NVDIMM Block Window Driver Writer's Guide

Example NFIT-Based NVDIMM
Block Window and Persistent Memory Interface Guide

April 2015

Contents

Figures

Tables

Introduction

Document Scope

This document is targeted to driver writers for NVDIMMs that adhere to the NFIT tables in the Advanced Configuration and Power Interface (ACPI) V6.0 specification, the Device Specific Method (DSM) specification and the NVDIMM Namespace Specification. This document specifically discusses the block window HW interface and persistent memory interface that Intel is proposing for NVDIMMs. These interfaces are utilized in the released Linux NVDIMM driver stack.

Related Documents

This document depends heavily on the documents listed in Table 1. Where possible, this document will call out specific references to these documents.

Table 1 - Related Documents

Title	Description
Advanced Configuration and Power Interface (ACPI) V6.0 - NVDIMM Firmware Interface Table (NFIT)	This document describes the ACPI-style table that the BIOS builds to describe NVDIMMs, including NVDIMM interleave information, and how the DIMMs are mapped in to system physical address space.
Device Specific Method (DSM) interface proposal	This document describes the proposed interface between the NVDIMM Driver and the ACPI SW stack for configuring, managing, and enumerating NVDIMMs.
NVDIMM Namespace Specification	This document describes the proposed label mechanism used to sub-divide the NVDIMM into BlockNamespaces and PmemNamespaces, as described by Namespace Labels, including the Block Translation Table (BTT) used to provide power-fail write atomicity. Included in this specification are on-media structure definitions for labels and the BTT, and rules for using those data structures.
NVDIMM Linux driver code released by Intel	Intel has released NVDIMM Linux sample code that utilizes the interfaces specified in this specification.

Terminology

Error! Reference source not found. provides a glossary of terms used in this ocument.

Table 2 - Terminology

Term	Description
BW	Block Window. A set of registers consisting of a command register, a status register, and an aperture allowing the NVDIMM driver to read and write blocks of data to any persistent area on an NVDIMM.
BTT	Block Translation Table. A software data structure, defined in the NVDIMM Namespace Specification, which prevents torn blocks when a write is interrupted by a system crash, hang, or power failure.
CLFLUSHOPT	A performance-optimized version of the Intel® Architecture CLFLUSH instruction. This instruction is weakly-ordered, allowing multiple flush instructions to occur in parallel.
DPA	DIMM Physical Address. An address within the memory in an NVDIMM.
External LBA	The host OS logical block address passed to the driver in an IO request. This term is used to distinguish between the LBA given to the driver, and where the actual IO happens due to BTT translation.
LBA	Logical Block Address. IO requests to block devices are typically in terms of LBAs. For a byte-addressable NVDIMM like the NVDIMMs, the LBA is converted into a byte offset.
NFIT	The NVDIMM Firmware Interface Table, which defines the ACPI-like information created by the BIOS to inform the OS about NVDIMMs in the system.
NVM	Non-Volatile Memory
NVDIMM	Non-Volatile DIMM. Non-volatile memory in a DIMM form factor.
NVDIMM Namespace Label	Labels, stored at a known location on NVDIMMs, that define the DIMM's contribution to NVDIMM Namespaces. This is a software mechanism; the DIMM itself just sees the labels as part of the overall data stored on the DIMM.
NVDIMM Namespace	Similar to an NVMe Namespace or a Logical Unit (LUN) on a SCSI disk, this is a software mechanism for managing ranges of persistence on NVDIMMs.
PCOMMIT	An Intel Architecture instruction that flushes outstanding stores to persistent memory from all memory controllers, system-wide. This instruction just initiates the flush; a fence operation is required to wait for the flush to complete.
Persistent Memory	Byte-addressable memory that retains its contents across power loss.
SPA	System Physical Address. A physical address on the host operating system.

NVDIMM Architecture

The NVDIMM technology is the memory subsystem architecture for server platforms incorporating next generation non-volatile memory (NVM) technology in the form of an NVDIMM. The following sections introduce the basic proposed SW architecture utilized by the sample Linux SW stack.

Proposed NVDIMM Software Architecture

The major components for the proposed NVDIMM software architecture are shown in Figure 1 – Basic NVDIMM Software Architecture.

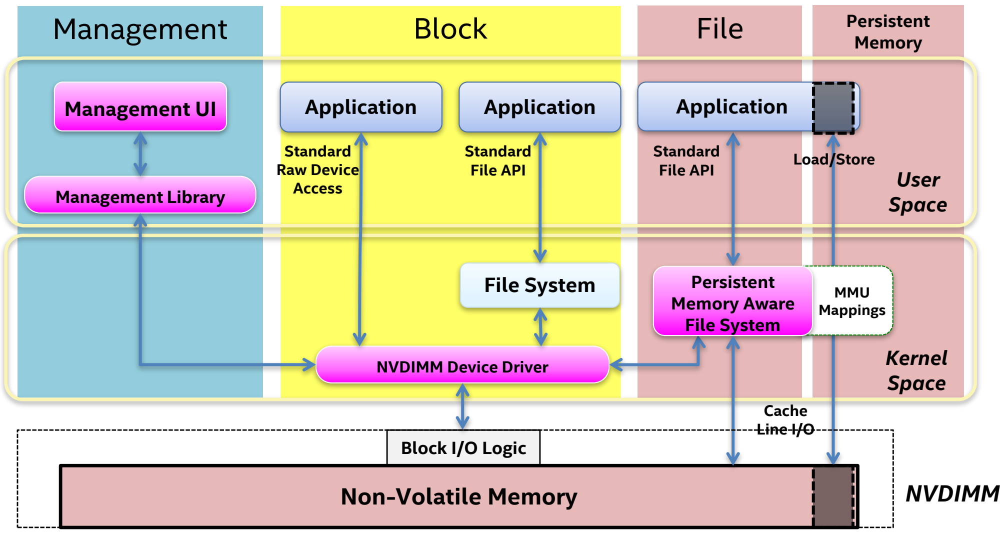


Figure 1 – Basic NVDIMM Software Architecture

The figure shows the NVDIMM at the bottom (systems can have multiple NVDIMMs). Those DIMMs are described by the BIOS to the OS via the ACPI-defined NVDIMM Firmware Interface Table (NFIT). The NVDIMM driver interfaces the OS application and file system components to the NVDIMM. The NVDIMM driver utilizes the proposed block window HW interface in the NVDIMM to move data to/from the persistent memory and the data buffers utilized by the OS. This interface is described in detail in Chapter 0 of this guide.

For applications to directly access persistent memory utilizing a standard byte addressable load/store interface, the NVDIMM driver exposes the persistent memory through a persistent memory-aware file system, or some other OS-specific mechanism. Once persistent memory-aware applications have access to the persistent memory region addresses, they can use the memory directly and perform loads and stores without the need for an NVDIMM driver, minimizing or eliminating the file system and storage overhead of the OS.

NVDIMM Driver

The combination of all of the components that implement the block window and persistent memory interfaces to the NVDIMM hardware are collectively called *the NVDIMM driver.* The NVDIMM block window interface described in Chapter 3 and the NVDIMM driver are designed to meet the following basic high-level requirements:

- Manage all NVDIMMs with a single driver instance – An NVDIMM-aware platform BIOS creates a single NFIT that the NVDIMM driver is loaded against. This single driver instance is expected to manage all of the NVDIMMs in the system.
- Support traditional OS storage stacks through the block window interface:
    - Direct attached SSDs - The NVDIMM Namespace Specification describes the NVDIMM namespace labels that are created from free space on the NVDIMM and are stored in a reserved namespace label data region on each NVDIMM. For block window regions of persistent memory, each NVDIMM namespace label describes one NVDIMM BlockNamespace or a portion of a BlockNamespace if the namespace is broken in to multiple, non-contiguous portions. Each BlockNamespace can be surfaced by the driver or other kernel components to the OS as an SSD direct-attached disk, interfacing with the rest of the traditional OS-specific block storage stack.
    - Support for Existing SW RAID_implementations – See the section below on how the block window HW implementation allows the use of traditional SW RAID stacks with NVDIMMs.
    - Provides traditional block storage driver error model – See the section below on how the NVDIMM block window HW architecture allows non-traditional DIMM errors to be routed to the storage stack and why that is important to system robustness.
    - Optional single-sector power-fail write atomicity – The NVDIMM Namespace Specification outlines the Block Translation Table (BTT) metadata that is stored on the NVDIMM. The BTT provides single-sector write atomicity by implementing a write LBA indirection system. See the section below that explains the importance of this feature to traditional OS storage stacks.
- Multiple logical block-size support including 512, 520, 528, 4096, 4160, and 4224 byte block sizes that support embedded metadata in each block, like DIF/DIX.
- NVDIMM management support

Most of the above requirements are self-explanatory, but a few key requirements deserve more detailed explanation:

Block Window Interface - Error Flow Requirement

The way errors propagate is quite different between storage and memory. Storage errors are typically propagated up the storage stack where components like SW RAID or applications themselves can react to them. Memory errors are often handled transparently to applications and in many cases can cause the system to crash if it is deemed unsafe to continue. One feature of the proposed NVDIMM block window HW interface is that errors are returned to the NVDIMM driver instead of being exposed as memory errors. Using a block window status register, the NVDIMM driver checks for errors after each block IO and propagates any errors as a block error through normal storage stack means. Without this HW support, storage mode errors would appear as memory errors instead of block storage errors, causing many systems to crash due to memory errors in kernel space, reducing overall system robustness.

Block Window Interface - SW RAID Requirement

In addition to the need to propagate errors to SW RAID as described above, it is also important for SW RAID to correctly understand the RAS boundaries of the underlying storage. The NVDIMM block window HW interface allows blocks to be placed directly on NVDIMMs, regardless of any memory controller-based interleaving currently in effect. The combination of this HW feature, memory error reporting on block transfers, and support in the driver allows SW RAID stacks to work correctly on NVDIMMs. The block window HW interface combined with NVDIMM BlockNamespaces provides the same level of SW RAID functionality as with separate discrete SSDs.

Some SW RAID examples include:
Redundancy - By isolating NVDIMM BlockNamespaces to specific NVDIMMs, RAID features such as RAID-5 can be used across NVDIMMs to provide data protection.

Capacity Aggregation - Since the maximum NVDIMM BlockNamespace capacity is limited to the size of the NVDIMM, this capacity limitation with smaller NVDIMMs can be overcome using SW RAID to stripe together multiple NVDIMM Namespaces to create a larger virtual volume.

Block Window Interface – Power-Fail Write Atomicity Requirement

A typical SSD will maintain some sort of error correcting code (ECC) on each block in order to detect data corruption. If the system loses power during a write to an SSD, these results are possible (listed from most commonly implemented to least commonly implemented):

- The sector completely contains the old data, or completely contains the new data. This happens when the SSD provides **power-fail write atomicity**. Many SSDs do this, at least for writes of 1 sector (it is a requirement of the NVM Express Specification, in fact).
- The sector is torn by a power failure, so the data is corrupt. But the ECC indicates the data is corrupt, so an error results when reading that sector.
- The sector is torn, but the ECC is somehow correct. So when SW reads the sector, a corrupt sector containing some old data and some new data is read. Since no error is indicated, this is silent data corruption. This has been reported on some HDDs and on early SSDs, but is quite rare, perhaps impossible, on modern SSDs. Most file systems and applications do nothing to detect this case (notable exceptions are check summing file systems like ZFS, btrfs, and ReFS).

The case of the torn write is particularly interesting for NVDIMMs. While each write to the DIMM is done using a store instruction, and store instructions to NVDIMMs are power-fail atomic, a block of data that incorporates a number of atomic store instructions will not result in an ECC error on a block, if power is lost in the middle of updating that block. The block will contain some new data, whatever had been stored, and the rest of the old data, but nothing to indicate a corrupt block. Because this causes a silent data corruption on many file systems and applications, it is a requirement that the driver provide at least single block power-fail write atomicity.

This requirement is implemented through the use of BlockNamespaces that contain a BTT in the NVDIMM metadata layout and is discussed in some detail below.

NVDIMM Labels, Namespaces, and BTTs

The following sections provide a high level description of the key elements of the NVDIMM on media metadata layout and architecture. These components include Namespace Labels, BlockNamespaces, PmemNamespaces, and BTTs (Block Translation Tables). The NVDIMM Namespace and BTT formats are OS-independent and are described in full detail in the NVDIMM Namespace Specification.

Namespace Labels

Each NVDIMM namespace is described by a set of namespace labels that reside on each NVDIMM contributing to the namespace in a reserved area called the namespace label data area. The driver is responsible for maintaining the namespace labels that describes the namespaces. Since each NVDIMM can contain multiple labels, each NVDIMM can therefore contain multiple namespaces. The labels describe the DIMM physical address (DPA) and range of that portion of the namespace on that given NVDIMM. Namespace labels are not interleaved across multiple NVDIMMs.

BlockNamespace labels on a given NVDIMM describe one or more BlockNamespaces. Since BlockNamespaces do not need to be contiguous, multiple BlockNamespace labels can be linked together on a given NVDIMM to describe a large logically contiguous region that is not physically contiguous on the NVDIMM. Drivers must account for any interleaving in affect when programing the block window HW interface, as described more in following sections.

PmemNamespace labels on a given NVDIMM describe that NVDIMM's portion of the interleave set utilized for the PmemNamespace. The driver typically does not need to understand the interleaving that is in affect for these namespaces.

BlockNamespaces

The NVDIMM Namespace Specification defines a BlockNamespace as a logical unit of storage conceptually similar to a standard disk drive, SSD, or SCSI LUN. A BlockNamespace is composed of ranges of persistent memory from one NVDIMM device. BlockNamespaces have a single logical block address (LBA) space that is numbered from 0 through (N – 1), where N is the number of logical blocks in the namespace. BlockNamespaces are created on regions of the NVDIMM that are enabled for persistent memory and start at high DPAs and grow down to minimize collisions with PmemNamespaces that start at low DPAs and grow up. BlockNamespaces can be created in portions of the persistent memory region that have an interleave set defined as long as no PmemNamespace is currently defined on the NVDIMM in that portion of the interleave set.

The driver is ultimately responsible for surfacing each valid BlockNamespace to the operating system as a separate device. When interfacing to the OS's storage stack, the BlockNamespace is usually in the form of a SCSI disk or LUN.

PmemNamespaces

The NVDIMM Namespace Specification defines a PmemNamespace as a container for a portion of the system physical address space that has been mapped by the BIOS across one or more NVDIMMs grouped together in an interleave set. These

PmemNamespaces tell the driver how much of the interleave set is used, leaving the remaining space for use with BlockNamespaces. The driver is ultimately responsible for surfacing each valid PmemNamespace to the OS as a separate device. Other OS SW components typically utilize these devices to create a persistent memory-aware file system. PmemNamespaces start at the low DPA of each NVDIMM and grow up to higher DPAs, minimizing collisions with BlockNamespaces that start at high DPA and grow down.

BTTs

Block Translation Tables (BTTs) are a software mechanism for providing power-fail write atomicity for NVDIMMs that do not have HW power-fail write atomicity available. As described in the previous section, providing power-fail write atomicity for a single block write is a block window requirement. This requirement is met using a BTT that maintains a small pool of free blocks used for writes. The BTT metadata and algorithms for using it are described fully in the NVMDIMM Namespace Specification.

Persistent Memory-Aware File Systems and Applications

A persistent memory aware-file system, kernel modules, or applications that wish to make use of byte-addressable persistent memory need to interface with the NVDIMM driver to obtain a list of PmemNamespace ranges that the driver has found through enumerating the NVDIMMs namespace labels directly. The SNIA NVM programming model specification calls out this action as *getranges* but the actual API name or method utilized is implementation-specific. Once the persistent memory-aware file system or kernel components get the physical ranges associated with a PmemNamespace, they need not ever call back to the NVDIMM driver to perform IO. Persistent memory-aware applications can execute direct loads, direct stores, cache flushing, and PCOMMIT instructions to the persistent memory regions of each NVDIMM.

NVDIMM Management Operations

An important function of the NVDIMM driver is to provide the appropriate interfaces to NVDIMM manageability SW. This interface provides methods to find, create, modify and delete BlockNamespaces and PmemNamespaces through management of the namespace labels (as described in the NVDIMM Namespace Specification), gather SMART and health information, and a number of other functions. Refer to the specific NVDIMM management specifications for details on the specific interfaces that the NVDIMM driver should implement.

The management interface is composed of a number of IOCTLs that fall in to several categories:

Pass-through IOCTLs

These are management requests whose inbound and outbound payloads, commands, and status are created and handled by the management applications and the NVDIMM. The NVDIMM driver should validate the buffer for sufficient input and output length for these requests and route the IOCTL to the vendor-specific command DSM mechanism described in this guide and the DSM specification. The NVDIMM supports a command effect log that reports to the driver all NVDIMM Opcode combinations, which combinations are supported by the NVDIMM, and a set of effect bits that describe the effects on the NVDIMM subsystem when the command is sent to the NVDIMM. The NVDIMM driver should retrieve the CommandEffectLog utilizing the GetCommandEffectLogSize DSM method and the GetCommandEffectLog DSM method. The driver can then check the vendor-specific command pass-through IOCTL payload opcode and handle the reported effects or potentially reject the pass-through command if the NVDIMM driver can't guarantee system stability.

IOCTLs to Retrieve Driver-Managed Information

In general, these IOCTL requests are from the management stack and are considered "native" driver IOCTLs. They retrieve information including current namespace and namespace label configurations, interleave set information, and driver capabilities. At boot time the NVDIMM driver will utilize DSM commands to read namespace labels from the reserved area of each NVDIMM, which includes BlockNamespaces and PmemNamespaces and cache the data internally in the driver. In response to these IOCTLs, the driver should validate the payload buffer for sufficient input and output length as well as completely validating the inbound payload for illegal payload values. The driver uses internally gathered data and context to fill in the output payload buffer and does not typically use a DSM command in direct response to these IOCTLs.

IOCTLs to Create, Delete, or Modify Driver Managed Information

In general these IOCTL requests are from the management stack and are considered "native" driver IOCTLs. They create or modify information like current BlockNamespaces or PmemNamespaces, and namespace label configurations, etc. The driver should validate the payload buffer for sufficient input and output length as well as completely validating the inbound payload for illegal payload values. The driver updates or modifies internally managed driver data and context utilizing the input IOCTL payload buffer and does not typically use a DSM command in response to these IOCTLs. One example where a DSM command is used in response to these IOCTLs is the SetNamespaceLabelData DSM command, which is utilized by the driver when an IOCTL to create, delete, or modify one or more BlockNamespaces or PmemNamespaces is requested.

Interfacing with NFIT-Based NVDIMMs

This section covers the NFIT and DSM interfaces necessary to produce an NVDIMM block window device driver. The existence of the NFIT in the ACPI V6.0 exposes ACPI namespace to the OS, typically causing the OS driver to be loaded and initialized. The initialization begins with the driver pulling the relevant information from the NFIT, as described in the sections below.

NFIT Parsing

Refer to the NFIT in the ACPI 6.0 specification and the following figure and discussion for details on the structure of the NFIT, the descriptor tables it encompasses, and the description of the specific fields. The NFIT is the primary means by which the NVDIMM-aware BIOS describes the physical HW topology of the system, including the number of nodes, the number of slots per node, the memory controllers per slot, and the topology of the NVDIMMs. The topology of the NVDIMMs includes the interleave; DIMM physical address (DPA); how they map to the system physical address (SPA) space; block window command, status, and aperture register definitions; and the unique NFIT device handle utilized with NVDIMM DSM commands.

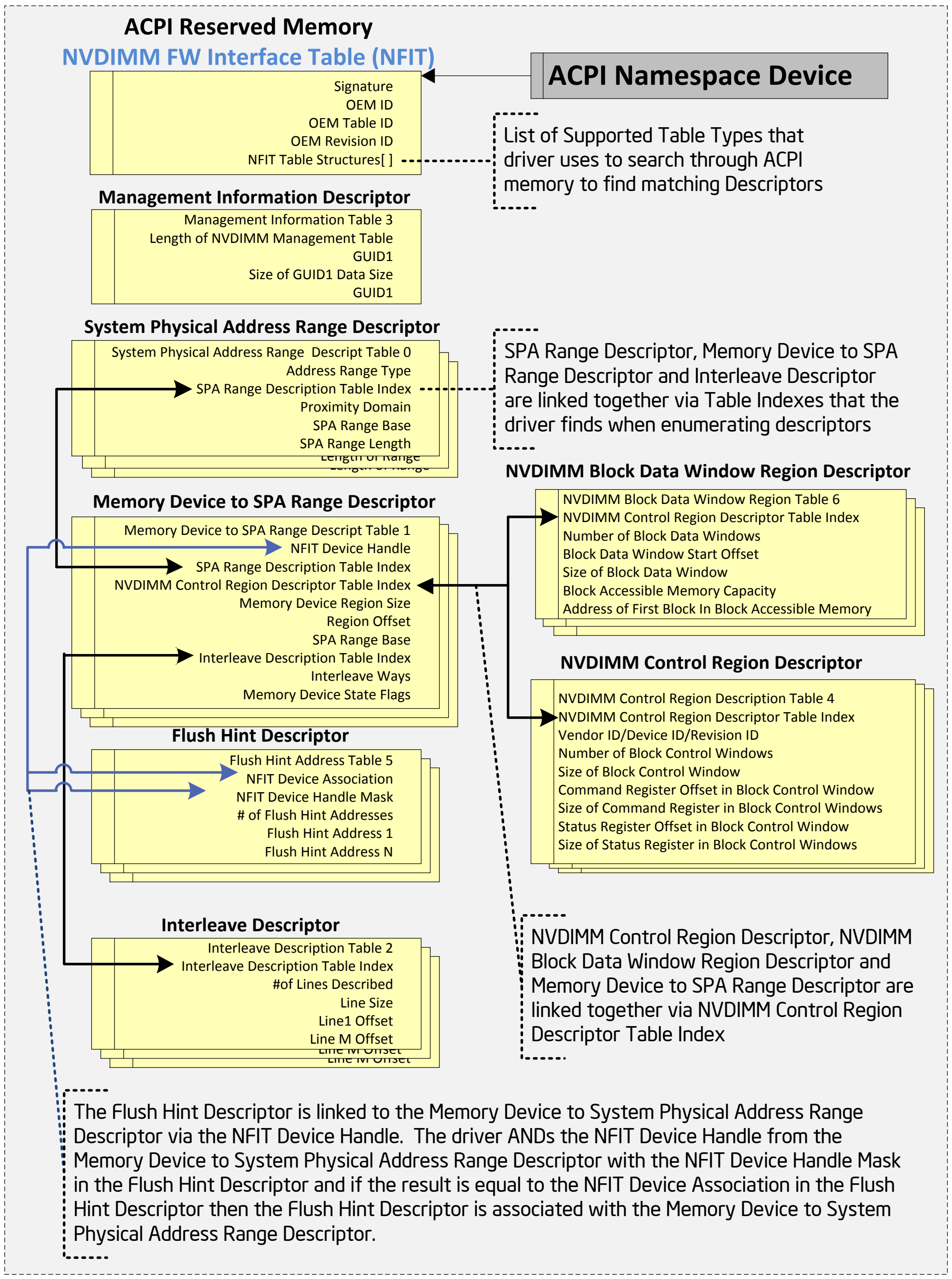

ACPI Reserved Memory
NVDIMM FW Interface Table (NFIT)
Signature
OEM ID
OEM Table ID
OEM Revision ID
NFIT Table Structures[ ]
ACPI Namespace Device
List of Supported Table Types that driver uses to search through ACPI memory to find matching Descriptors
Management Information Descriptor
Management Information Table 3
Length of NVDIMM Management Table
GUID1
Size of GUID1 Data Size
GUID1
System Physical Address Range Descriptor
System Physical Address Range Descript Table 0
Address Range Type
SPA Range Description Table Index
Proximity Domain
SPA Range Base
SPA Range Length
SPA Range Descriptor, Memory Device to SPA Range Descriptor and Interleave Descriptor are linked together via Table Indexes that the driver finds when enumerating descriptors
NVDIMM Block Data Window Region Descriptor
NVDIMM Block Data Window Region Table 6
NVDIMM Control Region Descriptor Table Index
Number of Block Data Windows
Block Data Window Start Offset
Size of Block Data Window
Block Accessible Memory Capacity
Address of First Block In Block Accessible Memory
Memory Device to SPA Range Descriptor
Memory Device to SPA Range Descript Table 1
NFIT Device Handle
SPA Range Description Table Index
NVDIMM Control Region Descriptor Table Index
Memory Device Region Size
Region Offset
SPA Range Base
Interleave Description Table Index
Interleave Ways
Memory Device State Flags
NVDIMM Control Region Descriptor
NVDIMM Control Region Description Table 4
NVDIMM Control Region Descriptor Table Index
Vendor ID/Device ID/Revision ID
Number of Block Control Windows
Size of Block Control Window
Command Register Offset in Block Control Window
Size of Command Register in Block Control Windows
Status Register Offset in Block Control Window
Size of Status Register in Block Control Windows
Flush Hint Descriptor
Flush Hint Address Table 5
NFIT Device Association
NFIT Device Handle Mask
# of Flush Hint Addresses
Flush Hint Address 1
Flush Hint Address N
Interleave Descriptor
Interleave Description Table 2
Interleave Description Table Index
#of Lines Described
Line Size
Line1 Offset
Line M Offset
NVDIMM Control Region Descriptor, NVDIMM Block Data Window Region Descriptor and Memory Device to SPA Range Descriptor are linked together via NVDIMM Control Region Descriptor Table Index
The Flush Hint Descriptor is linked to the Memory Device to System Physical Address Range Descriptor via the NFIT Device Handle. The driver ANDs the NFIT Device Handle from the Memory Device to System Physical Address Range Descriptor with the NFIT Device Handle Mask in the Flush Hint Descriptor and if the result is equal to the NFIT Device Association in the Flush Hint Descriptor then the Flush Hint Descriptor is associated with the Memory Device to System Physical Address Range Descriptor.

Figure 2 - NFIT Layout

Figure 2 - NFIT LayoutFigure 2 - NFIT Layout shows the basic components of the NFIT in reserved ACPI memory and the relationship of those components to each other. Note that the ACPI namespace device that is created by the BIOS describes the location of the NFIT in memory and is the device that the OS will load the NVDIMM driver against. This is in place of a more traditional HW PCI function that a storage device would typically use.

The NFIT is composed of a number of descriptors that are all placed in ACPI reserved memory space. The main NFIT table contains a list of NFIT table structures that are supported. The driver will utilize this list to search through memory and find all of the descriptors that match each supported list. Each descriptor begins with 2 bytes of table structure information that is used to find and enumerate all of the support descriptors. See the NFIT figure above for the layout.

The following descriptors are linked together in the NFIT:

- **Index Linked** – System physical address range descriptor and memory device to system physical address range descriptor tables are linked by the SPA range description table index. The memory device to system physical address range descriptor is linked to the interleave descriptors for each NVDIMM via the interleave description table index. The NVDIMM control region descriptor, NVDIMM block data window region descriptor, and memory device to system physical address range descriptor are linked together via the NVDIMM control region descriptor table index.
- **NFIT Device Handle Linked** – The memory device to system physical address range descriptor is linked to the flush hint descriptor table via the NFIT device handle. The driver ANDs the NFIT device handle from the memory device to system physical address range descriptor with the NFIT device handle mask in the flush hint descriptor. If the result is equal to the NFIT device association in the flush hint descriptor, then the flush hint descriptor is associated with the memory device to system physical address range descriptor.

In a system with multiple revisions of an NVDIMM implementation, the NFIT will contain unique instances of all of the above descriptors utilizing different range description table index, SPA range description table index, NVDIMM control region descriptor table index, interleave descriptor table index, node controller ID, socket ID, and memory controller ID. This allows different sizes and numbers of BW resources to be used in the same system and allows new NVDIMMs to be added without altering already established configurations.

The driver will create a number of derived resources based on the NFIT. See the following sections for more details on how the driver will make use of this information. Driver-derived NFIT information includes:

- **Number of NVDIMMs present in the system** - This is utilized throughout the driver for initialization and enumeration of the NVDIMMs.
- **Block Windows (BWs)** – The NFIT NVDIMM block data window region descriptor describes the starting offset, size, and number of BW data apertures on the NVDIMM. The NFIT NVDIMM control region descriptor describes the starting offset, size, and number of BW command and status registers present. See the NFIT figure and further discussion below for more details.

- **Interleave Information** – When utilizing the block window HW interface to move data in/out of the DIMM, the interleaving of that portion of the DIMMs must be understood and taken in to account. The NFIT interleave table will give the NVDIMM driver the information needed to determine the internal memory controller interleave in effect. Anytime the driver moves data with a BW aperture window, the starting offset of the aperture must be adjusted to account for interleaving. Likewise data movement larger than the smallest non-interleaved transfer size must take in to account "holes" that must be left in the aperture window to insure the data is only written to the NVDIMM being addressed. For details on handling memory controller interleave, see the section on Programming Block Windows, Calculating the System Physical Address (SPA).

Creating Internal Interleave Tables

One of the most important pieces of information that the driver will need to use for the I/O path is the information found in the NFIT interleave tables. In general the interleave information will be utilized for translating the DPA address offset of the BW command, status, and aperture registers into a virtual address that the driver will utilize to directly access the register while taking interleaving in to account. There is a basic hierarchical sequence the driver uses to find and retain that information from the various NFIT tables. Here is the basic NFIT enumeration sequence:

Find all of the NVDIMMs – Reference the NFIT system physical address range description tables for address range type 2, control regions, and find the SPA range description table index. Then reference all of the memory device to system physical address range mapping tables that have the same SPA range description table index. The interleave ways for any of the matching memory device to system physical address range mapping tables will be the number of NVDIMMs in the system. Since each NVDIMM will utilize separate control register regions, this count can be used as the number of NVDIMMs in the system.

Organize by Range Type - The driver enumerates each of the SPA range description tables and organizes the data by the address range type that each range describes. Type 1 ranges describe persistent memory and will need to be utilized for persistent memory accesses. Address range type 2 tables describe the ranges used for the BW command and status registers. Type 3 ranges describe the block window aperture registers. For each address range type the driver uses the start address and length to map the given physical region that the BIOS has set up in to the system virtual address space of the operating system. Additionally, the driver needs to keep track of the SPA range description table index for each range as this is used to find the interleave table pertaining to this NVDIMM.

Find Memory Device to SPA Range and Memcontroller ID – Using the SPA range description table index enumerate the memory device to system physical address range mapping tables that contain a SPA range description index that matches the indexes found in the previous step. This table describes the NVDIMM in detail including the memory controller ID. The driver must also save the **InterleaveWays** field contained in each of the memory device to SPA range mapping table copies as this will be utilized directly in determining the virtual addresses of the NVDIMM registers. Lastly, the driver uses the contained interleave description table index to find the interleave table that describes this NVDIMMs location in the interleave of the memory controller.

Save the Interleave LineSize, NumLines, and LineOffset[ ] – Using the interleave description table index, find all of the Interleave Tables that are present for each NVDIMM participating in each address range type. The important information in the interleave table is the NumLines, the LineSize, and the LineOffset[ ] array as described. These values will all be required to calculate the virtual address for a given register DPA address offset.

Create In-Memory Virtual Address Translation Table – It is recommended that the driver pre-calculate the BW command, status, and beginning aperture virtual address at driver initialization time. See the section on programming block windows for the details on how this per NVDIMM interleave information will be utilized by the driver to calculate the virtual addresses used when accessing the block windows.

Determining BW Control and Aperture Resources

The NVDIMM driver searches through ACPI reserved memory to find the NVDIMM control region description table header that identifies this table in memory. Once the table has been located the driver determines the size, number, and location offset of the BW control and status registers utilizing that descriptor table, specifically the size of block control window, number of block control windows, block control start offset fields. See Figure 4 to get an idea of what the table looks like. The descriptor also defines the size of the address and status fields and their field offsets within those registers, specifically, the size of address field in block control windows, size of status field in block cntrol windows, the address field offset in block control window, and status field offset in block control window fields.

The NVDIMM driver searches through ACPI-reserved memory to find the NVDIMM block data window region description table header, which identifies this table in memory. Once the table has been located, the driver determines the size, number, and location offset of the BW aperture registers utilizing that descriptor table, specifically the number of block control windows, block control start offset, size of block control window, command register offset in block control window, size of command register in block control windows, status register offset in block control window, and size of status register in block control windows fields. See Figure 4 to get an idea of what the table looks like.

By utilizing the NFIT to parse this information, the driver can avoid utilizing the NVDIMM controller-specific mechanism to retrieve this information, making the driver more portable and less specific to a particular NVDIMM controller implementation.

Note that with these NVDIMM descriptions each NVDIMM tilizes a unique vendor ID, device ID, and revision ID will have a separate unique NVDIMM block data window region and NVDIMM control region descriptor entries in the NFIT. This provides the flexibility to support multiple revisions of NVDIMMs with different block window characteristics in the same system.

NVDIMM Driver Device Discovery

At system power-on, the BIOS detects and initializes all NVDIMM devices installed in the system. The BIOS then creates the NFIT, which communicates appropriate details on all installed NVDIMMs to the operating system.

During initialization, it is expected that the driver will utilize the NFIT for most information and issue commands to each NVDIMM to collect the remaining necessary information utilizing the DSM interface. For details, refer to the ACPI 6.0 - NFIT section and the basic NFIT overview presented above. Here's a list of the information the driver would typically collect during initialization:

- **Determining DPA Register Offsets** – The driver reads the NFIT NVDIMM control region descriptor and the NFIT NVDIMM block data window region descriptor to obtain the offsets to all of the registers it will need to use to send BW commands, apertures, and receive BW status. This information is used in the NVDIMM HW register virtual address calculation described in detail further below in this document.
- **Determine NFIT Device Handle** – The driver utilizes the device mapping to system physical address range mapping table to find the NFIT device handle it will need to utilize when sending NVDIMM commands via the DSM pass-through mechanism specified in the NFIT.
- **NVDIMM List** - Determined from parsing the NFIT.
- **NVDIMM Memory Controller interleave** - Determined from the NFIT Interleave Table. Note that the interleave affecting the BW registers of all installed DIMMs are required by the driver.
- **Size, Number, and Location of BW Control Registers** – The driver parses the NFIT NVDIMM control region descriptor to determine the size of the BW control and status registers, the number of BW command and status registers supported by the NVDIMM, and the starting offset of the BW control and status registers in the NVDIMM register map. This information is utilized in the NVDIMM HW register virtual address calculation described in detail further below in this document. The NDIMM control region flag Bit 0 set to 1 is used as an indicator that the NVDIMM is buffered and will set the pending bit in the BW status register until the data transfer has completed.
- **Size, Number, and Location of BW Aperture Registers** – The driver parses the NFIT NVDIMM block data window region descriptor to determine the size of the BW aperture registers, the number of BW apertures supported by the NVDIMM and the starting offset of the BW aperture registers in the NVDIMM register map. This information is used in the NVDIMM HW register virtual address calculation described in detail below in this document.
- **Retrieving NVDIMM Namespace Labels** – The driver will need to utilize the native NFIT-specified DSM get namespace label data area size and get Namespace Label Data Area methods to retrieve the Namespace Labels that are stored on the NVDIMM. Additionally, the driver will need to use the native NFIT-specified DSM set namespace label data area method when creating new or modifying existing namespace labels on behalf of the storage management software.

- **Retrieving SMART Health Information and SMART Thresholds** – The driver can optionally use the native NFIT SMART and health information DSM method to retrieve per NVDIMM SMART status. Currently it is not possible to have the driver interrupt when SMART or health status has changed so if the driver wants to detect such events, it must currently poll, periodically issuing this DSM method to check for SMART status changes. The driver can also optionally determine the current SMART thresholds that have been set by utilizing the native NFIT GetSmartThresholds DSM method.
- **Address Range Scrubbing** – The driver can use the native NFIT QueryAddress RangeScrubCapabilities DSM method to determine if address range scrubbing (ARS) is supported by the system. Additionally, the driver can start an address range scrub to find poison locations in the range of the requested scrub by utilizing the native NFIT StartAddressRangeScrub DSM method. Lastly, the driver can determine if an ARS is in progress and retrieve the resulting error log for the platform when the ARS completes by utilizing the QueryAddressRangeScrubStatus DSM method. Note that these methods are all addressed to the root ACPI namespace device and do not require an NFIT device handle. See the section on error handling for more details on possible driver uses of these ARS commands.

Issuing DSM Commands to the NVDIMM

The DSM specification outlines the NVDIMM-agnostic interface that the driver should utilize for communicating NVDIMM commands with the NVDIMM required for NVDIMMs adhering to format interface code 1. This interface is based on the driver building a payload and sending it to the BIOS via a Device Specific Method or DSM. By having the BIOS handle these requests the implementation details of a specific NDIMMs interface are hidden from the driver, allowing the driver to implement a management interface that can work for multiple NVDIMM implementations. The NFIT specification outlines the DSM payload fields and specific data payloads for GetSmartHealthInformation, GetNamespaceLabelDataAreaSize, GetNamespaceLabelDataAreaData, SetNamespaceLabelData, address range scrub, and a vendor specific pass-through command. The latter interface is utilized by the driver for any commands not specifically specified in the NFIT specification.

The DSM interface is synchronous meaning that the call to the ACPI stack to send the message will not complete until the underlying command to the NVDIMM has completed. However, the address range scrubbing (ARS) DSM command will return once the scrub has been initiated (via the start ARS command) and the driver will need to re-submit the command (using the query ARS status) to poll for ARS completion.

The DSM message consists of a UUID that defines the DSM interface, the revision ID, the function index (essentially the opcode), followed by the DSM payload. In general the DSM payload contains the NFIT device handle, followed by some number of bytes of input data, status, and some number of bytes of output data. The NFIT device handle is found in the device mapping to system physical address range mapping table, as specified in the ACPI 6.0 NFIT Specification. The UUID and revision ID are also specified in the NFIT specification.

Preconditions:

- The NVDIMM driver has an OS-specific mechanism to send DSM messages to the BIOS.
- The NVDIMM driver has determined the NFIT device handle (from the NFIT) for the specific NVDIMM it wishes to send the NVDIMM command to.
- The NVDIMM driver has a NVDIMM command to send, formatted per the DSM specification.
- The NVDIMM driver has the NVDIMM command payload, if any, to send, formatted per the NFIT specification.

Steps:

1. The NVDIMM driver builds the DSM message per the NFIT specification formatting.
2. The NVDIMM driver sends the DSM message to the BIOS code and when the message is completed back to the driver, it checks the resulting Status in the output payload to verify the completion status of the command. Note that for the address range scrub command (start ARS) the driver will need to send further commands (query ARS status) to poll for the final completion.

Block Windows

The block window aperture mechanism allows software access to data in persistent memory without mapping the entire persistent memory range into the kernel's virtual address space. The accessible range of persistent memory therefore may exceed the available SPA space and/or kernel virtual address space supported by the host OS. A positive side-effect of the block window aperture architecture is that large ranges of persistent memory are not exposed to stray writes from faulty or malicious kernel code.

Each NVDIMM has some number of block windows (BWs) for use with the NVDIMM block window interface. The exact number of BWs, their location, and their size is found using the NFIT. See the following sections that outline the details of how the block window interface is programmed by the driver.

A Block Window contains three things:

- **Command Register** - Points the BW to a specific DPA, sets the direction for the transfer, and the size of the transfer
- **8k Aperture** - Provides a window to allow access to the data at the target DPA
- **Status Register** - Latches errors during transfers, cleared by setting the command register

Some basic points about using Block Windows:

- Every NVDIMM requires the use of block window registers that are specific to that NVDIMM.
- Since each NVDIMM has multiple block windows, many parallel operations can be carried out on the same NVDIMM at the same time.

- Each block window is used for the transfer of one logical block of data. Multi-block transfers are done by transferring individual blocks with BW command register programming in between each block.
- Block Windows can be reused immediately upon completion of a data transfer. There is no performance advantage to switching BWs between transfers.
- Moving data through a block window aperture utilizes load and store instructions that operate on virtual addresses. Because of this, drivers do not need to use physical DMA scatter gather lists and can simply use the virtual addresses of the host data buffers.
- Each NVDIMM supports an equal number of BW command, status, and aperture registers, so multiple overlapped IO can move data through the NVDIMM at the same time.

Driver threads taking ownership of a BW must follow these basic rules or risk data corruption:

- After the driver selects the BW command and aperture registers and calculates the DPA for the block window transfer, interleaving is taken in to account by calculating the virtual address for both registers. The driver uses the virtual address to program the BW command and to copy the data in/out of the aperture. The driver moves the data based on the LineSize of the interleave that is in effect and must re-calculate the register aperture virtual address for the data movement at each LineSize interval. The driver uses the NFIT interleave table to calculate these virtual register addresses, which takes in to account the interleaving in affect. See the following sections that outline the details of how the virtual register address is calculated.
- BW command register must be programmed with appropriate R/W bit and transfer size individually for each block transfer. The transfer size is the size of a logical block in 64-byte cache line increments. While the aperture size is 8k, only one block of data can be moved through an aperture at a time, and the any unused space in the aperture is not utilized.
- After writing the BW command register to reprogram the aperture, cached data from previous BW use may still be sitting in cache. So before reading from the aperture the data must be invalidated using CLFLUSHOPT (the lines cannot be dirty due to these rules, so CLFLUSHOPT acts like an invalidate operation). Exception: If the Block Data Window Invalidation Required bit is clear in the NVDIMM flags field returned by the GetBlockNvdimmFlags DSM command, then the NVDIMM does not require flushes before BW reads for that device.
- Writes done through the aperture must be done using non-temporal instructions or must be followed by CLFLUSHOPTs so no dirty cache lines exist when the BW is released.

For performance, not correctness, this rule should also be followed:

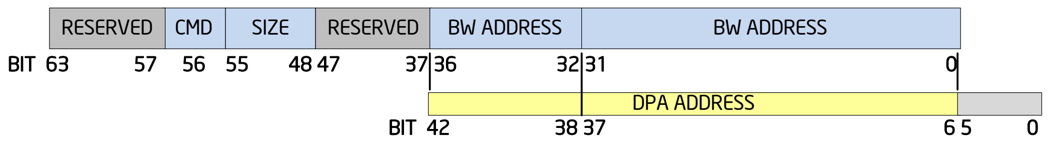
- A block write via a BW should not be preemptible (the entire block transfer happens on the same CPU).

Warning: Attempting to utilize the same physical portions of an NVDIMM for block windows and direct persistent memory access, at the same time, will lead to undefined behavior! While these regions can all be used on each NVDIMM at the same time, on the same NVDIMM, the regions cannot overlap one another. It is the responsibility of the driver and management SW to enforce this rule.

Block Window Register Definition

Figure 3 - BW Command and Address Register Format shows the format for the BW command and address register. The first 37 bits of a BW address are programmed with the DIMM physical address (DPA) for the block window transfer, shifted right 6 bits to make the address a 64-Byte cache line relative address.

BLOCK WINDOW HW REGISTER FORMAT – BW COMMAND/ADDRESS REGISTER

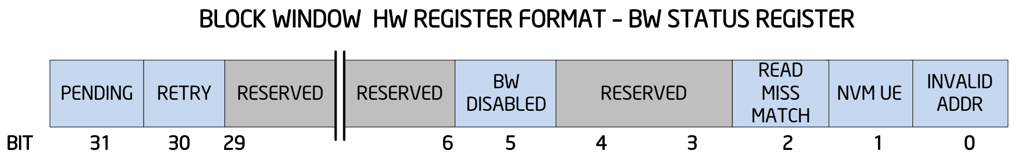


FIELD	DESCRIPTION
BW ADDRESS [31:0]	The BW Cache Line Relative DPA Address [37:6] for the request
BW ADDRESS [36:32]	The BW Cache Line Relative DPA Address [42:38] for the request
SIZE	The size of the BW transfer in number of cache lines.
COMMAND	Bit [56] is 1 – Write Command
	Bit [56] is 0 – Read Command

Figure 3 - BW Command and Address Register Format

The transfer size is the number of cache lines of data that the request will move based on a 64-Byte cache line utilized by all of the Intel® Architecture platforms that support the NVDIMM. The maximum transfer size supported by the NVDIMM is 128 cache lines for a maximum logical block size of 8192 bytes. Only the lowest bit 56 of the command is currently used where a 1 is a write and a 0 is a read request. The NVDIMM will use this setting to determine if any data moves through the aperture in an unexpected direction. See the status register description below for direction error reporting.

Error! Reference source not found. describes the format of the block window tatus register. The hardware clears this register after this register is read by the driver.



FIELD	DESCRIPTION
INVALID ADDRESS	The BW address programmed is outside the Memory Device Physical Address Region specified in the NFIT
NVM UE	An uncorrectable error occurred upon NVDIMM access to the given BW Address
READ MISS MATCH	Set when a BW Write Command is received when the Command Register is configured for a READ operation
BW DISABLED	Set when the BW request attempts to access a Block Window region of the NVDIMM that is disabled
RETRY	Set if the request should be retried
PENDING	Set if the NVDIMM is still executing the BW Aperture data transfer at the time the Status Register is read

Figure 4 - BW Status Register Format

The INVALID ADDRESS bit indicates the DPA specified in the BW address register is not a valid address for the NVDIMM being accessed. The NVM UE bit indicates that an uncorrectable ECC error occurred while attempting to access the NVDIMM at the DPA BW address programmed. The NVDIMM HW checks the direction of the data movement within the BW aperture against the command bit specified in the BW command register and indicates a read miss-match utilizing the READ MISSMATCH status bit. Finally a data movement utilizing a disabled BW region will result in setting the BW DISABLED bit. If all of the utilized bits described here are 0, the BW transfer succeeded.

Note: The driver cannot assume the value of the RESERVED bits in the status register are zero. These reserved bits need to be masked off, and the driver must avoid checking the state of those bits.

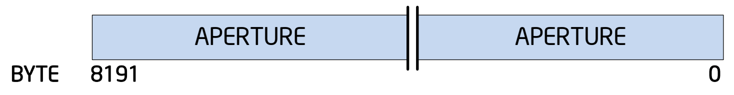
The NVDIMM will set the RETRY bit if the request failed and should be retried by the driver. The driver should only check this bit if the NFIT NVDIMM control region table, NDIMM control region flag bit 0 set to 1, indicating that the NVDIMM is buffered. It is proposed that un-buffered NVDIMMs will not require a retry.

The PENDING bit is utilized for those NVDIMM implementations that do not complete the BW aperture data transfer before the time the status register is read. The NFIT NVDIMM control region table, NVDIMM control region flag Bit 0 set to 1 indicates that the NVDIMM is buffered and will set the pending bit in the BW Status register until the data transfer has completed. This bit is for supporting NVDIMM implementations that require the driver to poll for BW data transfer completion. It is up to the driver author to determine if buffered NVDIMMs will be used with the driver and handle the pending bit being set.

Figure 5 – BW Aperture Register Format shows the 8KB aperture format utilized for block-sized data movement through the block window hardware. The driver must

calculate the SPA virtual address for each LineSize amount of data that is transferred to account for the memory controller's interleaving of data.

BLOCK WINDOW HW REGISTER FORMAT – BW APERTURE REGISTER



FIELD	DESCRIPTION
APERTURE	The NVDIMM HW Window into 8k of the Persistent Memory whose address is the lower 37 bits of the BW Command/Address register

Figure 5 – BW Aperture Register Format

Calculating Register SPA and Virtual Addresses

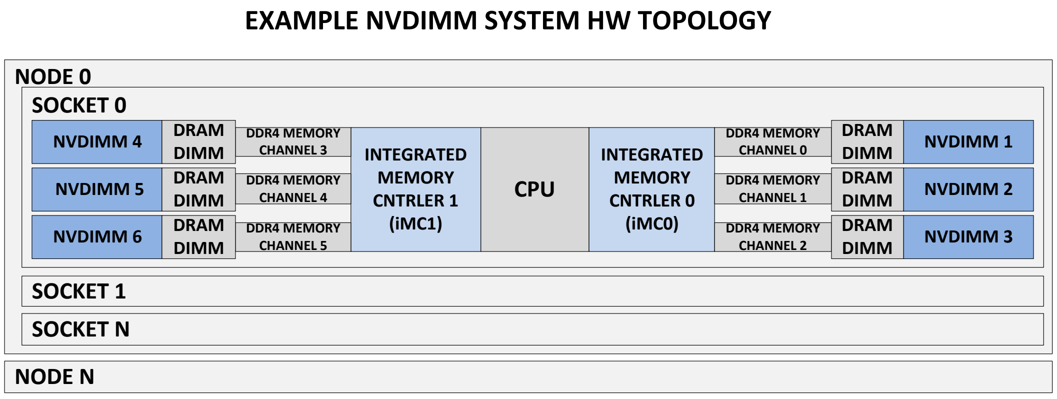
In order for the driver to take memory controller interleave in to account when reading and writing the BW registers, the register offset in the NVDIMM register layout is run through a series of calculations to find the relevant NVDIMM HW tegister system physical address (SPA). The SPA is then converted to a NVDIMM HW register virtual address. These calculations are based on the NIFT table in general and specifically the interleave table for each NVDIMM. So once the NVDIMM for the request has been determined, that NVDIMM's interleave table will be referenced by the driver to calculate the driver's virtual address used for each register access. This will account for the starting offset and will skip required holes in the address space for the interleave.

Before detailing the driver calculations that are performed to get the SPA and the NVDIMM HW register virtual address, it is important to know some of the background in how the HW is set up and how the interleave works. This will make the NFIT Interleave Tables and the NVDIMM HW Register Virtual Address calculation easier to understand, detailed later in this section.

Memory Controllers, Channels, Multi-Way Channel Interleave

BW registers can exist in an interleave of multiple NVDIMMs and because of this, the driver needs to account for the interleaving that's in effect when accessing BW registers. This will be discussed in detail in the Block Window section of this document.

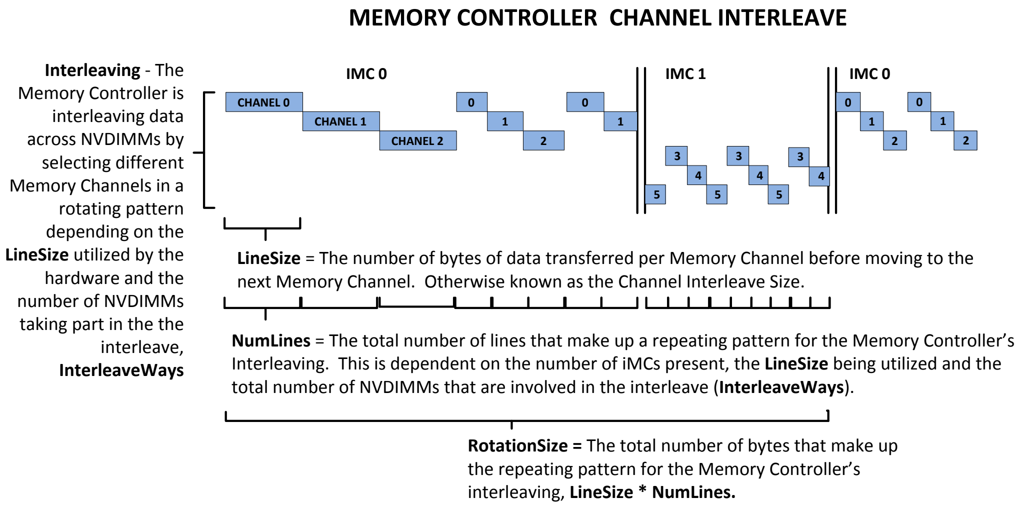
Figure 6 - Memory Controller TopologyFigure 6 - Memory Controller Topology shows a basic high-level diagram of a platform containing NVDIMMs to help describe how CPU, Internal Memory Controllers, Memory Channels, DRAM DIMMs, and NVDIMMs are related. The InterleaveWays is the number of NVDIMMs connected to each internal memory controller multiplied by the number of internal memory controllers being utilized for the socket. The driver uses the InterleaveWays value to calculate the NVDIMM HW register virtual address used for writing the block window command registers and BW apertures, and for reading the BW status register.



TERM	DESCRIPTION	EXAMPLE VALUE
#IMCs	Number of Integrated Memory Controllers (iMCs) that have DDR4 DIMMs or NVDIMMs attached	2
#WAYS PER IMC	Number of NVDIMMs connected per Integrated Memory Controller	3
InterleaveWays	Total number of NVDIMMs in this Interleave Set: #WAYS PER IMC * #IMCs	6

Figure 6 - Memory Controller Topology

Figure 7 - Memory Controller Interleave is a basic diagram explaining memory channel interleave and the concepts of interleaving, LineSize, NumLines, and RotationSize. The driver will utilize these variables when calculating the NVDIMM HW register virtual address used for writing the block window command and BW apertures, and for reading the BW status register.



Interleaving - The Memory Controller is interleaving data across NVDIMMs by selecting different Memory Channels in a rotating pattern depending on the **LineSize** utilized by the hardware and the number of NVDIMMs taking part in the the interleave, **InterleaveWays**

LineSize = The number of bytes of data transferred per Memory Channel before moving to the next Memory Channel. Otherwise known as the Channel Interleave Size.

NumLines = The total number of lines that make up a repeating pattern for the Memory Controller's Interleaving. This is dependent on the number of iMCs present, the **LineSize** being utilized and the total number of NVDIMMs that are involved in the interleave (**InterleaveWays**).

RotationSize = The total number of bytes that make up the repeating pattern for the Memory Controller's interleaving, **LineSize * NumLines.**

Figure 7 - Memory Controller Interleave

Interleaving with Persistent Memory

Regions of the NVDIMMs that are configured as persistent memory will utilize memory controller interleave, which is factored in to the SPA range. The SPA range is described in the NFIT, system physical address range descriptor with address range type 1. Since the SPA range is accessed directly with load/store instructions and does not make use of NVDIMM registers like the block window interface does, the driver does not need to account for interleaving when calculating virtual addresses.

If an NVDIMM is added to the system and is configured as persistent memory, a new interleave set will be created for the new persistent memory when the system is next rebooted. So any persistent memory that has been added to the system shows up as a new interleave set, leaving the existing sets undisturbed.

NVDIMM HW Register SPA and Virtual Address Calculation

The driver communicates with the NVDIMM via registers that the BIOS has mapped into the SPA space. Since these registers exist in memory that is interleaved by the memory controller, the driver must calculate the NVDIMM HW register virtual address for every BW command and status register or BW aperture it is going to access by first calculating the SPA of the register.

The NVDIMM HW register calculation requires the register address offset for the register being accessed by the driver and the specific NVDIMM described by the namespace for the IO being accessed. The register address offset for BW command registers is found by referencing the NFIT NVDIMM control region table, command register offset in block control region and size of command register in block control window. The register address offset for BW status registers is found by referencing the NFIT NVDIMM control region table, status register offset in block control region and size of status register in block control window. Likewise, to find the register address offset for BW aperture registers, reference the NFIT NVDIMM block data window region table, block data window start offset and size of block data window.

Virtual Address Calculation with Interleave

Figure 8 – Virtual Address Calculation with InterleaveFigure 8 – Virtual Address Calculation describes the calculations the driver must execute to determine the kernel system virtual address for any NVDIMM HW register when an interleave is present. The figure also contains the data required for each calculation and the NFIT Table references required to complete the calculation steps. The calculation will require access to the NFIT system physical address range description table, memory device to SPA range mapping table and the interleave table, probably cached in driver memory for quick efficient access. The NVDIMM described by the namespace for this IO is utilized for all references to the NFIT tables which are used to find the LineSize, NumLines, InterleaveWays, and LineOffset. Note that the StartingPhysAddress is calculated at driver initialization time when the NFIT is parsed for each of the NFIT address ranges. The same steps are used to determine the SPA for the address type 2 control registers including block control and status registers, and address type 3 block apertures simply by replacing the StartingPhysAddress with the address for that region.

Here are the basic execution steps for calculating the NVDIMM HW Register Address for any NVDIMM:

1. **Calculate the starting virtual address of the memory region** – The driver uses mapping address range type 2, control region, when finding the virtual address for block control, and block status registers. Likewise the mapping address range type 3 is used when finding the virtual address for the block window apertures. The desired address range type is used to access the NFIT system physical address range description table and find the physical address the BIOS has mapped, the StartingPhysAddress.
2. **InterleaveWays** –The driver determines the number of NVDIMMs taking part in the interleave of the memory controller by accessing the specific NFIT memory device to system physical address range mapping table. The driver determines the NVDIMM from the namespace for this IO and stores the reported InterleaveWays.
3. **LineSize, NumLines** – Using the NVDIMM described by the namespace for this IO, the driver selects the NFIT interleave description table and stores the reported LineSize and NumLines for use in calculating the LineNumber, RotationSize, and Remainder. The LineSize is the number of bytes the internal memory controller will transfer in one memory channel request. NumLines represents the minimum number of lines in the interleave set before the interleave pattern repeats.
4. **LineNumber = (Register Offset % RotationSize) / LineSize** – The specific line number the driver will use as an index in to the NFIT Interleave Table LineOffset[ ] array.
5. **RotationSize = LineSize * NumLines** – The minimum number of bytes in the interleave set before the interleave pattern repeats.
6. **RotationNumber = Register Offset / RotationSize** – Which rotation the register access is in based on the minimum number of bytes in the interleave set.
7. **Remainder = Register Offset % LineSize** – Any bytes less than the LineSize of 256Bytes is considered the remainder that must be added in to the NVDIMM HW register virtual address.
8. **LineOffset –** The driver accesses the NFIT interleave description table for the specific NVDIMM described by the namespace for this IO and uses the LineNumber calculated in step 4 to retrieve the LineOffset array value, LineOffset[LineNumber]. This is the interleave offset to be applied for the given LineNumber.
9. **Register SPA = RotationSize * RotationNumber * InterleaveWays + LineOffset + Remainder + StartingPhysAddress –** This is the physical address for the desired register.
10. **NVDIMM HW Register VA** – The virtual address for the register SPA is found using OS-specific means. This is virtual address the driver uses to access the desired register.

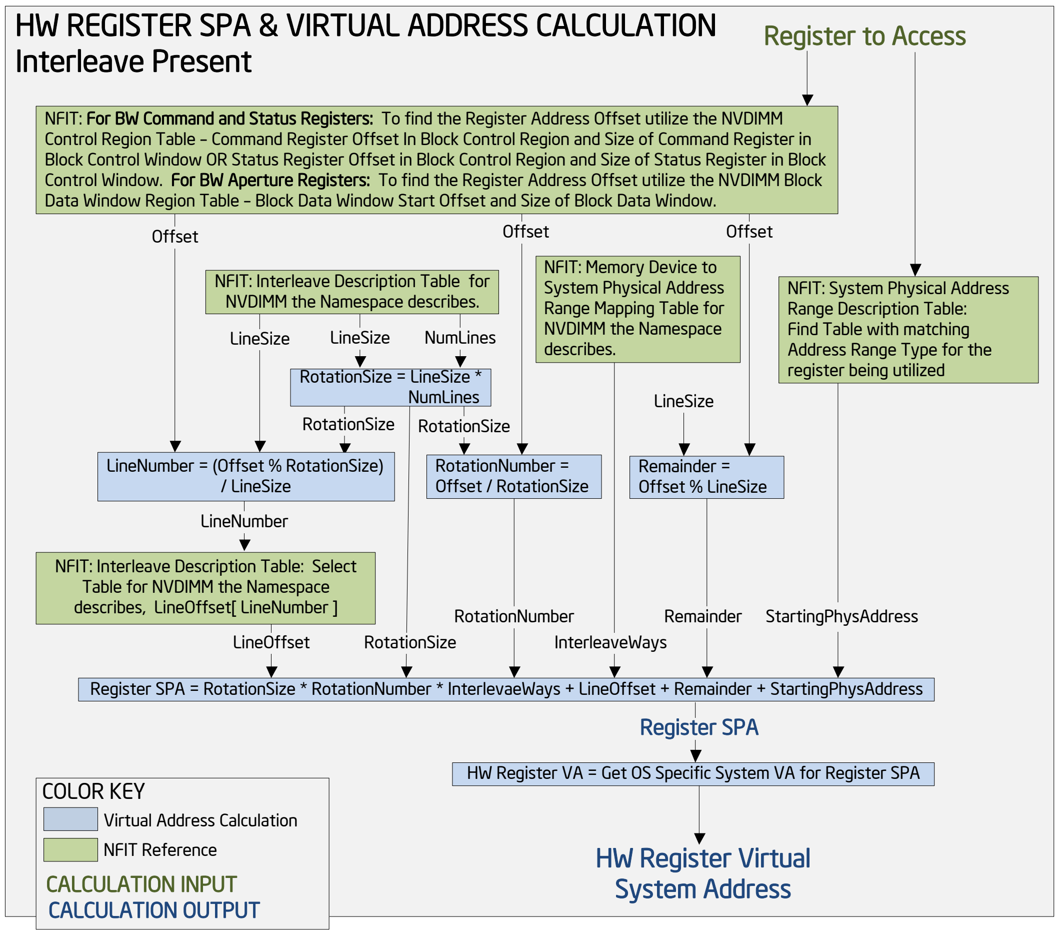

HW REGISTER SPA & VIRTUAL ADDRESS CALCULATION
Interleave Present
Register to Access
NFIT: For BW Command and Status Registers: To find the Register Address Offset utilize the NVDIMM Control Region Table – Command Register Offset In Block Control Region and Size of Command Register in Block Control Window OR Status Register Offset in Block Control Region and Size of Status Register in Block Control Window. For BW Aperture Registers: To find the Register Address Offset utilize the NVDIMM Block Data Window Region Table – Block Data Window Start Offset and Size of Block Data Window.
Offset
Offset
Offset
NFIT: Interleave Description Table for NVDIMM the Namespace describes.
NFIT: Memory Device to System Physical Address Range Mapping Table for NVDIMM the Namespace describes.
NFIT: System Physical Address Range Description Table: Find Table with matching Address Range Type for the register being utilized
LineSize
LineSize
NumLines
RotationSize = LineSize * NumLines
LineSize
RotationSize
RotationSize
LineNumber = (Offset % RotationSize) / LineSize
RotationNumber = Offset / RotationSize
Remainder = Offset % LineSize
LineNumber
NFIT: Interleave Description Table: Select Table for NVDIMM the Namespace describes, LineOffset[LineNumber]
RotationNumber
Remainder
StartingPhysAddress
LineOffset
RotationSize
InterleaveWays
Register SPA = RotationSize * RotationNumber * InterlevaeWays + LineOffset + Remainder + StartingPhysAddress
Register SPA
HW Register VA = Get OS Specific System VA for Register SPA
HW Register Virtual System Address
COLOR KEY
Virtual Address Calculation
NFIT Reference
CALCULATION INPUT
CALCULATION OUTPUT

Figure 8 – Virtual Address Calculation with Interleave

Virtual Address Calculation without Interleave

Figure 9 - Virtual Address Calculation without Interleave describes the Virtual Address calculation for each register when no interleave is present. This calculation simplifies to:

1. **Register SPA = Register Address Offset + StartingPhysAddress** . – The physical address for the desired register.
2. **NVDIMM HW Register VA** – The virtual address for the Register SPA is found using OS-specific means. The driver uses this virtual address to access the desired register.

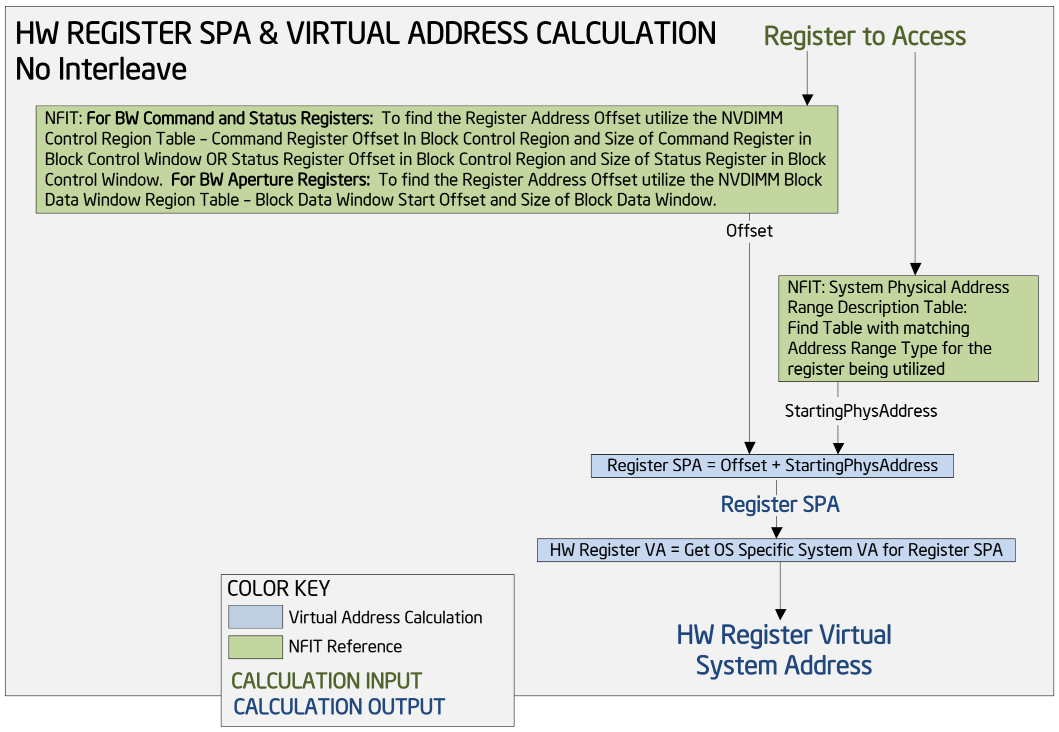

HW REGISTER SPA & VIRTUAL ADDRESS CALCULATION
No Interleave
Register to Access
NFIT: For BW Command and Status Registers: To find the Register Address Offset utilize the NVDIMM Control Region Table – Command Register Offset In Block Control Region and Size of Command Register in Block Control Window OR Status Register Offset in Block Control Region and Size of Status Register in Block Control Window. For BW Aperture Registers: To find the Register Address Offset utilize the NVDIMM Block Data Window Region Table – Block Data Window Start Offset and Size of Block Data Window.
Offset
NFIT: System Physical Address Range Description Table: Find Table with matching Address Range Type for the register being utilized
StartingPhysAddress
Register SPA = Offset + StartingPhysAddress
Register SPA
HW Register VA = Get OS Specific System VA for Register SPA
HW Register Virtual System Address
COLOR KEY
Virtual Address Calculation
NFIT Reference
CALCULATION INPUT
CALCULATION OUTPUT

Figure 9 - Virtual Address Calculation without Interleave

Block Window Aperture Programming

This section provides the details of the final BW Aperture memcopy loop, which moves Line Size number of Bytes of data to/from the Aperture using the NVDIMM HW register address calculation and the virtual address of the host I/O request buffer.

Figure 10 - Block Window Programming Sequence demonstrates the sequence the driver executes to transfer data utilizing a block window.

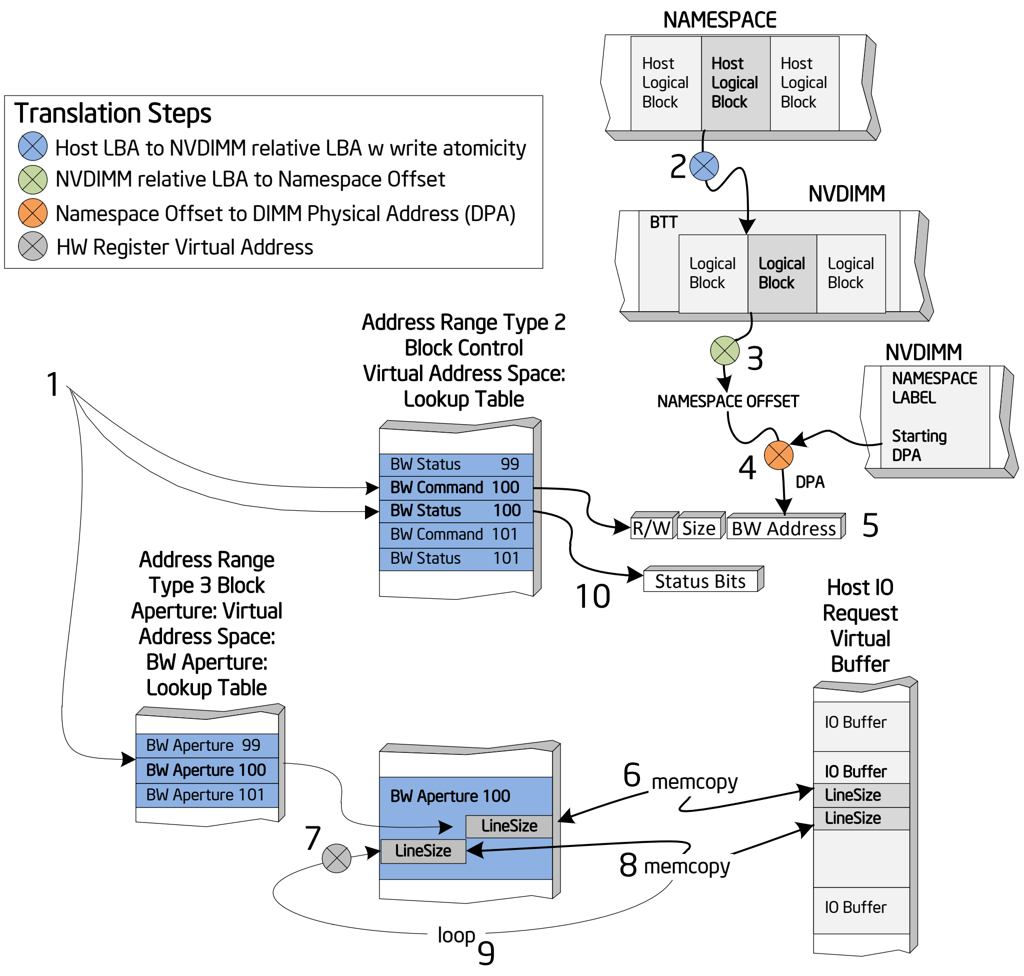


Figure 10 - Block Window Programming Sequence

The following BW programming steps are shown in Figure 10 - Block Window Programming Sequence:

STEP	DESCRIPTION
INIT TIME	Driver builds a virtual address table of BW command, status, and aperture registers
1	Driver selects a BW command, status, and initial aperture for the IO request, from virtual memory utilizing pre-initialized driver tables
2	Driver translates the host LBA to a NVDIMM relative LBA with write atomicity (if present) accounted for
3	Driver translates NVDIMM relative LBA in to a namespace offset
4	Driver calculates the DIMM physical address (DPA) from the namespace offset
5	Driver writes the DPA in to the virtual BW command register
6	Driver executes load/store instructions to memcopy the first LineSize amount of data utilizing the starting virtual BW aperture address and the IO buffer virtual address
7	Driver increments the BW aperture address by the LineSize and calculates the next virtual address for the next portion of the BW aperture
8	Driver executes load/store instructions to memcopy the next LineSize amount of data utilizing the BW apertures virtual address and the next portion of the IO buffer virtual address
9	Steps 7 and 8 are repeated until all data for the logical block has been transferred
10	The driver reads the final BW status using the block window status register's virtual address

Basic Driver Execution Flows

This section describes the basic high-level flows the NVDIMM driver uses to perform block mode reads and writes. For more in-depth execution flows, see the sample Linux Block Driver.

Additional notes that apply to the flows are outlined below:

- "if (FlushRequired)" – true if the NVDIMM requires CPU cache flushes after BWs have been moved. This is indicated in the Block Data Window Invalidation Required flag returned as a response to the GetBlockNvdimmFlags DSM command.
- The BW Status register is assumed to not set bit 31, the pending status bit, and these flows do NOT account for the pending status the NVDIMM returns. A future Driver Writer's Guide update will contain details for handling pending once those flows have been architected.
- Any of the steps that require a virtual address to be calculated for a register access can be implemented with a pre-initialized table. This allows the address translation to be calculated at driver initialization time, removing those calculations from the run time IO path.
- Note that the namespace labels specified in the NVDIMM Namespace Specification support a physical block size on the media that is larger than the logical block size reported to the host operating system. The physical block size utilized with the BW aperture must be a multiple of 64 bytes since data is moved with cache line granularity. The logical block size does not have such restrictions. The following flows do not show the required logic for handling logical block sizes < physical block size, but it is easy to use a scratch buffer to load or store the remaining number of bytes (physical block size – logical block size) after the required load/store transfer of the logical block size number of bytes to/from the host data buffer has occurred. Since the bytes transferred in/out of the scratch buffer are not part of the host IO transfer, the additional transfer is only used to satisfy the BW HW and the data is thrown away. This solution wastes space in between each logical block but makes the implementation easy.

Block Read

The following execution steps should be followed when reading a single logical block. This sequence of steps is repeated for each logical block being transferred:

1) Map the destination buffer for write access from kernel virtual.
2) Determine the external LBA for the I/O request.
3) Pass the external LBA to the BTT IO layer to calculate the proper post-map LBA for the request based on whether write atomicity is being utilized and the size and number of free blocks in the BTT. See the NVDIMM Namespace Specification for details on this step.
4) Translate the post-map LBA in to namespace offset.
5) Translate the namespace offset in to the DPA.
6) Using the NVDIMM described by the namespace for this IO, select an available BW command register, status register and BW aperture.
7) Calculate the command register virtual address using the driver-derived interleave information, and program the BW command register with the DPA, read mode, and LBA size.
8) Issue a SFENCE/PCOMMIT, followed by an SFENCE.
9) Flush cache if FlushRequired. This will flush cache of possible stale data from prior use of BW, using CLFLUSHOPT with the BW address for size of block.
10) Calculate the BW aperture register virtual address using the driver-derived interleave information, and copy the interleave LineSize number of bytes of data from the BW aperture to the destination buffer virtual address. Increment the BW aperture and destination buffer by the LineSize and repeat this step until the entire logical block has been copied.
11) Calculate the BW status register virtual address using the driver-derived interleave information, and check the corresponding BW status register for any errors.
12) Release the BW resources and unmap the destination buffer.

Block Write

The following execution steps should be followed when writing a single logical block. This sequence of steps is repeated for each logical block being transferred:

1) Map the source buffer for read access from kernel virtual address space.
2) Determine the external LBA for the IO request.
3) Pass the external LBA to the BTT IO layer to calculate the proper post-map LBA for the request based on whether write atomicity is being used and the size and number of free blocks in the BTT. See the NVDIMM Namespace Specification for details on this step.
4) Translate the Post-Map LBA in to namespace offset.
5) Translate the namespace offset in to the DPA.
6) Using the NVDIMM described by the namespace for this IO select the appropriate BW command register, status register and BW aperture.
7) Calculate the Command register virtual address using the driver derived interleave information, and program the BW command register with the DPA, write mode, LBA size.
8) Issue PCOMMIT, followed by an SFENCE.
9) Calculate the BW aperture register virtual address using the driver-derived interleave information, and copy the interleave LineSize number of bytes of data from the source buffer virtual address to the BW aperture (uses non-temporal store instructions). Increment the BW aperture and source buffer by the LineSize and repeat this step until the entire logical block has been copied.
10) Issue SFENCE, PCOMMIT, followed by an SFENCE.
11) Calculate the BW status register virtual address using the driver-derived interleave information, and check the corresponding BW status register for any error.
12) Using the allocated post-map LBA from step #3 above, utilize the BTT IO layer to update the on-media BTT metadata based on the write atomicity and free block settings configured. See the NVDIMM Namespace Specification for details on this step.
13) Release the BW resources and unmap the source buffer.

Block Flush

Since all block window writes are made persistent before the writes are completed back to the host, there is never any write data that needs to be made persistent at a later point in time. Thus block flush or synchronize cache requests from the host can be treated as NO-OPs.