection information is passed (read) or replaces the first or last 8 bytes of the metadata (write).											
0b	any	The protection information is passed (read and write).											
02:00	<p>Protection Information Check (PRCHK): The protection information check field indicates the fields that need to be checked as part of end-to-end data protection processing. This field is only used if the namespace is formatted to use end-to-end protection information. Refer to section 8.3.</p> <table><thead><tr><th>Bit</th><th>Definition</th></tr></thead><tbody><tr><td>02</td><td>If set to '1' enables protection information checking of the Guard field. If cleared to '0', the Guard field is not checked.</td></tr><tr><td>01</td><td>If set to '1' enables protection information checking of the Application Tag field. If cleared to '0', the Application Tag field is not checked.</td></tr><tr><td>00</td><td>If set to '1' enables protection information checking of the Logical Block Reference Tag field. If cleared to '0', the Logical Block Reference Tag field is not checked.</td></tr></tbody></table>	Bit	Definition	02	If set to '1' enables protection information checking of the Guard field. If cleared to '0', the Guard field is not checked.	01	If set to '1' enables protection information checking of the Application Tag field. If cleared to '0', the Application Tag field is not checked.	00	If set to '1' enables protection information checking of the Logical Block Reference Tag field. If cleared to '0', the Logical Block Reference Tag field is not checked.				
Bit	Definition												
02	If set to '1' enables protection information checking of the Guard field. If cleared to '0', the Guard field is not checked.												
01	If set to '1' enables protection information checking of the Application Tag field. If cleared to '0', the Application Tag field is not checked.												
00	If set to '1' enables protection information checking of the Logical Block Reference Tag field. If cleared to '0', the Logical Block Reference Tag field is not checked.												

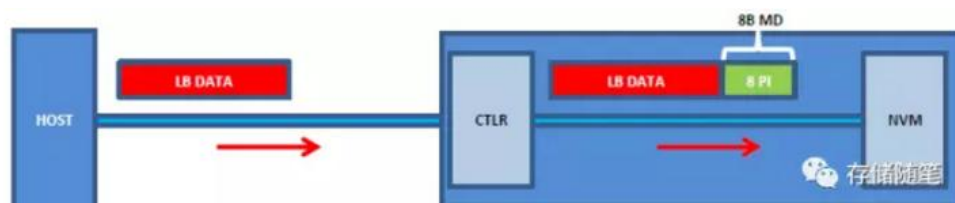
存储随笔

当 **PRACT=0** 时，不论 **Metadata** 的大小，**PI** 信息会全称负责保护数据的安全。在这个过程中，再考虑到 TLP 报本身的 LCRC/ECRC 以及 SSD 内部 RAID/LDPC 等保护措施，基本可以确保数据在传输过程中万无一失。



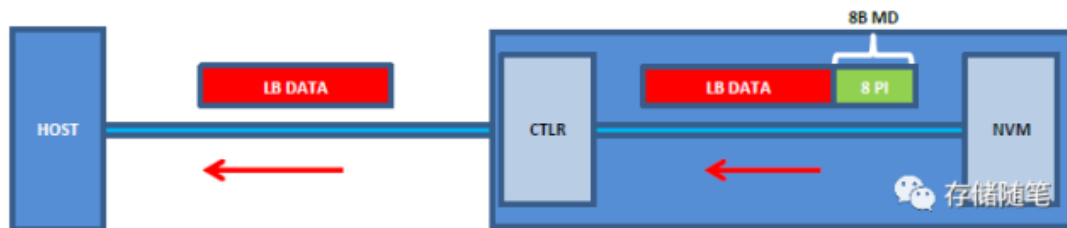
当 **PRACT=1** 时，需要注意 **Metadata** 大小的区别：

(1) 当 **PRACT=1, Metadata=8 Bytes** 时，Host 与 SSD 之间没有 E2E 保护，当数据到达 SSD 之后，主控会在计算出 PI 的信息并添加到 LBA data 之后，最后写入 NAND。

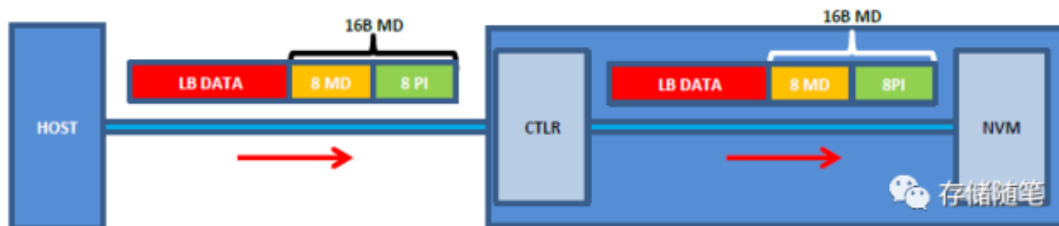




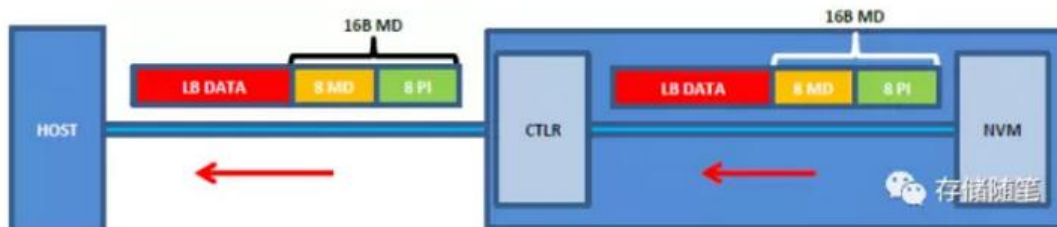
Host 读数据的时候，SSD 主控会将 PI 去掉之后的数据传送给 Host。



(2) 当 $PRACT=1$ ，Metadata>8 Bytes 时，在 Metadata 通过主控时，主控会重写 PI 区域，最后将数据和 Metadata+PI 一起写入 NAND。小编觉得这部分与 $PRACT=0$ 的 PI 处理过程差不多。



Host 读数据的时候，SSD 主控会检查 PI 是否正确，之后保持 PI 信息不变，与数据一起传送给 Host。



更多精彩内容，敬请关注微信公众号：存储随笔，Memory-logger.



声明：本教程版权归微信公众号“存储随笔”所有，支持原创，转载请注明出处！