A Comparison of NVMe and AHCI

By Don Walker, Dell

Version: 1.0

Date Created: 7/31/2012

Table of Contents

Revision History

Date	Revision #	Explanation of Changes	Author
04/02/2012	0.5	Initial working draft	Don H Walker
07/31/2012	1.0	Promoted after final review	Don H Walker

NVMe and AHCI

1 Introduction

NVMe and AHCI were created to solve different problems. Each design addresses the problems within the constraints that existed at the time they were architected and take advantage of and fully exploit the technologies that were available at the time they were defined.

AHCI came about due to advances in platform design, both in the client and enterprise space, and advances in ATA technology, specifically SATA. SATA in particular, which is an adaptor-to-device side interface, highlighted the need to provide an updated adaptor-to-host side interface. Many features of SATA were either not capable of being exploited or could only be exploited with difficulty through proprietary host side interfaces prior to the creation of AHCI.

AHCI was originally created to provide an abstracted device host interface to SATA hard disk drives that provided a queuing interface supporting an asynchronous command-response mechanism. Other important features, power management, a better architected application programming interface, etc., were also added. The most important feature, the queuing interface, was added in order to improve performance.

NVMe is designed as a device interface. More specifically it is designed as a device interface for devices that attach directly to the PCIe bus. The devices may function as either a storage device or something that, in the future, may function as a new device type sitting between the storage subsystem and system main memory.

NVMe was designed from the ground up to provide a very high performance, low latency interface for PCIe SSD devices that would additionally allow some of the special characteristics and attributes of NAND flash to be exposed and exploited by upper layers of the system.

Prior to the creation of NVMe AHCI, in a modified and proprietary form, has been used as an interface for PCIe SSDs. While this approach has worked in the absence of something more purpose designed and built, AHCI was not originally architected with this use in mind and therefore suffers from some shortcomings when used in that application.

This paper will provide an overview of the features of both interfaces to allow for comparisons and contrasts to be drawn between the two and allow the reader to better understand in what applications each may best be used. The focus will be on the higher levels of the interfaces, essentially the transports, protocols and upper level features. A discussion of the physical and link layers of each of the interfaces is beyond the scope of this paper and not essential to the topic.

A comparison of an AHCI HBA subsystem to an NVMe device is admittedly something of an apples to oranges comparison. In configuring a system it would not be expected that someone would be faced with making a decision as to whether that system should be configured with an AHCI HBA or NVMe PCIe SSDs. Each serves different purposes. The reason for this paper is educational. It is hoped that the reader can benefit from a comparison for the following reasons.

- If the reader is already familiar with AHCI then a comparison to NVMe will help in the understanding of NVMe and its benefits
- If the reader is unfamiliar with either interface an understanding of both will help to point out the context of how each can best be used.
- Finally this comparison will show how storage interfaces are evolving in response to advances in persistent memory technology and the systems in which they are used.

2 Overview

The differences between an interface designed as an adaptive, translational layer (AHCI) and an interface designed as a simple device adaptive layer (NVMe) drive fundamental architectural and design features that are inherently encompassed in the respective interface specifications. These differences are in addition to various features that are included in a specification designed to expose characteristics and attributes of the underlying attached device technology.

2.1 AHCI

Inherent in the problem of designing AHCI, in fact many interfaces, is that it serves as the logical element that tie two physical buses together, the system interconnect, PCI/PCIe, and the storage subsystem interconnect, SATA.

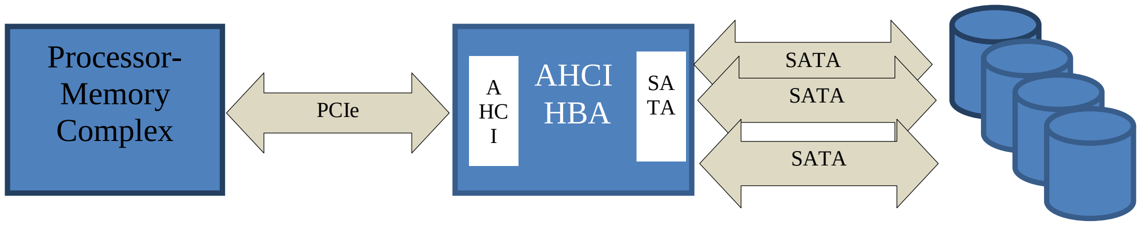


Figure 1 Host Bus Adaptor

Adapters built on such interfaces typically provide logical translation services between the two sides of the adapter. The electrical signaling and the physical transports and associated protocols on either side of the adapter are different, PCI/PCIe on the host side and SATA on the storage device side. The resulting logical transport and protocol implemented by the adaptor and used to bridge between the two physical interconnects is then shaped by the underlying physical and link layer elements of the buses which it connects.

At the time AHCI was conceived and designed the only devices connected via SATA were IO devices such as hard drives, optical drives, and other IO peripheral devices that were slow as compared to the processor-memory complex of the platform. Given this performance disparity an aggregation point in the system topology was needed, serving as a protocol/transport translator and as an elasticity buffer and alleviating the processor from managing this disparity. An HBA using AHCI as the host side interface and SATA as the device side interface serves this purpose quite well. This aggregation/translation point serves to smooth the bandwidth and latency disparity between the SATA storage interconnect and the system interconnect, in this case PCI or PCIe.

In providing such functionality latency is introduced into device access. For the originally intended environment of AHCI this is not an issue as the additional latency of the aggregation point, the HBA, is a miniscule component of the overall device access path latency.

AHCI serves its intended architecture and design goals well. The host is able to keep attached SATA storage devices busy while minimizing the effort needed to manage the devices and take full advantage of additional features that they may offer.

2.2 NVMe

The deployment of SSDs, with performance capabilities orders of magnitude greater than previous storage devices, especially the low latency characteristics of the devices, drove the transition from physical attachments based on traditional storage busses to physical interconnects more closely tied to the processor-memory complex, namely the PCIe bus.

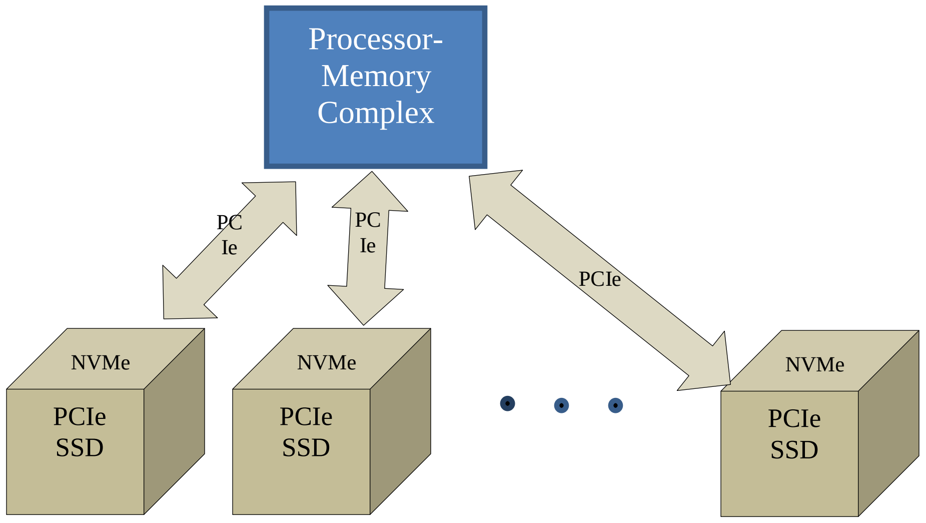
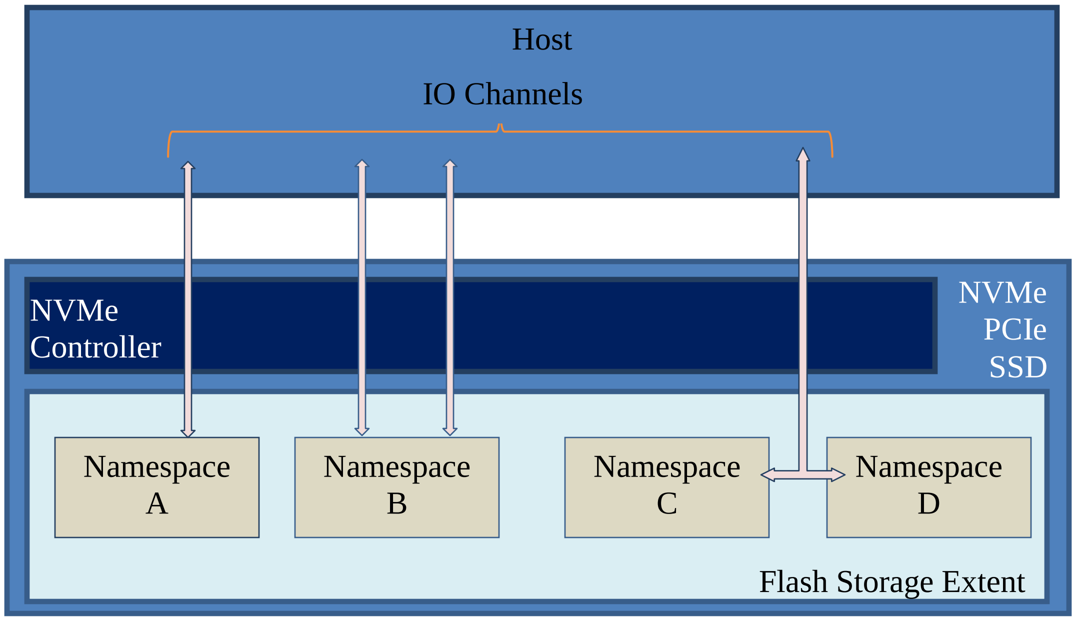


Figure 2 PCIe Attached Solid State Storage

With a storage device moving from a legacy storage interconnect to the low latency system interconnect the need for a new storage device interface that could span both the storage domain and function equally well within the system interconnect domain and unlock the full potential of these new devices was required. NVMe is that new interface.

The interface was also designed to be highly parallel and highly scalable. The scalability, parallelism and inherent efficiency of NVMe allow the interface to scale up and down in performance without losing any of the benefits. These features allow the interface to be highly adaptable to a wide variety of system configurations and designs from laptops to very high end, highly parallel servers.

Another important feature of the NVMe interface is its ability to support the partitioning of the physical storage extent into multiple logical storage extents, each of which can be accessed independently of other logical extents. These logical storage extents are called Namespaces. Each NVMe Namespace may have its own pathway, or IO channel, over which the host may access the Namespace. In fact, multiple IO channels may be created to a single Namespace and be used simultaneously (Note that an IO channel, i.e. a submission/completion queue pair is not limited to addressing one and only one Namespace; see the NVMe specification for details.).



The ability to partition a physical storage extent into multiple logical storage extents and then to create multiple IO channels to each extent is a feature of NVMe that was architected and designed to allow the system in which it is used to exploit the parallelism available in upper layers of today's platforms and extend that parallelism all the way down into the storage device itself.

Multiple IO channels that can be dedicated to cores, processes or threads eliminate the need for locks, or other semaphore based locking mechanisms around an IO channel. This ensures that IO channel resource contention, a major performance killer in IO subsystems, is not an issue.

2.3 Layering Aspects of Both

Both AHCI and NVMe are interfaces that rely on PCI/PCIe to provide the underlying physical interfaces and transports. An AHCI HBA will plug into a PCI/PCIe bus. A PCIe SSD implementing the NVMe interface will plug into a PCIe bus. Either interface could be enhanced to work with devices that plug into a systems interconnect of a different type but this has not been done to date as PCI/PCIe is the dominant systems interconnect and a need for additional physical interconnect support is not there.

That said, AHCI is commonly implemented within the processor chipset. The processor side physical bussing is implementation dependent and may or may not be PCI/PCIe. When the AHCI controller is implemented within the processor or processor support chipset and connected via the PCI/PCIe bus it is a referred to as a Root Integrated Endpoint. The bus connecting the processor to the AHCI adapter within a processor or processor support chipset may be proprietary. Those cases are not germane to this discussion and any further discussion on this topic is beyond the scope of this paper. The remainder of this paper will assume that the host side physical interconnect is PCI/PCIe.

3 Anatomy of the Interfaces

Before going into a detailed analysis of the two interfaces it will be helpful to establish a common understanding of the elements of an interface and establish a common terminology associated with an interface. In this section I'll present my own ontology and an associated terminology. It should serve the purpose of creating the foundations needed to position each of the discussed interfaces with respect to one another.

3.1 Overview

An interface can be decomposed into various components. This paper uses the following decomposition.

- Transport
  - Logical
  - Physical
- Register Set
- Queuing Layer or Model
- Protocol
- Command Set

3.1.1 Transport

Fundamentally a transport is the mechanism by which information is conveyed or moved from one point in a system to another in an orderly and predictable manner. A transport is a combination of other elements of the interface. Interfaces depend on transports. Transports can be physical or logical. Logical transports can run on top of physical transports. Transports can be layered or tunneled. Transports are also sometimes referred to as protocols, but in this paper the two terms have distinctly different meanings.

As described in the previous section both AHCI and NVMe are layered on top of PCIe. PCIe has its own transport and protocols that include a physical and data link layer that is not included in either AHCI or NVMe. Both AHCI and NVMe are logical transports that are layered on top of the underlying physical bus transports and protocols on which they are implemented.

3.1.2 Register Set

Each interface defines a register set that is mapped into various PCI address space types. In addition to the registers required by the PCI specification, registers that map into the PCI Configuration Space, there are registers that are defined to describe and control the device itself. These latter registers map into the PCI/PCIe bus address space which are in turn mapped into the system's address space. These register sets can be grouped into,

- Capabilities Registers
- Configuration Registers
- Status Registers

Registers of these types exist in both interfaces. They allow the user to configure operational aspects of the device, provide status on the device and to operate access channels to the storage extents within the various device domains.

3.1.3 Queuing Layer

Most interfaces use queues for moving commands, command status and/or data across the transport. Queues are typically used for smoothing the flow of information and data, functioning as elasticity buffers, and possibly maintaining command ordering of some type. They provide the buffers needed for temporary storage of command and command status until they can be processed by the host or device.

Queuing layers must have usage rules associated with them. The mechanisms used to update queue head and tail pointers and notifications of when a queue has been updated to interested parties are all part of the queuing layer and its associated protocols.

3.1.4 Protocol

The elements that make up an interface, whether physical or logical, have rules of usage. These rules make up a protocol. The term "protocol" will mean different things to different people and may even mean slightly different things in usage by the same person depending on context. In this paper my usage of "protocol" is as follows.

"The set of rules that apply to the usage of the various physical and logical components of an interface or transport that must be adhered to by all entities, whether initiators or

targets/requestors or responders, that connect to an interface or transport, either physical or logical, in order to facilitate the correct and orderly exchange of information."

3.1.5 Command Set

The definition of an interface may include the definition of a command set. In this paper a command set will be defined as the set of tasks to which the interface is dedicated. For a storage interface typical commands would be read, write, trim, etc.

4 AHCI

This section will describe the various components of the AHCI interface and how these elements combine to form a functional interface.

4.1 Interface Elements

4.1.1 Register Sets

The complete set of registers exposed by an AHCI HBA interface are described in the SATA AHCI specification, and not duplicated here. Some key registers are;

- Capabilities registers - Describe support for optional features of the AHCI interface as well as optional features of the attached SATA devices. Examples of the former include support for 64 bit addressing and power state capabilities. Examples of the latter include support for NCQ, staggered spin-up and interface (SATA) speeds.

- Configuration registers - Allow the host to configure the HBA's operational modes. Examples of specifics include the ability to enable/disable AHCI mode in the HBA, enable/disable interrupts and reset the HBA.

- Status registers - These registers report on such things as pending interrupts, implemented "ports" (described in the following paragraphs) timeout values, interrupt/command coalescing and HBA readiness.

4.1.1.1 Port Registers

AHCI implements the concept of ports. A port is a portal through which a SATA attached device has its interface exposed to the host and allows host direct or indirect access depending on the operational mode of the AHCI HBA. Each port has an associated set of registers that are duplicated across all ports. Up to a maximum of 32 ports may be implemented. Port registers provide the low level mechanisms through which the host access attached SATA devices. Port registers contain primarily either address descriptors or attached SATA device status.

4.1.2 Transport/Protocols

The AHCI transport and associated protocols were shaped in large part by the need to be compatible with a large body of software created for IDE/ATA devices. This aspect of the interface is presented to the host through the AHCI programming API.

The physical transport and protocol for the storage side of an AHCI adaptor is SATA. When operating an AHCI HBA in legacy mode the transport/protocol emulate the legacy IDE/ATA interface. When operating in AHCI mode SATA still determines the transport and associated protocols, but the API exposed to the upper layers of the driver stack and host system software are determined by both the queuing model of AHCI as well as the ATA command set and the additional fields of the AHCI Command Table. The details of the ATA command set and the AHCI Command Table fields are beyond of the scope of this paper. See the ATA/ATAPI Command Set – ACS and the Serial ATA Advanced Host Controller Interface specifications for that information.

4.1.3 Queuing Model

The queuing model implemented by AHCI allows SATA device performance to be exploited to full advantage while simultaneously removing some of the device management burden from the processor. An AHCI interface has the ability to support up to 32 SATA ports. Each port has a command list associated with it. A port command list may have up to 32 entries.

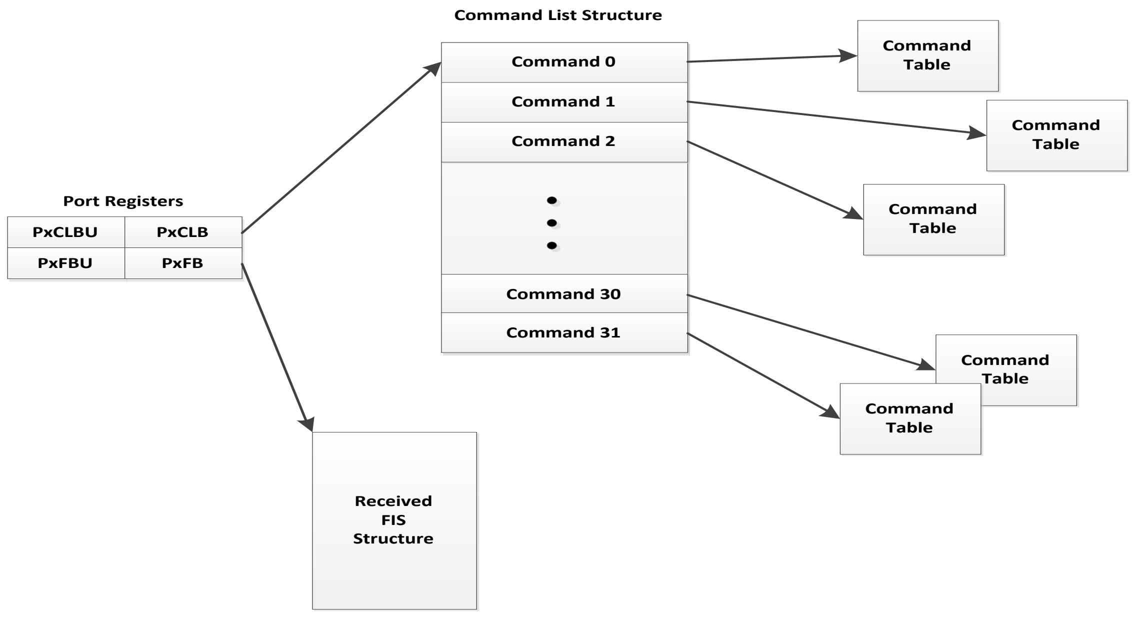


Figure 4 AHCI Command List and Command Tables

A Command List is a memory resident table data structure, not a queue. Each port will have a register which contains the base address of the Command List. Since this is a simple table, and not a true queue, no head and tail pointers, as is typical of a queue, exist.

The Command Table is a structure residing in host memory that is pointed to by an entry in the Command List. Each Command Table will contain one command with associated command parameters and other command metadata.

Return status is contained within another host memory based table termed the Received FIS Structure. Each port has its own Received FIS Structure. This table can contain up to four known FIS types as well as space to hold a FIS of unknown type. The details of each of the FIS types are beyond the scope of this paper. See the ATA specification for details.

Each port then has its own set of command and command return status data structures and elements independent of any other port. Both command submission and command completion data structures are table based, not queue based. How these data structures are used is described in later sections in this paper.

4.1.4 Command Set

The command set used by AHCI is the ATA command set. When an AHCI HBA is operating in AHCI mode the Command Table contains additional information that is associated with an ATA command. See the SATA AHCI specification for details.

4.2 How It Works

4.2.1 ATA Frame Information Structure

Before we begin to explore the way commands are issued in AHCI an understanding of the basics of the ATA Frame Information Structure is helpful. This section will provide a very brief overview. For more information see the ATA specification.

The programming interface of an IDE/ATA device appears as a set of registers that taken together are called a Frame Information Structure. The physical instantiation of these registers has long been replaced by software constructs that “shadow” the original physical registers. To software a Frame Information Structure (FIS) still appears as a register set. Each device implements a set of host-to-device and device-to-host FIS “registers” to support the legacy mode of AHCI HBA operation. In pure AHCI operation

ATA commands are still transferred but additional information is provided by the host side software and driver stack to support the greater feature set available in AHCI.

Originally the FIS registers were used in a simple synchronous manner in which commands were written via the host-to-device FIS (H2D FIS) and status returned via the device-to-host FIS (D2H FIS). The synchronous, interlocked nature of information exchange limited the performance of the interface.

AHCI created a queued interface where the host could build command lists that are processed by the HBA and device en-mass. This command batching provides the underlying mechanism that allows greatly improved SATA subsystem performance and is implemented via a queuing model explained in the following section.

4.2.2 Command Issuance

Issuance of a command to a SATA device connected to the host through an AHCI adaptor is a matter of constructing the command, staging it within an area of host memory accessible to the adaptor and then notifying the adaptor that it has commands staged and ready to be sent to the storage end device.

A command is contained within a data structure called a Command Table. A Port may have up to 32 Command Tables associated with it. Each Command Table contains one command and associated parameters and other metadata. The Command Table contains a number of fields. Briefly the fields are a Command FIS, ATAPI Command and a Physical Region Descriptor Table. Depending on the requested operation not all fields will contain valid data.

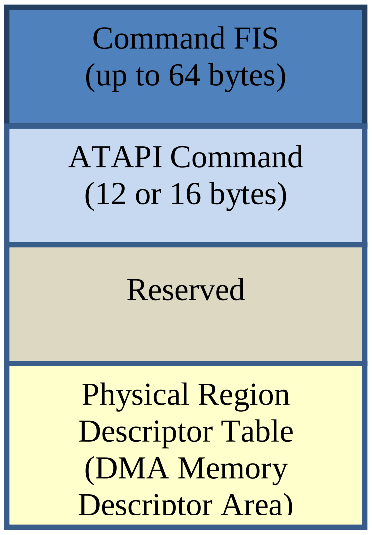
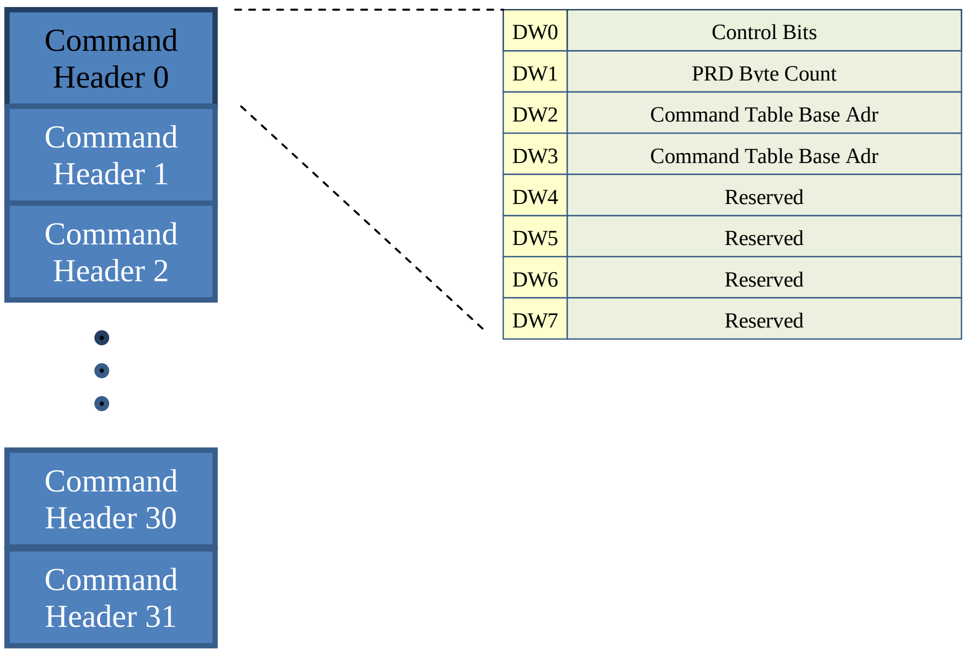


Figure 5 AHCI Command Table

A Command Table is in turn pointed to by an entry in another data structure called a Command List. A Command List is simply a table of pointers to Command Tables. A Command List is associated with each of the enabled 32 possible ports within an adaptor. The Command List contains, in addition to the pointer to the Command Table, meta-data associated with the processing of the command contained within the Command Table and the operational aspects of the port itself. A Port's Command List is pointed to by two registers associated with that port and contains the port's Command List base address. Both the Command List and the Command Table are resident within host memory.



DW0	Control Bits
DW1	PRD Byte Count
DW2	Command Table Base Adr
DW3	Command Table Base Adr
DW4	Reserved
DW5	Reserved
DW6	Reserved
DW7	Reserved

Figure 6 AHCI Command List Structure

The first step in constructing a command is for the host to allocate memory for the Command Table. Memory for Command Tables can be allocated statically when the host initializes the adaptor or can be dynamically allocated each time a command is issued. Given that the memory will be kernel memory the static allocation method is the method more commonly utilized.

The memory for each port's Command List is allocated statically due to the fact that adaptor registers must be initialized with the base address of the Command List and it is inefficient (and incompatible with the functioning of an AHCI adaptor), to rewrite hardware based registers repeatedly.

Once memory has been allocated for the Command List and Command Table the Command List is initialized with pointers to the Command Tables and other metadata associated with the command. Information such as the FIS length and direction of the command (i.e. whether this is a read or write command, etc.) is placed into the Command List entry. The Command Table is initialized if it has not been done previously. The Command Table will contain the command and its associated parameters in the form of a FIS and possibly associated scatter/gather lists depending on the command type.

A Command Table is variable length, but will have a minimum size of 128 bytes and a maximum size of 1048688 bytes. The ultimate sized being determined by whether or not a scatter/gather list is included and if so how many scatter/gather list entries make up the list.

Prior to initializing the Command List and Command Table entries host software (typically the AHCI driver) must ensure that any previous commands occupying the Command List slot(s) have completed. This is done by a series of adaptor port register reads. Two registers, the Port Command Issue and the Port Serial ATA Active register are used for this purpose; both registers serve dual purposes.

The Port Command Issue register serves to indicate to the host whether or not a command at a particular Command List entry has completed or is currently being processed. This register is a bit mapped register with each bit position mapping to one of the 32 possible associated Command List slots. In operation the host reads the register looking for a command that has completed. If the command is an NCQ command it must also perform a similar function on the Port Serial ATA Active register. Since NCQ commands have an associated TAG, this register is used to indicate whether a particular command TAG is outstanding or is available.

Once an available command slot has been located, and an available TAG, if it is a NCQ command, has been identified, and the Command List and Command Table(s) have been initialized the adaptor is notified that new command(s) are available for processing. This is done by host software writing the Port Command Issue register, setting the register bit position that corresponds to the newly initialized Command Table entry. Prior to writing the Port Command Issue register a similar action is performed on the Port Serial ATA Active register if the command is a NCQ command. At that point the Command List slot and the associated Command Table must not be touched by the host until the command completes. It effectively belongs to the adaptor.

A port’s Command List and Command Table resources, combined with the adaptor’s port registers that provide associated control and status information, effectively form the core of the transport and associated protocols for AHCI. These resources are not sharable between multiple threads. If two or more threads or processes wish to access the same port or a port's attached devices then a mechanism, a lock, mutex, or other semaphore, must be used to coordinate access to the port. Such coordination is expected to take place at a layer above the adaptor. A description of how this might work is beyond the scope of the AHCI specification and this paper.

4.2.3 Command Completion

Command completion is provided through mechanisms and constructs that are built on the SATA protocols. On command completion SATA devices return a Device-to-Host Frame Information Structure, D2H FIS for short. Additional FIS types may play a role in command completion depending on the type of command that was issued and how it was relayed to the device, i.e. the FIS types used to communicate the command to the device. Regardless of the FIS types used, the purpose of the completion FIS is to communicate command completion status as well as to update overall device status[1].

The return status FIS is contained within a host memory based table termed the Received FIS Structure. This table can contain up to four known FIS types, DMA Setup FIS, PIO Setup FIS, D2H Register FIS and a SDBFIS. The table also contains space to hold a FIS of unknown type. The details of each of the FIS types are beyond the scope of this paper. See the ATA specification for details.

At the time the host initializes the adaptor it will allocate host memory for the purpose of accepting received device FIS information. Each port of an adaptor has its own area of host memory reserved for this purpose. The received FIS is copied into the host memory area by the adaptor. This buffer area is used in a specific manner. Certain FIS types are placed into specific offsets within the Received FIS buffer area. The buffer area contains only enough space for each one of the FIS types at a time.

The Received FIS area only needs to be read if the command completed with an error or was not processed due to an error. A check of a few status registers will indicate whether the command completed successfully or not. If no error condition is present the Received FIS is not of interest and the host doesn't care if it is overwritten by the next received FIS.

Notification of command completion can be via interrupt or polling. The controller may be configured to generate an interrupt on command completion or the host may choose to poll the port's Command Issue register and, if the command is a NCQ command, the Serial ATA Active registers. If the host chooses to be notified of command completion via interrupts, then on interruption the host will have to read the contents of three,

[1] This is the way non-NCQ command return status works. NCQ commands use a slightly different process which is not described here.

possibly four, controller registers. The host will have to read the controller's interrupt status register to determine which port has caused the interrupt, read the port interrupt status register to discover the reason for the interrupt, read the port's Command Issue register to determine which command has completed and finally, if the command is an NCQ command, read the port's Serial ATA Active register to determine the TAG for the queued command.

4.2.4 Error and Exception Handling

Errors only occur on the device or the device transport. There are no errors that occur on the adaptor. The adaptor is capable of detecting and reporting a wide variety of device and device interface errors. These errors are typically recorded in status registers where they may then generate an interrupt.

However not all errors will cause interrupts. It is expected that host software will poll the status registers to detect the occurrence of these errors and take corrective action. Some errors that occur may result in the cessation of Command List processing on the port.

In general it is expected that the host play a significant role in error/exception handling.

4.2.5 Additional Features of the Interface

4.2.5.1 Power management

SATA defined features to support power states for both attached devices, i.e. hard disk drives, as well as the SATA interconnect itself. There are four power states defined for SATA devices that are somewhat modeled on the PCIe device power states. They are,

- D0 – Device is fully powered and working
- D1 – Device is an IDLE mode.
- D2 – Device is in STANDBY mode
- D3 – Device is in SLEEP mode.

The device state for each of these modes is defined in the SATA specification. Exit latencies, how a device enters and exists these power states and other characteristics are defined in the SATA specification and not repeated here.

The power state of the SATA interconnect can also be controlled. The elements of the interconnect that consume power, the phys, are defined to have three power states,

- Phy Ready – Fully powered and functional
- Partial – Phy is powered in a reduced state.

- Slumber – Phy is powered in a reduced state.
- DevSleep – Phy may be completely powered down

The primary difference in Partial, Slumber and DevSleep, aside from power consumption, is the exit latency. It will take a phy longer to return to full power from DevSleep than it will from Slumber and longer from Slumber than it will from Partial.

The HBA, as a PCI/PCIe device, has power states that are defined and controlled by the PCI specification.

The SATA device and SATA interconnect power states are controlled through a variety of mechanism. The device may change power states under the control of itself, the HBA or system software. If a SATA device supports the low power states and is placed into the appropriate mode it may transition into and out of low power states based on activity, i.e. whether or not it has received commands within some designated period of time. It may also be placed into various power states by commands issued to it by the HBA or the system software when either determines that appropriate conditions, such as periods of inactivity, have been met. SATA interconnect power states are entered into and exited by similar means and mechanism.

The SATA and AHCI specifications define mechanisms that support all of the above features/functions.

4.2.5.2 Hot Plug

Hot plug of SATA devices is supported in AHCI through both out-of-band and side-band signaling mechanisms. Out-of-band is supported via external signals that are detected by the AHCI HBA causing status bits to be set within the HBA's registers. The registers are defined by the AHCI specification. Optionally, depending on how the AHCI interface is configured, the HBA may generate an interrupt notifying the host that a device has been removed.

Side-band signaling [2]of device insertion and removal is also supported. SATA COMINIT and COMRESET are used to support this functionality. This method relies on detection of electrical presence detect at the phy level. As a result when combined with enabled power management features the detection of insertion and removal is not always

[2] The use of the terms side-band versus out-of-band signaling may not be consistent with the way these terms are used today. My usage is that of the original radio communications based usage of these terms. Side-band is essentially communication of information, possibly with a different protocol, over the same carrier as the in-band information flow. Out-of-band communications is the flow of information over a carrier separate from the in-band communications carrier.

reliable. If hot plug functionality is to be supported solely through side-band signaling mechanisms it is recommended that power management features of AHCI be disabled.

Regardless of whether out-of-band or side-band signaling is used, this represents a significant feature of AHCI and helps standardize and simplify the support of this feature in the host.

4.2.5.3 Miscellaneous

In addition to hot plug and power management a number of other features are supported by AHCI. Support of these features in the interface via a standard method helps to simplify host support. Without going into detail, the additional features are,

- Staggered Spin-up
- Command Completion Coalescing
- Port Multipliers
- Enclosure Management

Taken as a whole these features provide for significant advancement in support for SATA devices that allow a greatly enhanced user experience through performance, management and system architecture and design.

5 NVMe

This section will describe the various components of the NVMe interface and how these elements combine to form a functional interface.

5.1 Interface Elements

5.1.1 Register Sets

The register sets of an NVMe interface are conceptually very similar to those of AHCI and many other interfaces. Like AHCI there are PCI Configuration Space registers that are defined by the PCI specification and there are capability, control and status registers that reside in PCI bus address space which is in turn mapped into the processor/system address space.

Just as with the AHCI interface NVMe controller capabilities registers provide for a mechanism through which the host can discover the implemented capabilities of the device. Remember that unlike an AHCI HBA which serves as an aggregation point and is the portal through which the host may access multiple SATA devices, an NVMe interface is 1-to-1 with the end-point device[3]. Examples of key registers and their functions are;

- Capabilities registers - Command submission/completion queue size, arbitration mechanisms supported and command sets supported. More on these features later.
- Configuration registers - Allow for setting the submission/completion queue sizes, selecting arbitration mechanisms, enabling the device controller and other such device global functions.
- Status registers - Allow for the reporting of global device controller status such as the transitional device states during an orderly shutdown and global error/exception conditions.

There are numerous other device state control and status information details that can be obtained and set through device controller, configuration and status register fields and bits. See the NVMe specification for more detail.

5.1.2 Transport/Protocols

Unlike AHCI the NVMe architecture and design started with a clean sheet of paper. There were no legacy interfaces and associated software stacks with which backward compatibility needed to be maintained. This has allowed an interface design that is optimized to take advantage of those very characteristics that make NAND flash based devices so attractive. The transport and associated protocols are optimally designed to exploit the advances in both the underlying flash technology and the architectural advances of the platform and operating systems that run them.

Multiple IO channels or communications pathways between the host and device that carry commands and return status may be created. These IO channels are implemented through command submission and command completion queue pairs. Having the ability to create multiple IO channels, which may operate in parallel and independent of one another, is fundamental to the transport model of NVMe.

Optimizations in the design of the interface were considered at every stage of the architecture and design process. The information unit (IU) that is conveyed over the PCIe bus, carrying commands and status, is designed to be optimized for performance

[3] Ignoring for the moment that a PCIe Endpoint may implement multiple PCIe Functions and each Function may implement its own NVMe interface. I'll not go into this level of complexity here for the sake of ease of explanation.

yet also adaptable to future technology advances. The size of an IU is fixed[4] to support easy parsing of the information stream. A minimal number of PCIe posted writes are needed to move commands and status in and out of the command and status queues. No non-cacheable reads are required.

And finally the NVMe transport supports a QoS mechanism based on command queue arbitration. Priorities may be assigned to command queues. NVMe will provide inherent support for simple round robin, weighted round robin with urgent priority or a vendor defined priority scheme.

In order to ensure that the interface was not an obstacle in the performance path, now or in the future, every aspect of the transport was designed to be as efficient as possible. Since performance, in particular low latency, is the primary value proposition of NAND flash and upcoming resistive memory technologies, NVMe is designed to ensure that the full value of devices based on these persistent memory technologies can be realized.

5.1.3 Queuing Model

NVMe uses a command submission/completion queue pair based mechanism. The basic model of a command submission queue associated with a completion queue returning command status is the fundamental concept on which NVMe queues are built and is not new. What is new about the queuing model of NVMe is that it allows the creation of multiple submission and completion queues and allows for multiple submission queues to be associated with a completion queue. *See Figure 7 Command Submission and Completion Queues*

NVMe has provisions that allow a system configurator to create up to 64K command and completion queues. Each queue may contain up to 64K commands. Of course smaller configurations are possible. A minimum of one submission/completion queue pair is supported. This is in addition to a mandatory admin submission/completion queue pair.

Multiple command and status queues can be created to support the parallelism that is now common in processors and exploited in operating systems with growing regularity. The queues are also designed in a way that does not require locking mechanisms to be put into place to ensure that queue updates maintain the integrity of the queue contents. Register reads/writes that are used to notify the device and host of queue updates are kept to an absolute minimum.

[4] The interface defines a mechanism where the size of the IU may be changed but will always be of a fixed size. This feature allows for future upgrades and features without radical changes to the interface.

Each completion queue may have its own interrupt vector associated with it. Having the ability to assign each completion queue its own interrupt vector allows interrupts to be targeted to the core, thread or process that initiated the IO request. Such targeted interrupts allows for a more efficient utilization of compute resources. This feature further supports the parallelism available in the device and host.

While parallelism has been supported in the core elements of the platform for some time support for the same level of parallelism down to storage device is new to storage subsystems with the advent of NVMe. The queuing model of NVMe was designed specifically to enable this capability.

5.1.4 Command Set

NVMe defines its own command set. A command set that is small, a total of eight at this time, and streamlined was defined. It is optimized for both performance and to expose enhanced functionality in support of persistent memory technology on which the devices are built.

During the architectural definition phase of NVMe it was decided that a new command set was both needed and desired. Command sets for existing interfaces were deemed to be suffering from both feature bloat and from a legacy of backward device compatibility (some devices being antiquated) and would have hampered the performance and development of a new device type, PCIe SSDs.

While it was decided to create a new command set, features were also added to the interface that would allow other command sets to be added to NVMe with little or no modification to the transport. Indeed features have been added to the transport that will make the support of new, additional command sets relatively painless.

5.2 How it Works

5.2.1 Command Issuance

In order to create an interface that does not “get in the way” of high bandwidth, low latency PCIe SSD devices the interface was architected to allow multiple command submission/completion pathways or IO channels based on the queue pairs to be created.

A pathway, or IO channel, is composed of one or more command submission queues associated with a particular command completion queue. Command submission/completion queues can be created dynamically. This allows command submission/completion queue pairings to be created and assigned to cores, processes or threads as they are created and have a need for an IO channel or channels to the device. If multiple submission queues are created as a part of an IO channel an arbitration mechanism is defined such that the user may define the order in which commands are pulled from the submission queues for processing by the device.

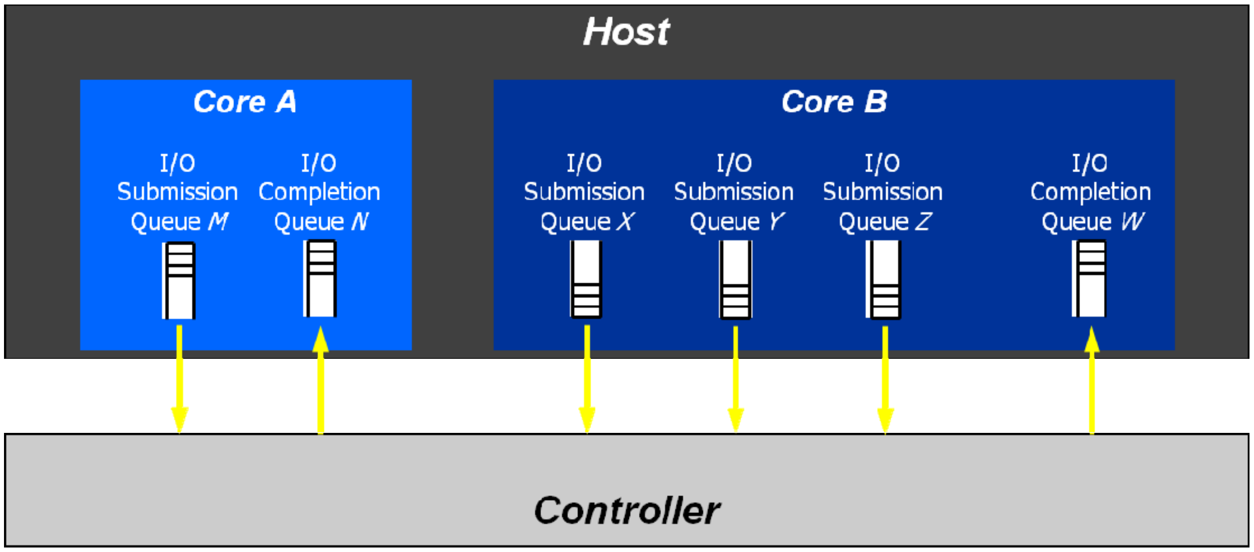


Figure 7 NVMe Command Submission and Completion Queues

IO channels can be used to access any NVMe Namespace (See section 2.2 for an explanation of NVMe Namespaces) and an NVMe Namespace may have multiple IO channels associated with it. While the ability to create multiple channels allows the system to take advantage of any inherent parallelism in the platform that alone is not enough to ensure maximum subsystem throughput. Multiple IO channels with queue pairs dedicated to cores, processes or threads eliminates the needs for locks, or other semaphore based locking mechanism around queue pairs.

However simply creating multiple IO channels to a Namespace is not sufficient to ensure maximum interface performance. The interface is designed with still other performance optimizations in mind. Every interface requires device register access. The problem with this, from a performance perspective, is that register accesses can be orders of magnitude slower than memory access. NVMe is designed so that device register accesses are kept to a bare minimum. In fact NVMe does not require any register reads in the command submission-completion cycle and only two register writes, that are PCIe posted, are needed.

Multiple command submission queues may exist at any one time, each with its own doorbell register[5]. Doorbell registers are fixed at 16 bits in length and are contiguous within the device's register address space. In this way the doorbell register index into the device's doorbell register address space may also serve as the command submission queue identifier.

One write to a doorbell register notifies the device that the host has updated the command submission queue. Another write to a completion queue doorbell register by the host will notify the device that the completion queue head pointer has been updated. Both of these writes can be amortized over multiple commands, i.e. one submission queue doorbell register write can notify the device that several new commands are available for processing.

Additional efficiency is achieved by defining the command queue entry so that the scatter/gather list associated with read and write commands can be combined into the command frame for IO operations up to 8KBs in length. This helps to eliminate additional DMA operations required to move the scatter/gather list into controller memory.

More efficiency is achieved by using fixed size entries in the submission queues. This makes parsing the queues and queue entries simpler and therefore quicker. Submission queues are circular buffers allocated within host system memory with each submission queue entry being a fixed 64 bytes in length.

The host submits a command by filling in the next empty command slot in a command submission queue. The empty slot will be pointed to by the tail pointer (tail from the perspective of the device, head from the perspective of the host). The device's doorbell register corresponding to that queue is then written with the new pointer value. The act of writing the device's doorbell register is the notification to the device that new entries have been placed on the submission queue and is ready for processing. The head pointer is updated by the host once it receives notification through the completion queue that the command(s) has been fetched.

5.2.2 Command Completion

Just as with the command submission queue the command completion queue is implemented in host memory as a circular buffer. Each completion entry is 16 bytes in

[5] A doorbell register is a hardware based register, in this case it is defined as a part of the NVMe interface and is resident on the PCIe SSD device. It holds a memory descriptor, one for each submission queue and another for each completion queue. A write to a doorbell register will, in most implementations, cause an interrupt to be generated to the device controller.

size. As with the submission queue the entries in the completion queue are of fixed size for efficiency in parsing and processing.

Performance was also a key issue in the architecture and design of the completion path. Particular attention was paid to the “good path” so that command processing overhead on the host was kept to an absolute minimum. A completion queue entry contains enough information so the host can quickly make a determination as to whether or not the command completed with or without error and a small amount of additional information needed for queue management.

A quick check of one byte out of the 16 byte field will indicate whether the command completed with error. If it completed without error then additional fields are checked to identify the completed command, (a command identifier is assigned to each command by the host when it is submitted) the submission queue from which it came (also assigned an identifier at queue creation and when taken together with the command identifier forms a unique command identifier) and a field that serves to update the submission queue head pointer.

While the completion queue and the completion queue entries are structured for efficiency, the notification of the process or thread that originated the IO request is also a very important factor in ensuring a high performance, low latency command submission/completion path. NVMe has optimized this last leg of the completion path by making full use of PCIe MSI/MSI-X capabilities.

Each completion queue can be assigned an interrupt that is specific to that queue. Up to 2K unique interrupt/queue mappings can be assigned. After the 2K limit is reached completion queues would have to share interrupts with other completion queues. The limit of 2K interrupts is not an NVMe limitation but is one of PCIe MSI-X and that is a “soft” limit which may be increased without significant effort. The ability to create completion queues which each has its own ability to interrupt the host on command completion queue update is key to supporting the parallelism and low latency requirements of many systems and applications.

To avoid possible interrupt storms created by the rapid completion of many commands that are processed quickly, NVMe defines a command completion/interrupt coalescing feature. This feature allows for the system or user to set both or either of a minimum number of command completions before an interrupt is generated or a timeout value which ensures that a small number of completions not reaching a coalescing high water mark, do not suffer undue latencies prior to system notification.

A host can choose to be interrupted or can poll for command completions. In either case once a command completion arrives processing that completion is a simple matter of checking the completion structure to identify the source of the command (submission queue identifier and command identifier) checking the completion status, a quick read of one byte from the completion structure, and then reading the submission queue head pointer update field so that the associated submission queue head pointer can be updated. The last step is removing the completion entry from the queue by updating the completion queue head pointer register on the device. So in total a completion will consist of a DMA of 16 bytes of completion status from the device into host memory followed by an interrupt (optional) and then, once the host has processed the information contained in the completion structure, an update of the device's completion queue head pointer register. Compared to most storage interfaces this is a very quick process with very low processing overhead.

5.2.3 Error/Exception Handling

Errors are communicated to the host through a hierarchical architecture. A coarse granularity of error reporting is contained within the command completion structure. The field in this structure, the Status Code field, is qualified by an additional field in the completion structure known as the Status Type field. A field in the completion structure will indicate whether the reported error is transport, command or media related. An additional field will indicate whether or not the device believes a retry of the same command may succeed. Between these fields it is possible for the host to quickly decide on the course of error correction/recovery it should attempt. If it is determined that more error information is needed than is provided in the initial completion structure the host may query the device via the Get Log Page command.

One of the additional benefits of having the ability to create multiple submission/completion queue pairs that are independent of one another is that an error on one IO channel should not interrupt or inhibit IO traffic on other IO channels. In this way the host can take corrective action on the miss-behaving IO channel, up to and including destroying the channel, without affecting the availability of the device via other IO channels.

5.2.4 Additional Features of the Interface

5.2.4.1 Power Management

Unlike AHCI, where an adaptor might have a number of devices connected to it each of which may implement power management features independently of the adaptor, an NVMe device is self-contained. That is power management features exposed by the interface act only on the NVMe device itself. NVMe provides mechanisms through which the host can manage subsystem power statically or dynamically.

As a PCIe device, the interface may implement any of the various power states as defined in the PCI power management specification. Each of the D0 through D3 power states may be supported, however due to the typical usage models of PCIe SSDs it is not recommended that D1, D2 be implemented. Additional details of the static power states supported and constraints placed on the device through the interface are available in the NVMe specification.

Dynamic power states of the interface are defined as sub-states of the PCI D0 static power state and are modeled after and compatible with the PCIe Dynamic Power Allocation (DPA) functionality. Support for this feature is optional.

As with PCIe DPA up to a maximum of 32 power states are defined. At least one must be supported and any number up to the maximum of 32 may be supported. These sub-states of the PCIe D0 power state are accessed through the Set and Get Features and Identify NVMe commands.

Note that these NVMe power states are modeled after the PCIe DPA states and are compatible with them but not dependent on them. Platforms that do not implement PCIe DPA may still take advantage of the NVMe Dynamic Power State features and functions.

Each possible power state describes read/write performance, latency and entry and exit times (state change latency) and the maximum amount of power that the device will consume in that particular state. Setting a device into any power state other than the default, full power state, must be done by host software.

5.2.4.2 Hot Plug

Hot Plug is not specifically defined by the NVMe specification. The definition is instead left to the PCIe specification.

5.2.4.3 End-to-End Data Protection

Data Protection is implemented in NVMe as an optional feature and is compatible with either or both of T10 Data Integrity Field (DIF) or the SNIA defined Data Integrity Extensions (DIX) specifications. As described in following sub-sections a generic metadata area may be associated with each data block in an LBA. The protection information is included within this defined meta-data area. In fact a meta-data field of at least eight bytes must be defined for an LBA to contain the protection information bytes if the data protection feature is to be enabled. Support for DIF types 1, 2 and 3 are defined within the NVMe specification.

5.2.4.4 Security

Security is provided for in the NVMe interface by providing optional support for user data encryption and device secure locking based on the TCG defined protocols. The interface also supports secure erase features for user data. Data may be encrypted on a per NVMe Namespace basis. Device locking/unlocking and other security features are supported. The TCG protocols are fully supported using the T10 Send and Security Receive commands.

Secure erase is supported by two different mechanisms, Cryptographic erase and User Data Erase. Cryptographic erase will delete the data encryption key associated with a NVMe Namespace. The User Data Erase functions will write over the specified data area with bits that are implementation dependent, zeros or ones[6].

5.2.4.5 SR-IOV

Single Root-IO Virtualization (SR-IOV) is supported within the NVMe specification. Physical Function (PF) implementation is defined by the PCIe SR-IOV specification. The Virtual Function (VF) implementation is vendor specific but also must be defined within the context and constraints of the PCIe SR-IOV specification.

5.2.4.6 Meta-Data

An optional feature that allows a meta-data field to be defined and associated with a particular LBA is included in the NVMe specification. The content of this field is not dictated by the specification. This field is provided as a convenience for use by upper layers of the software stack except as noted in the above section on end-to-end data protection.

5.2.4.7 Multi-Pathing

Multi-pathing is being defined at the time this paper was written. It is expected to be included in the NVMe 1.1 specification release. The functionality currently being discussed will provide for support for a fault tolerant implementation and enhance the ability of the device to function in an SR-IOV environment.

6 Interface differences

This section summarizes and outlines the important differences between AHCI and NVMe. Keep in mind that the most significant difference is in the performance goals of the two interfaces. NVMe was architected from the ground up to provide the most bandwidth and lowest latency possible with today's systems and devices. While

[6] Note that there is some disagreement as to whether the NVMe specification defines whether this data pattern is required to be written or whether the command will result in this data being returned on a read.

performance was important to AHCI, it was in the context of SATA HDDs which do not place the same demands on the surrounding infrastructure and support matrix as PCIe SSDs.

The following sections provide a summary of the differences in the two interfaces as explained in detail earlier.

6.1 Aggregation Point vs. End Point

As stated in the opening sections of this paper one of the most fundamental differences between AHCI and NVMe is that NVMe is designed as an end point device interface, while AHCI is designed as an aggregation point that also serves to translate between the protocols of two different transports, PCI and SATA. The differences drive different architectural and design goals into the interface specifications.

6.2 64K Queues vs. 32 Ports

While both interfaces support parallelism the way in which they do this is influenced by their differing design goals. NVMe can support up to 64K command submission/completion queue pairs. It can also support multiple command submission queues where command completion status is placed on a common command completion queue. The entire queue structure of NVMe was architected to enable the platform itself to establish multiple, independent IO channels down to the device. This then allows the device itself to be fully integrated into those components of the platform that are themselves parallelized; the cores, threads and IO paths of the operating systems. By providing features in the interface that enable multiple IO channels into the device from the host the motivation for the device itself to implement a parallel architecture internally is provided.

AHCI also provides a high degree of parallelism in its architecture and design. AHCI however provides this functionality as a means of allowing a host adaptor (HBA) to serve as an effective fan-out connection point to end devices; historically SATA HDDs. With AHCI host side parallelism is a side effect of this end point device fan-out capability and not an inherent architectural or design goal. The host side parallelism needed for a very highly performant device is not needed for a SATA device environment and would add unnecessary expense to an AHCI interface and subsystem.

6.3 64K Command Queue Depth vs. 32 Command Queue Depth

The difference in this component of the two interfaces is due strictly to the differences in end point performance. It is envisioned that an NVMe device will be capable of up to a 1 million IOPs in the foreseeable future. Devices based on resistive memory technologies will undoubtedly be capable of even greater performance. It would seem that a device capable of such performance would not need a queue at all. Can't the device satisfy an IO request as fast as the processor can generate them? That would be true only if the device were capable of much higher performance than the system in which it is used. This is not the case. A single high end processor with multiple cores is capable of driving approximately 750,000 IOPs today. In a multi-socket system it is obvious that a device still cannot keep pace with the load that the processor complex is capable of placing on it. With both the processor complex and the end device capable of very high IOP rates, but still of different rates, the need for an elasticity buffer not only remains but now must be of significant size. If the queue were not capable of such depth, the processor would be spending a great deal of time replenishing the command queue in order to keep the device busy. A very deep queue allows the overhead of queue management to be amortized over a greater period. The additional time between the draining of the queue by the device allows the processor to spend time on other tasks.

The 32 deep command buffers of each AHCI device port is more than sufficient to keep all attached SATA devices busy and provide for a long periods between times the processor will need to replenish the command queue. This will allow the processor to spend its time attending to other tasks rather than filling the command queue.

6.4 9 Register Read/Writes vs. 2 Register Writes for a Command Issue/Completion Cycle

AHCI requires the following steps to submit and process a completed command. These steps form the command issue/completion cycle.

Command Submission to a port.

- Read PxCI[7] to find an empty command slot in the command table.
- Read PxSACT to find an unused command tag. This step is only necessary for NCQ commands
- Write PxCLB to set the command header
- Write PxSACT to reserve the command tag. This step is only necessary for NCQ commands.
- Write PxCI to issue the command.

Command Completion assuming a good completion path and an interrupt driven completion process.

[7] Refer to the AHCI specification for a definition and explanation of AHCI register acronyms

- Read IS, the controller interrupt status register to discover which port has generated the interrupt
- Read PxIS to determine why the port has generated the interrupt.
- Read the PxSACT register to determine the tag of the completed command. This step is only necessary for NCQ commands
- Read the PxCI register to determine which command in the Command Table has completed.

If the operation is a non-NCQ operation this totals 6 register reads/writes. If the operation is an NCQ operation then this becomes 9 register read/writes. All of the reads and writes listed are controller register reads and writes. These operations are uncachable meaning the processor is not allowed to store any of the previous reads in its caches and must generate a PCIe bus access any time a new read or write is performed. It is possible for the overhead detailed here to be amortized over as many as 32 commands.

In contrast, NVMe requires just 2 register writes for the command issue/completion cycle.

Command Submission

- Write the Submission Queue Doorbell register with the new value of the submission queue pointer head.

Command Completion

- Write the Command Completion Queue Doorbell register with the new value of the completion queue pointer head.

Both submission and completion queue updates can be amortized over as many as 64K commands.

The higher command issue/completion overhead of AHCI is acceptable in an environment where the host can manage the relatively leisurely command issue/completion rate of SATA HDDs, the environment for which AHCI was designed. This is no longer reasonable with the advent of PCIe SSDs[8], which require an interface with an overall queue management protocol that can provide the lowest latency possible. This is exactly the environment for which NVMe is architected and designed.

[8] An upcoming paper will explore these same requirements in the context of SATA Express where either AHCI or NVMe can be used as a PCIe SSD interface.

6.5 Single Interrupt vs. Support For Multiple MSI-X Interrupts

AHCI has a single interrupt to the host versus the support for an interrupt per completion queue of NVMe. The differences are again driven by the requirements of the environments in which the devices that implement the interfaces are used. NVMe is architected and designed to run in a platform environment that is highly parallelized. AHCI is not as the devices which it aggregates are not capable of performing at a rate that could be utilized in such an environment. The single interrupt of AHCI is adequate for the subsystem it is designed for. The multiple interrupt capability of NVMe allows for the platform to partition compute resources in a way that is most efficient for rapid command completion, i.e. dedicated cores, threads.

6.6 Command Completion Queue vs. Completion Buffer

The command queuing design of NVMe requires that command completion status carry with it the information needed to maintain the command submission queue. It is also essential that with the possible rapid command completion rate of an NVMe device that command completion status not be required to be processed in a synchronous manner. These requirements drove the architecture and design of the NVMe command completion queue mechanisms. This is in contrast to AHCI where each port or IO channel has a buffer space that holds command completion information for a single command at a time. The relatively slow command completion rate of ATA commands by the SATA device coupled with the need for the host to consume the completion information contained within the buffer only in the event the command doesn’t complete successfully means that the need for a more elaborate completion queuing feature does not exist in AHCI. However this simplified command completion status mechanism makes it less than ideal when used in an environment where the device has a high rate of command completion.

6.7 Parallel Access to a Single Namespace vs. Serial Access to a Port

As discussed in previous sections of this paper NVMe supports the concept of Namespaces, logical partitioning of the NVMe device physical storage extent. The high command processing rate of an NVMe device, supported in large part by the parallelism built into the interface, makes it feasible to share a single device or a NVMe Namespace across multiple command initiators. In contrast the relatively slow command processing rate of SATA devices means that there is no need to support an architecture that would allow a host-to HBA interface that supports a high flow of commands to the controller. The 32 deep command queue of an AHCI port, with the attendant protocol used to fill the queue and process completed commands, is more than sufficient to support the command bandwidth of attached SATA devices.

6.8 Various Miscellaneous Feature/Function Support

Since both interfaces are storage interfaces each support features and functions that one could expect in almost any current storage interface; power management, support for data integrity and hot plug are a few. The biggest differences in feature sets come from the different basic storage technologies, rotating magnetic media versus solid state, which the two are targeted towards. Staggered spin-up for instance is a function not needed by solid state storage devices and therefore not supported in NVMe. NVMe has support for things like data hinting; a feature where an LBA range can have usage hint attributes associated with it by upper layers of the stack that will allow the end device to better manage its non-volatile memory. Other features such as multi-pathing are also being defined as this paper is written to better support the concurrent access down to the device that will better enable the parallelism found throughout the platform.

7 Use Cases

In engineering form follows function. This is absolutely true for storage subsystems and devices. Applications, the function, define IO traffic profiles. IO traffic profiles are expressed in terms of things like, access type (random versus sequential), block sizes, data set sizes, bandwidth, latency requirements and other such characteristics and attributes. These characteristics and attributes are then used to create a set of requirements that drive the architecture and design of the storage subsystem and all of the elements that go to make up the storage subsystem, the form.

What we’ll now explore is how the IO requirements of various application classes ultimately drove the development and definition of both AHCI and NVMe. From this discussion it should become apparent how devices implementing each interface are best used and how the architecture of the interface itself was driven by the functions it was intended to perform.

Enterprise applications present the most varied and demanding requirements on IO subsystems. For this reason those are the applications used here to develop the IO traffic profiles that will in turn be used to describe how each interface can best be leveraged to provide the IO services needed by the various application classes.

If we look at each of the possible IO characteristics that we commonly use, access type, bandwidth, latency, block size, data set size, and take the extreme of each, and then build a list of all possible combinations of all of those characteristics we would have a very large list. We could also find an application to match each member of that list whose IO requirements are described by any particular list element. Even database applications, commonly thought of as a single category, when examined in detail will reveal IO requirements ranging from one extreme of large data set size, large block, very low latency requirements, and sequential access patterns to those that have just the opposite, small data set size, small block, ambivalence towards latency and random access patterns.

What is common across all applications is that end users and the system administrators responsible for configuring and maintaining those systems expect IO subsystems and the systems in which they reside to perform at the highest level. This expectation is more a perception of overall system and subsystem efficiency than anything else. It is the goal of system and subsystem architects and designers to provide platforms that meet the efficiency and performance expectations of the end users regardless of the IO traffic profiles they may be required to run.

What is also common is that the amount of data that we create and retain is also growing exponentially and the demands to process that data, analyze it in some way to provide meaningful and useful information, is also growing. These last two requirements are the keys to understanding how each of the interfaces may best be exploited.

As more and more data is collected the ability to both economically store that data for a relatively long term but yet also keep the data immediately accessible for processing is a constant and growing challenge. AHCI can be used to build solutions that help solve one of those problems while NVMe can be used to build solutions that help solve the other. When used together to build IO subsystems we can create solutions that solve both the problems of economical long term storage and have the data immediately available for use.

IO subsystems used to store large amounts of data for relatively longer terms in an economical manner are currently referred to as “data tubs”. This term is very commonly associated with storage subsystems composed of large numbers of SATA HDDs. In such systems cost is a major consideration and usually the primary consideration.

The need for high performance, and high cost, HDD based storage subsystems is being replaced by a tiered storage system composed of appropriately sized and expensed elements. These elements frequently include low cost SATA HDDs and high performance PCIe SSDs. If a storage cache or tier is built using high performance, and admittedly high cost NVMe PCIe SSDs, and combined with lower cost devices comprising a data tub then a storage subsystem can be built that provides both the low cost, long term attributes needed while also providing a subsystem that has the highest available performance in the industry today. By choosing the appropriate balanced mix of AHCI/SATA HDDs and NVMe PCIe SSDs, a mix dependent on the application’s IO requirements, a cost effective and highly performant storage subsystem can be configured.

There are some classes of applications that will need only one or the other of these IO subsystem types. Some data acquisition applications acquire data at a rate that is far below the capabilities of an AHCI/SATA based storage subsystem. The data is retained for very long periods of time. Some is retained for decades. When the data is needed the time to retrieve it is not critical, as long as it is not unreasonable, and even the value of each bit is not high as the value lies in the larger data set. Such data sets include seismic and video.

At the other extreme are applications where the value of the data is directly tied to its age. The older the data the less valuable it is. The quicker the data can be analyzed the more valuable it is. Financial trading is the prime example. In these applications milliseconds can determine whether the data is critical and of extreme value or useless and of no value. In these applications NVMe based storage subsystems, with access latencies that are the lowest possible, are best utilized.

8 Summary

8.1 Futures of the Interfaces

Even though AHCI has been around for less than a full decade it is currently a mature standard. The last major [9]version of the specification, 1.3, was released in June of 2008. Not much chance of it evolving to fill other niches. SATA devices are not expected to evolve in any significant way that would require significant architectural or design changes to the AHCI specification.

NVMe is a new specification. Currently at 1.0c, the first release was in March of 2011. The next release, 1.1, is scheduled for later this year (2012). It will add many new features, multi-path and persistent reservations, etc. The interface is designed to evolve with the evolution of persistent memory technologies. It is also designed to help system implementations as they evolve and push the load-store semantics of the processor towards wider and larger domains.

8.2 Conclusion

NVMe as an interface to devices that have extremely low latency and high bandwidth characteristics has endeavored to enable the full benefit of the device to be realized by the system in which they are used. Efficiency in the transfer of commands and status was made a top priority in the interface design. Parallelism in the interface was also a

[9] A minor release, 1.3.1, was released in early 2012 to add support for DevSleep; a power management feature targeted to low power SATA mobile applications.

priority so that the highly parallel systems of today could take full advantage of multiple concurrent IO paths all the way down to the device itself. The interface supports features and functions which will allow the device to be shared among multiple initiators whether they are physical or virtual. The interface is designed so that it is scalable in a way that makes it possible to build device implementations that are cost effective for the low end system, laptops and even mobile devices, and at the same time scale up to the highest performing, most parallel servers available today. Finally, every attempt has been made in the architecture and design of the interface to ensure that it is future proofed. The eminent advent of resistive memory technologies will continue to drive innovation into platform architecture and design and into storage subsystem architecture and design. NVMe is designed to ensure that it can evolve to support such new horizon technologies without disruptive re-architectures and re-designs of either the interface or the subsystems built around it.

AHCI has been successful in realizing its original architecture and design goals. It has enabled SATA devices to be used almost ubiquitously throughout systems both small and large; from laptops to servers. It is now much easier to integrate a large number of SATA devices into a system and achieve the full performance of those devices than it was previously. The parallelism built into AHCI, while not as great as NVMe, is more than sufficient for the relatively slower SATA devices it is intended to serve. The relatively advanced features and functions introduced into SATA devices over PATA devices are exposed to the system through AHCI very successfully. Things like power management, hot plug and command coalescing are significant features that are fully enabled by AHCI and not to be overlooked.

Finally NVMe is a new interface and is still being enhanced to extend functionality even more than what is possible today. Things like multi-pathing and support for features that will allow NVMe devices to be used in shared, clustered environments will be appearing in the next release of the specification. The future for NVMe is still very much ahead of it.

AHCI, while less than a decade old, is already a mature, stable interface. It serves its intended purpose quite well. It is evolving slowly and only in response to pressures from the devices it supports, SATA devices, and the slowly evolving environments in which they are used. SATA devices, and the AHCI interface as an aggregator of those devices, may find a new usefulness as a cost effective component of a storage subsystems that are used as very large data tubs and closely integrated with NVMe based devices serving as the high performance tier.