An Architecture For A System Involving SDXI and NVMe Components

March 23, 2026

ABSTRACT: This whitepaper discusses a reference architecture for a system that is enabled with NVMe and SDXI Components.

Technical White Paper

USAGE

DISCLAIMER

The information contained in this publication is subject to change without notice. SNIA makes no warranty of any kind with regard to this specification, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this specification.

Suggestions for revisions should be directed to https://www.snia.org/feedback/.

Abbreviations

CMB	NVMe Controller Memory Buffer
CQ	NVMe Completion Queue
CQE	NVMe Completion Queue Entry
CRC	Cyclic Redundancy Check
DIF	Data Integrity Field
DMA	Direct Memory Access
HBA	Host Bus Adaptor
IDE	PCIe Integrity and Data Encryption
IOMMU	Input Output Memory Management Unit
ISA	Instruction Set Architecture
LBA	Logical Block Address
NIC	Network Interface Controller
NVM	Non-Volatile Memory
NVMe	Non-Volatile Memory Express ® Protocol
PASID	Process Address Space ID
PCIe	Peripheral Component Interconnect Express ® Protocol
PMR	NVMe Persistent Memory Region
SDXI	Smart Data Accelerator Interface
SED	Self-Encrypting Drive
SQ	NVMe Submission Queue
SQE	NVMe Submission Queue Entry
SMMU	System Memory Management Unit

Executive Summary

This whitepaper discusses a reference architecture for a system that is enabled with NVMe and SDXI Components. In such an architecture, this whitepaper envisions facilities available in NVMe and SDXI to complement a storage read and write flow to NVMe subsystems. No extensions to SDXI or NVMe specifications have been proposed in this whitepaper.

Motivation

SDXI is an industry standard memory-to-memory data movement and acceleration interface. Memory transfers are integral to storage access. Transparent memory data movement within and across storage nodes remains an active area of optimization for storage systems.

NVMe is an industry leading storage access protocol. Data is transferred by NVMe controllers from host memory to device memory or from device memory to host memory. Leveraging SDXI for memory data movement and transformation enables various solutions, including solutions for storage. Integrating SDXI data transformations as part of storage access may present an optimization opportunity to system designers, accelerator vendors, and NVMe drive vendors. This applies to scenarios where SDXI data transformation accelerators either exist outside an NVM subsystem or are integrated within the NVM subsystem for access by the Host.

This whitepaper provides an overview of NVMe and SDXI operations, a discussion on the case to combine NVMe and SDXI, and example read and write flows where SDXI accelerators exist outside an NVM subsystem.

This whitepaper is intended for system designers and developers who may have little to no understanding of NVMe or SDXI but are interested in learning more about them and the benefits a combination provides. This whitepaper focuses on standards and specifications available at the time of publication of this document.

1 NVMe High-Level Overview

NVMe is a protocol defined by the NVM Express organization to address the needs of PCIe and Fabric connected NAND Flash memory-based storage systems. An NVM subsystem consists of controllers that communicate with the host and namespaces that store user data or operate on user data. The interface between the host and the controller provides a high-performance command submission and completion path,

including support for parallel operations of up to 65,535 outstanding commands on each of up to 65,535 queues.

NVMe defines a memory-based model for PCIe connectivity, and a message-based model for Fabric connectivity. SQs are used to supply commands from the host to the NVMe controller and CQs are used to return status information from the NVMe controller to the host.

NVMe also defines multiple command sets. NVMe provides several command sets for the persistent storage of user data such as the NVM Command Set, the Zoned Namespace Command Set, and the Key Value Command Set. In addition, there are command sets associated with performing operations on user data such as the Computational Programs Command Set and the Subsystem Local Memory Command Set.

2 A Typical NVMe I/O

To perform an I/O operation, the host places requests into entries on a Submission Queue (in an SQE). The SQE contains information about the type of I/O request (e.g., a Read or a Write), the address in the namespace (e.g., the LBA, or Key), a buffer descriptor (e.g., source address for a Write, or destination address for a Read), and other information. An example of the Submission Queues and Completion Queues are shown in

Figure 1.

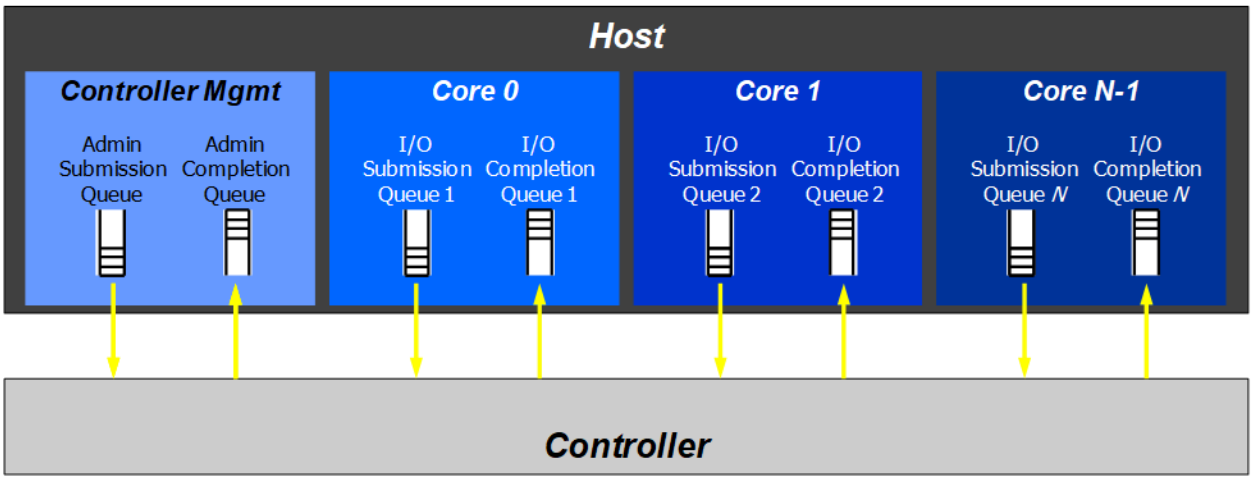

Figure 1: NVMe Submission and Completion Queues

For memory-based transports, a controller doorbell register is written to cause the controller to fetch the entries from the submission queue. For message-based

transports, the entries are transferred to the controller by the Fabric transport. In both cases, the controller begins to process the I/O requests.

Upon completion of the requested command processing, the controller returns status information to the host by placing an entry on a CQ. The host processes each CQE to determine the status of each command (whether the command succeeded or failed).

2.1 NVMe Deployments

NVMe Controllers may be virtualized as Physical Functions or Virtual Functions.

2.1.1 Host NVMe Storage Access

Hosts can access NVMe storage directly using memory-based transports or message-based transports. A privileged Host (e.g., Hypervisor) typically manages Physical Functions and its associated Virtual Functions. While Memory-based transports may require an NVMe PCIe function, message-based transports may require a NIC/HBA PCIe function to access remote NVMe storage across a fabric.

Host operating systems and drivers make efforts to minimize buffer copies. If buffer copies become necessary, an accelerator could be very useful to improve performance. Introducing an accelerator needs to be standard and complimentary to Host NVMe data buffer accesses.

2.1.2 Virtualized Guest NVMe Storage Access

A Host (e.g. Hypervisor) may provide Virtualized Guests access to NVMe storage directly using memory-based transports or message-based transports by direct-assigning PCIe NVMe Controller Functions or NIC/HBA Functions to guests, respectively. Hosts may also emulate NVMe storage by assigning Virtual NVMe Controller Functions to guests or emulating NVMe storage as generic block interfaces.

Most modern hypervisors and drivers make efforts to minimize buffer copies. If buffer copies become necessary, an accelerator could be very useful to improve performance. Introducing an accelerator needs to be standard and complimentary to Host and Guest NVMe data buffer accesses.

3 SDXI High-Level Overview

SDXI introduces a new model of accelerating memory to memory data movement, as show in

Figure 2.

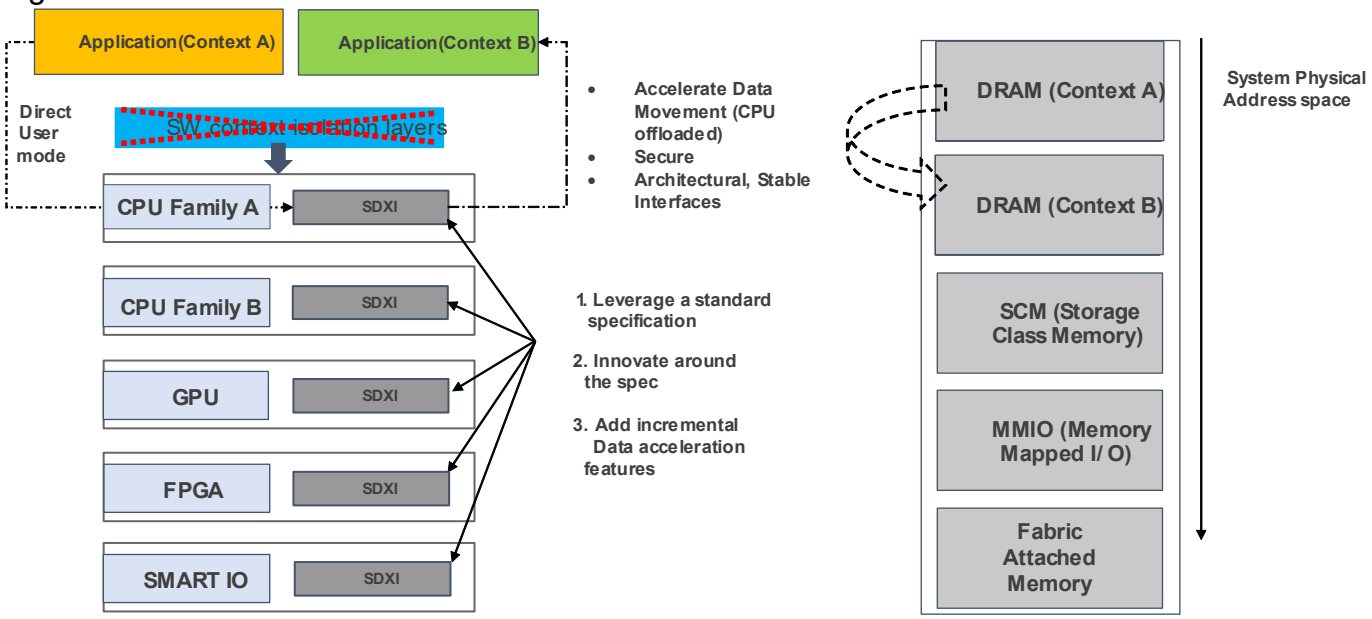

Figure 2: SDXI Architectural Overview

Software memcpy is the current data movement standard. While this has served the industry well due to stable CPU ISA, this approach is running into challenges in the following ways:

- Takes away from application performance
- Incurs software overhead to provide context isolation.
- Offload DMA engines and their interfaces are vendor-specific and not standardized for user-level software.

SNIA's SDXI specification is solving this problem with a standard for a memory-to-memory data movement and acceleration interface that is Extensible, Forward-compatible, and Independent of I/O interconnect technology.

Figure 3 shows a basic SDXI architecture.

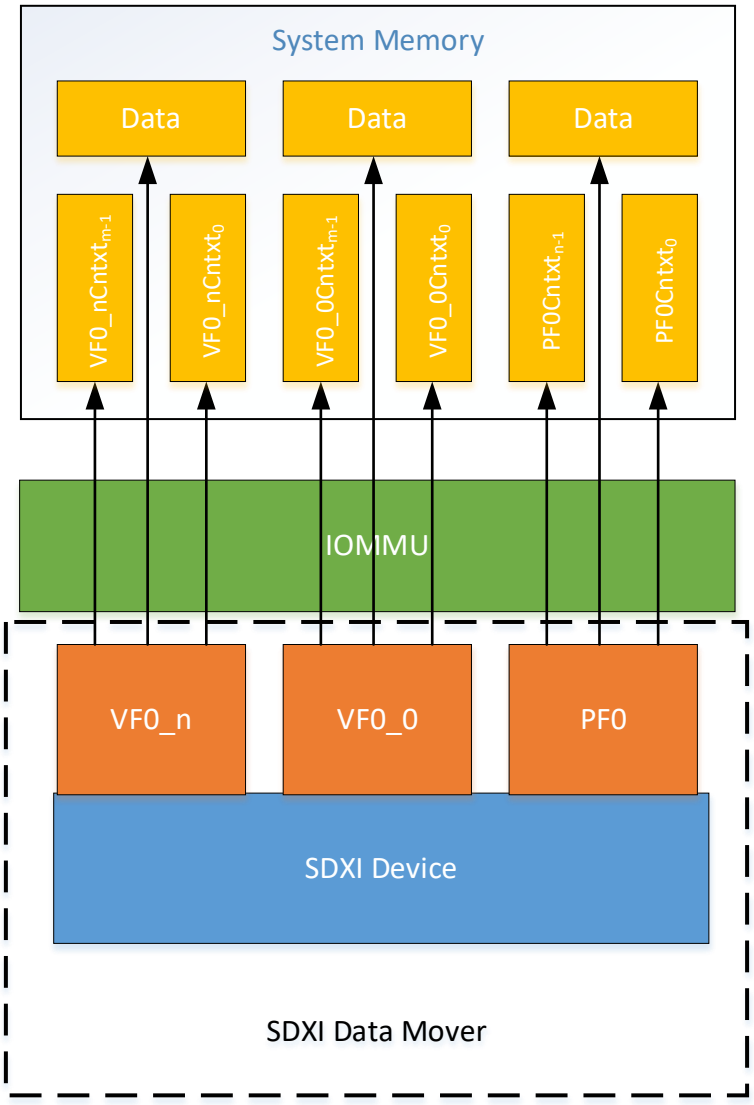

Figure 3: SDXI Architecture

3.1 SDXI Use Cases

While many use cases can be envisioned with an architectural data mover that is part of the system design, a few of them deserve mention for SDXI's initial goals.

3.1.1 Offloading Application memcpy

As the number and size of buffer memory copies increase, applications spend more compute cycles performing memory copies. SDXI function implementations improve application performance by giving back compute cycles otherwise used to perform memory copies.

3.1.2 Virtualization and Multi-Tenant Use Cases

Virtualization allows multiple tenants to share the same host resources and remain separate with respect to the data they use or produce. Providing tenant isolation can require tenants separated with resources in different Guest Virtual Machines (VMs) running on the same host. When they need to send/receive data between each other or share data with each other, they often resort to costly buffer copies within the same host that consume CPU cycles, or resort to using I/O devices such as NICs that don't perform memory to memory DMA between the Guest Virtual Machine's user address spaces. SDXI provides tenant Isolation and Authorized VM-to-VM memory data movement using an accelerated interface that bypasses context layers in an architected way. SDXI is also architected for virtualization design requirements like Live Migration enabling a performant accelerated experience while preserving manageability.

3.1.3 Data Transformations with Data Movement

Many data transformations like compression, encryption, CRC, T10 DIF calculations, involve sourcing data from memory and storing transformed data back to memory. SDXI provides a framework to standardize such operations. In addition, SDXI enables vendors to define their own class of memory operations without requiring all vendors to implement them to stay compliant.

3.1.4 Storage Platforms

Storage systems perform many intra-node and inter-node data movement and data transformations in memory. SDXI optimizes and standardizes all such memory data movements in Storage Nodes, and Storage Node clusters transparent to Host visible interfaces.

3.2 Base SDXI Concepts for Multi-Address Space Data Movement

SDXI is a standards-based memory data mover with well-defined structures and definitions. SDXI is designed to ensure secure, accelerated memory data movement. To achieve secure DMA access, SDXI takes advantage of platform security technologies, such as IOMMU, to improve memory to memory data movement. The following sections describe some of the features defined in the SDXI specification to implement multi-address space data movement.

3.2.1 Access Key (Akey) Table Entries

AKey table entries identify remote resources like memory address spaces, and interrupts on a per context basis. Each AKey Table entry is indexed with an Akey id.

Applications (SDXI Producers) are granted access to allowed AKey ids using a Connection Manager (currently under discussion for the SDXI specification) that will help broker a connection between two or more authorized SDXI applications. SDXI function(s) obtain the latest copy of the AKey table entry indexed by the AKey id. The Akey id itself is obtained from the SDXI descriptor enqueued by the application (producer) in system memory before initiating access to a remote address space. The AKey entry may identify more granular attributes such as:

- **sfunc** that maps to a unique identifier on the DMA bus. For a PCIe based DMA bus, this maps to a PCIe Requester ID.
  or
- **A PASID** that is used to identify a unique process address space.

Collectively, these attributes are used to authenticate access to a target address space. For example, these attributes may be used to form the TLPs on a PCIe bus accessing the remote memory address space.

3.2.2 Receiver Access Key (RKey) Table Entries

RKey Table Entries control remote function's access to local resources, such as memory and interrupts. Receiver Access Key entries are found in a per function Receive Access Key Table. Each entry may be associated with a remote function, controlled and managed by privileged software. A function allows remote requesting functions access to local resources by consulting the latest copy of its RKey table and the RKey Table Entry referenced.

AKey and RKey work collaboratively for request function control and target function control, respectively. This ensures data access across tenant boundaries is not possible without authorization.

3.2.3 IOMMU

Host Memory access by SDXI functions is protected using IOMMU page protection tables. In some architectures, IOMMUs are known as SMMU. With the help of PASIDs, memory accesses by SDXI functions are protected from unauthorized and unpinned page memory access to user addresses.

3.2.4 Reducing Context Isolation Layers for Performance and Security

Reducing the Trusted Computing Base (TCB) is essential to lowering rogue actors' attack surface. Since SDXI enables direct user mode access for applications to produce work for SDXI functions, code complexity is greatly simplified. It eliminates various software layers otherwise needed to improve context isolation.

4 A Typical SDXI Operation

Figure 4 uses an example of a descriptor-based operation in a single context to demonstrate the memory data structures involved.

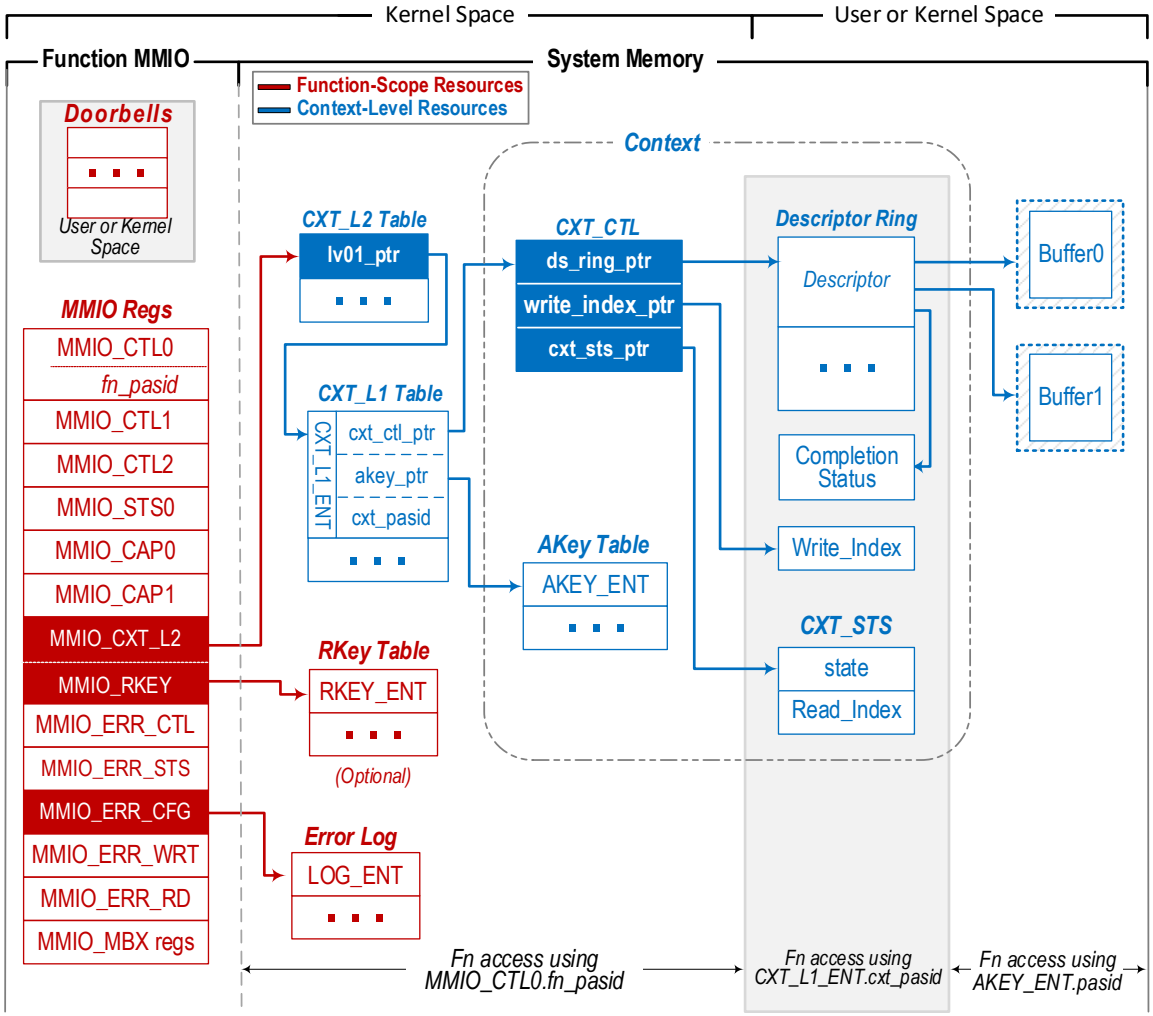

Figure 4: Example of Descriptor-Based Operation

A typical SDXI operation requires an application (producer) to obtain an SDXI Context. An SDXI Context contains all the required structures used by a producer to interface with an SDXI Function (consumer). As shown in Figure 4, an SDXI Context contains among other things:

- A Descriptor Ring
- Read_Index

- Write_Index
- Completion Status Block
- Context Status
- AKey table

Once a context has been set up, a producer may enqueue descriptors to the Descriptor Ring while the SDXI function processes the operations described by descriptors without intervention from privileged software. See

Figure 5 for an example of an SDXI descriptor ring.

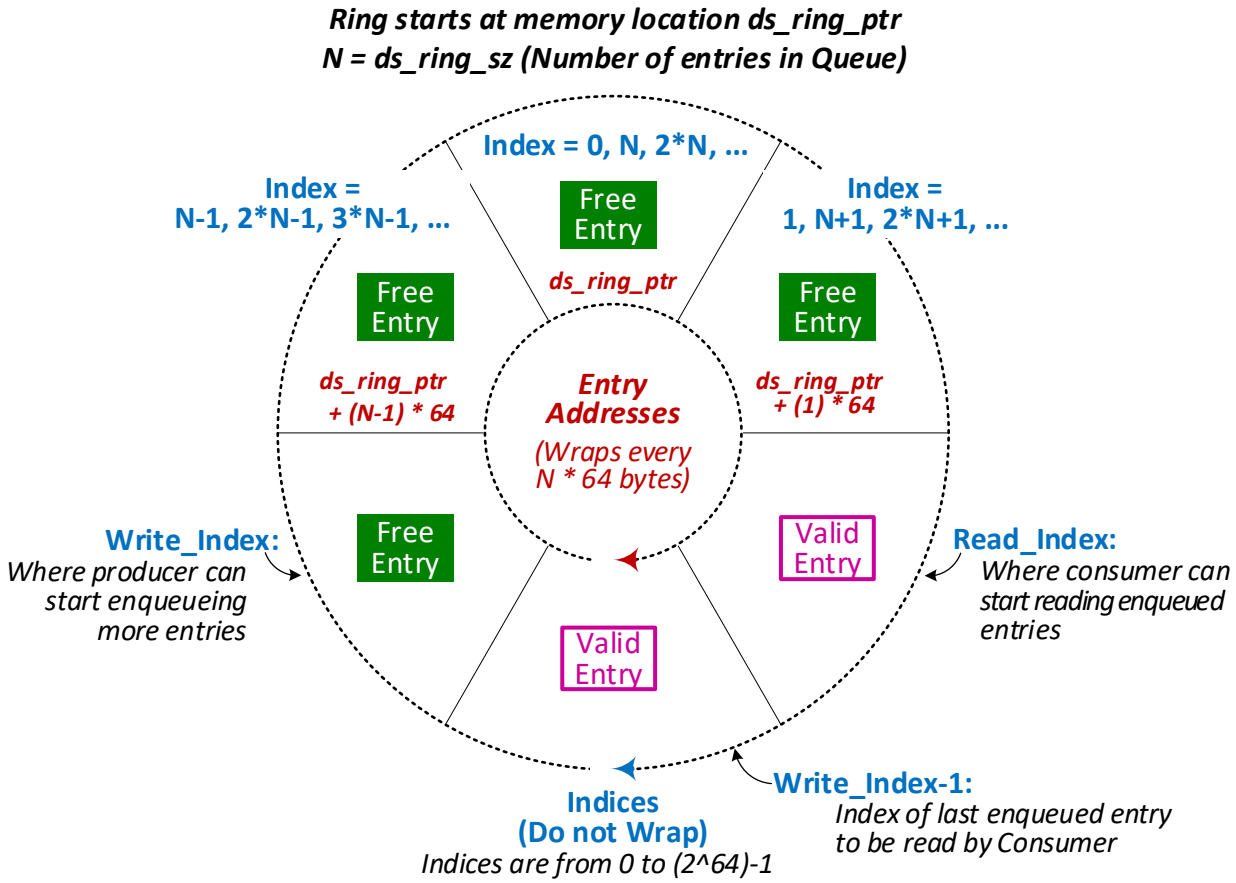

Figure 5: SDXI Descriptor Ring

Descriptors in the descriptor ring are processed (descriptor is fetched and validated) in-order by the SDXI Function, executed (requested operation is performed) out-of-order, and completed (posting completion signals) out-of-order. The Write_Index for the context is incremented by the producer and the Read_Index for the context is

incremented by the SDXI Function (consumer). The SDXI Function may aggressively read valid descriptors between read and write indices without waiting on doorbells from producers. Doorbells ensure new descriptors are recognized. The SDXI specification enables implementations to be optimized with the goal of maximizing parallel operations. Quiescing and serializing state is achieved at well-defined boundaries. Refer to the SNIA SDXI Specification for a detailed description of the various operations supported by SNIA SDXI Specification.

5 A Case for Combining NVMe and SDXI

NVMe controllers are equipped with DMA engines and implement the standard NVMe protocol interface however, DMA implementations are not required to be standard. NVMe DMA implementations are also not optimized for memory to memory data transfers and transformations.

For designers that have an existing NVMe controller design, and well tested built-in DMA engines, SDXI, as an additional standard DMA engine, presents an opportunity to augment data flows and transformations without affecting shipping NVMe controllers. For designers looking into a new DMA engine design for their NVMe controllers, standard SDXI DMA engines offer an off the shelf, standards-based alternative to designing a new DMA engine from scratch. System designers may also combine off the shelf NVMe controllers with off the shelf standard SDXI DMA engines for a more optimized data movement solution.

There are several benefits to using SDXI for each of the described cases. As a standard memory data mover, SDXI is designed for memory access, memory ordering intricacies, and byte-size addressability. This enables NVM subsystems to employ standard DMA implementations for their data movement needs and benefit from an ecosystem of SDXI enabled DMA hardware implementations. This simplifies the design of the subsystem using off the shelf components.

The SDXI data movement can leverage built-in transformation engines to manipulate the data while it is transferred.

NVM subsystems that incorporate standard SDXI based DMA future proof software investments, innovations, and integrations enabled by the SDXI based memory data movement and transformation ecosystem.

Finally, SDXI is designed for applications in such a way that it reduces buffer copies.

There are other benefits that are beyond the scope of this whitepaper.

The next sections detail an NVMe Write I/O flow and an NVMe Read I/O flow with SDXI.

6 Application NVMe Write I/O Flow with SDXI

SDXI is able to be involved in the processing of an NVMe I/O write. Section 2 described a traditional NVMe I/O write. Figure 6 below shows a system topology involving NVM I/O flows enhanced by SDXI. The SDXI enhanced I/O write assumes that the SDXI instance is external to the NVM subsystem (it may be in the host, for example).

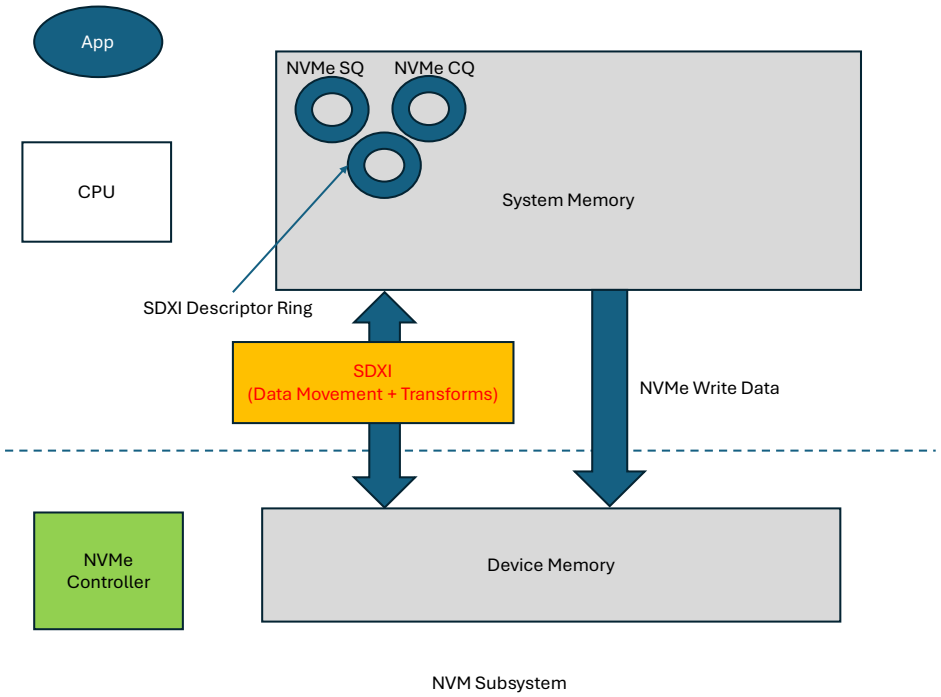

Figure 6: SDXI and NVMe Write Flow

Described below is an example of how a host may operate when it has SDXI and NVMe functions available while performing NVMe Write I/O using a PCIe transport. See Figure 6 for such a topology.

1. The host issues an SDXI descriptor to perform a memory-to-memory copy of data from the host memory to the NVMe controller's CMB/PMR.
2. This device memory is host addressable and can easily be referenced by SDXI in the host.
3. While transferring the data, SDXI engines can optionally perform transformations to the data before writing the data to CMB/PMR.
4. Once the data has been written to CMB/PMR, the host issues an NVMe write command or NVMe copy command that refers to the data in CMB/PMR to transfer the data to persistent storage.

Section 6.1 describes this application in more detail.

If NVMe I/O is over a fabric, CMB/PMR is not available, and alternate host buffers (e.g., bounce buffers) may be used to stage the data for SDXI data transformations.

In virtualized scenarios, a combination of SDXI and NVMe abstractions may be used by applications. See sections 2.1.2 and 3.1.2 for discussion of NVMe and SDXI in virtualized environments.

6.1 Write Use Cases

SDXI enables movement of data between memory and storage tiers. This data movement can be done under the direction of background processes. These processes decide when to move data from volatile DRAM to Non-Volatile Storage (e.g., NVMe PMR). SDXI offloads that data movement to the appropriate memory/storage tier. The SDXI architecture enables several types of transformations that may be performed on data as it is moved. For example, data may be compressed, encrypted, or hashed, by SDXI engines, before it is stored in persistent storage. These transformations may also be initiated by applications as part of an I/O request pipeline. Thus, if an application wants to transform data blocks in application memory and store it to flash via NVMe Write I/O, SDXI provides an architectural interface that enables reduction in data copies that may occur as part of an NVMe Write request pipeline.

SDXI may be used by kernel mode or user mode processes.

7 Application NVMe Read I/O Flow with SDXI

Similar to Section 6, SDXI is able to be involved in the processing of an NVMe I/O read. Section 2 described a traditional NVMe I/O read. The SDXI enhanced I/O read assumes that the SDXI instance is external to the NVM subsystem (it may be in the host, for example).

Described below is an example of how a host may operate when it has SDXI and NVMe functions available while performing NVMe Read I/O using a PCIe transport. See Figure 7 for such a topology.

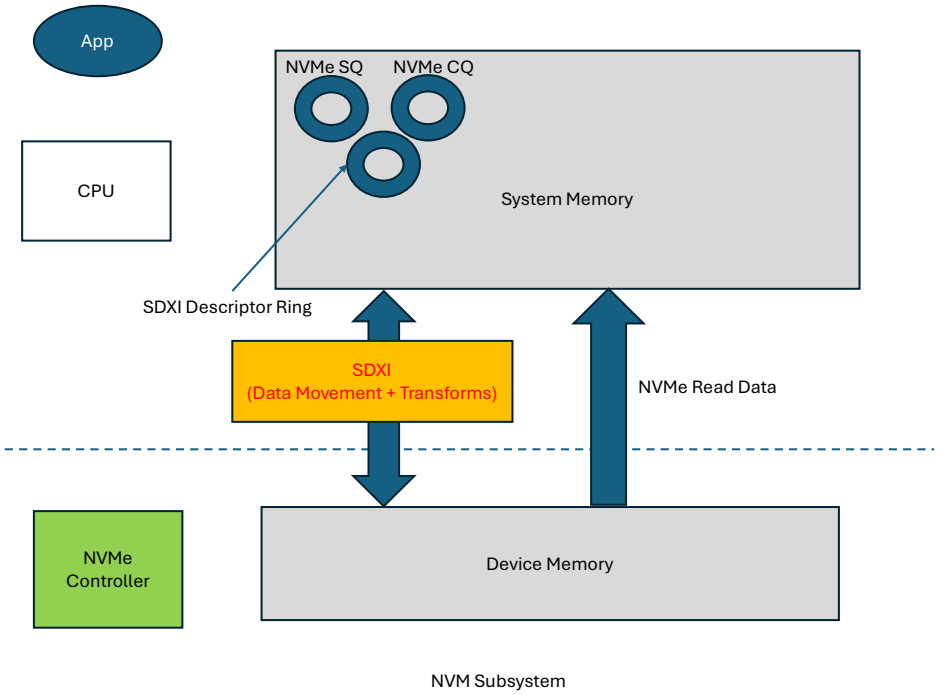

Figure 7: SDXI and NVMe Read Flow

1. The host issues an NVMe read command to read the data from persistent storage and write the data to CMB/PMR.
2. The CMB/PMR device memory is host addressable and can easily be referenced by SDXI in the host.
3. Now that the data is in CMB/PMR, the host issues an SDXI descriptor to transfer the data from CMB/PMR to host memory.
4. While transferring the data, SDXI can optionally perform transformations to the data before writing the data to host memory.

Section 7.1 describes this application in more detail.

If NVMe I/O is over a fabric, CMB/PMR is not available, and alternate host buffers (e.g., bounce buffers) may be used to stage the data for SDXI data transformations.

In virtualized scenarios, a combination of SDXI and NVMe abstractions may be used by applications. See sections 2.1.2 and 3.1.2 for discussion of NVMe and SDXI in virtualized environments.

7.1 Read Use Cases

SDXI enables movement of data between memory and storage tiers. A read request can certainly result in data movement from a slower tier to a faster tier for immediate

system access. In addition, this data movement can be done under the direction of background processes based upon access patterns or other heuristics. These background processes may decide to move data from Non-Volatile Storage (e.g., NVMe Persistent Memory Region (PMR)) to volatile DRAM. SDXI offloads data movement from the appropriate memory/storage tier. The SDXI architecture enables several types of transformations that may be performed on data as it is moved. For example, data may be decompressed, decrypted, or a hash compared, by SDXI engines, as it is moved from persistent storage to volatile DRAM. These transformations may also be initiated by applications as part of an I/O request pipeline. SDXI enables reduction in data copies as it transforms and moves data blocks to application memory after reading it from flash via an NVMe Read request pipeline.

Staging the read data in device memory and using SDXI to transform and move it to host memory avoids multiple system memory transfers and buffer allocations. For example, if NVMe controllers are used to read data into system memory, and then SDXI is used to perform memory transformations in system memory, it will triple the memory bandwidth usage and double the host buffer allocations.

SDXI may be used in kernel mode or user mode processes.

8 Peer to Peer Example with NVMe and SDXI

The SDXI Function may be employed by the host to perform data transfers between host memory and device memory or peer to peer transfers between two or more peer devices. In such situations, the host may delegate data transfers to the SDXI Function. This may require the host to enable the SDXI Function to access memory space(s) available to the host. Memory spaces may include those exposed by NVMe CMB/PMR in peer devices.

While many peer-to-peer data transfer flows are possible, the following example describes a flow where SDXI may be used to assist peer-to-peer transfer of data from one NVMe Function's (Function A's) CMB to another NVMe Function's (Function B's) CMB:

1. The host issues an NVMe read command to Function A to read the data from its persistent storage to Function A's CMB.
2. The CMB device memory in Function A and Function B are host addressable.
3. Now that the data is in NVMe Function A's CMB, the host issues an SDXI descriptor to transfer the data from NVMe Function A's CMB to Function B's CMB.
4. While transferring the data, SDXI engines can optionally perform transformations to the data before writing the data to NVMe Function B's CMB.
5. Once the data has been written to NVMe Function B's CMB, the host issues an NVMe write command or NVMe copy command that refers to the data in Function B's CMB to transfer the data to persistent storage.

This example shows how SDXI may act as an efficient 3rd party data mover between various NVMe Function's CMB memory spaces.

9 Security Considerations

SDXI enables multi-address space memory to memory data movement. It enables isolation of memory address spaces also known as an airgap. An airgap between memory in different address spaces (e.g., guest kernel and host kernel address spaces) is a common security practice. So, employing SDXI to help with data movement can be a useful security feature.

The combination of SDXI external to the NVM subsystem does not materially change the security posture of the interaction of the NVM subsystem with the rest of the system. Host based security policies can be easily applied to SDXI based data movement similar to non-SDXI based data movement.

For deployments where interconnects are encrypted (e.g., Link Encryption), the combination of SDXI and NVMe does not change the security posture of the NVM subsystem with respect to the rest of the system. Data movement is layered above the encrypted secure links and therefore takes advantage of the services provided by those encrypted links. SDXI merely accelerates data movement over encrypted links.

SED technology provides data encryption at rest. Other technologies may be used to provide data encryption over links such as PCIe IDE. SDXI may be used to move unencrypted or encrypted data. SDXI may also encrypt data as it is being moved. To provide additional protection when non-SED drives are part of a system, SDXI may be used to encrypt data before being stored.

When augmenting system security with SDXI, it is important to recognize that SDXI may be implemented in a variety of form factors in different parts of a system topology. For example, SDXI may be implemented in a CPU, an intermediary device such as a PCIe switch, a lookaside device such as an accelerator, or in the drive itself. Depending on where SDXI is in the system topology, SDXI encryption may be applied to achieve protection on different access links. System implementers should carefully evaluate the security implications of their individual design.

Figure 8 describes some possible combinations. For example, when reading data from system memory and writing to CMB/PMR memory as highlighted in the picture, SDXI engine A may read (unencrypted) data from system memory using Memory controller A, encrypt it and then write to CMB/PMR memory in drive A. The data flows through the PCIe Root Port 1, PCIe-switches through Upstream Port 1 and Down Stream Port 1 and is written into drive A's CMB/PMR memory. The Write data flowing through the Root Port1 may be encrypted using PCIe IDE-stream encryption keys. Finally, the data in CMB/PMR may be encrypted using SED encryption keys when the host issues an NVMe write command to send the data to persistent storage from CMB/PMR.

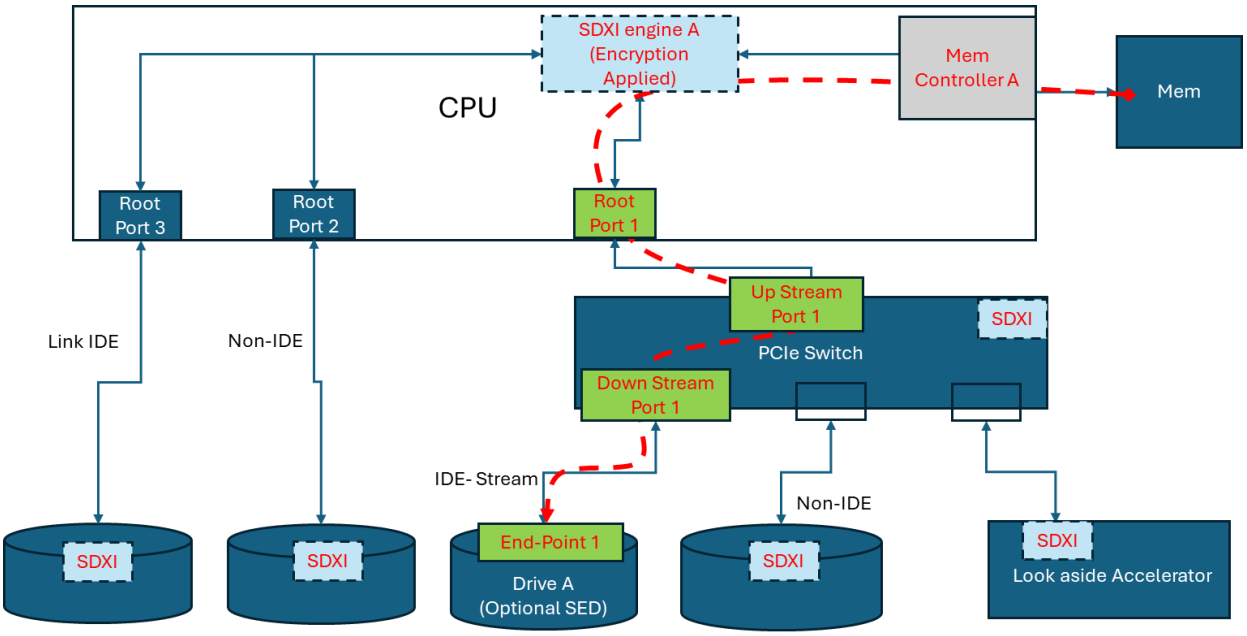

Figure 8: Security Examples

It may be noted as shown in the figure that SDXI engines could also be implemented in PCIe Switches, Look aside Accelerators, Drives, etc.

10 Summary

This paper discussed the advantages of complementing existing NVMe based read, write, and peer-to-peer storage flows with existing SDXI standard accelerators for data movement and transformation. This paper discussed flows for transforming data as it is being read from NVM storage, written to NVM storage, or as it is moved from one NVM storage device to another. The above use-case involves system topologies and architectures with SDXI outside an NVM subsystem and data movement within system addressable memory regions. Other architectures or implementation choices are possible.

SNIA welcomes further discussions, enhancements, proposals, and other use cases.

11 For More Information

For more information, visit:

[1] www.snia.org/sdxi
[2] www.snia.org/feedback