

NVDIMM Namespace Specification

April 2015

Revision 1.0





You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request. Copies of documents which have an order number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at <http://www.intel.com/design/literature.htm>.

Intel and the Intel logo are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2015, Intel Corporation



Contents

1	Introduction.....	6
1.1	Document Scope	6
1.2	Related Documents.....	7
1.2.1	NVDIMM Firmware Interface Table (NFIT).....	7
1.2.2	NVDIMM DSM Example Definition.....	7
1.3	Terminology	7
1.4	Overview	10
1.4.1	NVDIMMs and the System Physical Address Space	10
1.4.2	NVDIMM Access via Block Window	12
1.4.3	Namespaces	13
1.4.4	Driver Software	17
2	Namespaces.....	20
2.1	Namespace Label Index Block Layout	21
2.1.1	The Label Index Block <code>seq</code> Field	24
2.1.2	The Label Index Block <code>free</code> Field	25
2.2	Namespace Label Layout	25
2.2.1	The Label <code>uuid</code> Field	28
2.2.2	The Label <code>name</code> Field	28
2.2.3	The Label <code>flags</code> Field.....	28
2.2.4	The Label <code>nlabel</code> and <code>position</code> Fields.....	29
2.2.5	The Label <code>isetcookie</code> Field	30
2.2.6	The Label <code>lbasize</code> Field	31
2.3	Namespace Label Rules.....	31
2.3.1	Validating Label Index Blocks and Labels	32
2.3.2	Reading Namespace Labels	32
2.3.3	Recovery Steps on Namespace Labels	33
2.3.4	Assembling Namespace Labels into Complete Sets	34
2.3.4.1	Incomplete Namespaces	34
2.3.5	Updating the Contents of the Label Storage Area	35
2.3.6	Writing New Namespace Labels.....	35
2.3.6.1	Writing New Persistent Memory Namespace Labels ..	36
2.3.6.2	Writing New Block Mode Namespace Labels.....	36
2.3.7	Updating Existing Namespace Labels	37
2.3.7.1	Updating Persistent Memory Namespace Labels.....	37



2.4	The Label-less Namespace	38
2.5	Virtualization Considerations	38
2.6	I/O on Namespaces	38
3	Block Translation Table (BTT)	40
3.1	BTT Data Structures	41
3.1.1	The BTT Info Block	41
3.1.2	The BTT Data Area	44
3.1.3	The BTT Map	44
3.1.4	The BTT Flog	46
3.2	BTT Theory of Operation	47
3.2.1	The Read Path	48
3.2.2	The Write Path	48
3.2.3	BTT Recovery	49
4	Example C Structure Definitions	51
4.1	Namespace Label Index Block Structure	52
4.2	Namespace Label Structure	53
4.3	BTT Info Block Structure	53
4.4	BTT Flog Structure	54
5	Example Algorithms	56
5.1	The Fletcher64 Checksum	56



Figures

Figure 1: Typical NVDIMM Software Architecture	7
Figure 2: Persistent Memory Mapped into the System Physical Address Space .	11
Figure 3: 2-Way Interleave Set of Persistent Memory	12
Figure 4: DPA-Based Access Through a Block Window (BW)	13
Figure 5: 2-Way Interleave Set Containing a Persistent Memory Namespace.....	14
Figure 6: Block Namespaces Associated with Individual DIMMs	15
Figure 7: NVDIMMs Contributing to Both Persistent Memory and Block Namespaces.....	15
Figure 8: Creating a Multi-Range Namespace Due to Fragmentation.....	16
Figure 9: Example Software Organization for Block I/O	17
Figure 10: Example Software Organization for Namespace Management	18
Figure 11: Namespace Label Storage on the NVDIMM	21
Figure 12: Cyclic Sequence Numbers in Label Index Block.....	24
Figure 13: I/O on a Multi-Range Namespace	39
Figure 14: The BTT Data Structures in a Block Namespace	40
Figure 15: A BTT With Multiple Arenas in a Large Block Namespace.....	41
Figure 16: Cyclic Sequence Numbers for Flog Entries	47
Figure 17: BTT Read Path Overview	48
Figure 18: BTT Write Path Overview (Error Cases Not Shown)	49
Figure 19: Namespace Label Index Block Structure	52
Figure 20: Namespace Label Structure.....	53
Figure 21: BTT Info Block Structure	54
Figure 22: BTT Flog Structure	55
Figure 23: The Fletcher64 Algorithm Used in this Specification.....	56

Tables

Table 1: Terminology.....	10
Table 2: Namespace Label Index Block Fields	24
Table 3: Namespace Label Fields.....	28
Table 4: BTT Info Block Fields	43
Table 5: BTT Map Layout	45
Table 6: BTT Flog Layout.....	47



1 Introduction

This document describes a mechanism for sub-dividing Non-Volatile DIMMs (NVDIMMs) into *namespaces*, which are logic units of storage similar to SCSI LUNs or NVM Express namespaces. The primary audience for this document is driver writers and NVDIMM manageability software developers, although NVDIMM designers and platform OEMs may also find this specification useful.

This chapter contains an overview of the motivation for NVDIMM namespaces, covers the related terminology, and provides examples of the software architecture that might utilize this specification. Chapter 2 defines NVDIMM namespaces and provides the on-media data structures and rules for using them. Chapter 3 defines the Block Translation Table (BTT) mechanism, which is an optional layout within a namespace for providing block writes that cannot be torn by a system interruption such as a power failure. Finally, Chapters 4 and 5 contain detailed examples of code and algorithms to help clarify the intention of this specification and enable interoperating implementations.

1.1 Document Scope

This document exists primarily to document the NVDIMM on-media data structures shown in the following tables:

- Table 2: Namespace Label Index Block Fields
- Table 3: Namespace Label Fields
- Table 4: BTT Info Block Fields
- Table 5: BTT Map Layout
- Table 6: BTT Flog Layout

Virtually all other text in this document is here to describe the semantics of those data structures. Software architecture diagrams and algorithms described here are meant to provide one possible implementation; various implementations are possible provided the rules are followed as outlined in chapters 2 and 3.

Fully function code examples for some algorithms described in this specification (the BTT algorithm, for example) are available as open source guides to help with individual implementations. See

<https://github.com/pmem/nvml/blob/master/src/libpmemblk/btt.c>.

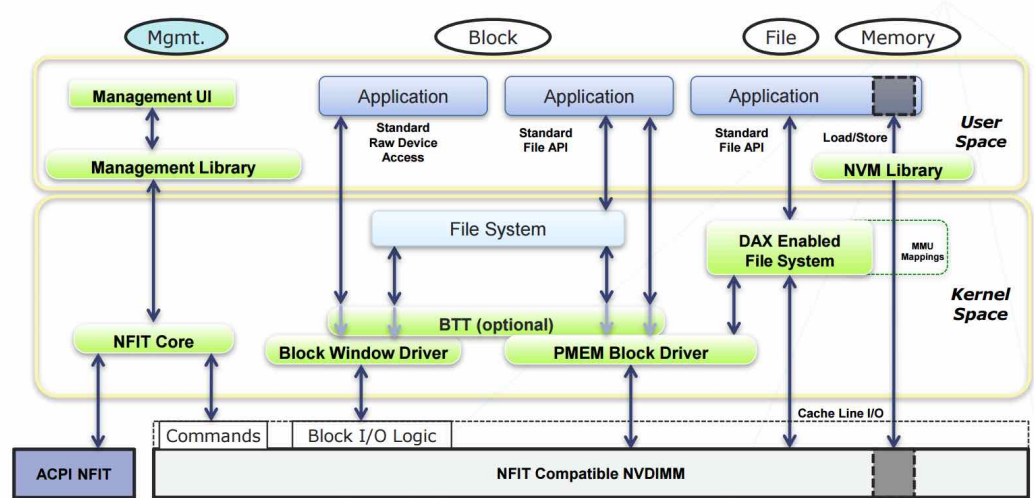


Figure 1: Typical NVDIMM Software Architecture

The software architecture overview in Figure 1 shows one way that the hardware NVDIMM might interact with software components. An NVDIMM NFIT-driver stack sits at the heart of the architecture and is responsible for managing namespaces, including creating, deleting, updating them, and the block I/O path for reading and writing data blocks in a namespace.

1.2 Related Documents

This document depends heavily on the following related documents:

1.2.1 NVDIMM Firmware Interface Table (NFIT)

This ACPI table, defined in ACPI 6.0, enables platform firmware to describe NVDIMMs to an OSPM.

1.2.2 NVDIMM DSM Example Definition

This document describes a hardware interface for NVDIMMs which enables block mode access by providing Block Windows (BWs) which the driver uses to address NVM on a specific DIMM. Although the namespaces defined by this specification could be used with a variety of NVDIMM interfaces, the BW interface is used in many examples for clarity.

1.3 Terminology

This following table provides a brief glossary of terms used by this document.



Term	Description
Block Mode	In the context of this specification, Block Mode refers to block-organized NVM, used as a block storage device like an SSD. The namespaces described in this specification are either Block Mode or Persistent Memory namespaces.
Block Mode Namespace	A block-organized namespace which is associated with a single DIMM. Although the namespace only spans NVM on a single DIMM, it does not necessarily use the entire DIMM; the DIMM may contribute to other namespaces as well.
Block Window (or BW)	The NVDIMM DSM Example Definition describes a hardware interface to NVDIMMs that supports Block Mode. The heart of that interface is the Block Window mechanism, a set of programmable apertures used by driver software for NVM block I/O.
BTT	Block Translation Table: A software mechanism for turning a byte-addressable Persistent Memory range into a block-organized range with powerfail write atomicity when a block is updated.



Term	Description
DIMM-local Namespace	The Block Mode namespaces described in this specification are associated with a single DIMM, and are therefore <i>DIMM-local namespaces</i> . Compare this with an <i>Interleave Set based namespace</i> , which is potentially spread across multiple DIMMs that are interleaved together.
DPA	DIMM Physical Address: A memory address from a DIMM's perspective, that is, the offset into the DIMM's memory, starting with DPA zero as the lowest addressable byte of the DIMM.
Fletcher64	A simple, position-dependent checksum algorithm producing a 64-bit result (see section 5.1 for an algorithmic description).
Interleave Set	A byte-addressable, contiguous range of system physical address space interleaved across one or more DIMMs. The NVDIMM Firmware Interface Table Specification describes an entry in the <i>SPA Table</i> whose interleave is described by the <i>Interleave Description Table</i> . That SPA entry corresponds to an <i>Interleave Set</i> .
Interleave Set based Namespace	The Persistent Memory namespaces described in this specification are associated with an Interleave Set and are therefore <i>Interleave Set based</i> . Compare this with <i>Block Mode namespaces</i> which are DIMM-local.
Label Storage Area	A persistent area of an NVDIMM reserved for namespace Label storage.
Little Endian	Byte order for storing multi-byte values where the least-significant byte is stored in the lowest address in memory. This is the x86 native byte order. All on-media integers described in this specification are stored in little endian byte order.
Memory Interleave	A feature of most memory controllers that interleaves data between multiple DIMMs for performance reasons.
Namespace	A namespace defines a contiguously-addressed range of Non-Volatile Memory similar to a SCSI Logical Unit (LUN) or a NVM Express namespace. Namespaces described by this specification can be either Persistent Memory namespaces or Block Mode namespaces.
NVDIMM	A non-volatile DIMM. This term is generic, describing the non-volatile nature but not the implementation which could be anything from non-volatile memory on the DIMM, to battery-backed up RAM.
NVM	Non-Volatile Memory.



Term	Description
Persistent Memory (or PMEM)	Byte-addressable memory which retains its contents across power loss. A DIMM primarily containing non-volatile memory.
Powerfail Write Atomicity	A feature allowing writes (or stores to memory) of a certain size that cannot be torn by a system interruption like a power failure. After recovery, the area being written will contain either the old value or the new value, but not a mixture of the two.
SPA	System Physical Address: A physical address as accessed by the CPU.
UUID	Universally-Unique Identifier: A 128-bit ID designed to be practically unique. Described in RFC 4122.

Table 1: Terminology

1.4 Overview

The NVDIMMs targeted by this specification provide byte-addressable non-volatile memory mapped into the System Physical Address (SPA) space. Alternatively, they may only provide access via a Block Window or similar mechanism, or they may allow a combination of both of these modes of operation.

1.4.1 NVDIMMs and the System Physical Address Space

As shown in Figure 2, an NVDIMM may appear in the System Physical Address (SPA) space where software can access it as memory using loads and stores. Software typically uses virtual addresses to access memory, and from there the MMU translates those to addresses in the SPA space. From that point the system memory controller determines which DIMM is being addressed and translates the access to an offset into the DIMM's Physical Address (DPA) space.

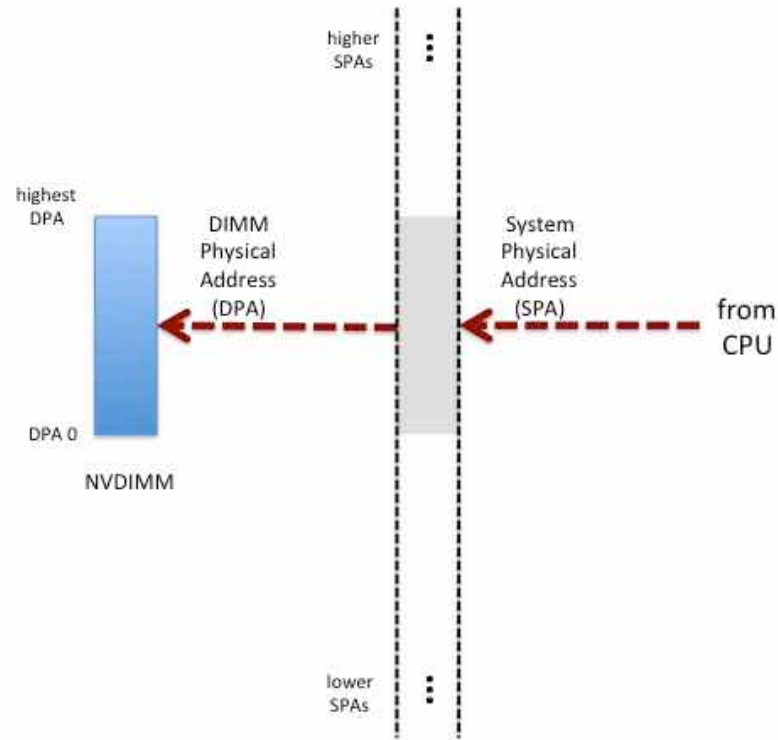


Figure 2: Persistent Memory Mapped into the System Physical Address Space

The on-DIMM data structures described in this specification use either relative offsets or DPAs when referring to other on-DIMM data structures, as the exact location of a DIMM in the SPA may vary from boot to boot due to other platform configuration changes.

The simple 1-to-1 mapping of a SPA range to a DPA range shown in Figure 2 is often not possible due to cross-DIMM interleaving done by the memory controller.

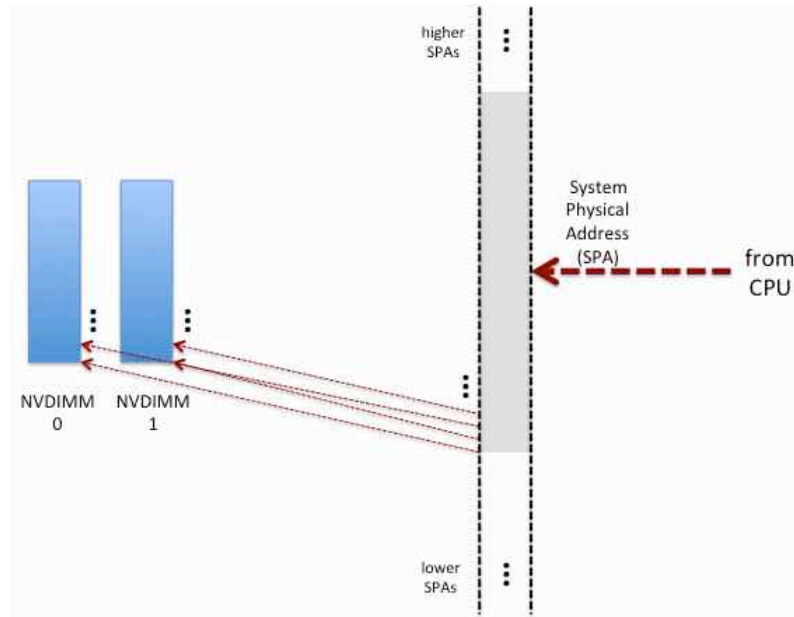


Figure 3: 2-Way Interleave Set of Persistent Memory

Figure 3 shows a simple example of interleaving where two NVDIMMs are combined to make a contiguous range of SPA space. The dashed red arrows in the picture illustrate how accesses from the CPU are sent to the NVDIMMs alternatively based on their address. In this document, the term *Interleave Set* is used to refer to a contiguous range of SPA providing byte-addressable access to NVDIMMs. The interleave set can be 1-way, 2-way, etc., and can include fairly complex interleave math and attributes such as memory controller based mirroring between interleave sets. The primary feature of an interleave set as far as this document is concerned is the fact that every byte of the interleave set is usable by software as *Persistent Memory*.

1.4.2 NVDIMM Access via Block Window

Sometimes it is preferable to access the storage on an NVDIMM without interleaving between multiple DIMMs. This may be because software is building a RAID-style array from multiple NVDIMMs and wants to maintain the RAS boundaries of the DIMMs similarly to the RAS boundaries of multiple disks in a RAID array. Or this may be because the NVDIMM is not capable of the Persistent Memory style access described in the previous section.

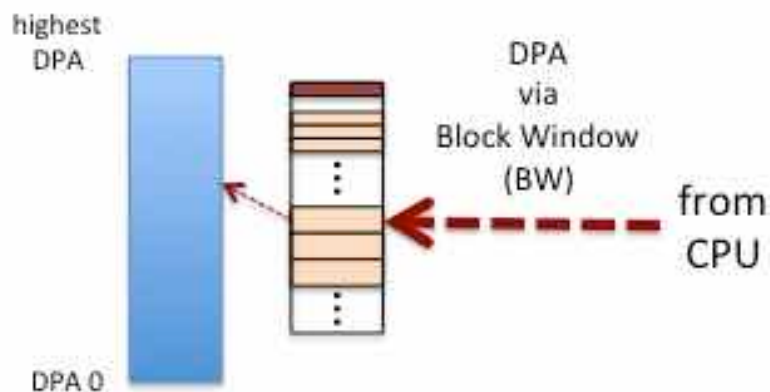


Figure 4: DPA-Based Access Through a Block Window (BW)

In Figure 4, one way that accesses are isolated to a single NVDIMM is shown. NVDIMMs which comply with the *NVDIMM DSM Example Definition* include Block Windows (BW), which are apertures through which software can read and write block-sized chunks of NVM. The BWs include a control register for programming the target location, and a status register for error detection. The main point of the BW, however, is to send I/O specifically to a single DIMM.

1.4.3 Namespaces

Just as a large SAN storage array can be subdivided into some set of SCSI LUNs, and an NVM Express PCIe SSD can be carved into namespaces, *NVDIMM namespaces* are very much the same idea. This is not to be confused with other subdivision mechanisms like disk partitions or virtual volumes provided by many SW RAID stacks – those things still exist on top of NVDIMM namespaces just as they do on top of any direct-attached storage. Note that unlike things like disk partitions, namespaces may have attributes unavailable through other means, like different block sizes for block devices, the choice of powerfail write atomicity, and the ability to be accessed as Persistent Memory.

In this specification, there are two main types of namespaces: a *Persistent Memory namespace* and a *Block Mode namespace*.

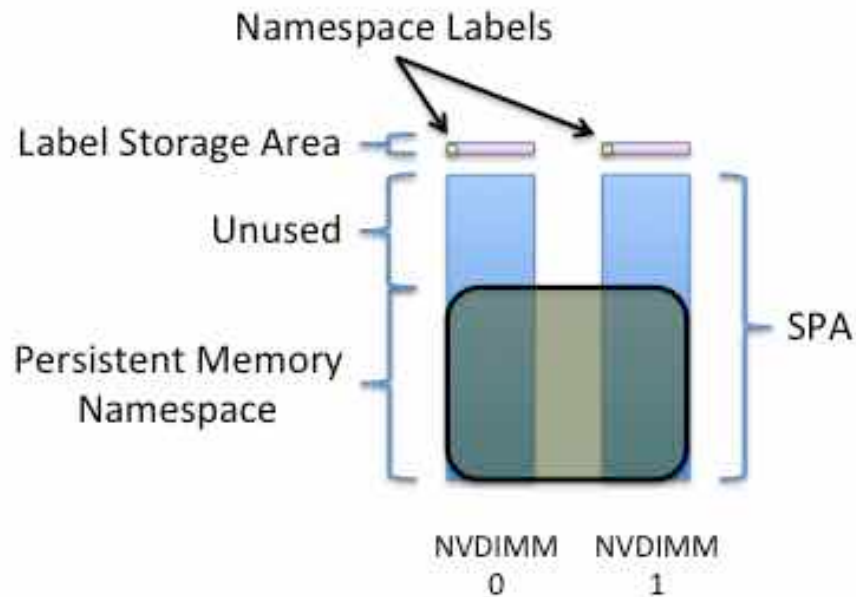


Figure 5: 2-Way Interleave Set Containing a Persistent Memory Namespace

A Persistent Memory namespace is associated with an Interleave Set, since the primary reason for a Persistent Memory namespace is to be addressed as memory in the SPA. Figure 5 shows a typical Persistent Memory namespace, along with some other relevant details. The example in the figure starts with a 2-way interleave set where two NVDIMMs are completely mapped into the SPA as interleaved space (note the SPA bracket on the right). The example has a single Persistent Memory namespace created on the Interleave Set. In fact, an Interleave Set is allowed to contain at most one namespace, but the namespace size may be smaller than the Interleave Set size, as shown in the figure, allowing some of the space to remain unused (the bracket on the left). Finally, Figure 5 also depicts the labels that define the namespace, stored in a *Label Storage Area* shown at the top of the figure. The exact location of the label storage area is NVDIMM-specific, but NVDIMMs following the *NVDIMM DSM Example Definition* access the label storage area using a firmware device-specific-method.

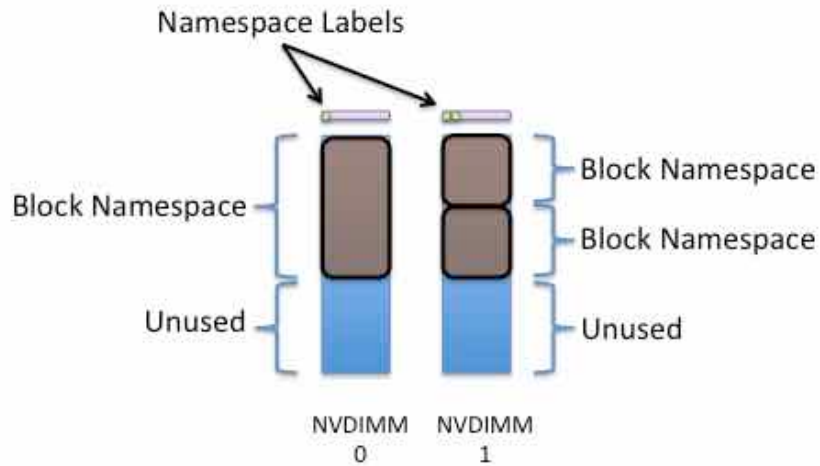


Figure 6: Block Namespaces Associated with Individual DIMMs

A Block Mode namespace is associated with a specific DIMM (not an Interleave Set like Persistent Memory namespaces). Just as the previous figure, Figure 6 is showing that namespaces need not use the entire capacity available to them. Unlike a Persistent Memory namespace, which can appear at most once per Interleave Set, multiple Block Mode namespaces are allowed on each NVDIMM. Additional Block Mode namespaces can be created until the NVDIMM space is exhausted, or the label storage area space is exhausted.

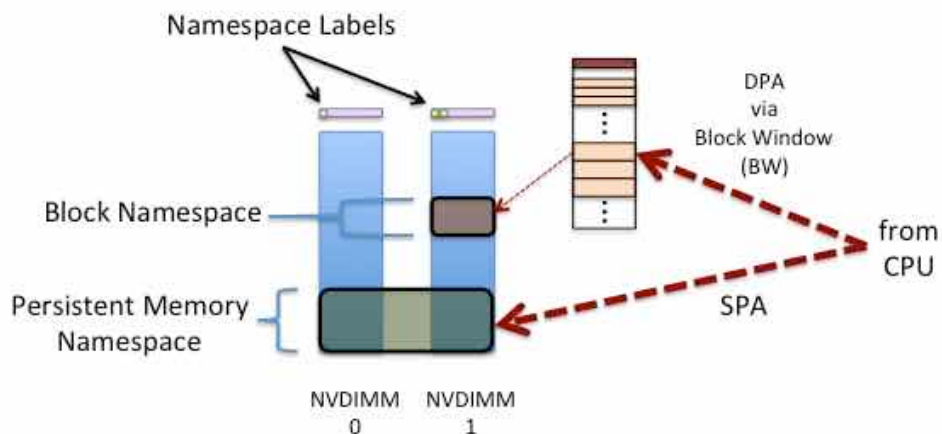


Figure 7: NVDIMMs Contributing to Both Persistent Memory and Block Namespaces

Figure 7 puts the two types of namespaces together, showing how they co-exist. Notice how the space used for the Persistent Memory namespace always starts at the lowest DPA, potentially leaving the higher DPA space for use with Block Mode namespaces. In the example, the Persistent Memory namespace only uses a fraction of the interleave set, and each NVDIMM contains a label in its label storage area describing that NVDIMM's contribution to the Persistent



Memory namespace (the labels are depicted as little yellow boxes in the label storage area). Additionally, the figure shows some of the free space on NVDIMM1 was used for a Block Mode namespace, requiring that NVDIMM to have another label in its label storage area to describe that namespace.

It is up to software to determine exactly where the namespaces are placed, but some rules must be followed. Those rules are described in chapter 2, and include the requirement that namespaces cannot overlap, and that the Persistent Memory namespace, if it exists, must end at the beginning (lowest address in the SPA) of an Interleave Set.

The Block Mode namespace on NVDIMM 1 shown on the right side of Figure 7 consists of a single range of DPA. In order to overcome the fragmentation that naturally occurs from creating and deleting namespaces over time, Block Mode namespaces are allowed to consist of multiple ranges of DPA.

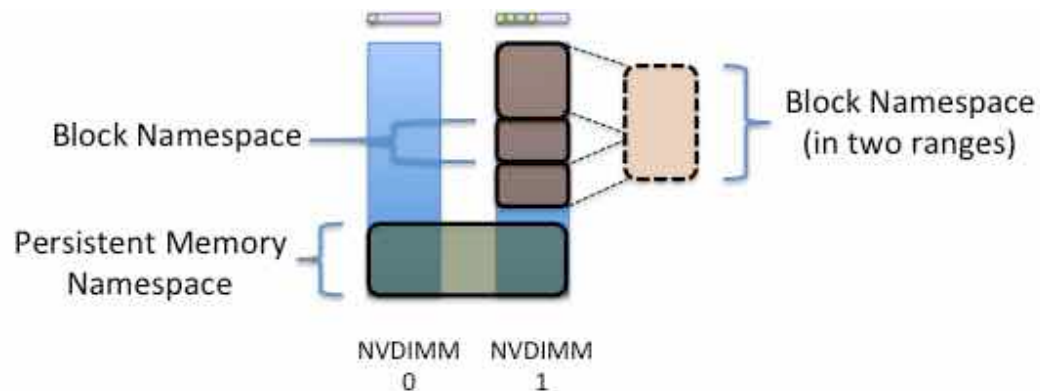


Figure 8: Creating a Multi-Range Namespace Due to Fragmentation

As an example, imagine a Block Mode namespace of 4GB is created, followed by the creation of a 2GB Block Mode. At some later point the original 4GB namespace is deleted and the admin wishes to create a 6GB namespace. In our example, 6GB of storage is available, but not contiguously due to the second 2GB namespace stuck in the middle. This is solved, as shown in Figure 8, by creating a Block Mode namespace made up of two ranges (shown on the right). Each range takes a label in the label storage area (chapter 2 describes how the labels are used together to describe the namespace). The resulting labels in Figure 8 are one on NVDIMM 0 (to describe that DIMM's contribution to the Persistent Memory namespace), and four on NVDIMM 2 (one for that DIMM's contribution to the Persistent Memory namespace, one for the 2GB namespace indicated by the bracket on the left, and two for the two-range namespace indicated by the bracket on the right).

A key point of Figure 8 is to show that, although the Block Mode namespace on the right is made up of two DPA ranges, it still implements a contiguous address range of logical blocks. This is typically implemented by the driver examining the logical block address (LBA) of an I/O request, matching it against the ranges that name up a namespace, and issuing the I/O to the DPA+offset appropriate to the range being accessed.

1.4.4 Driver Software

There are many ways an NVDIMM driver can be organized, one such organization might be to arrange the drive routines as shown in Figure 9.

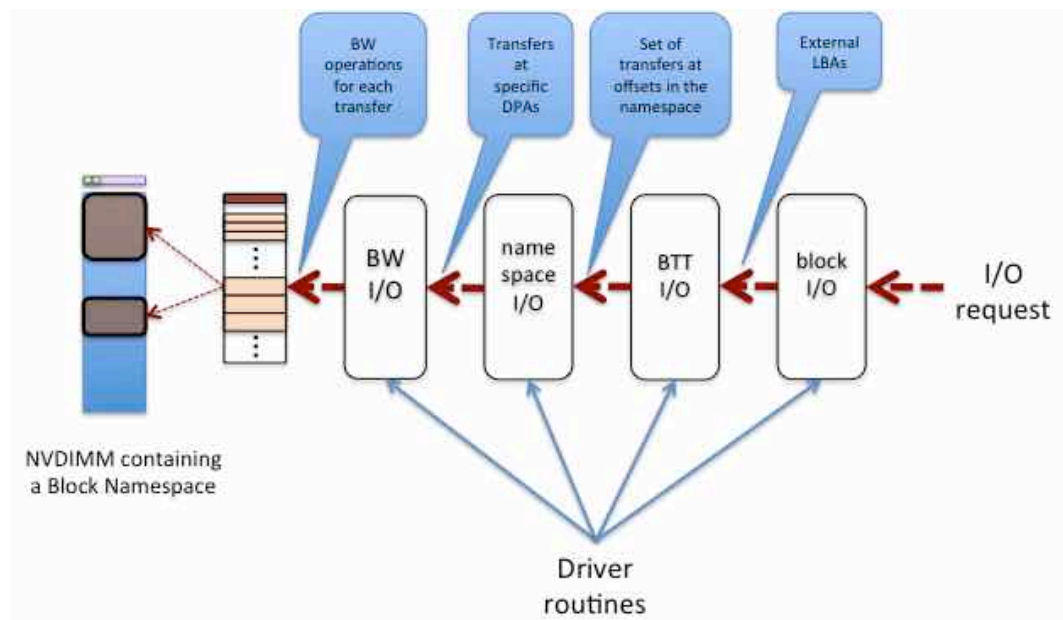


Figure 9: Example Software Organization for Block I/O

Walking through the above example driver organization from right to left, the driver starts by taking an I/O request from the software in the stack above it. That request is broken up into individual blocks, each associated with an LBA. The term *External LBA* is used here to indicate the LBA is in the range 0 to the highest LBA advertised to components outside the driver – the external LBA range does not include extra blocks that may be allocated by the driver to support write atomics, for example.

Requests based on external LBAs are submitted to the BTT I/O routine, which implements the translation table based write atomicity described in chapter 3.



The BTT algorithm will take the request and turn it into multiple requests to specific offsets in the namespace. For example, a read request will be turned into a small read from the BTT map data structure followed by a full block read from the location indicated by the map (much more detail on this in chapter 3).

Continuing to the left in Figure 9, the namespace I/O routine will take the namespace offset based requests and convert them to DPA based requests using the ranges associated with the namespace (from the namespace labels).

Finally, the example uses Block Windows so the BW I/O routine takes the DPA based requests and programs the BWs to perform the I/O, as described in the *NVDIMM DSM Example Definition*.

Just as it is useful to describe a possible implementation of the driver data path, it is also useful to describe a possible driver organization for managing namespaces.

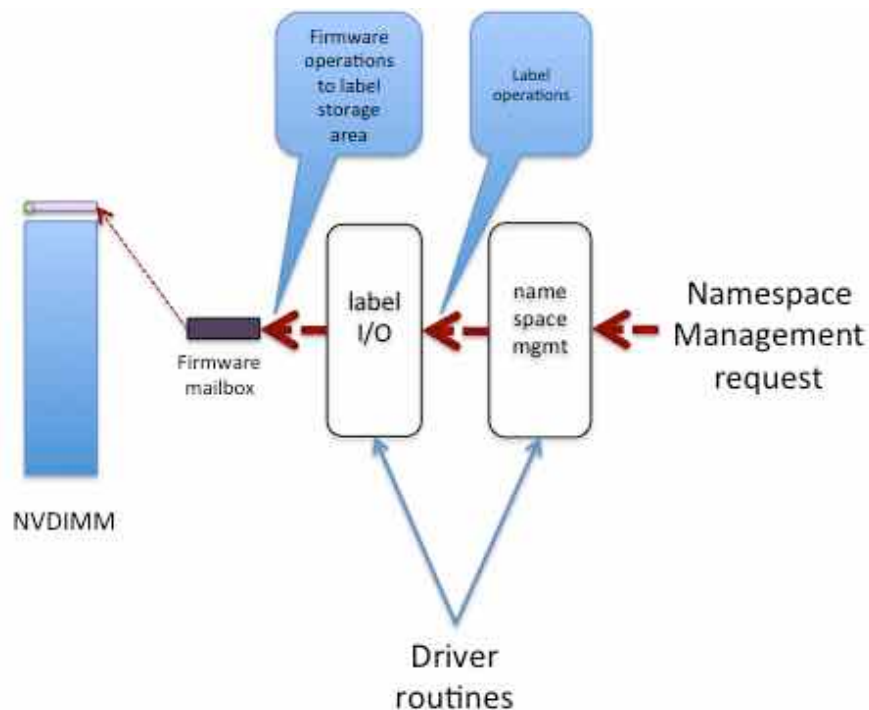


Figure 10: Example Software Organization for Namespace Management

Figure 10 shows how namespace management in the driver might be divided into routines that deal with the namespace rules described in this spec, and a label I/O routine and understands how to access the label storage area. The



figure shows the label storage area described by the *NVDIMM DSM Example Definition*, where firmware device-specific-methods are used to access data structures in the label storage area.



2 Namespaces

A *namespace* defines a contiguously-addressed range of Non-Volatile Memory similar to a SCSI Logical Unit (LUN) or an NVM Express namespace. This specification provides two categories of namespace:

- **Persistent Memory namespace**
This type of namespace is associated with an **Interleave Set**, where a combination of one or more DIMMs, interleaved together by the system memory controller, provides a byte-addressable range of Persistent Memory in the system physical address space. These persistent memory namespaces are typically exposed via a *Persistent Memory Aware File System* as shown in the Persistent Memory stack in Figure 1.
- **Block Mode namespace**
This type of namespace is associated with a **single DIMM**, where a range of NVM on that DIMM is organized into logical blocks. These DIMM-local namespaces are typically exposed via the system's block storage interfaces as shown in the Block stack in Figure 1.

Namespaces are defined by Namespace Labels which are stored in a *Label Storage Area* on each DIMM. NVDIMMs providing an isolated Label Storage Area as described in the *NVDIMM DSM Example Definition* access the labels using Firmware Device-specific-methods. However, any NVDIMM could use the label format specified in this document by storing the labels in a well-known location.

Figure 11 shows the organization of the Label Storage Area. A header called the *Namespace Label Index Block* appears twice at the top of the Label Storage Area. This provides a powerfail-safe method for updating the information in the storage area by alternating between the two index blocks when writing (more details on this mechanism below).

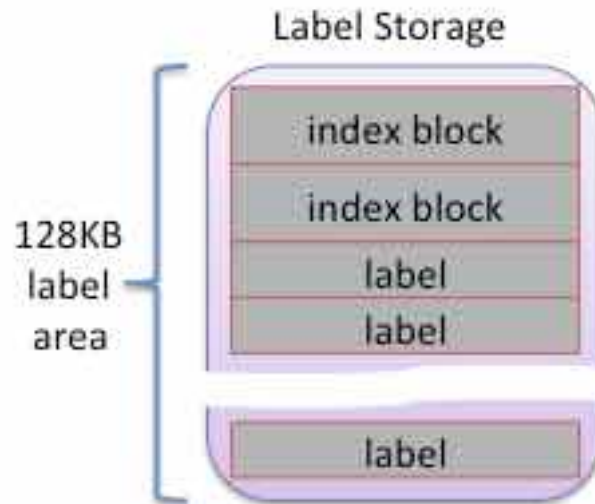


Figure 11: Namespace Label Storage on the NVDIMM

Following the index blocks, an array for storing labels takes up the remainder of the label storage area. NVDIMM vendors define the size of their label storage area and, therefore, the number of labels it holds. NVDIMMs following the *NVDIMM Block Mode Specification* use an area at least 128KB in size, which holds around 1000 labels. The index blocks contain a bitmap which indicates which label slots are currently free and which are in use. The same powerfail-safe mechanism used for updating the index blocks covers the update of labels in the label area.

The powerfail-safe update mechanism depends on a principle used several times in this specification where writes to *active* metadata are avoided. Instead, a shadow copy is updated and checksums and sequence numbers are used to make the last written copy active (a complete description of this mechanism is in section 2.3).

Similarly, the labels themselves are never updated in-place. Instead, a free label slot is first updated, followed by an update to the index block to mark the old label slot free and the new label slot in use. At least one slot in the storage area must be free, ensuring it is always possible to update labels using this method.

2.1 Namespace Label Index Block Layout

This section describes the layout of the Namespace Label Index Block. Like all on-media structures defined in this specification, all multi-byte integer fields in the index block are stored in little endian byte order.



The size of an index block depends on how many label slots fit into the label storage area. The minimum size of an index block is 256 bytes and the size must be a multiple of 256 bytes. As necessary, padding with zero bytes at the end of the structure is used to meet these size requirements. For a storage area of 128KB, as described in the *NVDIMM DSM Example Definition*, the corresponding index block size is 256 bytes:

Size of the Label Index Block field up to the free field, as described in Table 2	72 bytes
Bytes required for a bitmask of 1024 labels (the number of 128-byte labels that fit into a 128KB label storage area)	128 bytes
Padding to meet minimum size of 256 bytes	56 bytes
Total size of Label Index Block on NVDIMMs following the <i>NVDIMM DSM Example Definition</i>	256 bytes

The following table describes the layout in a Namespace Label Index Block. See section 4.1 for an example C structure definition of the Namespace Label Index Block.

Field	Byte Length	Byte Offset	Description
sig	16	0x0000	Must be "NAMESPACE_INDEX\0".
flags	4	0x0010	Boolean attributes of this label storage area. There are no flag bits defined at this time, so this field must be zero.



Field	Byte Length	Byte Offset	Description
seq	4	0x0014	Sequence number used to identify which of the two label index blocks is current. Only the least-significant two bits of this field are used in the current definition, rotating through the values depicted in Figure 12. The other bits must be zero.
myoff	8	0x0018	The offset of this Label Index Block in the label storage area.
mysize	8	0x0020	The size of this Label Index Block in bytes. This field must be a multiple of 256 bytes.
otheroff	8	0x0028	The offset of the other Label Index Block paired with this one (stored adjacent to this one in the label storage area).
labeloff	8	0x0030	The offset of the first slot where labels are stored in this label storage area.
nlabel	4	0x0038	The total number of slots for storing labels in this label storage area.
major	2	0x003c	Major version number. Currently at version 1. Software must support this version exactly, or decline to support the DIMM contents.
minor	2	0x003e	Minor version number. Currently at version 1. Software may use this to check for backward-compatible features in the label.
checksum	8	0x0040	64-bit <i>Fletcher64</i> checksum of all fields in this Label Index Block (see section 5.1 for details on the Fletcher64 checksum). This field is considered zero when the checksum is computed.



Field	Byte Length	Byte Offset	Description
free	Bytes required to hold nlabel bits + padding	0x0048	Array of unsigned bytes implementing a bitmask that tracks which label slots are free. The size of this field is the number of bytes required to hold the bitmask with nlabel bits, padded with addition zero bytes to make the Label Index Block size a multiple of 256.

Table 2: Namespace Label Index Block Fields

Most of the fields are sufficiently defined by the above table, but some fields require a more detailed explanation:

2.1.1 The Label Index Block **seq** Field

The sequence number held by the **seq** field is two bits in size (the remaining bits in the **seq** field must be zero). Each time an index block is written, the sequence number of the current index block is “incremented” by moving to the next value clockwise as shown in Figure 12.

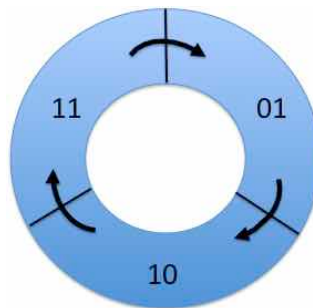


Figure 12: Cyclic Sequence Numbers in Label Index Block

Since there are two index blocks, written alternatively with successive sequence numbers, the older index block’s sequence number will be immediately behind (counter-clockwise to) the current index block’s sequence number. This property is used during software initialization to identify the current index block.

The sequence number 00 is used to indicate an uninitialized or invalid index block. Software never writes the sequence number 00, so a correctly checksummed index block with this sequence number probably indicates a software bug. When software discovers this case it treats it as an invalid index block indication.



Two index blocks with matching sequence numbers is also unexpected and likely indicates a software bug. However, for deterministic behavior this specification defines how matching sequence numbers are handled (section 2.3 explains the index block at the higher offset in the label storage area is considered the valid block in this case).

2.1.2 The Label Index Block **free** Field

The **free** bitmask is organized in the usual way where the label slot with the lowest offset in the label storage area is tracked by the least significant bit of the first byte of the free array. Missing from the above layout is a total count of free slots. Since the common use case for the label storage area is to read all labels during software initialization, it is recommended that software create a total free count (or in use count, or both), maintained at run-time. Rules for maintaining the on-media index blocks are described in section 2.3 below.

2.2 Namespace Label Layout

Each slot in the label storage area is either free or contains an active namespace label. Single namespace may be described by a single label, but often it takes multiple labels to fully describe a namespace. This happens for two reasons:

- **Persistent Memory namespace**

This type of namespace is associated with an **Interleave Set**. For interleave sets that involve more than a single DIMM, each DIMM involved will contain a namespace label describing that DIMM's contribution to the namespace.

- **Block Mode namespace**

This type of namespace is **DIMM-local**, associated with a single DIMM and not interleaved across DIMMs. Due to potential fragmentation of the NVM on a DIMM, block mode namespaces may be described as a list of ranges on that DIMM. In this case, each range will be stored as a label in the label storage area on that DIMM.

In the cases where multiple labels are used to describe a namespace, the label fields **nlabel** and **position** provide an ordering ("label one of two, label two of two") so that incomplete label sets can be detected.

The following table describes the layout of the Namespace Label. See section 4.2 for an example C structure definition of the Namespace Label.



Field	Byte Length	Byte Offset	Description								
uuid	16	0x0000	UUID per RFC 4122								
name	64	0x0010	Optional name, NULL-terminated.								
flags	4	0x0050	Boolean attributes of this namespace.								
			<table><tr><th>Bit</th><th>Meaning</th></tr><tr><td>0x00000001</td><td>ROLABEL: label is read-only. This indicates the namespace is exported to a domain where configuration changes to the label are not allowed, such as a virtual machine.</td></tr><tr><td>0x00000002</td><td>LOCAL: namespace is local to this DIMM (DIMM-based namespaces are not spread across DIMMs; Interleave Set based namespaces will have this bit clear).</td></tr><tr><td>0x00000004</td><td>UPDATING: label set is being updated.</td></tr></table>	Bit	Meaning	0x00000001	ROLABEL: label is read-only. This indicates the namespace is exported to a domain where configuration changes to the label are not allowed, such as a virtual machine.	0x00000002	LOCAL: namespace is local to this DIMM (DIMM-based namespaces are not spread across DIMMs; Interleave Set based namespaces will have this bit clear).	0x00000004	UPDATING: label set is being updated.
			Bit	Meaning							
			0x00000001	ROLABEL: label is read-only. This indicates the namespace is exported to a domain where configuration changes to the label are not allowed, such as a virtual machine.							
0x00000002	LOCAL: namespace is local to this DIMM (DIMM-based namespaces are not spread across DIMMs; Interleave Set based namespaces will have this bit clear).										
0x00000004	UPDATING: label set is being updated.										
nlabel	2	0x0054	Total number of labels describing this namespace. For Interleave Set based								



Field	Byte Length	Byte Offset	Description
			namespaces, this number represents the number of DIMMs involved since each DIMM will have a label describing that DIMM's contribution to this namespace. For DIMM-local namespaces, this field is zero.
position	2	0x0056	Position of this label in list of labels for this namespace, such that: $0 \leq position < nlabel$ For DIMM-local namespaces, this field is zero.
isetcookie	8	0x0058	For an Interleave Set based namespace, this cookie identifies the Interleave Set. The label is considered invalid if the actual Interleave Set cookie doesn't match the cookie stored here. For DIMM-local namespaces, this field is zero.
lbasize	8	0x0060	Zero for a <i>Persistent Memory</i> namespace, a non-zero LBA size in bytes for a block-structured namespace.
dpa	8	0x0068	The <i>DPA</i> where the NVM contributing to this namespace begins on this DIMM.
rawsize	8	0x0070	The extent of the DPA contributed by this label.
slot	4	0x0078	Current slot in the label storage area where this label is stored.
unused	4	0x007c	Must be zero.

**Table 3: Namespace Label Fields**

The existence of a label in the label storage area is not enough to consider the label valid. The rules for managing the label metadata must be applied, as described in section 2.3. Most of the fields are sufficiently defined by the above table, but some fields require a more detailed explanation:

2.2.1 The Label **uuid** Field

This field provides two functions. First, the namespace is associated with a UUID that drivers can use to uniquely identify it, providing a way for it to be matched up with applications using it, etc. Second, when multiple labels are required to describe a namespace (either multiple labels on a single DIMM because a Block Mode namespace consists of multiple ranges, or multiple labels spread across DIMMs because a Persistent Memory namespace is on a multi-DIMM Interleave Set), the UUID is the mechanism used to group the labels together. Section 2.3.4 describes the process for grouping the labels together by UUID, checking for missing labels, recovering from partial label changes, etc.

2.2.2 The Label **name** Field

The **name** field is optionally used by manageability software to store a more friendly name for the namespace. When this field is unused, it contains zeros. For a Block Mode namespace, it is only necessary to store the name in the first label of the range set. All subsequent **name** fields for that Block Mode namespace are ignored and are expected to be zeros. But for Persistent Memory namespaces, storing the name in every label for the namespace allows for better error messages when exposing incomplete namespaces (if the name were only stored in the first label, and that label is missing, there's no way to display the name in error messages).

The **name** field can be set at label creation time, or updated by following the rules for updating labels. When updating the name of a Block Mode namespace, a single, atomic update of the first label is used. When updating the name of a Persistent Memory namespace, the two-phase approach using the **UPDATING** flag is used to update all labels atomically, as described in section 2.3.7.

2.2.3 The Label **flags** Field

There are several varieties of namespaces, determined by the bits set in the **flags** field:

- **Persistent Memory namespace**



These namespaces have the **LOCAL** bit clear (since Persistent Memory namespaces are associated with Interleave Sets, not the local DIMM). The **lbasize** field in the label is unused in this case and should contain zero.

- **Block Mode namespace**

These namespaces have the **LOCAL** bit set (since Block Mode namespaces are associated with the local DIMM). The **lbasize** field in the label contains the logical block size for the namespace – all I/O must be done in multiples of this size.

- **Block Mode namespace with Powerfail Write Atomicity**

These Block Mode namespaces use a BTT (Block Translation Table) to provide single-block powerfail write atomicity. That is, a write of a block cannot be torn by system interruption such as a power failure. This feature is indicated by the presence of a BTT info block located at an offset of 4K bytes from the start of the namespace/GPT-partition.

- **Persistent Memory namespace being Created or Updated**

During a cross-DIMM operation, such as creating a new namespace, or updating the **name** field for a Persistent Memory namespace, the **UPDATING** flag is used to make the update atomic across interruptions. Updates happen in two phases, first writing the label with the UPDATING flag set, second writing the updated label without the UPDATING flag. As described in section 2.3.7, this allows recovery actions during software initialization to either *roll back* or *roll forward* the cross-DIMM change.

- **A Read-Only Namespace Label**

The **ROLABEL** field indicates that device drivers and manageability software should refuse to make changes to the namespace labels. This is not a security mechanism, but a usability feature instead. In cases where **ROLABEL** is set, such as virtual machine guests, attempting to make configuration changes that affect the namespace labels will fail (i.e. because the VM guest is not in a position to make the change correctly). For these cases, the VMM can set the **ROLABEL** bit on the label exposed to the guest to provide a better user experience where manageability refuses to make changes with a friendlier error message.

2.2.4 The Label **nlabel** and **position** Fields

The **nlabel** field contains the number of labels required to describe an interleave-setnamespace. Each label is numbered as to its position in the list of labels using the **position** field. For example, the common case where a Persistent Memory Mode namespace requires exactly one label, **nlabel** will be 1 and **position** will be 0. If a Persistent Memory namespace is built on an Interleave Set that spans 4 DIMMs, each DIMM will contain a label with



increasing **position** values to show the labels position in the set. For Block Mode namespaces 'nlabel' and 'position' must be zero.

2.2.5 The Label **isetcookie** Field

When a Persistent Memory namespace is defined, it is associated with an Interleave Set. The **isetcookie** field in each label for that namespace is set to a checksum the associated entry in the NFIT Interleave Description Table. This value is used to detect a change in the Interleave Set configuration, rendering the label invalid. The cookie must be robust in the case of DIMMs moving physical location. Platform firmware may or may not be able to re-establish an existing interleave set if DIMMs move location. The following algorithm is used to calculate the **isetcookie** field.

For each interleave set create a data structure of the form:

```
struct interleave_set_info {
    struct interleave_set_info_map {
        uint64_t region_spa_offset;
        uint32_t serial_number;
        uint32_t zero;
    } mapping[INTERLEAVE_WAYS];
};
```

- **INTERLEAVE_WAYS** is the number of memory devices (DIMMs) in the interleave set as specified by the number of *Memory Device to System Physical Address Range Mapping Structure* entries that reference the *System Physical Address Range Structure* that defines the interleave set.
- **region_spa_offset** is the *Region Offset* field of the *Memory Device to System Physical Address Range Mapping Structure* for a given DIMM. This determines the DIMM's position in the interleave set.
- **serial_number** is the *Serial Number* field from the *NVDIMM Control Region Structure* associated with the *Memory Device to System Physical Address Range Mapping Structure* for the given DIMM.
- **zero** is zero-filled padding.

The **isetcookie** is then calculated by sorting the **mapping[]** array by **region_spa_offset** and then taking the Fletcher64 sum of the total **interleave_set_info** structure.



For Block Mode namespaces, the **isetcookie** field must be zero.

2.2.6 The Label **lbase** Field

Block Mode namespaces have a block size associated with them and this is stored in the **lbase** field. All I/O to the namespace is addressed in multiples of this size.

For Persistent Memory namespaces, the **lbase** field must be zero.

2.3 Namespace Label Rules

In this section, the rules for managing namespace label information are described. Reading and writing the label storage area is a device-dependent operation. For NVDIMMs following the NVDIMM DSM Example Definition, this involves invoking platform firmware device-specific-methods to read or write the label space.

There are a variety of ways software can choose to manage the label metadata. A set of example algorithms are described in chapter 5. Unlike the BTT data structures described in chapter 3, the namespace labels are not designed for high-performance, parallel access. All the algorithms related to labels in this specification assume single-threaded execution, or some sort of *label metadata lock* for drivers proving label management.

Software must maintain certain invariants to use the on-media data structures correctly and to inter-operate with other software components. This section describes the rules that must be followed for the on-media data structures in the label storage area.

At all times, the following must be true:

- The size of the label storage area is known (this must be true even if no namespace metadata has been written yet).
- The label storage area either contains no valid Label Index Blocks, indicating there are no labels on the DIMM (all slots free), or the validation rules below produce a single, valid, Label Index Block.
- The number of free label slots is at least 1
- Only fully written, active labels, and full-written labels with the **UPDATING** flag are marked in-use by the label index block
- Write to in-use label slots are not allowed; all updated to labels must be done by writing to free slots and then updating the label index block to make them active



2.3.1 Validating Label Index Blocks and Labels

The existence of a valid Namespace Label Index Block depends on the following tests passing (typically done during driver initialization):

1. Based on the size of the label storage area, the size of the Label Index Block is calculated and both index blocks must be read successfully from the label storage area.
2. Any index block with an incorrect signature field is discarded
3. Any index block with an incorrect checksum is discarded
4. Any index block with an incorrect myoff, mysize, or otheroff field is discarded
5. Any index block with a sequence number of zero is discarded
6. If two index blocks remain, after passing all the above tests, and their sequence numbers match, the index block at the lower offset in the label storage area is discarded
7. If two index blocks remain, after passing all the above tests, their sequence numbers are compared and the block whose sequence number is the predecessor of the other (immediately counter-clockwise to it, as shown in Figure 16) is discarded.
8. If one index block remains, that is the current, valid block and software should make note that the next update to the index will write the other block. However, if no valid index blocks remain, all slots are considered free and the next update to the index will write to the lower-addressed block location (i.e. the start of the label storage area). Note that there's no reason to write a valid index block until the first update to a label takes place, although software is free to write an initial block no valid block is found.

The existence of valid namespace labels on a DIMM depends on the slot being marked in-use by a valid label index block and, in the case of a Persistent Memory namespace, on the **isetcookie** field in the label matching the current NFIT information.

2.3.2 Reading Namespace Labels

Namespace labels are typically scanned, in their entirety, during software initialization, for example when a driver is making a list of valid namespaces to surface as NVDIMM devices. Once a list of labels has been created, the steps in the following sections are followed for recovery of label state after a system interruption and to assemble labels into valid namespaces. Since Persistent Memory namespaces are associated with Interleave Sets and not individual DIMMs, software should read all labels associated with an Interleave Set before performing the next steps on them. An even simpler algorithm (and the one



described in section **Error! Reference source not found.**) simply makes a list of all labels from all NVDIMMs on the system and then performs the recovery and namespace assembly steps on the entire list.

For a given DIMM, the following steps are used to read all labels once the Label Index Block validation steps in the previous section are complete:

0. Pre-condition: both Label Index Blocks have been read and the rules in section 2.3.1 have been followed to determine the current index state.
1. Step through the **free** bitmask field in the index, starting with bit 0 and ending with bit **nlabel - 1**
 - a. Read the label in that slot using the label I/O method for the NVDIMM. For NVDIMMs following the NVDIMM DSM Example Definition, this means issuing a firmware mailbox command to access the label-sized chunk of data at the offset given by $(2 * \text{index_block_size} * \text{slot} * \text{label_size})$.

After reading all the labels during software initialization, the next steps are typically to perform the recovery steps described in the next section, and then assemble the labels into complete sets, as described in the section after next.

2.3.3 Recovery Steps on Namespace Labels

After creating a list of labels, as described in the previous section, a driver must perform recovery steps to return the labels to a stable state. This is to recover from an unexpected system interruption while labels are being written. Since the Label Index Blocks are written by alternating to the unused copy of the index, and checksum indicate completed writes, no recovery is necessary for that data structure or for label updates made on a single DIMM. The only recovery required for the label storage area is due to the interruption of a multi-DIMM update to a set of labels.

Section 2.3.7 below describes how the **UPDATING** flag in a label is used to indicate a multi-DIMM label operation. For recovery, the following steps must be applied to the set of labels:

0. Pre-condition: The set of labels have been read as described in the previous section.
1. For each set of labels with the same UUID, if no labels in the set are found with the **UPDATING** flag set, then no recovery is required for that set



2. For the sets where **UPDATING** appears at least once, if the set is incomplete (some DIMMs in the Interleave Set do not contain a label with the UUID), the recovery action is to roll back the interrupted create operation that left this state:
 - a. For each DIMM in the Interleave Set containing a label with the UUID:
 - i. Delete the label
3. For set where **UPDATING** appears at least once and the set is otherwise complete (each DIMM in the Interleave Set contains a label with the UUID, some with **UPDATING** set, some with **UPDATING** clear), the recovery action is to roll forward the change that was interrupted:
 - a. For each DIMM in the Interleave Set:
 - i. If **UPDATING** is set, write an updated label with **UPDATING** clear and with the name field copied from the first label in the set (the label with a **position** field of 0).

2.3.4 Assembling Namespace Labels into Complete Sets

After creating a list of labels and performing the recovery actions on the list, as described in the previous two sections, a driver must follow the steps in this section to assemble complete sets of labels representing usable namespaces:

0. Precondition: Labels have been read and the recovery actions have been taken.
1. For each set of labels with the same UUID
 - a. If the set is complete (**position** fields are found for every position from 0 to **nlabel - 1**), the namespace is complete and may be surfaced by the driver as a usable namespace (if the driver supports that type of namespace)

2.3.4.1 Incomplete Namespaces

A driver may find incomplete namespaces in the case where an Interleave Set is incomplete. In this case, the BIOS will prevent the Interleave Set from appearing in the SPA, but the individual DIMMs will still exist along with their labels for the incomplete Persistent Memory namespace. In this case, it is recommended that the driver expose the incomplete namespace somehow, even if only for manageability software, so that users see evidence of the missing DIMMs). No I/O should be allowed on an incomplete namespace. Manageability software should allow incomplete namespaces to be deleted (i.e. for the case where the admin knows the missing DIMM will never return).



2.3.5 Updating the Contents of the Label Storage Area

More specific cases are covered below, but in general **adding a label** to a DIMM's label storage area requires the driver to follow these steps:

0. Pre-conditions: the driver has a new label constructed to be written to a specific DIMM's label storage area. There are at least 2 free slots in the label storage area so that, after adding the label, at least 1 free slot remains.
1. The driver chooses a free slot from the Label Index Block, fills in that slot number in the label's **slot** field
2. The driver writes the new label to that slot in the label storage area
3. The driver updates the Label Index Block by taking the current index block, clearing the appropriate bit in the **free** field, incrementing the sequence number as shown in Figure 12, and then writing the index block over the inactive index block location (making this location the new active index block if the write succeeds)

Similarly, **updating an existing label** in the label storage area requires the driver to follow these steps:

0. Pre-conditions: the driver has an updated label constructed to be written to a specific DIMM's label storage area. There is at least 1 free slot in the label storage area.
1. The driver chooses a free slot from the Label Index Block, fills in that slot number in the label's **slot** field
2. The driver writes the updated label to that slot in the label storage area
3. The driver updates the Label Index Block by taking the current index block, setting the appropriate bit in the **free** field to make the old version of the label inactive and clearing the appropriate bit in the **free** field to make the new version active, incrementing the sequence number as shown in Figure 12, and then writing the index block over the inactive index block location (making this location the new active index block if the write succeeds)

2.3.6 Writing New Namespace Labels

When creating namespaces, the driver's namespace management routines have two interesting cases: creating labels for a new Persistent Memory namespace and creating labels for a new Block Mode namespace. These operations are similar, but not identical because of the issues related to writing labels across multiple DIMMs for Persistent Memory namespaces. The following two sub-sections describe each case.



The flows described in this section assume that the label management software serializes all label updates, so that only one thread at a time is executing these flows. The pre-conditions state a certain number of free slots must be available before and after the operations. If there are not enough of these free slots in the label storage area, the operations describe should fail before any steps are taken (the failure reason should be something similar to “out of label space for the requested operation – delete some namespaces to free up label space.”).

2.3.6.1 Writing New Persistent Memory Namespace Labels

Although rare, a system interruption such as a power failure during namespace creation must be handled to make sure the labels are not left in a corrupt state. When updating labels on a single DIMM, the Label Index Block update rules provide atomicity. But when an operation involves multiple DIMMs, that mechanism alone is not sufficient for atomicity, and the **UPDATING** flag in the label is used as well. When creating a new Persistent Memory namespace, the driver must follow these steps:

0. Pre-conditions: the labels to be written to each DIMM in the Interleave Set have been constructed, each with a unique **position** field from 0 to **nlabel - 1**, and all labels with the same new UUID. All DIMMs involved have at least 2 label slots free, so that after the new labels are written, they will have at least 1 free label slot left.
1. For each DIMM in the Interleave Set, the new label is written with the **UPDATING** flag set, using the **adding a label** flow described above in section 2.3.6
2. For each DIMM in the Interleave Set, the new label is updated with the same contents as the previous step, but with the **UPDATING** flag clear, using the updating an existing label flow described above in section 2.3.5

In the case of an unexpected system interruption, the above flows leave either a partial set of labels, all with the new UUID, with the **UPDATING** flag set, or a complete of labels is left where some of them have the **UPDATING** flag set. The recovery steps in section 2.3.3 comprehend these two cases and either roll the change back or roll it forward as appropriate, making the cross-DIMM update atomic with respect to system interruption (common driver multi-threaded locking is still responsible for making the update atomic with respect to other concurrent update attempts).

2.3.6.2 Writing New Block Mode Namespace Labels

Updating labels that are all on the same DIMM is powerfail atomic by nature of the Label Index Block update rules. Since Block Mode namespaces are always



DIMM-local, the use of the **UPDATING** flag and multi-pass update described in the previous section are not necessary. Drivers creating new Block Mode namespaces must follow these steps:

0. Pre-conditions: the labels to be written to the DIMM have been constructed, each with a unique **position** field from 0 to **nlabel - 1**, and all labels with the same new UUID. The DIMMs involved has at least **nlabel + 1** label slots free, so that after the new labels are written, it will have at least 1 free label slot left.
1. All labels are written to free slots and made active in one step using steps similar to the **adding a label** flow described above in section 2.3.6:
 - a. Free slots are identified using the current index block, the **slot** field in each label is updated accordingly
 - b. All new labels are written into their free slots
 - c. The new index block is constructed so the the new label slots are no longer marked free, the sequence number is advanced as shown in Figure 12, and then the new index block is written over the inactive index block location (making this location the new active index block if the write succeeds)

2.3.7 Updating Existing Namespace Labels

At the current version of this specification, there's only one reason to write out an updated label: updating the **name** field in the label.

2.3.7.1 Updating Persistent Memory Namespace Labels

To update the **name** field associated with a Persistent Memory namespace, the driver must follow these steps:

0. Pre-conditions: the namespace must already exist. Each DIMM in the Interleave Set must have at least 1 free slot.
1. For each DIMM in the Interleave Set, the label on that DIMM is updated with a label with the new **name** field and the **UPDATING** flag set. The "for each DIMM" operation in this step must start with the DIMM containing the label whose **position** field is zero.
2. For each DIMM in the Interleave Set, the label is updated with the same contents as the previous step, but with the **UPDATING** flag clear, using the updating an existing label flow described above in section 2.3.5



If the above steps are interrupted unexpectedly, the recovery steps in section 2.3.3 handle the case where a **name** update is incomplete and finish the update.

2.4 The Label-less Namespace

Implementations may choose to expose some ranges of NVM as a namespace without any labels existing. An example of this would be a simple NVDIMM product with no label storage space or support for Block Mode. In this case, the entire interleave set should be exposed as a single namespace which is the full size of the Persistent Memory range in the SPA. Implementations may find it useful to provide this degenerate case by having the driver construct a label on the fly that represents the Persistent Memory namespace as if the label was read from a label storage area. But since this case doesn't actually use a real label storage area the **name** field are not supported.

2.5 Virtualization Considerations

Implementations allowing NVDIMM namespaces to appear in VM Guests may choose to disallow configuration changes done from within the guest. The **ROLABEL** flag describe in Table 3 provides this mechanism. The VMM would set this flag in the label when it constructs the label that the VM Guest sees, and that tells the driver in the guest that making configuration changes that write to the label storage area does not make sense and would fail if attempted.

2.6 I/O on Namespaces

A Persistent Memory namespace, by definition, is a contiguous range of System Physical Address space. So reading and writing at a given offset in the namespace is a matter of reading and writing at the same offset from the beginning of the SPA range exposing the namespace. But for Block Mode namespaces, the conversion from namespace offset to an NVM location is required. The Block Mode namespace labels contain a **dpa** field which tells the driver, for each range of NVM that is part of the namespace, the DIMM Physical Address where that range starts.

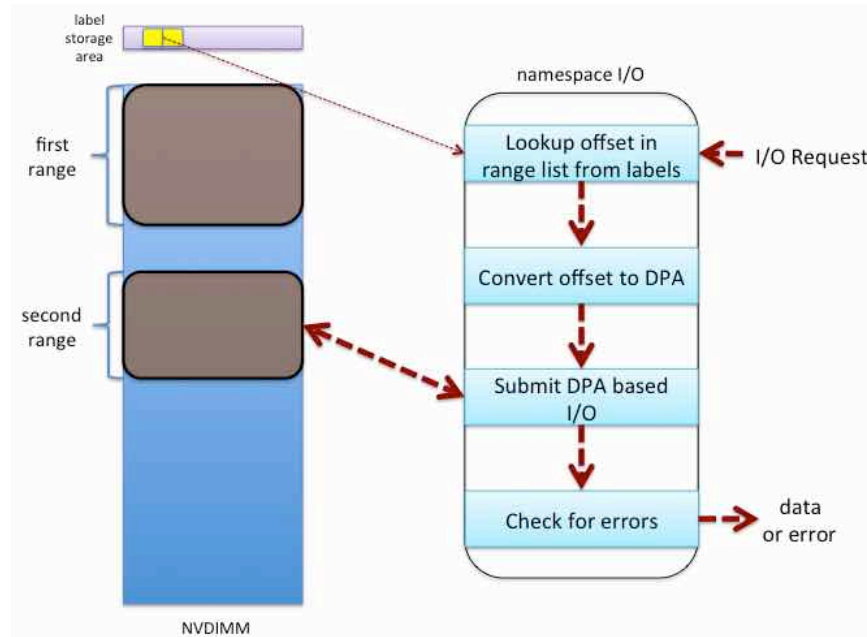


Figure 13: I/O on a Multi-Range Namespace

As shown in Figure 13 above, when a Block Mode namespace consists of multiple ranges, the driver performs a look up that maps the offset into the namespace to the specific range. This is done by comparing the namespace offset to the **rawsize** field in the labels (chapter 5 contains a full algorithmic description of this process).



3 Block Translation Table (BTT)

A block namespace may contain a Block Translation Table (BTT), which is a layout and set of rules for doing block I/O that provide powerfail write atomicity of a single block. Traditional block storage, including hard disks and SSDs, usually protect against *torn sectors*, which are sectors partially written when interrupted by power failure. Existing software, mostly file systems, depend on this behavior, often without the authors realizing it. To enable such software to work correctly on NVDIMM block devices, the BTT adds several data structures as shown in Figure 14.

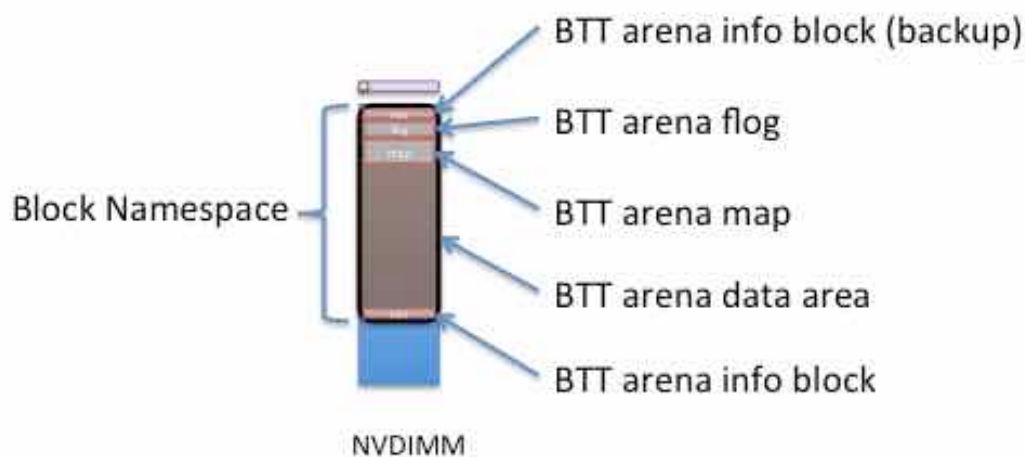


Figure 14: The BTT Data Structures in a Block Namespace

Each block namespace using a BTT is broken into arenas, each of which can handle a maximum of 512 Gigabytes. Each area will contain the layout shown in Figure 14: the info block, data area, map, flog, and a backup info block. Each of these areas is described in the following sections. When the namespace is larger than 512 Gigabytes, multiple arenas are required by the BTT layout, as shown in Figure 15.

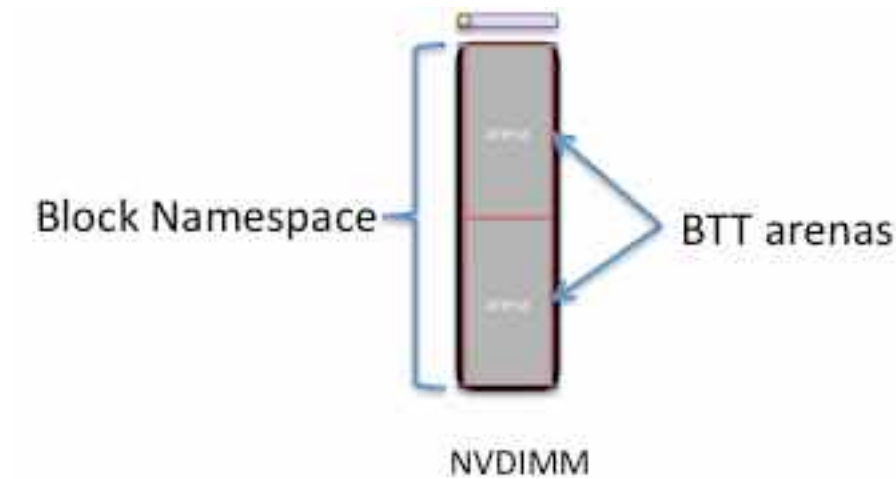


Figure 15: A BTT With Multiple Arenas in a Large Block Namespace

A full reference implementation of the BTT layout and algorithms is available as open source code. It is described at <http://pmem.io/2014/09/23/btt.html>, which includes pointers to the source code.

3.1 BTT Data Structures

All BTT fields are stored as Little Endian, unsigned integers unless stated otherwise in the tables below. Rules for the BTT layout are described below, but it is highly recommended that the reference implementation, specifically the function `write_layout()`, be used as a reference since that code has been validated extensively. The layout code can be found at <https://github.com/pmem/nvml/blob/master/src/libpmemblk/btt.c>.

3.1.1 The BTT Info Block

The following table describes the layout of the BTT Info Block. See section 4.3 for an example C structure definition of the BTT Info Block. The BTT Info Block starts at 4K offset from the start of the hosting Namespace or GPT partition.

Field	Byte Length	Byte Offset	Description
sig	16	0x0000	Must be "BTT_ARENA_INFO\0\0".
uuid	16	0x0010	UUID identifying this BTT instance



Field	Byte Length	Byte Offset	Description	
parent_uuid	16	0x0020	UUID of containing namespace or GPT partition	
flags	4	0x0030	Boolean attributes of this namespace.	
			Bit	Meaning
			0x00000001	ERROR: arena is read-only due to errors (metadata inconsistent, for example)
major	2	0x0034	Major version number. Currently at version 1. Software must support this version exactly, or decline to support the DIMM contents.	
minor	2	0x0036	Minor version number. Currently at version 1. Software may use this to check for backward-compatible features in the label.	
external_lbasize	4	0x0038	Advertised LBA size in bytes. I/O requests must be in this size chunk.	
external_nlba	4	0x003c	Advertised number of LBAs in this arena.	
internal_lbasize	4	0x0040	Internal LBA size. Each block in the arena data area is this size in bytes. This may be larger than the external_lbasize due to alignment padding between LBAs.	
internal_nlba	4	0x0044	Number of blocks in the arena data area.	



Field	Byte Length	Byte Offset	Description
nfree	4	0x0048	Number of free blocks maintained for writes to this arena. In the current layout definition, nfree will always be equal to internal_nlba – external_nlba.
infosize	4	0x004c	The size of the info block. Must be 4096.
nextoff	8	0x0050	Offset of next arena, relative to the beginning of this arena's info block.
dataoff	8	0x0058	Offset of the data area for this arena, relative to the beginning of this arena's info block. The internal-LBA number zero lives at this offset.
mapoff	8	0x0060	Offset of the map for this arena, relative to the beginning of this arena's info block.
flogoff	8	0x0068	Offset of the flog for this arena, relative to the beginning of this arena's info block.
infooff	8	0x0070	Offset of the backup copy of this arena's info block, relative to the beginning of this arena's primary info block.
unused	3968	0x0078	Must be zero.
checksum	8	0x0ff8	64-bit <i>Fletcher64</i> checksum of all fields. This field is considered as containing zero when the checksum is computed.

Table 4: BTT Info Block Fields



The existence of a valid *BTT info block* is used to determine whether a block namespace is used as a raw block device (with no powerfail atomicity guarantees) or as a BTT block device. This is similar to the way a file system *superblock* is used to indicate the existence of a file system on a block device. The block namespace *encapsulates* the BTT layout, which in turn encapsulates the OS partition table (if any), and partitions then encapsulate file systems, if any. Each BTT arena starts with a BTT info block, aligned on a 4096-byte boundary, and ends with a backup BTT info block, in the highest 4096-byte aligned block available in the arena. When writing the BTT layout, implementations should write out the info blocks from the highest arena to the lowest, writing the backup info block and other BTT data structures before the primary info block. Writing the layout in this manner will ensure that a valid BTT layout is only detected after the entire layout has been written.

3.1.2 The BTT Data Area

The BTT data area starts immediately after the BTT info block and extends to the beginning of the BTT map data structure. The amount of data that can be stored in an arena is calculated by first calculating the necessary space required for the BTT info blocks, map, and flog (plus any alignment required), subtracting that amount from the total arena size, and then calculating how many blocks fit into the resulting space.

3.1.3 The BTT Map

The BTT map area maps an LBA that indexes into the arena, to its actual location. The terminology *pre-map LBA* and *post-map LBA* is used to describe the input and output values of this mapping. The BTT map is located as high as possible in the arena, after room for the backup info block and flog (and any required alignment) has been taken into account.

The following table describes the layout of the BTT Map.

Entry	Byte Length	Byte Offset	Description	
pre-map LBA 0	4	0x0000	Map entry contains:	
			Bits	Meaning
			[29:0]	Post-map LBA number (block number in this



Entry	Byte Length	Byte Offset	Description	
				arena's data area)
			[30]	ERROR: When set, reads on this block return an error. Writes to this block clear this flag.
			[31]	ZERO: When set, reads on this block return a full block of zeros. Writes to this block clear this flag.
pre-map LBA 1	4	0x0004	“	
...	Repeat for external_nlba entries in the map.	

Table 5: BTT Map Layout

The **ERROR** and **ZERO** bits indicate conditions that cannot both be true at the same time, so that combination is used to indicate a *normal* map entry, where no error or zeroed block is indicated. In other words, the error condition is indicated only when the **ERROR** bit is set and the **ZERO** bit is clear, with similar logic for the zero block condition. When neither condition is indicated, both **ERROR** and **ZERO** are set. This leaves the case where both **ERROR** and **ZERO** are bits are zero, which is typically the initial state of the map. Both bits zero means that the map entry contains the initial *identity* mapping where the pre-map LBA is mapped to the same post-map LBA. Defining the map this way allows an implementation to leverage the case where the initial contents of the namespace is known to be zero, requiring no writes to the map when writing the layout. This can greatly improve the layout time since the map is the largest BTT data structure written during layout.



3.1.4 The BTT Flog

The BTT *flog* is so named to illustrate that it is both a free list and a log, rolled into one data structure. The flog size is determined by the **nfree** field in the BTT info block. The flog location is the highest address in the arena after space for the backup info block and alignment requirements have been taken in account.

The following table describes the layout of the BTT Flog. See section 4.4 for an example C structure definition of the BTT Flog.

Field	Byte Length	Byte Offset	Description
<code>lba</code>	4	0x0000	Last pre-map LBA written using this flog entry. This value is used as an index into the BTT map when updating it to complete the transaction.
<code>old_map</code>	4	0x0004	Old post-map LBA. This is the old entry in the map when the last write using this flog entry occurred. If the transaction is complete, this LBA is now the free block associated with this flog entry.
<code>new_map</code>	4	0x0008	New post-map LBA. This is the block allocated when the last write using this flog entry occurred. By definition, a write transaction is complete if the BTT map entry contains this value.
<code>seq</code>	4	0x000c	This sequence number field is written last to mark the flog entry as updated. Only the least-significant two bits of this field are used in the current definition, rotating through the values depicted in Figure 16. The other bits must be zero.
<code>lba'</code>	4	0x0010	Alternate lba entry.
<code>old_map'</code>	4	0x0014	Alternate old entry.



Field	Byte Length	Byte Offset	Description
<code>new_map'</code>	4	0x0018	Alternate new entry.
<code>seq</code>	4	0x001c	Alternate seq entry.
...	The above fields repeat nfree times.

Table 6: BTT Flog Layout

The **seq** field in each flog entry is used to determine which set of fields is newer. Updates to a flog entry must always be made to the older set of fields and must be implemented carefully so that the **seq** bits are only written after the other fields are known to be committed to persistence. Figure 16 shows the progression of the **seq** bits over time, where the newer entry is indicated by a value that is clockwise of the older value.

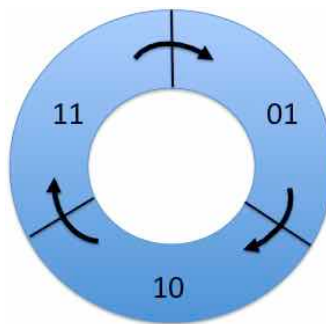


Figure 16: Cyclic Sequence Numbers for Flog Entries

3.2 BTT Theory of Operation

The reference implementation at <https://github.com/pmem/nvml/blob/master/src/libpmemblk/btt.c> contains an up-to-date and validated implementation of the BTT, so it is recommended as the best reference for BTT operation. The layout described in this document is implemented by the header file in the reference implementation, available at https://github.com/pmem/nvml/blob/master/src/libpmemblk/btt_layout.h.



3.2.1 The Read Path

As shown in the reference implementation and illustrated in Figure 17 below, reading a block from a BTT block namespace starts by calculating the arena, then looking the block number up in that arena's map. The **ERROR** and **ZERO** bits are checked and the block data itself is read if appropriate.

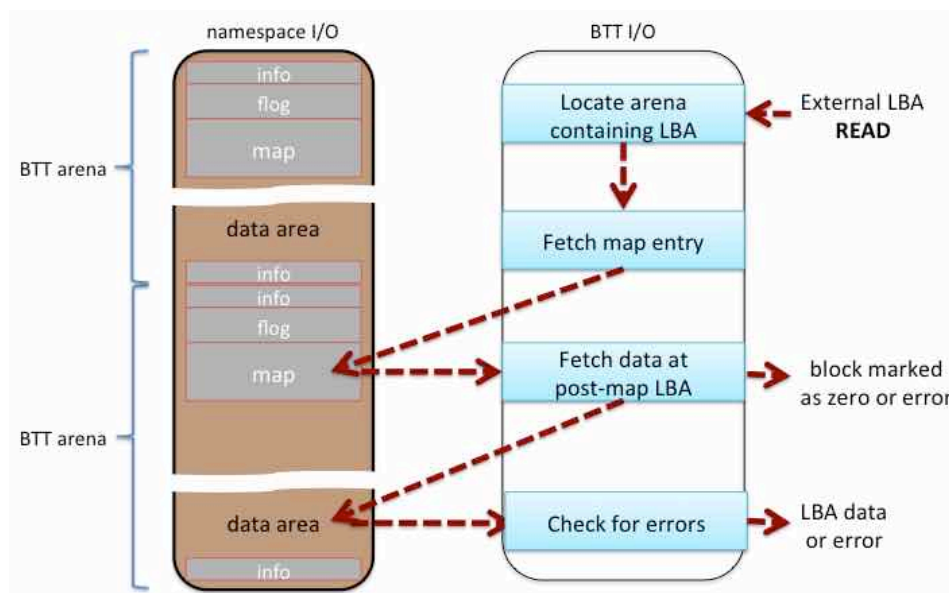


Figure 17: BTT Read Path Overview

3.2.2 The Write Path

The BTT write path is more complex than the read path described above. Each write to a BTT block namespace is an *allocating write*, avoiding the situation where an existing block is being overwritten since that would allow a block to be torn by a power failure.

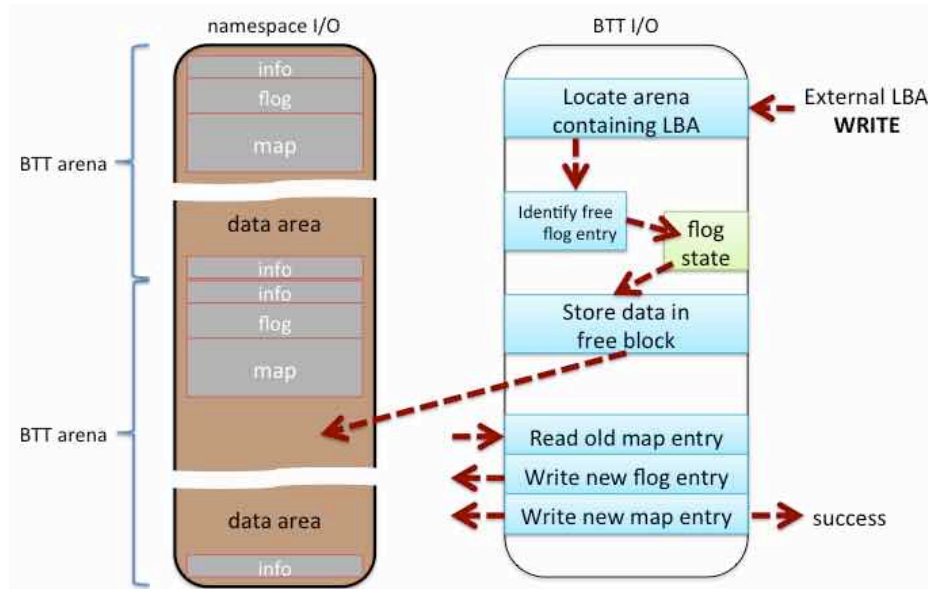


Figure 18: BTT Write Path Overview (Error Cases Not Shown)

As shown above, the write path starts with the arena calculation. A free block is found in the flog, typically by grabbing a run-time lock for a flog entry. In the reference implementation, this run-time lock is part of the *lane* mechanism. Once the volatile flog state is updated in memory, the new block is written and the persistent flog and map states are updated. These steps are carefully ordered to ensure recovery is possible after a power failure at any point in the write flow, without resulting in a torn block.

3.2.3 BTT Recovery

The reference implementation performs the recovery steps in the routine `read_flog_pair()` available at <https://github.com/pmem/nvml/blob/master/src/libpmemblk/btt.c>. Since the persistent flog and map states are not updated until the free block is written with new data, a power failure at any point during the data transfer is harmless, simply leaving the partially written data in a free block that remains free. Once the flog is updated (made atomic by the `seq` bits in the flog entry), the algorithm is committed to the update and a power failure from this point in the write flow onwards will be handled by completing the update to the map. The flog contains all the information required to complete the update.

BTT recovery is intended to happen single-threaded, on an inactive BTT (before the BTT block namespace is allowed to accept I/O requests). The maximum amount of time required for recovery is determined by `nfree`, but is only a few



loads and a single store (and the corresponding cache flushes) for each incomplete write discovered.



4 Example C Structure Definitions

This chapter provides example C structure definitions for the data structures defined by this specification. Since these are on-media structures, before storing them all multi-byte integer fields should be converted to little endian byte order. The definitions shown here use POSIX standard data types as defined in the include file `<stdint.h>`.



4.1 Namespace Label Index Block Structure

The figure below shows an example C declaration of a Label Index Block, two copies of which are stored at the beginning of the label storage area on each NVDIMM.

```

/*
 * Layout of namespace index. All integers are stored little-endian.
 */

#define NSINDEX_SIG_LEN 16
#define NSINDEX_ALIGN 256

struct namespace_index {
    char sig[NSINDEX_SIG_LEN]; /* must be "NAMESPACE_INDEX\0" */
    uint32_t flags; /* see flag bits below */
    uint32_t seq; /* sequence number for this index */
    uint64_t myoff; /* offset of this index in label area */
    uint64_t mysize; /* size of this index struct */
    uint64_t otheroff; /* offset of other index */
    uint64_t labeloff; /* offset of first label slot */
    uint32_t nslot; /* total number of label slots */
    uint16_t major; /* label area major version */
    uint16_t minor; /* label area minor version */
    uint64_t checksum; /* Fletcher64 of all fields */
    uint8_t free[]; /* bitmap, nlabel bits */
    /*
     * The size of free[] is rounded up so the total struct size
     * is a multiple of NSINDEX_ALIGN bytes. Any bits this
     * allocates beyond nlabel bits must be zero.
     */
};

/*
 * Definitions for flags mask for namespace_index struct above.
 *
 * (no flags defined at this time)
 */

```

Figure 19: Namespace Label Index Block Structure



The `nslot` field is calculated by taking the number of labels that would fit into the label storage area. For example:

```
nslot = label_storage_area_size / sizeof (struct namespace_label)
```

The size of the `free[]` field in the above struct definition is calculated by taking the number of labels that would fit into the label storage area and creating a bitmask large enough to represent that many label slots, and finally rounding that value up so the index block size is a multiple of 256. For example:

```
free_size = roundup(howmany(nslot, 8), 256)
```

4.2 Namespace Label Structure

The figure below shows an example C declaration of a Namespace Label.

```
/*
 * Layout of namespace label. All integers are stored little-endian.
 */

#define NSLABEL_UUID_LEN 16
#define NSLABEL_NAME_LEN 64

struct namespace_label {
    uint8_t uuid[NSLABEL_UUID_LEN]; /* UUID per RFC 4122 */
    char name[NSLABEL_NAME_LEN]; /* optional name (NULL-terminated) */
    uint32_t flags; /* see flag bits below */
    uint16_t nlabel; /* num labels to describe this ns */
    uint16_t position; /* labels position in set */
    uint64_t isetcookie; /* interleave set cookie */
    uint64_t lbasize; /* LBA size in bytes or 0 for pmem */
    uint64_t dpa; /* DPA of NVM range on this DIMM */
    uint64_t rawsize; /* size of namespace */
    uint32_t slot; /* slot of this label in label area */
    uint32_t unused; /* must be zero */
};

/*
 * Definitions for flags mask for namespace_label struct above.
 */
#define NSLABEL_FLAG_ROLABEL 0x00000001 /* read-only label */
#define NSLABEL_FLAG_LOCAL 0x00000002 /* DIMM-local namespace */
#define NSLABEL_FLAG_BTT 0x00000004 /* namespace contains a BTT */
#define NSLABEL_FLAG_UPDATING 0x00000008 /* label being updated */
```

Figure 20: Namespace Label Structure

4.3 BTT Info Block Structure

The figure below shows an example C declaration of a BTT Info Block.

```
/*
 * Layout of BTT info block. All integers are stored little-endian.
 */
```



```

#define BTT_ALIGNMENT 4096                /* alignment of all BTT structures */
#define BTTINFO_SIG_LEN 16
#define BTTINFO_UUID_LEN 16

struct btt_info {
    char sig[BTTINFO_SIG_LEN];             /* must be "BTT_ARENA_INFO\0\0" */
    uint8_t uuid[BTTINFO_UUID_LEN];        /* BTT UUID */
    uint8_t parent_uuid[BTTINFO_UUID_LEN]; /* UUID of container */
    uint32_t flags;                        /* see flag bits below */
    uint16_t major;                        /* major version */
    uint16_t minor;                       /* minor version */
    uint32_t external_lbasize;             /* advertised LBA size (bytes) */
    uint32_t external_nlba;                /* advertised LBAs in this arena */
    uint32_t internal_lbasize;             /* size of data area blocks (bytes) */
    uint32_t internal_nlba;                /* number of blocks in data area */
    uint32_t nfree;                        /* number of free blocks */
    uint32_t infosize;                     /* size of this info block */
    /*
     * The following offsets are relative to the beginning of
     * the btt_info block.
     */
    uint64_t nextoff;                      /* offset to next arena (or zero) */
    uint64_t dataoff;                      /* offset to arena data area */
    uint64_t mapoff;                       /* offset to area map */
    uint64_t flogoff;                      /* offset to area flog */
    uint64_t infooff;                     /* offset to backup info block */

    char unused[3968];                     /* must be zero */

    uint64_t checksum;                     /* Fletcher64 of all fields */
};

/*
 * Definitions for flags mask for btt_info structure above.
 */
#define BTTINFO_FLAG_ERROR 0x00000001 /* error state (read-only) */

/*
 * Current on-media format versions.
 */
#define BTTINFO_MAJOR_VERSION 1
#define BTTINFO_MINOR_VERSION 1

```

Figure 21: BTT Info Block Structure

4.4 BTT Flog Structure

The figure below shows an example C declaration of a BTT Flog structure. These entries are paired so that each Flog entry contains two of the structs defined below.

```

/*
 * Layout of a BTT "flog" entry. All integers are stored little-endian.
 *
 * The "nfree" field in the BTT info block determines how many of these
 * flog entries there are, and each entry consists of two of the following
 * structs (entry updates alternate between the two structs).
 */

struct btt_flog {
    uint32_t lba;                          /* last pre-map LBA using this entry */
    uint32_t old_map;                       /* old post-map LBA (the freed block) */
    uint32_t new_map;                       /* new post-map LBA */
    uint32_t seq;                           /* sequence number (01, 10, 11) */
};

```



Figure 22: BTT Flog Structure



5 Example Algorithms

This chapter contains example algorithms related to the namespace labels and BTT structures defined by this specification. In some cases, C code is provided, but most of the examples in this section are shown in an informal pseudocode format.

5.1 The Fletcher64 Checksum

The figure below shows a Fletcher64 checksum implementation that produces the correct result for the data structures in this specification when run on a 64-bit system. When checksumming a structure, any multi-byte integer fields should be in host byte order. If the structure contains its own checksum, as is commonly the case, that field should contain zero when this checksum routine is called.

```
/*  
 * checksum -- compute a Fletcher64 checksum  
 */  
uint64_t  
checksum(void *addr, size_t len)  
{  
    uint32_t *p32 = addr;  
    uint32_t *p32end = addr + len;  
    uint32_t lo32 = 0;  
    uint32_t hi32 = 0;  
  
    while (p32 < p32end) {  
        lo32 += *p32++;  
        hi32 += lo32;  
    }  
  
    return (uint64_t)hi32 << 32 | lo32;  
}
```

Figure 23: The Fletcher64 Algorithm Used in this Specification