SNIA

Advancing storage &
information technology

NVM Programming Model (NPM)

Version 1.1

Abstract: This SNIA document defines recommended behavior for software supporting Non-Volatile Memory (NVM).

Working Draft

November 7, 2014

USAGE

The SNIA hereby grants permission for individuals to use this document for personal use only, and for corporations and other business entities to use this document for internal use only (including internal copying, distribution, and display) provided that:

1. Any text, diagram, chart, table or definition reproduced shall be reproduced in its entirety with no alteration, and,

2. Any document, printed or electronic, in which material from this document (or any portion hereof) is reproduced shall acknowledge the SNIA copyright on that material, and shall credit the SNIA for granting permission for its reuse.

Other than as explicitly provided above, you may not make any commercial use of this document, sell any or this entire document, or distribute this document to third parties. All rights not explicitly granted are expressly reserved to SNIA.

Permission to use this document for purposes other than those enumerated above may be requested by e-mailing tcmd@snia.org. Please include the identity of the requesting individual and/or company and a brief description of the purpose, nature, and scope of the requested use.

All code fragments, scripts, data tables, and sample code in this SNIA document are made available under the following license:

BSD 3-Clause Software License

Copyright (c) 2014, the Storage Networking Industry Association.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of The Storage Networking Industry Association (SNIA) nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

DISCLAIMER

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

Revision History

Update 1 Revision 1

Date

November 7, 2014

Changes Incorporated

- USAGE: added BSD 3-clause license (new SNIA template)
- 1 Scope: last sentence: clarified expectation that sharing data was consistent with native hardware (as well as OS) behavior
- 3.4 Conventions: remove text on numeric conventions because they are not used
- 4.1 bullet #1: changed hard-coded section number to xref
- 4.3.3 remove bullet saying NVM.PM.VOLUME addresses errors
- 6.8 last paragraph: changed *Programming* to have 2 “m”s
- 6,10 Aligned operations on fundamental data types::moved here from former Consistency Annex (Ballot-Proposal-00008)
- 7.2.7 NVM.BLOCK.SCAR last paragraph: added “r” to “scarred” in “A block stays scarred until it is updated by a write operation.”
- 8.4.3.2.3, Failure Scenarios, paragraph 1: removed word “fundamental”
- 9.1 remove bullet saying NVM.PM.VOLUME addresses errors
- 10.1 NVM.PM.FLE Overview: changes “bold red line” to “red wavy line”
- 10.1.1 Applications and PM Consistency:: Content moved from former Consistency Annex (Ballot-Proposal-00008)
- 10.2.4 NVM.PM.FILE.SYNC paragraph 3: deleted “An annex to this specification is proposed to address this.” This refers to former Consistency Annex content which is now in 10.1.1 (Ballot-Proposal-00008)
- 10.2.23 NVM.PM.FILE.MAP paragraph 3: add OPTIMIZED-FLUSH and OPTIMIZED_FLUSH_AND_SYNC to the list of sync actions (Ballot-Proposal-00006)
- 10.2.5 NVM.PM.FILE.OPTIMIZED_FLUSH and 10.2.7 NVM.PM.FILE.OPTIMIZED_FLUSH_AND_SYNC: added “Requires: NVM.PM.FILE.MAP” (Ballot-Proposal-00006)
- 10.3.3 NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY first sentence: add reference to new 6.10 discussion of aligned operations on fundamental data types (Ballot-Proposal-00008)
- Annex B Consistency deleted (Ballot-Proposal-00008)

Table of Contents

Table of Figures

FOREWORD

The SNIA NVM Programming Technical Working Group was formed to address the ongoing proliferation of new non-volatile memory (NVM) functionality and new NVM technologies. An extensible NVM Programming Model is necessary to enable an industry wide community of NVM producers and consumers to move forward together through a number of significant storage and memory system architecture changes.

This SNIA specification defines recommended behavior between various user space and operating system (OS) kernel components supporting NVM. This specification does not describe a specific API. Instead, the intent is to enable common NVM behavior to be exposed by multiple operating system specific interfaces.

After establishing context, the specification describes several operational modes of NVM access. Each mode is described in terms of use cases, actions and attributes that inform user and kernel space components of functionality that is provided by a given compliant implementation.

Acknowledgements

The SNIA NVM Programming Technical Working Group, which developed and reviewed this standard, would like to recognize the significant contributions made by the following members:

Organization Represented	*Name of Representative*
EMC	Bob Beauchamp
Hewlett Packard	Hans Boehm
NetApp	Steve Byan
Hewlett Packard	Joe Foster
Fusion-io	Walt Hubis
Red Hat	Jeff Moyer
Fusion-io	Ned Plasson
Rougs, LLC	Tony Roug
Intel Corporation	Andy Rudoff
Microsoft	Spencer Shepler
Fusion-io	Nisha Talagata
Hewlett Packard	Doug Voigt
Intel Corporation	Paul von Behren

1 Scope

This specification is focused on the points in system software where NVM is exposed either as a hardware abstraction within an operating system kernel (e.g., a volume) or as a data abstraction (e.g., a file) to user space applications. The technology that motivates this specification includes flash memory packaged as solid state disks and PCI cards as well as other solid state non-volatile devices, including those which can be accessed as memory.

It is not the intent to exhaustively describe or in any way deprecate existing modes of NVM access. The goal of the specification is to augment the existing common storage access models (e.g., volume and file access) to add new NVM access modes. Therefore this specification describes the discovery and use of capabilities of NVM media, connections to the NVM, and the system containing the NVM that are emerging in the industry as vendor specific implementations. These include:

- supported access modes,
- visibility in memory address space,
- atomicity and durability,
- recognizing, reporting, and recovering from errors and failures,
- data granularity, and
- capacity reclamation.

This revision of the specification focuses on NVM behaviors that enable user and kernel space software to locate, access, and recover data. It does not describe behaviors that are specific to administrative or diagnostic tasks for NVM. There are several reasons for intentionally leaving administrative behavior out of scope.

- For new types of storage programming models, the access must be defined and agreed on before the administration can be defined. Storage management behavior is typically defined in terms of how it enables and diagnoses the storage programming model.
- Administrative tasks often require human intervention and are bound to the syntax for the administration. This document does not define syntax. It focuses only on the semantics of the programming model.
- Defining diagnostic behaviors (e.g., wear-leveling) as vendor-agnostic is challenging across all vendor implementations. A common recommended behavior may not allow an approach optimal for certain hardware.

This revision of the specification does not address sharing data across computing nodes. This revision of the specification assumes that sharing data between processes and threads follows the native OS and hardware behavior.

2 References

The following referenced documents are indispensable for the application of this document.

For references available from ANSI, contact ANSI Customer Service Department at (212) 642-49004980 (phone), (212) 302-1286 (fax) or via the World Wide Web at http://www.ansi.org.

SPC-3	ISO/IEC 14776-453, SCSI Primary Commands – 3 [ANSI INCITS 408-2005] Approved standard, available from ANSI.
SBC-2	ISO/IEC 14776-322, SCSI Block Commands - 2 [T10/BSR INCITS 514] Approved standard, available from ANSI.
ACS-2	ANSI INCITS 482-2012, Information technology - ATA/ATAPI Command Set -2 Approved standard, available from ANSI.
NVMe 1.1	NVM Express Revision 1.1, Approved standard, available from http://nvmexpress.org
SPC-4	SO/IEC 14776-454, SCSI Primary Commands - 4 (SPC-4) (T10/1731-D) Under development, available from http://www.t10.org.
SBC-4	ISO/IEC 14776-324, SCSI Block Commands - 4 (SBC-4) [BSR INCITS 506] Under development, available from http://www.t10.org.
T10 13-064r0	T10 proposal 13-064r0, Rob Elliot, Ashish Batwara, SBC-4 SPC-5 Atomic writes Proposal, available from http://www.t10.org.
ACS-2 r7	Information technology - ATA/ATAPI Command Set – 2 r7 (ACS-2) Under development, available from http://www.t13.org.
Intel SPG	Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, and 3C: System Programming Guide*, Parts 1 and 2, available from http://download.intel.com/products/processor/manual/325384.pdf

3 Definitions, abbreviations, and conventions

For the purposes of this document, the following definitions and abbreviations apply.

3.1 Definitions

3.1.1 durable

committed to a persistence domain (see 3.1.7)

3.1.2 load and store operations

commands to move data between CPU registers and memory

3.1.3 memory-mapped file

segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file

3.1.4 non-volatile memory

any type of memory-based, persistent media; including flash memory packaged as solid state disks, PCI cards, and other solid state non-volatile devices

3.1.5 NVM block capable driver

driver supporting the native operating system interfaces for a block device

3.1.6 NVM volume

subset of one or more NVM devices, treated by software as a single logical entity

See 4.2 NVM device models

3.1.7 persistence domain

location for data that is guaranteed to preserve the data contents across a restart of the device containing the data

See 6.9 Persistence domain

3.1.8 persistent memory

storage technology with performance characteristics suitable for a load and store programming model

3.1.9 programming model

set of software interfaces that are used collectively to provide an abstraction for hardware with similar capabilities

3.2 Keywords

In the remainder of the specification, the following keywords are used to indicate text related to compliance:

3.2.1 **mandatory**

a keyword indicating an item that is required to conform to the behavior defined in this standard

3.2.2 **may**

a keyword that indicates flexibility of choice with no implied preference; “may” is equivalent to “may or may not”

3.2.3 **may not**

keywords that indicate flexibility of choice with no implied preference; “may not” is equivalent to “may or may not”

3.2.4 **need not**

keywords indicating a feature that is not required to be implemented; “need not” is equivalent to “is not required to”

3.2.5 **optional**

a keyword that describes features that are not required to be implemented by this standard; however, if any optional feature defined in this standard is implemented, then it shall be implemented as defined in this standard

3.2.6 **shall**

a keyword indicating a mandatory requirement; designers are required to implement all such mandatory requirements to ensure interoperability with other products that conform to this standard

3.2.7 **should**

a keyword indicating flexibility of choice with a strongly preferred alternative

3.3 Abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
NVM	Non-Volatile Memory
PM	Persistent Memory
SSD	Solid State Disk

3.4 Conventions

Representation of modes in figures

Modes are represented by red, wavy lines in figures, as shown below:

The wavy lines have labels identifying the mode name (which in turn, identifies a clause of the specification).

4 Overview of the NVM Programming Model (informative)

4.1 How to read and use this specification

Documentation for I/O programming typically consists of a set of OS-specific Application Program Interfaces (APIs). API documentation describes the syntax and behavior of the API. This specification intentionally takes a different approach and describes the behavior of NVM programming interfaces, but allows the syntax to integrate with similar operating system interfaces. A recommended approach for using this specification is:

1. Determine which mode applies (read 4.3 NVM programming modes).

2. Refer to the mode section to learn about the functionality provided by the mode and how it relates to native operating system APIs; the use cases provide examples. The mode specific section refers to other specification sections that may be of interest to the developer.

3. Determine which mode actions and attributes relate to software objectives.

4. Locate the vendor/OS mapping document (see 5.2) to determine which APIs map to the actions and attributes.

For an example, a developer wants to update an existing application to utilize persistent memory hardware. The application is designed to bypass caches to assure key content is durable across power failures; the developer wants to learn about the persistent memory programming model. For this example:

1. The NVM programming modes section identifies NVM.PM.FILE mode (see 10 NVM.PM.FILE) as the starting point for application use of persistent memory.

2. The NVM.PM.FILE mode text describes the general approach for accessing PM (similar to native memory-mapped files) and the role of PM aware file system.

3. The NVM.PM.FILE mode identifies the NVM.PM.FILE.MAP and NVM.PM.FILE.SYNC actions and attributes that allow an application to discover support for optional features.

4. The operating system vendor’s mapping document describes the mapping between NVM.PM.FILE.MAP/SYNC and API calls, and also provides information about supported PM-aware file systems.

4.2 NVM device models

4.2.1 Overview

This section describes device models for NVM to help readers understand how key terms in the programming model relate to other software and hardware. The models presented here generally apply across operating systems, file systems, and hardware; but there are differences across implementations. This specification strives to discuss the model generically, but mentions key exceptions.

One of the challenges discussing the software view of NVM is that the same terms are often used to mean different things. For example, between commonly used management applications, programming interfaces, and operating system documentation, *volume* may refer to a variety of things. Within this specification, *NVM volume* has a specific meaning.

An NVM volume is a subset of one or more NVM devices, treated by software as a single logical entity. For the purposes of this specification, a volume is a container of storage. A volume may be block capable and may be persistent memory capable. The consumer of a volume sees its content as a set of contiguous addresses, but the unit of access for a volume differs across different modes and device types. Logical addressability and physical allocation may be different.

In the examples in this section, “NVM block device” refers to NVM hardware that emulates a disk and is accessed in software by reading or writing ranges of blocks. “PM device” refers to NVM hardware that may be accessed via load and store operations.

4.2.2 Block NVM example

Consider a single drive form factor SSD where the entire SSD capacity is dedicated to a file system. In this case, a single NVM block volume maps to a single hardware device. A file system (not depicted) is mounted on the NVM block volume.

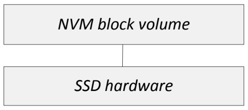


Figure 1 Block NVM example

The same model may apply to NVM block hardware other than an SDD (including flash on PCIe cards).

4.2.3 Persistent memory example

This example depicts a NVDIMM and PM volume. A PM-aware file system (not depicted) would be mounted on the PM volume.

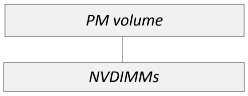


Figure 2 PM example

The same model may apply to PM hardware other than an NVDIMM (including SSDs, PCIe cards, etc.).

4.2.4 NVM block volume using PM hardware

In this example, the persistent memory implementation includes a driver that uses a range of persistent memory (a PM volume) and makes it appear to be a block NVM device in the legacy block stack. This emulated block device could be aggregated or de-aggregated like legacy block devices. In this example, the emulated block device is mapped 1-1 to an NVM block volume and non-PM file system.

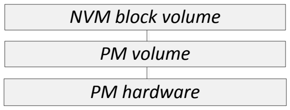


Figure 3 Block volume using PM HW

Note that there are other models for connecting a non-PM file system to PM hardware.

4.3 NVM programming modes

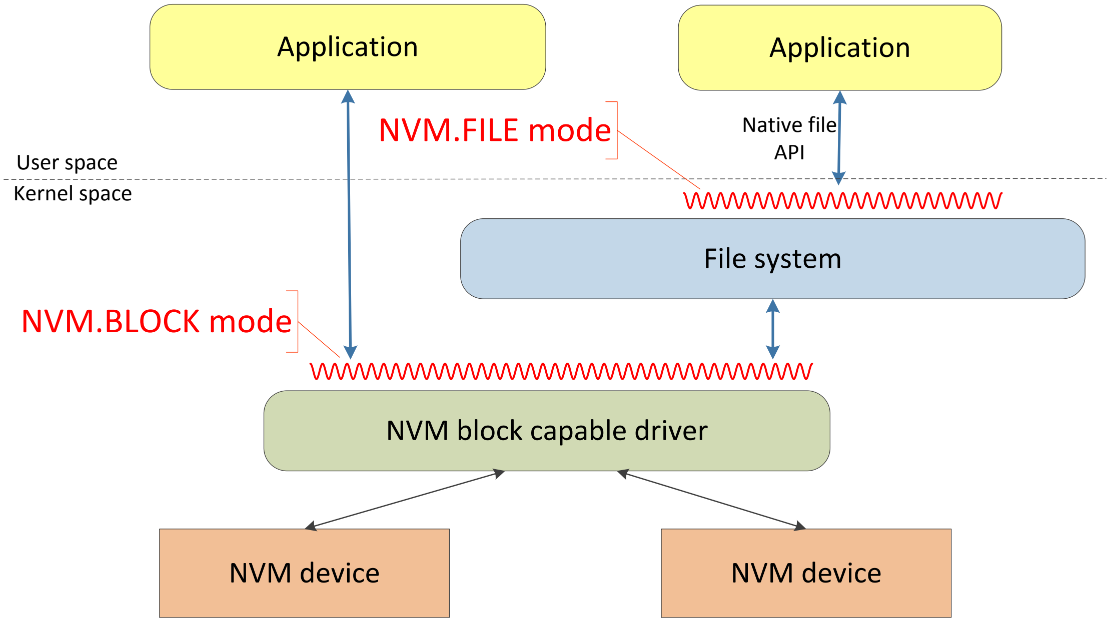
4.3.1 **NVM.BLOCK mode overview**

NVM.BLOCK and NVM.FILE modes are used when NVM devices provide block storage behavior to software (in other words, emulation of hard disks). The NVM may be exposed as a single or as multiple NVM volumes. Each NVM volume supporting these modes provides a range of logically-contiguous blocks. NVM.BLOCK mode is used by operating system components (for example, file systems) and by applications that are aware of block storage characteristics and the block addresses of application data.

This specification does not document existing block storage software behavior; the NVM.BLOCK mode describes NVM extensions including:

- Discovery and use of atomic write and discard features
- The discovery of granularities (length or alignment characteristics)
- Discovery and use of ability for applications or operating system components to mark blocks as unreadable

Figure 4 NVM.BLOCK and NVM.FILE mode examples



4.3.2 **NVM.FILE mode overview**

NVM.FILE mode is used by applications that are not aware of details of block storage hardware or addresses. Existing applications written using native file I/O behavior should work unmodified with NVM.FILE mode; adding support in the application for NVM extensions may optimize the application.

An application using NVM.FILE mode may or may not be using memory-mapped file I/O behavior.

The NVM.FILE mode describes NVM extensions including:

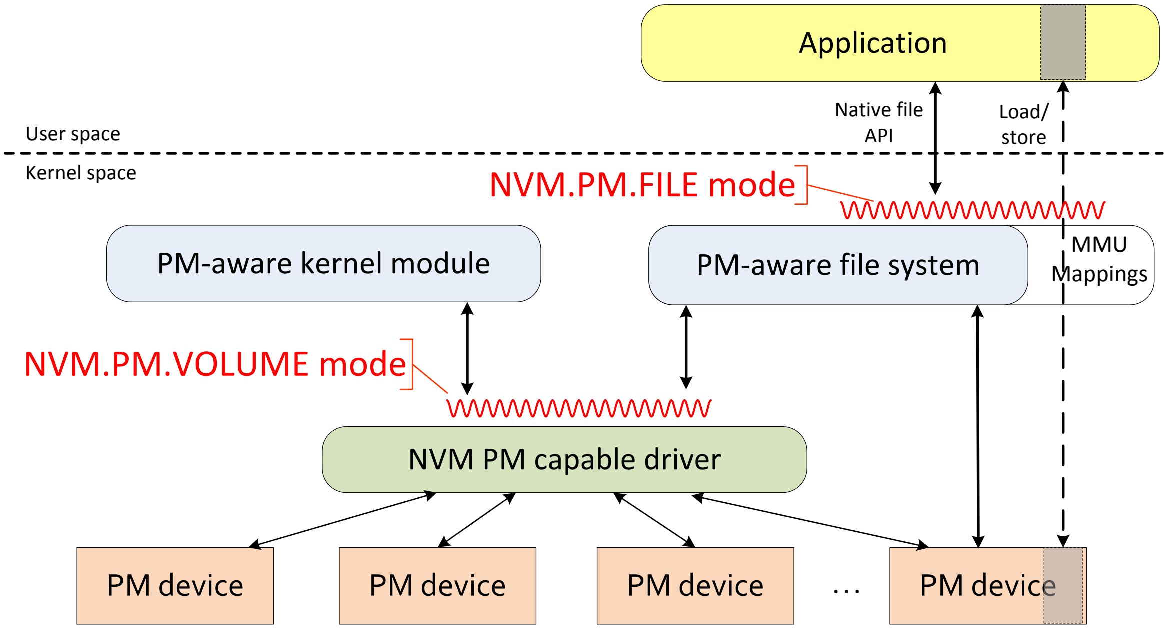
- Discovery and use of atomic write features
- The discovery of granularities (length or alignment characteristics)

4.3.3 NVM.PM.VOLUME mode overview

NVM.PM.VOLUME mode describes the behavior for operating system components (such as file systems) accessing persistent memory. NVM.PM.VOLUME mode provides a software abstraction for Persistent Memory hardware and profiles functionality for operating system components including:

- the list of physical address ranges associated with each PM volume

Figure 5 NVM.PM.VOLUME and NVM.PM.FILE mode examples



4.3.4 NVM.PM.FILE mode overview

NVM.PM.FILE mode describes the behavior for applications accessing persistent memory. The commands implementing NVM.PM.FILE mode are similar to those using NVM.FILE mode, but NVM.PM.FILE mode may not involve I/O to the page cache. NVM.PM.FILE mode documents behavior including:

- mapping PM files (or subsets of files) to virtual memory addresses
- syncing portions of PM files to the persistence domain

4.4 Introduction to actions, attributes, and use cases

4.4.1 **Overview**

This specification uses four types of elements to describe NVM behavior. Use cases are the highest order description. They describe complete scenarios that accomplish a goal. Actions are more specific in that they describe an operation that represents or interacts with NVM. Attributes comprise information about NVM. Property Group Lists describe groups of related properties that may be considered attributes of a data structure or class; but the specification allows flexibility in the implementation.

4.4.2 **Use cases**

In general, a use case states a goal or trigger and a result. It captures the intent of an application and describes how actions are used to accomplish that intent. Use cases illustrate the use of actions and help to validate action definitions. Use cases also describe system behaviors that are not represented as actions. Each use case includes the following information:

- a purpose and context including actors involved in the use case;
- triggers and preconditions indicating when a use case applies;
- inputs, outputs, events and actions that occur during the use case;
- references to related materials or concepts including other use cases that use or extend the use case.

4.4.3 **Actions**

Actions are defined using the following naming convention:

<context>.<mode>.<verb>

The actions in this specification all have a context of “NVM”. The mode refers to one of the NVM models documented herein (or “COMMON” for actions used in multiple modes). The verb states what the action does. Examples of actions include “NVM.COMMON.GET_ATTRIBUTE” and “NVM.FILE.ATOMIC_WRITE”. In some cases native actions that are not explicitly specified by the programming model are referenced to illustrate usage.

The description of each action includes:

- parameters and results of the action
- details of the action’s behavior
- compatibility of the action with pre-existing APIs in the industry

A number of actions involve options that can be specified each time the action is used. The options are given names that begin with the name of the action and end with a descriptive term that is unique for the action. Examples include NVM.PM.FILE.MAP_COPY_ON_WRITE and NVM.PM.FILE.MAP_SHARED.

A number of actions are optional. For each of these, there is an attribute that indicates whether the action is supported by the implementation in question. By convention these attributes end with the term “CAPABLE” such as NVM.BLOCK.ATOMIC_WRITE_CAPABLE. Supported options are also enumerated by attributes that end in “CAPABLE”.

4.4.4 Attributes

Attributes describe properties or capabilities of a system. This includes indications of which actions can be performed in that system and variations on the internal behavior of specific actions. For example attributes describe which NVM modes are supported in a system, and the types of atomicity guarantees available.

In this programming model, attributes are not arbitrary key value pairs that applications can store for unspecified purposes. Instead the NVM attributes are intended to provide a common way to discover and configure certain aspects of systems based on agreed upon interpretations of names and values. While this can be viewed as a key value abstraction it does not require systems to implement a key value repository. Instead, NVM attributes are mapped to a system’s native means of describing and configuring those aspects associated with said attributes. Although this specification calls out a set of attributes, the intent is to allow attributes to be extended in vendor unique ways through a process that enables those extensions to become attributes and/or attribute values in a subsequent version of the specification or in a vendor’s mapping document.

4.4.5 Property group lists

A **property group** is set of property values used together in lists; typically **property group lists** are inputs or outputs to actions. The implementation may choose to implement a property group as a new data structure or class, use properties in existing data structures or classes, or other mechanisms as long as the caller can determine which collection of values represent the members of each list element.

5 Compliance to the programming model

5.1 Overview

Since a programming model is intentionally abstract, proof of compliance is somewhat indirect. The intent is that a compliant implementation, when properly configured, can be used in such a way as to exhibit the behaviors described by the programming model without unnecessarily impacting other aspects of the implementation.

Compliance of an implementation shall be interpreted as follows.

5.2 Documentation of mapping to APIs

In order to be considered compliant with this programming model, implementations must provide documentation of the mapping of attributes and actions in the programming model to their counterparts in the implementation.

5.3 Compatibility with unspecified native actions

Actions and attributes of the native block and file access methods that correspond to the modes described herein shall continue to function as defined in those native methods. This specification does not address unmodified native actions except in passing to illustrate their usage.

5.4 Mapping to native interfaces

Implementations are expected to provide the behaviors specified herein by mapping them as closely as possible to native interfaces. An implementation is not required to have a one-to-one mapping between actions (or attributes) and APIs – for example, an implementation may have an API that implements multiple actions.

NVM Programming Model action descriptions do not enumerate all possible results of each action. Only those that modify programming model specific behavior are listed. The results that are referenced herein shall be discernible from the set of possible results returned by the native action in a manner that is documented with action mapping.

Attributes with names ending in _CAPABLE are used to inform a caller whether an optional action or attribute is supported by the implementations. The mandatory requirement for _CAPABLE attributes can be met by the mapping document describing the implementation's default behavior for reporting unsupported features. For example: the mapping document could state that if a flag with a name based on the attribute is undefined, then the action/attribute is not supported.

6 Common programming model behavior

6.1 Overview

This section describes behavior that is common to multiple modes and also behavior that is independent from the modes.

6.2 Conformance to multiple file modes

A single computer system may include implementations of both NVM.FILE and NVM.PM.FILE modes. A given file system may be accessed using either or both modes provided that the implementations are intended by their vendor(s) to interoperate. Each implementation shall specify its own mapping to the NVM Programming Model.

A single file system implementation may include both NVM.FILE and NVM.PM.FILE modes. The mapping of the implementation to the NVM Programming Model must describe how the actions and attributes of different modes are distinguished from one another.

Implementation specific errors may result from attempts to use NVM.PM.FILE actions on files that were created in NVM.FILE mode or vice versa. The mapping of each implementation to the NVM Programming Model shall specify any limitations related multi-mode access.

6.3 Device state at system startup

Prior to use, a file system is associated with one or more volumes and/or NVM devices.

The NVM devices shall be in a state appropriate for use with file systems. For example, if transparent RAID is part of the solution, components implementing RAID shall be active so the file system sees a unified virtual device rather than individual RAID components.

6.4 Secure erase

Secure erase of a volume or device is an administrative act with no defined programming model action.

6.5 Allocation of space

Following native operating system behavior, this programming model does not define specific actions for allocating space. Most allocation behavior is hidden from the user of the file, volume or device.

6.6 Interaction with I/O devices

Interaction between Persistent Memory and I/O devices (for example, DMA) shall be consistent with native operating system interactions between devices and volatile memory.

6.7 NVM State after a media or connection failure

There is no action defined to determine the state of NVM for circumstances such as a media or connection failure. Vendors may provide techniques such as redundancy algorithms to address this, but the behavior is outside the scope of the programming model.

6.8 Error handling for persistent memory

The handling of errors in memory-mapped file implementations varies across operating systems. Existing implementations support memory error reporting however there is not sufficient similarity for a uniform approach to persistent memory error handling behavior. Additional work is required to define an error handling approach. The following factors are to be taken into account when dealing with errors.

- The application is in the best position to perform recovery as it may have access to additional sources of data necessary to rewrite a bad memory address.
- Notification of a given memory error occurrence may need to be delivered to both kernel and user space consumers (e.g., file system and application)
- Various hardware platforms have different capabilities to detect and report memory errors
- Attributes and possibly actions related to error handling behavior are needed in the NVM Programming model

A proposal for persistent memory error handling appears as an appendix; see Annex B.

6.9 Persistence domain

NVM PM hardware supports the concept of a persistence domain. Once data has reached a persistence domain, it may be recoverable during a process that results from a system restart. Recoverability depends on whether the pattern of failures affecting the system during the restart can be tolerated by the design and configuration of the persistence domain.

Multiple persistence domains may exist within the same system. It is an administrative act to align persistence domains with volumes and/or file systems. This must be done in such a way that NVM Programming Model behavior is assured from the point of view of each compliant volume or file system.

6.10 Aligned operations on fundamental data types

Data alignment means putting the data at a memory offset equal to some multiple of the word size, which increases the system's performance due to the way the CPU handles memory (from Wikipedia “Data structure alignment”). Data types are ***fundamental*** when they are native to programming languages or libraries.

Aligned operations on data types are usually exactly the same operations that under normal operation become visible to other threads/data producers atomically. They are already well-defined for most settings:

- Instruction Set Architectures already define them.
    - E.g., for x86, MOV instructions with naturally aligned operands of at most 64 bits qualify.

- They're generated by known high-level language constructs, e.g.:
    - C++11 lock-free atomic<T>, C11 _Atomic(T), Java & C# volatile, OpenMP atomic directives.

For optimal performance, fundamental data types fit within CPU cache lines.

6.11 Common actions

6.11.1 NVM.COMMON.GET_ATTRIBUTE

Requirement: mandatory

Get the value of one or more attributes. Implementations conforming to the specification shall provide the get attribute behavior, but multiple programmatic approaches may be used.

Inputs:

- reference to appropriate instance (for example, reference to an NVM volume)
- attribute name

Outputs:

- value of attribute

The vendor's mapping document shall describe the possible errors reported for all applicable programmatic approaches.

6.11.2 NVM.COMMON.SET_ATTRIBUTE

Requirement: optional

Note: at this time, no settable attributes are defined in this specification, but they may be added in a future revision.

Set the value of one attribute. Implementations conforming to the specification shall provide the set attribute behavior, but multiple programmatic approaches may be used.

Inputs:

- reference to appropriate instance
- attribute name
- value to be assigned to the attribute

The vendor's mapping document shall describe the possible errors reported for all applicable programmatic approaches.

6.12 Common attributes

6.12.1 **NVM.COMMON.SUPPORTED_MODES**

Requirement: mandatory

SUPPORTED_MODES returns a list of the modes supported by the NVM implementation.

Possible values: NVM.BLOCK, NVM.FILE, NVM.PM.FILE, NVM.PM.VOLUME

NVM.COMMON.SET_ATTRIBUTE is not supported for NVM.COMMON.SUPPORTED_MODES.

6.12.2 **NVM.COMMON.FILE_MODE**

Requirement: mandatory if NVM.FILE or NVM.PM.FILE is supported

Returns the supported file modes (NVM.FILE and/or NVM.PM.FILE) provided by a file system.

Target: a file path

Output value: a list of values: “NVM.FILE” and/or “NVM.PM.FILE”

See 6.2 Conformance to multiple file modes.

6.13 Use cases

6.13.1 **Application determines which mode is used to access a file system**

Purpose/triggers:

An application needs to determine whether the underlying file system conforms to NVM.FILE mode, NVM.PM.FILE mode, or both.

Scope/context:

Some actions and attributes are defined differently in NVM.FILE and NVM.PM.FILE; applications may need to be designed to handle these modes differently. This use case describes steps in an application’s initialization logic to determine the mode(s) supported by the implementation and set a variable indicating the preferred mode the application will use in subsequent actions. This application prefers to use NVM.PM.FILE behavior if both modes are supported.

Success scenario:

1) Invoke NVM.COMMON.GET_ATTRIBUTE (NVM.COMMON.FILE_MODE) targeting a file path; the value returned provides information on which modes may be used to access the data.
2) If the response includes “NVM.FILE”, then the actions and attributes described for the NVM.FILE mode are supported. Set the preferred mode for this file system to NVM.FILE.

3) If the response includes “NVM.PM.FILE”, then the actions and attributes described for the NVM.PM.FILE mode are supported. Set the preferred mode for this file system to NVM.PM.FILE.

Outputs:

Postconditions:

A variable representing the preferred mode for the file system has been initialized.

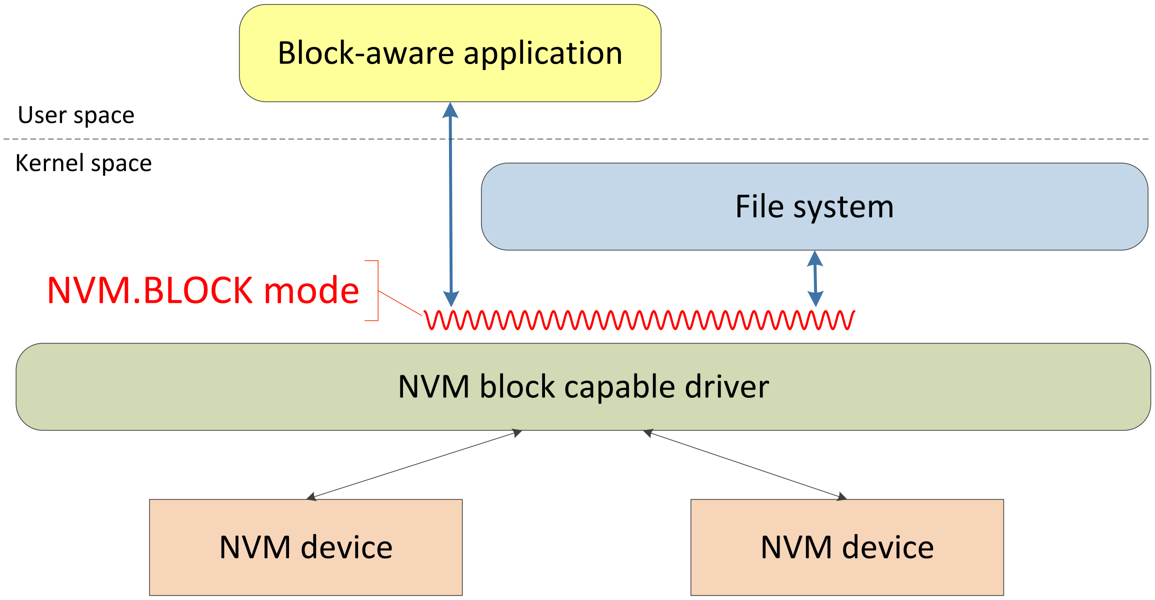
See also:
6.2 Conformance to multiple file modes
6.12.2 NVM.COMMON.FILE_MODE

7 NVM.BLOCK mode

7.1 Overview

NVM.BLOCK mode provides programming interfaces for NVM implementations behaving as block devices. The programming interfaces include the native operating system behavior for sending I/O commands to a block driver and adds NVM extensions. To support this mode, the NVM devices are supported by an NVM block capable driver that provides the command interface to the NVM. This specification does not document the native operating system block programming capability; it is limited to the NVM extensions.

Figure 6 NVM.BLOCK mode example



Support for NVM.BLOCK mode requires that the NVM implementation support all behavior not covered in this section consistently with the native operating system behavior for native block devices.

The NVM extensions supported by this mode include:

- Discovery and use of atomic write and discard features
- The discovery of granularities (length or alignment characteristics)
- Discovery and use of per-block metadata used for verifying integrity
- Discovery and use of ability for applications or operating system components to mark blocks as unreadable

7.1.1 Discovery and use of atomic write features

Atomic Write support provides applications with the capability to assure that all the data for an operation is written to the persistence domain or, if a failure occurs, it appears that no operation took place. Applications may use atomic write operations to assure consistent behavior during a failure condition or to assure consistency between multiple processes accessing data simultaneously.

7.1.2 The discovery of granularities

Attributes are introduced to allow applications to discover granularities associated with NVM devices.

7.1.3 Discovery and use of capability to mark blocks as unreadable

An action (NVM.BLOCK.SCAR) is defined allowing an application to mark blocks as unreadable.

7.1.4 NVM.BLOCK consumers: operating system and applications

NVM.BLOCK behavior covers two types of software: NVM-aware operating system components and block-optimized applications.

7.1.4.1 NVM.BLOCK operating system components

NVM-aware operating system components use block storage and have been enhanced to take advantage of NVM features. Examples include file systems, logical volume managers, software RAID, and hibernation logic.

7.1.4.2 Block-optimized applications

Block-optimized applications use a hybrid behavior utilizing files and file I/O operations, but construct file I/O commands in order to cause drivers to issue desired block commands. Operating systems and file systems typically provide mechanisms to enable block-optimized application. The techniques are system specific, but may include:

- A mechanism for a block-optimized application to request that the file system move data directly between the device and application memory, bypassing the buffering typically provided by the file system.
- The operating system or file system may require the application to align requests on block boundaries.

The file system and operating system may allow block-optimized applications to use memory-mapped files.

7.1.4.3 Mapping documentation

NVM.BLOCK operating system components may use I/O commands restricted to kernel space to send I/O commands to drivers. NVM.BLOCK applications may use a constrained set of file I/O operations to send commands to drivers. As applicable, the implementation shall provide documentation mapping actions and/or attributes for all supported techniques for NVM.BLOCK behavior.

The implementation shall document the steps to utilize supported capabilities for block-optimized applications and the constraints (e.g., block alignment) compared to NVM.FILE behavior.

7.2 Actions

7.2.1 Actions that apply across multiple modes

The following actions apply to NVM.BLOCK mode as well as other modes.

NVM.COMMON.GET_ATTRIBUTE (see 6.11.1)
NVM.COMMON.SET_ATTRIBUTE (see 6.11.2)

7.2.2 NVM.BLOCK.ATOMIC_WRITE

Requirement: mandatory if ATOMIC_WRITE_CAPABLE (see 7.3.1) is true

Block-optimized applications or operating system components may use ATOMIC_WRITE to assure consistent behavior during a power failure condition. This specification does not specify the order in which this action occurs relative to other I/O operations, including other ATOMIC_WRITE or ATOMIC_MULTIWRITE actions. This specification does not specify when the data written becomes visible to other threads.

Inputs:

- the starting memory address
- a reference to the block device
- the starting block address
- the length

The interpretation of addresses and lengths (block or byte, alignment) should be consistent with native write actions. Implementations shall provide documentation on the requirements for specifying the starting addresses, block device, and length.

Return values:

- Success shall be returned if all blocks are updated in the persistence domain
- an error shall be reported if the length exceeds ATOMIC_WRITE_MAX_DATA_LENGTH (see 7.3.3)
- an error shall be reported if the starting address is not evenly divisible by ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY (see 7.3.4)
- an error shall be reported if the length is not evenly divisible by ATOMIC_WRITE_LENGTH_GRANULARITY (see 7.3.5)
- If anything does or will prevent all of the blocks from being updated in the persistence domain before completion of the operation, an error shall be reported and all the logical blocks affected by the operation shall contain the data that was present before the device server started processing the write operation (i.e., the old data, as if the atomic write operation had no effect). If the NVM and processor are both impacted by a power failure, no error will be returned since the execution context is lost.
- the different errors described above shall be discernible by the consumer and shall be discernible from media errors

Relevant attributes:

ATOMIC_WRITE_MAX_DATA_LENGTH (see 7.3.3)

ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY (see 7.3.4)
ATOMIC_WRITE_LENGTH_GRANULARITY (see 7.3.5)
ATOMIC_WRITE_CAPABLE (see 7.3.1)

7.2.3 NVM.BLOCK.ATOMIC_MULTIWRITE

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6) is true

Block-optimized applications or operating system components may use ATOMIC_MULTIWRITE to assure consistent behavior during a power failure condition. This action allows a caller to write non-adjacent extents atomically. The caller of ATOMIC_MULTIWRITE provides a Property Group List (see 4.4.5) where the properties describe the memory and block extents (see Inputs below); all of the extents are written as a single atomic operation. This specification does not specify the order in which this action occurs relative to other I/O operations, including other ATOMIC_WRITE or ATOMIC_MULTIWRITE actions. This specification does not specify when the data written becomes visible to other threads.

Inputs:

A Property Group List (see 4.4.5) where the properties are:

- memory address starting address
- length of data to write (in bytes)
- a reference to the device being written to
- the starting LBA on the device

Each property group represents an I/O. The interpretation of addresses and lengths (block or byte, alignment) should be consistent with native write actions. Implementations shall provide documentation on the requirements for specifying the ranges.

Return values:

- Success shall be returned if all block ranges are updated in the persistence domain
- an error shall be reported if the block ranges overlap
- an error shall be reported if the total size of memory input ranges exceeds ATOMIC_MULTIWRITE_MAX_DATA_LENGTH (see 7.3.8)
- an error shall be reported if the starting address in any input memory range is not evenly divisible by ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY (see 7.3.9)
- an error shall be reported if the length in any input range is not evenly divisible by ATOMIC_MULTIWRITE_LENGTH_GRANULARITY (see 7.3.10)
- If anything does or will prevent all of the writes from being applied to the persistence domain before completion of the operation, an error shall be reported and all the logical blocks affected by the operation shall contain the data that was present before the device server started processing the write operation (i.e., the old data, as if the atomic write operation had no effect). If the NVM and processor are both impacted by a power failure, no error will be returned since the execution context is lost.
- the different errors described above shall be discernible by the consumer

Relevant attributes:

ATOMIC_MULTIWRITE_MAX_IOS (see 7.3.7)
ATOMIC_MULTIWRITE_MAX_DATA_LENGTH (see 7.3.8)
ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY (see 7.3.9)
ATOMIC_MULTIWRITE_LENGTH_GRANULARITY (see 7.3.10)
ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6)

7.2.4 NVM.BLOCK.DISCARD_IF_YOU_CAN

Requirement: mandatory if DISCARD_IF_YOU_CAN_CAPABLE (see 7.3.17) is true

This action notifies the NVM device that some or all of the blocks which constitute a volume are no longer needed by the application. This action is a hint to the device.

Although the application has logically discarded the data, it may later read this range. Since the device is not required to physically discard the data, its response is undefined: it may return successful response status along with unknown data (e.g., the old data, a default “undefined” data, or random data), or it may return an unsuccessful response status with an error.

Inputs: a range of blocks (starting LBA and length in logical blocks)

Status: Success indicates the request is accepted but not necessarily acted upon.

7.2.5 NVM.BLOCK.DISCARD_IMMEDIATELY

Requirement: mandatory if DISCARD_IMMEDIATELY_CAPABLE (see 7.3.18) is true

Requires that the data block be unmapped (see NVM.BLOCK.EXISTS 7.2.6) before the next READ or WRITE reference even if garbage collection of the block has not occurred yet,

DISCARD_IMMEDIATELY commands cannot be acknowledged by the NVM device until the DISCARD_IMMEDIATELY has been durably written to media in a way such that upon recovery from a power-fail event, the block is guaranteed to remain discarded.

Inputs: a range of blocks (starting LBA and length in logical blocks)

The values returned by subsequent read operations are specified by the DISCARD_IMMEDIATELY_RETURNS (see 7.3.19) attribute.

Status: Success indicates the request is completed.

See also EXISTS (7.2.6), DISCARD_IMMEDIATELY_RETURNS (7.3.19), DISCARD_IMMEDIATELY_CAPABLE (7.3.18).

7.2.6 NVM.BLOCK.EXISTS

Requirement: mandatory if EXISTS_CAPABLE (see 7.3.12) is true

An NVM device may allocate storage through a thin provisioning mechanism or one of the discard actions. As a result, a block can exist in one of three states:

- **Mapped**: the block has had data written to it
- **Unmapped**: the block has not been written, and there is no memory allocated
- **Allocated**: the block has not been written, but has memory allocated to it

The EXISTS action allows the NVM user to determine if a block has been allocated.

Inputs: an LBA

Output: the state (mapped, unmapped, or allocated) for the input block

Result: the status of the action

7.2.7 NVM.BLOCK.SCAR

Requirement: mandatory if SCAR_CAPABLE (see 7.3.13) is true

This action allows an application to request that subsequent reads from any of the blocks in the address range will cause an error. This action uses an implementation-dependent means to insure that all future reads to any given block from the scarred range will cause an error until new data is stored to any given block in the range. A block stays scarred until it is updated by a write operation.

Inputs: reference to a block volume, starting offset, length

Outputs: status

Relevant attributes:

NVM.BLOCK.SCAR_CAPABLE (7.3.13) – Indicates that the SCAR action is supported.

7.3 Attributes

7.3.1 Attributes that apply across multiple modes

The following attributes apply to NVM.BLOCK mode as well as other modes.

NVM.COMMON.SUPPORTED_MODES (see 6.12.1)

7.3.2 NVM.BLOCK.ATOMIC_WRITE_CAPABLE

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.BLOCK.ATOMIC_WRITE action.

7.3.3 NVM.BLOCK.ATOMIC_WRITE_MAX_DATA_LENGTH

Requirement: mandatory if ATOMIC_WRITE_CAPABLE (see 7.3.1) is true.

ATOMIC_WRITE_MAX_DATA_LENGTH is the maximum length of data that can be transferred by an ATOMIC_WRITE action.

7.3.4 NVM.BLOCK.ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY

Requirement: mandatory if ATOMIC_WRITE_CAPABLE (see 7.3.1) is true.

ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY is the granularity of the starting memory address for an ATOMIC_WRITE action. Address inputs to ATOMIC_WRITE shall be evenly divisible by ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY.

7.3.5 NVM.BLOCK.ATOMIC_WRITE_LENGTH_GRANULARITY

Requirement: mandatory if ATOMIC_WRITE_CAPABLE (see 7.3.1) is true.

ATOMIC_WRITE_LENGTH_GRANULARITY is the granularity of the length of data transferred by an ATOMIC_WRITE action. Length inputs to ATOMIC_WRITE shall be evenly divisible by ATOMIC_WRITE_LENGTH_GRANULARITY.

7.3.6 NVM.BLOCK.ATOMIC_MULTIWRITE_CAPABLE

Requirement: mandatory

ATOMIC_MULTIWRITE_CAPABLE indicates that the implementation is capable of the NVM.BLOCK.ATOMIC_MULTIWRITE action.

7.3.7 NVM.BLOCK.ATOMIC_MULTIWRITE_MAX_IOS

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6) is true

ATOMIC_MULTIWRITE_MAX_IOS is the maximum length of the number of IOs (i.e., the size of the Property Group List) that can be transferred by an ATOMIC_MULTIWRITE action.

7.3.8 NVM.BLOCK.ATOMIC_MULTIWRITE_MAX_DATA_LENGTH

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6) is true

ATOMIC_MULTIWRITE_MAX_DATA_LENGTH is the maximum length of data that can be transferred by an ATOMIC_MULTIWRITE action.

7.3.9 NVM.BLOCK.ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6) is true

ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY is the granularity of the starting address of ATOMIC_MULTIWRITE inputs. Address inputs to ATOMIC_MULTIWRITE shall be evenly divisible by ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY.

7.3.10 NVM.BLOCK.ATOMIC_MULTIWRITE_LENGTH_GRANULARITY

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 7.3.6) is true

ATOMIC_MULTIWRITE_LENGTH_GRANULARITY is the granularity of the length of ATOMIC_MULTIWRITE inputs. Length inputs to ATOMIC_MULTIWRITE shall be evenly divisible by ATOMIC_MULTIWRITE_LENGTH_GRANULARITY.

7.3.11 NVM.BLOCK.WRITE_ATOMICITY_UNIT

Requirement: mandatory

If a write is submitted of this size or less, the caller is guaranteed that if power is lost before the data is completely written, then the NVM device shall ensure that all the logical blocks affected by the operation contain the data that was present before the device server started processing the write operation (i.e., the old data, as if the atomic write operation had no effect).

If the NVM device can't assure that at least one LOGICAL_BLOCKSIZE (see 7.3.14) extent can be written atomically, WRITE_ATOMICITY_UNIT shall be set to zero.

The unit is NVM.BLOCK.LOGICAL_BLOCKSIZE (see 7.3.14).

7.3.12 NVM.BLOCK.EXISTS_CAPABLE

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.BLOCK.EXISTS action.

7.3.13 NVM.BLOCK.SCAR_CAPABLE

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.BLOCK.SCAR (see 7.2.7) action.

7.3.14 NVM.BLOCK.LOGICAL_BLOCK_SIZE

Requirement: mandatory

LOGICAL_BLOCK_SIZE is the smallest unit of data (in bytes) that may be logically read or written from the NVM volume.

7.3.15 NVM.BLOCK.PERFORMANCE_BLOCK_SIZE

Requirement: mandatory

PERFORMANCE_BLOCK_SIZE is the recommended granule (in bytes) the caller should use in I/O requests for optimal performance; starting addresses and lengths should be multiples of this attribute. For example, this attribute may help minimizing device-implemented read/modify/write behavior.

7.3.16 NVM.BLOCK.ALLOCATION_BLOCK_SIZE

Requirement: mandatory

ALLOCATION_BLOCK_SIZE is the recommended granule (in bytes) for allocation and alignment of data. Allocations smaller than this attribute (even if they are multiples of LOGICAL_BLOCK_SIZE) may work, but may not yield optimal lifespan.

7.3.17 NVM.BLOCK.DISCARD_IF_YOU_CAN_CAPABLE

Requirement: mandatory

DISCARD_IF_YOU_CAN_CAPABLE shall be set to true if the implementation supports DISCARD_IF_YOU_CAN.

7.3.18 NVM.BLOCK.DISCARD_IMMEDIATELY_CAPABLE

Requirement: mandatory

Returns true if the implementation supports DISCARD_IMMEDIATELY.

7.3.19 NVM.BLOCK.DISCARD_IMMEDIATELY_RETURNS

Requirement: mandatory if DISCARD_IMMEDIATELY_CAPABLE (see 7.3.18) is true

The value returned from read operations to blocks specified by a DISCARD_IMMEDIATELY action with no subsequent write operations. The possible values are:

- A value that is returned to each read of an unmapped block (see NVM.BLOCK.EXISTS 7.2.6) until the next write action
- Unspecified

7.3.20 NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE

Requirement: mandatory

FUNDAMENTAL_BLOCK_SIZE is the number of bytes that may become unavailable due to an error on an NVM device.

A zero value means that the device is unable to provide a guarantee on the number of adjacent bytes impacted by an error.

This attribute is relevant when the device does not support write atomicity.

If FUNDAMENTAL_BLOCK_SIZE is smaller than LOGICAL_BLOCK_SIZE (see 7.3.14), an application may organize data in terms of FUNDAMENTAL_BLOCK_SIZE to avoid certain torn write behavior. If FUNDAMENTAL_BLOCK_SIZE is larger than LOGICAL_BLOCK_SIZE, an application may organize data in terms of FUNDAMENTAL_BLOCK_SIZE to assure two key data items do not occupy an extent that is vulnerable to errors.

7.4 Use cases

7.4.1 Flash as cache use case

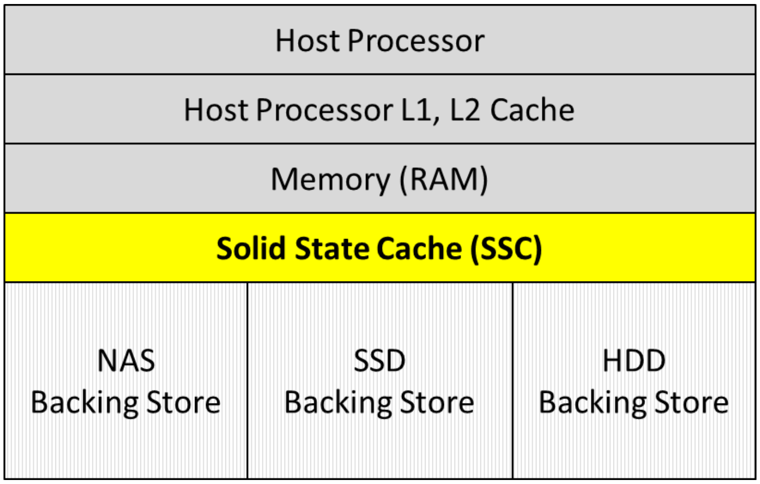
Purpose/triggers:

Use Flash based NVM as a data cache.

Scope/context:

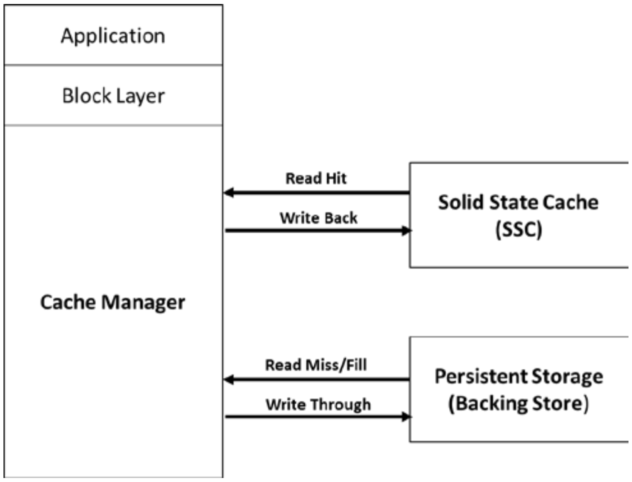
Flash memory’s fast random I/O performance and non-volatile characteristic make it a good candidate as a Solid State Cache device (SSC). This use case is described in Figure 7 SSC in a storage stack.

Figure 7 SSC in a storage stack



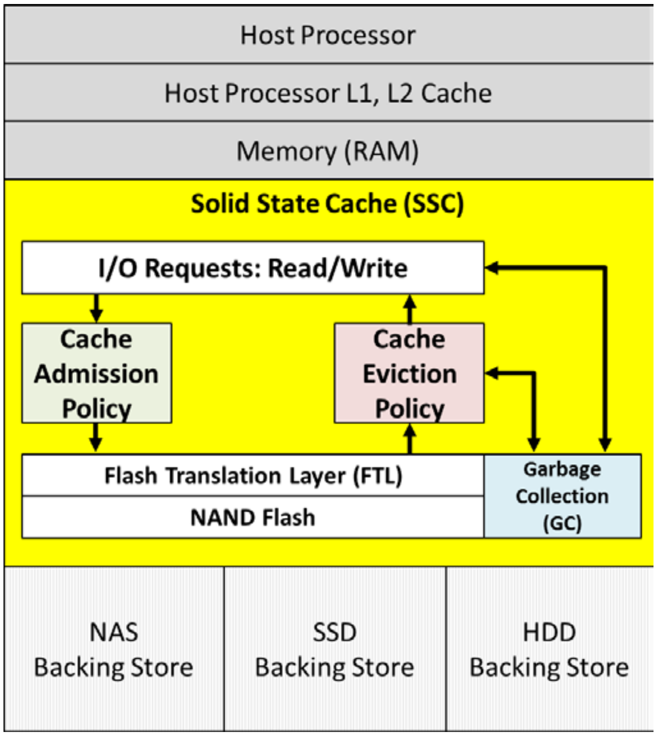
A possible software application is shown in Figure 8 SSC software cache application. In this case, the cache manager employs the Solid State Cache to improve caching performance and to maintain persistence and cache coherency across power fail.

Figure 8 SSC software cache application



It is also possible to use an enhanced SSC to perform some of the functions that the cache manager must normally contend with as shown in Figure 9 SSC with caching assistance.

Figure 9 SSC with caching assistance



In this use case, the Solid State Cache (SSC) provides a sparse address space that may be much larger than the amount of physical NVM memory and manages the cache through its own admission and eviction policies. The backing store is used to persist the data when the cache becomes full. As a result, the block state for each block of virtual storage in the cache must be maintained by the SSC. The SSC must also present a consistent cache interface that can persist the cached data across a power fail and never returns stale data.

In either of these cases, two important extensions to existing storage commands must be present:

> **Eviction:** An explicit eviction mechanism is required to invalidate cached data in the SSC to allow the cache manager to precisely control the contents of the SSC. This means that the SSC must insure that the eviction is durable before completing the request. This mechanism is generally referred to as a persistent trim. This is the NVM.BLOCK.DISCARD_IMMEDIATELY functionality.
>
> **Exists:** The EXISTS action allows the cache manager to determine the state of a block, or of a range of blocks, in the SSC. This action is used to test for the presence of data in the cache, or to determine which blocks in the SSC are dirty and need to be flushed to backing storage. This is the NVM.BLOCK.EXISTS functionality.

The most efficient mechanism for a cache manager would be to simply read the requested data from the SSC which would the return either the data or an error indicated that the requested data was not in the cache. This approach is problematic, since most storage drivers and software require reads to be successful and complete by returning data - not an error. Device that return errors for normal read operations are usually put into an offline state by the system drivers. Further, the data that a read returns must be consistent from one read operation to the next, provided that no intervening writes occur. As a result, a two stage process must be used by the cache manager. The cache manager first issues an EXISTS action to determine if the requested data is present in the cache. Based on the result, the cache manager decides whether to read the data from the SSC or from the backing storage.

Success scenario:

The requested data is successfully read from or written to the SSC.

See also:

- 7.2.5 NVM.BLOCK.DISCARD_IMMEDIATELY
- 7.2.6 NVM.BLOCK.EXISTS
- Ptrim() + Exists(): Exposing New FTL Primitives to Applications, David Nellans, Michael Zappe, Jens Axboe, David Flynn, 2011 Non-Volatile Memory Workshop. See: http://david.nellans.org/files/NVMW-2011.pdf
- FlashTier: a Lightweight, Consistent, and Durable Storage Cache, Mohit Saxena, Michael M. Swift and Yiying Zhang, University of Wisconsin-Madison. See: http://pages.cs.wisc.edu/~swift/papers/eurosys12_flashtier.pdf

  HEC: Improving Endurance of High Performance Flash-based Cache Devices, Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, Swaminathan Sundararaman, Robert Wood, Fusion-io, Inc., SYSTOR ’13, June 30 - July 02 2013, Haifa, Israel
- Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory, Eunji Lee, Hyokyung Bahn, and Sam H. Noh. See: https://www.usenix.org/system/files/conference/fast13/fast13-final114_0.pdf

7.4.2 SCAR use case

Purpose/triggers:

Demonstrate the use of the SCAR action

Scope/context:

This generic use case for SCAR involves two processes.

- The "detect block errors process" detects errors in certain NVM blocks, and uses SCAR to communicate to other processes that the contents of these blocks cannot be reliably read, but can be safely re-written.
- The "recover process" sees the error reported as the result of SCAR. If this process can regenerate the contents of the block, the application can continue with no error.

For this use case, the "detect block errors process" is a RAID component doing a background scan of NVM blocks. In this case, the NVM is not in a redundant RAID configuration so block READ errors can't be transparently recovered. The "recover process" is a cache component using the NVM as a cache for RAID volumes. Upon receipt of the SCAR error on a read, this component evaluates whether the block contents also reside on the cached volume; if so, it can copy the corresponding volume block to the NVM. This write to NVM will clear the SCAR error condition.

Preconditions:

The "detect block errors process" detected errors in certain NVM blocks, and used SCAR to mark these blocks.

Success scenario:

1. The cache manager intercepts a read request from an application
2. The read request to the NVM cache returns a status indicating the requested blocks have been marked by a SCAR action
3. The cache manager uses an implementation-specific technique and determines the blocks marked by a SCAR are also available on the cached volume
4. The cache manager copies the blocks from the cached volume to the NVM
5. The cache manager returns the requested block to the application with a status indicating the read succeeded

Postconditions:

The blocks previously marked with a SCAR action have been repaired.

Failure Scenario:

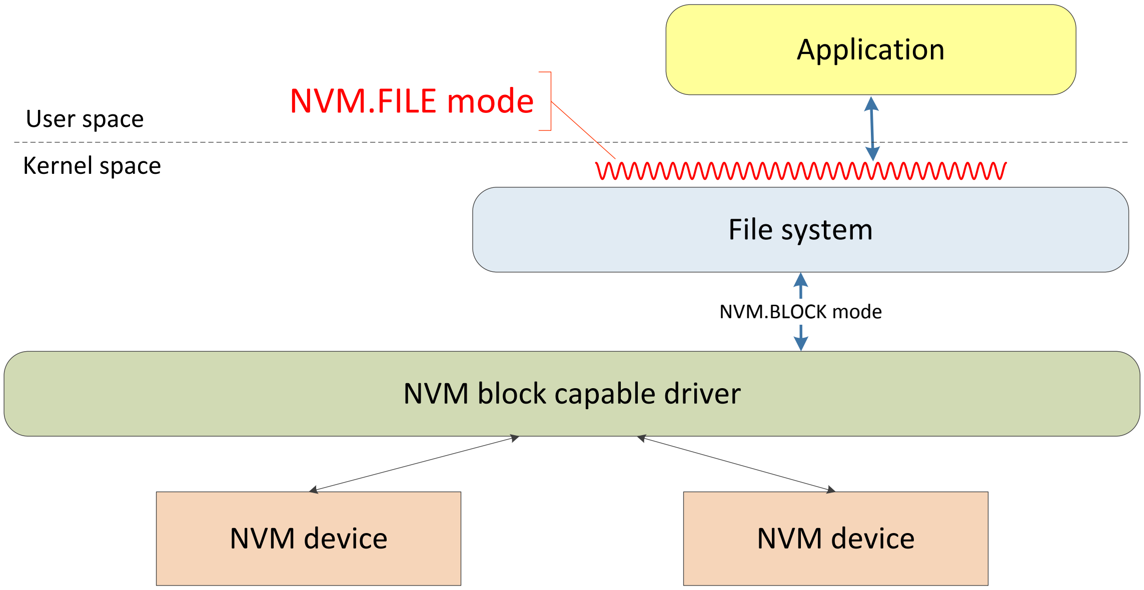
1. In Success Scenario step 3 or 4, the cache manager discovers the corresponding blocks on the volume are invalid or cannot be read.
2. The cache manager returns a status to the application indicating the blocks cannot be read.

8 NVM.FILE mode

8.1 Overview

NVM.FILE mode addresses NVM-specification extensions to native file I/O behavior (the approach to I/O used by most applications). Support for NVM.FILE mode requires that the NVM solution ought to support all behavior not covered in this section consistently with the native operating system behavior for native block devices.

Figure 10 NVM.FILE mode example



8.1.1 Discovery and use of atomic write features

Atomic Write features in NVM.FILE mode are available to block-optimized applications (see 7.1.4.2 Block-optimized applications).

8.1.2 The discovery of granularities

The NVM.FILE mode exposes the same granularity attributes as NVM.BLOCK.

8.1.3 Relationship between native file APIs and NVM.BLOCK.DISCARD

NVM.FILE mode does not define specific action that cause TRIM/DISCARD behavior. File systems may invoke NVM.BLOCK DISCARD actions when native operating system APIs (such as POSIX truncate or Windows SetEndOfFile).

8.2 Actions

8.2.1 Actions that apply across multiple modes

The following actions apply to NVM.FILE mode as well as other modes.

NVM.COMMON.GET_ATTRIBUTE (see 6.11.1)
NVM.COMMON.SET_ATTRIBUTE (see 6.11.2)

8.2.2 NVM.FILE.ATOMIC_WRITE

Requirement: mandatory if ATOMIC_WRITE_CAPABLE (see 8.3.2) is true

Block-optimized applications may use ATOMIC_WRITE to assure consistent behavior during a failure condition. This specification does not specify the order in which this action occurs relative to other I/O operations, including other ATOMIC_WRITE and ATOMIC_MULTIWRITE actions. This specification does not specify when the data written becomes visible to other threads.

The inputs, outputs, and error conditions are similar to those for NVM.BLOCK.ATOMIC_WRITE, but typically the application provides file names and file relative block addresses rather than device name and LBA.

Relevant attributes:

ATOMIC_WRITE_MAX_DATA_LENGTH
ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY
ATOMIC_WRITE_LENGTH_GRANULARITY
ATOMIC_WRITE_CAPABLE

8.2.3 NVM.FILE.ATOMIC_MULTIWRITE

Requirement: mandatory if ATOMIC_MULTIWRITE_CAPABLE (see 8.3.6) is true

Block-optimized applications may use ATOMIC_MULTIWRITE to assure consistent behavior during a failure condition. This action allows a caller to write non-adjacent extents atomically. The caller of ATOMIC_MULTIWRITE provides properties defining memory and block extents; all of the extents are written as a single atomic operation. This specification does not specify the order in which this action occurs relative to other I/O operations, including other ATOMIC_WRITE and ATOMIC_MULTIWRITE actions. This specification does not specify when the data written becomes visible to other threads.

The inputs, outputs, and error conditions are similar to those for NVM.BLOCK.ATOMIC_MULTIWRITE, but typically the application provides file names and file relative block addresses rather than device name and LBA.

Relevant attributes:

ATOMIC_MULTIWRITE_MAX_IOS
ATOMIC_MULTIWRITE_MAX_DATA_LENGTH
ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY
ATOMIC_MULTIWRITE_LENGTH_GRANULARITY
ATOMIC_MULTIWRITE_CAPABLE

8.3 Attributes

Some attributes share behavior with their NVM.BLOCK counterparts. NVM.FILE attributes are provided because the actual values may change due to the implementation of the file system.

8.3.1 **Attributes that apply across multiple modes**

The following attributes apply to NVM.FILE mode as well as other modes.

NVM.COMMON.SUPPORTED_MODES (see 6.12.1)

NVM.COMMON.FILE_MODE (see 6.12.2)

8.3.2 **NVM.FILE.ATOMIC_WRITE_CAPABLE**

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.BLOCK.ATOMIC_WRITE action.

8.3.3 **NVM.FILE.ATOMIC_WRITE_MAX_DATA_LENGTH**

Requirement: mandatory

ATOMIC_WRITE_MAX_DATA_LENGTH is the maximum length of data that can be transferred by an ATOMIC_WRITE action.

8.3.4 **NVM.FILE.ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY**

Requirement: mandatory

ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY is the granularity of the starting memory address for an ATOMIC_WRITE action. Address inputs to ATOMIC_WRITE shall be evenly divisible by ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY.

8.3.5 **NVM.FILE.ATOMIC_WRITE_LENGTH_GRANULARITY**

Requirement: mandatory

ATOMIC_WRITE_LENGTH_GRANULARITY is the granularity of the length of data transferred by an ATOMIC_WRITE action. Length inputs to ATOMIC_WRITE shall be evenly divisible by ATOMIC_WRITE_STARTING_ADDRESS_GRANULARITY.

8.3.6 **NVM.FILE.ATOMIC_MULTIWRITE_CAPABLE**

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.FILE.ATOMIC_MULTIWRITE action.

8.3.7 **NVM.FILE.ATOMIC_MULTIWRITE_MAX_IOS**

Requirement: mandatory

ATOMIC_MULTIWRITE_MAX_IOS is the maximum length of the number of IOs (i.e., the size of the Property Group List) that can be transferred by an ATOMIC_MULTIWRITE action.

8.3.8 NVM.FILE.ATOMIC_MULTIWRITE_MAX_DATA_LENGTH

Requirement: mandatory

ATOMIC_MULTIWRITE_MAX_DATA_LENGTH is the maximum length of data that can be transferred by an ATOMIC_MULTIWRITE action.

8.3.9 NVM.FILE.ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY

Requirement: mandatory

ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY is the granularity of the starting address of ATOMIC_MULTIWRITE inputs. Address inputs to ATOMIC_MULTIWRITE shall be evenly divisible by ATOMIC_MULTIWRITE_STARTING_ADDRESS_GRANULARITY.

8.3.10 NVM.FILE.ATOMIC_MULTIWRITE_LENGTH_GRANULARITY

Requirement: mandatory

ATOMIC_MULTIWRITE_LENGTH_GRANULARITY is the granularity of the length of ATOMIC_MULTIWRITE inputs. Length inputs to ATOMIC_MULTIWRITE shall be evenly divisible by ATOMIC_MULTIWRITE_LENGTH_GRANULARITY.

8.3.11 NVM.FILE.WRITE_ATOMICITY_UNIT

See 7.3.11 NVM.BLOCK.WRITE_ATOMICITY_UNIT

8.3.12 NVM.FILE.LOGICAL_BLOCK_SIZE

See 7.3.14 NVM.BLOCK.LOGICAL_BLOCK_SIZE

8.3.13 NVM.FILE. PERFORMANCE_BLOCK_SIZE

See 7.3.15 NVM.BLOCK.PERFORMANCE_BLOCK_SIZE

8.3.14 NVM.FILE.LOGICAL_ALLOCATION_SIZE

See 7.3.16 NVM.BLOCK.ALLOCATION_BLOCK_SIZE

8.3.15 NVM.FILE.FUNDAMENTAL_BLOCK_SIZE

See 7.3.20 NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE

8.4 Use cases

8.4.1 Block-optimized application updates record

Update a record in a file without using a memory-mapped file

Purpose/triggers:

An application using block NVM updates an existing record. The application requests that the file system bypass cache; the application conforms to native API requirements when bypassing cache – this may mean that read and write actions must use multiples of a page cache size. For simplicity, this application uses fixed size records. The record size is defined by application data considerations, not disk or page block sizes. The application factors in the PERFORMANCE_BLOCK_SIZE granularity to avoid device-side inefficiencies such as read/modify/write.

Scope/context:

Block NVM context; this shows basic behavior.

Preconditions:

- The administrator created a file and provided its name to the application; this name is accessible to the application – perhaps in a configuration file
- The application has populated the contents of this file
- The file is not in use at the start of this use case (no sharing considerations)

Inputs:

The content of the record, the location (relative to the file) where the record resides

Success scenario:

1) The application uses the native OPEN action, passing in the file name and specifying appropriate options to bypass the file system cache
2) The application acquires the device's optimal I/O granule size by using the GET_ATTRIBUTE action for the PERFORMANCE_BLOCK_SIZE.
3) The application allocates sufficient memory to contain all of the blocks occupied by the record to be updated.
    a. The application determines the offset within the starting block of the record and uses the length of the block to determine the number of partial blocks.
    b. The application allocates sufficient memory for the record plus enough additional memory to accommodative any partial blocks.
    c. If necessary, the memory size is increased to assure that the starting address and length read and write actions are multiples of PERFORMANCE_BLOCK_SIZE.
4) The application uses the native READ action to read the record by specifying the starting disk address and the length (the same length as the allocated memory buffer). The application also provides the allocated memory address; this is where the read action will put the record.
5) The application updates the record in the memory buffer per the inputs
6) The application uses the native write action to write the updated block(s) to the same disk location they were read from.
7) The application uses the native file SYNC action to assure the updated blocks are written to the persistence domain

8) The application uses the native CLOSE action to clean up.

Failure Scenario 1:

The native read action reports a hardware error. If the unreadable block corresponds to blocks being updated, the application may attempt recovery (write/read/verify), or preventative maintenance (scar the unreadable blocks). If the unreadable blocks are needed for a read/modify/write update and the application lacks an alternate source; the application may inform the user that an unrecoverable hardware error has occurred.

Failure Scenario 2:

The native write action reports a hardware error. The application may be able to recover by rewriting the block. If the rewrite fails, the application may be able to scar the bad block and write to a different location.

Postconditions:

The record is updated.

8.4.2 Atomic write use case

Purpose/triggers:

Used by a block-optimized application (see Block-optimized applications) striving for durability of on-disk data

Scope/context:

Assure a record is written to disk in a way that torn writes can be detected and rolled back (if necessary). If the device supports atomic writes, they will be used. If not, a double write buffer is used.

Preconditions:

The application has taken steps (based on NVM.BLOCK attributes) to assure the record being written has an optimal memory starting address, starting disk LBA and length.

Success scenario:

- Use GET_ATTRIBUTE to determine whether the device is ATOMIC_WRITE_CAPABLE (or ATOMIC_MULTIWRITE_CAPABLE)
- Is so, use the appropriate atomic write action to write the record to NVM
- If the device does not support atomic write, then
  - Write the page to the double write buffer
  - Wait for the write to complete
  - Write the page to the final destination
- At application startup, if the device does not support atomic write
  - Scan the double write buffer and for each valid page in the buffer check if the page in the data file is valid too.

Postconditions:

After application startup recovery steps, there are no inconsistent records on disk after a failure caused the application (and possibly system) to restart.

8.4.3 Block and File Transaction Logging

Purpose/Triggers:

An application developer wishes to implement a transaction log that maintains data integrity through system crashes, system resets, and power failures. The underlying storage is block-granular, although it may be accessed via a file system that simulates byte-granular access to files.

Scope/Context:

NVM.BLOCK or NVM.FILE (all the NVM.BLOCK attributes mentioned in the use case are also defined for NVM.FILE mode).

For notational convenience, this use case will use the term “file” to apply to either a file in the conventional sense which is accessed through the NVM.FILE interface, or a specific subset of blocks residing on a block device which are accessed through the NVM.BLOCK interface.

Inputs:

- A set of changes to the persistent state to be applied as a single transaction.
- The data and log files.

Outputs:

- An indication of transaction commit or abort

Postconditions:

- If an abort indication was returned, the data was not committed and the previous contents have not been modified.
- If a commit indication was returned, the data has been entirely committed.
- After a system crash, reset, or power failure followed by system restart and execution of the application transaction recovery process, the data has either been entirely committed or the previous contents have not been modified.

Success Scenario:

The application transaction logic uses a log file in combination with its data file to atomically update the persistent state of the application. The log may implement a before-image log or a write-ahead log. The application transaction logic should configure itself to handle torn or interrupted writes to the log or data files.

8.4.3.1 NVM.BLOCK.WRITE_ATOMICITY_UNIT >= 1

If the NVM.BLOCK.WRITE_ATOMICITY_UNIT is one or greater, then writes of a single logical block cannot be torn or interrupted.

In this case, if the log or data record size is less than or equal to the NVM.BLOCK.LOGICAL_BLOCK_SIZE, the application need not handle torn or interrupted writes to the log or data files.

If the log or data record size is greater than the NVM.BLOCK.LOGICAL_BLOCK_SIZE, the application should be prepared to detect a torn write of the record and either discard or recover such a torn record during the recovery process. One common way of detecting such a torn write is for the application to compute hash of the record and record the hash in the record. Upon reading the record, the application re-computes the hash and compares it with the recorded hash; if they do not match, the record has been torn. Another method is for the application to insert the transaction identifier within each logical block. Upon reading the record, the application compares the transaction identifiers in each logical block; if they do not match, the record has been torn. Another method is for the application to use the NVM.BLOCK.ATOMIC_WRITE action to perform the writes of the record.

8.4.3.2 NVM.BLOCK.WRITE_ATOMICITY_UNIT = 0

If the NVM.BLOCK.WRITE_ATOMICITY_UNIT is zero, then writes of a single logical block can be torn or interrupted and the application should handle torn or interrupted writes to the log or data files.

In this case, if a logical block were to contain data from more than one log or data record, a torn or interrupted write could corrupt a previously-written record. To prevent propagating an error beyond the record currently being written, the application aligns the log or data records with the NVM.BLOCK.LOGICAL_BLOCK_SIZE and pads the record size to be an integral multiple of NVM.BLOCK.LOGICAL_BLOCK_SIZE. This prevents more than one record from residing in the same logical block and therefore a torn or interrupted write may only corrupt the record being written.

8.4.3.2.1 NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE >= NVM.BLOCK.LOGICAL_BLOCK_SIZE

If the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE is greater than or equal to the NVM.BLOCK.LOGICAL_BLOCK_SIZE, the application should be prepared to handle an interrupted write. An interrupted write results when the write of a single NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE unit is interrupted by a system crash, system reset, or power failure. As a result of an interrupted write, the NVM device may return an error when any of the logical blocks comprising the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE unit are read. (See also SQLite.org, *Powersafe Overwrite*, http://www.sqlite.org/psow.html.) This presents a danger to the integrity of previously written records that, while residing in differing logical blocks, share the same fundamental block. An interrupted write may prevent the reading of those previously written records in addition to preventing the read of the record in the process of being written.

One common way of protecting previously written records from damage due to an interrupted write is to align the log or data records with the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE and pad the record size to be an integral multiple of NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE. This prevents more than one record from residing in the same fundamental block. The application should be prepared to discard or recover the record if the NVM device returns an error when subsequently reading the record during the recovery process.

8.4.3.2.2 NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE < NVM.BLOCK.LOGICAL_BLOCK_SIZE

If the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE is less than the NVM.BLOCK.LOGICAL_BLOCK_SIZE, the application should be prepared to handle both interrupted writes and torn writes within a logical block.

As a result of an interrupted write, the NVM device may return an error when the logical block containing the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE unit which was being written at the time of the system crash, system reset, or power failure is subsequently read. The application should be prepared to discard or recover the record in the logical block if the NVM device returns an error when subsequently reading the logical block during the recovery process.

A torn write results when an integral number of NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE units are written to the NVM device but the entire NVM.BLOCK.LOGICAL_BLOCK_SIZE has not been written. In this case, the NVM device may not return an error when the logical block is read. The application should therefore be prepared to detect a torn write of a logical block and either discard or recover such a torn record during the recovery process. One common way of detecting such a torn write is for the application to compute a hash of the record and record the hash in the record. Upon reading the record, the application re-computes the hash and compares it with the recorded hash; if they do not match, a logical block within the record has been torn. Another method is for the application to insert the transaction identifier within each NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE unit. Upon reading the record, the application compares the transaction identifiers in each NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE unit; if they do not match, the logical block has been torn.

8.4.3.2.3 NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE = 0

If the NVM.BLOCK.FUNDAMENTAL_BLOCK_SIZE is zero, the application lacks sufficient information to handle torn or interrupted writes to the log or data files.

Failure Scenarios:

Consider the recovery of an error resulting from an interrupted write on a device where the NVM.BLOCK.WRITE_ATOMICITY_UNIT is zero. This error may be persistent and may be returned whenever the affected block is read. To repair this error, the application should be prepared to overwrite such a block.

One common way of ensuring that the application will overwrite a block is by assigning it to the set of internal free space managed by the application, which is never read and is available to be allocated and overwritten at some point in the future. For example, the block may be part of a circular log. If the block is marked as free, the transaction log logic will eventually allocate and overwrite that block as records are written to the log.

Another common way is to record either a before-image or after-image of a data block in a log. During recovery after a system crash, system reset, or power failure, the application replays the records in the log and overwrites the data block with either the before-image contents or the after-image contents.

See also:

- SQLite.org, *Atomic Commit in SQLite*, http://www.sqlite.org/atomiccommit.html
- SQLite.org, *Powersafe Overwrite*, http://www.sqlite.org/psow.html
- SQLite.org, *Write-Ahead Logging*, http://www.sqlite.org/wal.html

9 NVM.PM.VOLUME mode

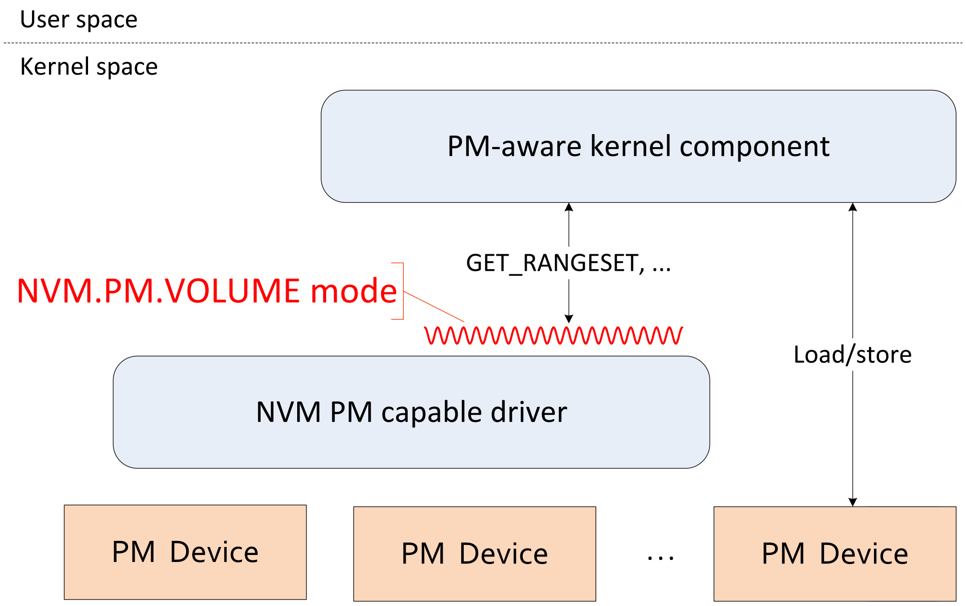
9.1 Overview

NVM.PM.VOLUME mode describes the behavior to be consumed by operating system abstractions such as file systems or pseudo-block devices that build their functionality by directly accessing persistent memory. NVM.PM.VOLUME mode provides a software abstraction (a PM volume) for persistent memory hardware and profiles functionality for operating system components including:

- list of physical address ranges associated with each PM volume

The PM volume provides memory mapped capability in a fashion that traditional CPU load and store operations are possible. This PM volume may be provided via the memory channel of the CPU or via a PCIe memory mapping or other methods. Note that there should not be a requirement for an operating system context switch for access to the PM volume.

Figure 11 NVM.PM.VOLUME mode example



9.2 Actions

9.2.1 Actions that apply across multiple modes

The following actions apply to NVM.PM.VOLUME mode as well as other modes.

NVM.COMMON.GET_ATTRIBUTE (see 6.11.1)
NVM.COMMON.SET_ATTRIBUTE (see 6.11.2)

9.2.2 NVM.PM.VOLUME.GET_RANGESET

Requirement: mandatory

The purpose of this action is to return a set of processor physical address ranges (and relate properties) representing all of the content for the identified volume.

When interpreting the set of physical addresses as a contiguous, logical address range; the data underlying that logical address range will always be the same and in the same sequence across PM volume instantiations.

Due to physical memory reconfiguration, the number and sizes of ranges may change in successive get ranges calls, however the total number of bytes in the sum of the ranges does not change, and the order of the bytes spanning all of the ranges does not change. The space defined by the list of ranges can always be addressed relative to a single base which represents the beginning of the first range.

Input: a reference to the PM volume

Returns a Property Group List (see 4.4.5) where the properties are:

- starting physical address (byte address)
- length (in bytes)
- connection type – see below
- sync type – see below

For this revision of the specification, the following values (in text) are valid for connection type:

- *“memory”*: for persistent memory attached to a system memory channel
- “*PCIe*”: for persistent memory attached to a PCIe extension bus

For this revision of the specification, the following values (in text) are valid for sync type:

- “none”: no device-specific sync behavior is available – implies no entry to NVM.PM.VOLUME implementation is required for flushing
- “VIRTUAL_ADDRESS_SYNC”: the caller needs to use VIRTUAL_ADDRESS_SYNC (see 9.2.3) to assure sync is durable
- “PHYSICAL_ADDRESS_SYNC”: the caller needs to use PHYSICAL_ADDRESS_SYNC (see 9.2.4) to assure sync is durable

9.2.3 **NVM.PM.VOLUME.VIRTUAL_ADDRESS_SYNC**

Requirement: optional

The purpose of this action is to invoke device-specific actions to synchronize persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain. VIRTUAL_ADDRESS_SYNC is used by a caller that knows the addresses in the input range are virtual memory addresses.

Input: virtual address and length (range)

See also: PHYSICAL_ADDRESS_SYNC

9.2.4 **NVM.PM.VOLUME.PHYSICAL_ADDRESS_SYNC**

Requirement: optional

The purpose of this action is to synchronize persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain. This action is used by a caller that knows the addresses in the input range are physical memory addresses.

See also: VIRTUAL_ADDRESS_SYNC

Input: physical address and length (range)

9.2.5 **NVM.PM.VOLUME.DISCARD_IF_YOU_CAN**

Requirement: mandatory if DISCARD_IF_YOU_CAN_CAPABLE (see 9.3.6) is true

This action notifies the NVM device that the input range (volume offset and length) are no longer needed by the caller. This action may not result in any action by the device, depending on the implementation and the internal state of the device. This action is meant to allow the underlying device to optimize the data stored within the range. For example, the device can use this information in support of functionality like thin provisioning or wear-leveling**.**

Inputs: a range of addresses (starting address and length in bytes). The address shall be a logical memory address offset from the beginning of the volume.

Status: Success indicates the request is accepted but not necessarily acted upon.

9.2.6 **NVM.PM.VOLUME.DISCARD_IMMEDIATELY**

Requirement: mandatory if DISCARD_IMMEDIATELY_CAPABLE (see 9.3.7) is true

This action notifies the NVM device that the input range (volume offset and length) are no longer needed by the caller. Similar to DISCARD_IF_YOU_CAN, but the implementation is required to unmap the range before the next READ or WRITE action, even if garbage collection of the range has not occurred yet.

Inputs: a range of addresses (starting address and length in bytes). The address shall be a logical memory address offset from the beginning of the volume.

The values returned by subsequent read operations are specified by the DISCARD_IMMEDIATELY_RETURNS (see 9.3.8) attribute.

Status: Success indicates the request is completed.

9.2.7 **NVM.PM.VOLUME.EXISTS**

Requirement: mandatory if EXISTS_CAPABLE (see9.3.9) is true

A PM device may allocate storage through a thin provisioning mechanism or one of the discard actions. As a result, memory can exist in one of three states:

- **Mapped**: the range has had data written to it
- **Unmapped**: the range has not been written, and there is no memory allocated
- **Allocated**: the range has not been written, but has memory allocated to it

The EXISTS action allows the NVM user to determine if a range of bytes has been allocated.

Inputs: a range of bytes (starting byte address and length in bytes)

Output: a Property Group List (see 4.4.5) where the properties are the starting address, length and state. State is a string equal to “mapped”, “unmapped”, or “allocated”.

Result: the status of the action

9.3 Attributes

9.3.1 Attributes that apply across multiple modes

The following attributes apply to NVM.PM.VOLUME mode as well as other modes.

NVM.COMMON.SUPPORTED_MODES (see 6.12.1)

9.3.2 NVM.PM.VOLUME.VOLUME_SIZE

Requirement: mandatory

VOLUME_SIZE is the volume size in units of bytes. This shall be the same as the sum of the lengths of the ranges returned by the GET_RANGESETS action.

9.3.3 NVM.PM.VOLUME.INTERRUPTED_STORE_ATOMICITY

Requirement: mandatory

INTERRUPTED_STORE_ATOMICITY indicates whether the device supports power fail atomicity of store actions.

A value of true indicates that after a store interrupted by reset, power loss or system crash; upon restart the contents of persistent memory reflect either the state before the store or the state after the completed store. A value of false indicates that after a store interrupted by reset, power loss or system crash, upon restart the contents of memory may be such that subsequent loads may create exceptions depending on the value of the FUNDAMENTAL_ERROR_RANGE attribute (see 9.3.4).

9.3.4 NVM.PM.VOLUME.FUNDAMENTAL_ERROR_RANGE

Requirement: mandatory

FUNDAMENTAL_ERROR_RANGE is the number of bytes that may become unavailable due to an error on an NVM device.

This attribute is relevant when the device does not support write atomicity.

A zero value means that the device is unable to provide a guarantee on the number of adjacent bytes impacted by an error.

A caller may organize data in terms of FUNDAMENTAL_ERROR_RANGE to avoid certain torn write behavior.

9.3.5 NVM.PM.VOLUME.FUNDAMENTAL_ERROR_RANGE_OFFSET

Requirement: mandatory

The number of bytes offset from the beginning of a volume range (as returned by GET_RANGESET) before FUNDAMENTAL_ERROR_RANGE_SIZE intervals apply.

A fundamental error range is not required to start at a byte address evenly divisible by FUNDAMENTAL_ERROR_RANGE. FUNDAMENTAL_ERROR_RANGE_OFFSET shall be set to the difference between the starting byte address of a fundamental error range rounded down to a multiple of FUNDAMENTAL_ERROR_RANGE.

Figure 12 Zero range offset example depicts an implementation where fundamental error ranges start at bye address zero; the implementation shall return zero for FUNDAMENTAL_ERROR_RANGE_OFFSET.

Figure 12 Zero range offset example

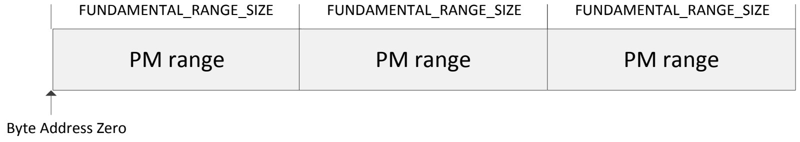
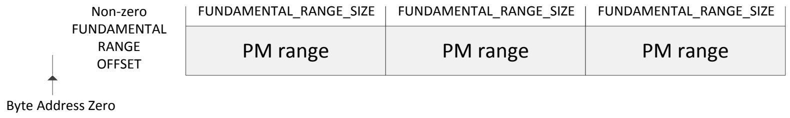


Figure 13 Non-zero range offset example depicts an implementation where fundamental error ranges start at a non-zero offset; the implementation shall return the difference between the starting byte address of a fundamental error range rounded down to a multiple of FUNDAMENTAL_ERROR_RANGE.

Figure 13 Non-zero range offset example



9.3.6 NVM.PM.VOLUME.DISCARD_IF_YOU_CAN_CAPABLE

Requirement: mandatory

Returns true if the implementation supports DISCARD_IF_YOU_CAN.

9.3.7 NVM.PM.VOLUME.DISCARD_IMMEDIATELY_CAPABLE

Requirement: mandatory

Returns true if the implementation supports DISCARD_IMMEDIATELY.

9.3.8 NVM.PM.VOLUME.DISCARD_IMMEDIATELY_RETURNS

Requirement: mandatory if DISCARD_IMMEDIATELY_CAPABLE (see 9.3.7) is true

The value returned from read operations to bytes specified by a DISCARD_IMMEDIATELY action with no subsequent write operations. The possible values are:

- A value that is returned to each load of bytes in an unmapped range until the next store action
- Unspecified

9.3.9 NVM.PM.VOLUME.EXISTS_CAPABLE

Requirement: mandatory

This attribute indicates that the implementation is capable of the NVM.PM.VOLUME.EXISTS action.

9.4 Use cases

9.4.1 Initialization steps for a PM-aware file system

Purpose/triggers:

Steps taken by a file system when a PM-aware volume is attached to a PM volume.

Scope/context:

NVM.PM.VOLUME mode

Preconditions:

- The administrator has defined a PM volume
- The administrator has completed one-time steps to create a file system on the PM volume

Inputs:

- A reference to a PM volume
- The name of a PM file system

Success scenario:

1. The file system issues a GET_RANGESET action to retrieve information about the ranges comprised by the PM volume.
2. The file system uses the range information from GET_RANGESET to determine physical address range(s) and offset(s) of the file system’s primary metadata (for example, the primary superblock), then loads appropriate metadata to determine no additional validity checking is needed.
3. The file system sets a flag in the metadata indicating the file system is mounted by storing the updated status to the appropriate location

    a. If the range containing this location requires VIRTUAL_ADDRESS_SYNC or PHYSICAL_ADDRESS_SYNC is needed (based on GET_RANGESET's sync mode property), the file system invokes the appropriate SYNC action

Postconditions:

The file system is usable by applications.

9.4.2 Driver emulates a block device using PM media

Purpose/triggers:

The steps supporting an application write action from a driver that emulates a block device using PM as media.

Scope/context:

NVM.PM.VOLUME mode

Preconditions:

PM layer FUNDAMENTAL_SIZE reported to driver is cache line size.

Inputs:

The application provides:

- the starting address of the memory (could be volatile) memory containing the data to write
- the length of the memory range to be written,
- an OS-specific reference to a block device (the virtual device backed by the PM volume),
- the starting LBA within that block device

Success scenario:

1. The driver registers with the OS-specific component to be notified of errors on the PM volume. PM error handling is outside the scope of this specification, but may be similar to what is described in (and above) Figure 15 Linux Machine Check error flow with proposed new interface.
2. Using information from a GET_RANGESET response, the driver splits the write operating into separate pieces if the target PM addresses (corresponding to application target LBAs) are in different ranges with different "sync type" values. For each of these pieces:
    a. Using information from a GET_RANGESET response, the driver determines the PM memory address corresponding to the input starting LBA, and performs a memory copy operation from the callers input memory to the PM
    b. The driver then performs a platform-specific flush operation
    c. Using information from a GET_RANGESET response, the driver invokes the PHYSICAL_ADDRESS_SYNC or VIRTUAL_ADDRESS_SYNC action as needed

3. No PM errors are reported by the PM error component, the driver reports that the write action succeeded.

Alternative Scenario 1:

In step 3 in the Success Scenario, the PM error component reports a PM error. The driver verifies that this error impacts the PM range being written and returns an error to the caller.

Postconditions:

The target PM range (i.e., the block device LBA range) is updated.

See also:

4.2.4 NVM block volume using PM hardware

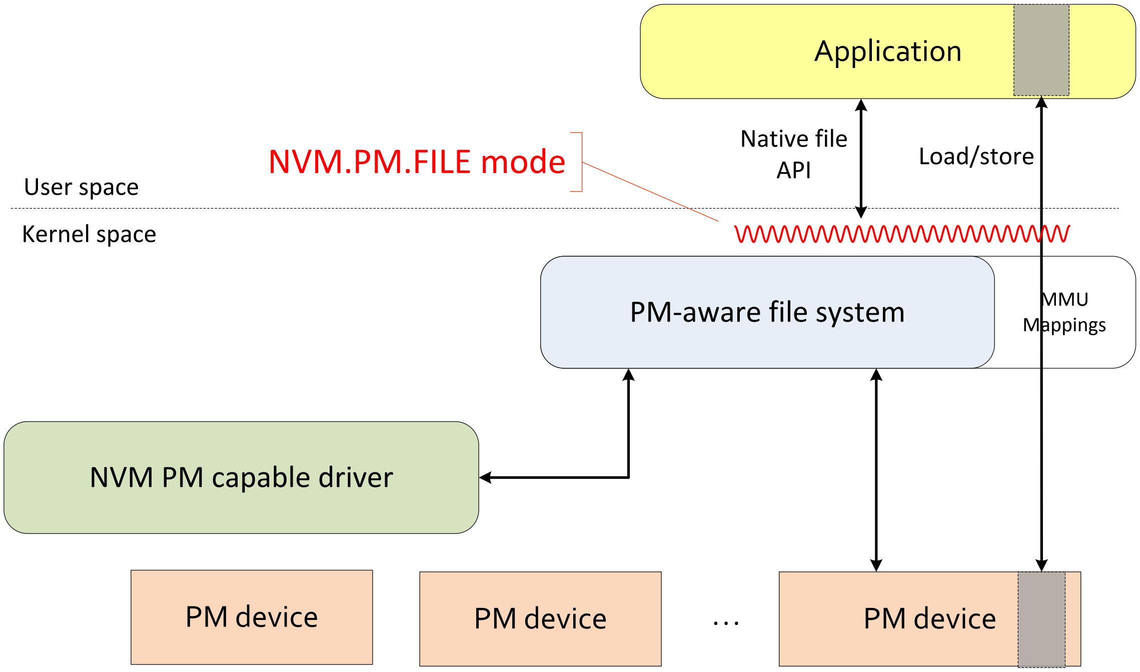
10 NVM.PM.FILE

10.1 Overview

The NVM.PM.FILE mode access provides a means for user space applications to directly access NVM as memory. Most of the standard actions in this mode are intended to be implemented as APIs exported by existing file systems. An NVM.PM.FILE implementation behaves similarly to preexisting file system implementations, with minor exceptions. This section defines extensions to the file system implementation to accommodate persistent memory mapping and to assure interoperability with NVM.PM.FILE mode applications.

Figure 14 NVM.PM.FILE mode example shows the context surrounding the point in a system (the red, wavy line) where the NVM.PM.FILE mode programming model is exposed by a PM-aware file system. A user space application consumes the programming model as is typical for current file systems. This example is not intended to preclude the possibility of a user space PM-aware file system implementation. It does, however presume that direct load/store access from user space occurs within a memory-mapped file context. The PM-aware file system interacts with an NVM PM capable driver to achieve any non-direct-access actions needed to discover or configure NVM. The PM-aware file system may access NVM devices for purposes such as file allocation, free space or other metadata management. The PM-aware file system manages additional metadata that describes the mapping of NVM device memory locations directly into user space.

Figure 14 NVM.PM.FILE mode example



Once memory mapping occurs, the behavior of the NVM.PM.FILE mode diverges from NVM.FILE mode because accesses to mapped memory are in the persistence domain as soon as they reach memory. This is represented in Figure 14 NVM.PM.FILE mode example by the arrow that passes through the "MMU Mappings" extension of the file system. As a result of persistent memory mapping, primitive ACID properties arise from CPU and memory infrastructure behavior as opposed to disk drive or traditional SSD behavior. Note that writes may still be retained within processor resident caches or memory controller buffers before they reach a persistence domain. As with NMV.FILE.SYNC, the possibility remains that memory mapped writes may become persistent before a corresponding NVM.PM.FILE.SYNC action.

The following actions have behaviors specific to the NVM.PM.FILE mode:

NVM.PM.FILE.MAP – Add a subset of a PM file to application's address space for load/store access.

NVM.PM.FILE.SYNC – Synchronize persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain.

10.1.1 Applications and PM Consistency

Applications (either directly or using services of a library) rely on CPU and kernel tools to achieve consistency of data in PM. These tools cause PM to exhibit certain data consistency properties enabling applications to operate correctly:

- PM is usable as volatile (not just persistent) memory
- Data residing in PM is consistent and durable even after a failure

Consistency is defined relative to the application’s objectives and design. For example, an application can utilize a write-ahead log (see SQLite.org, Write-Ahead Logging, http://www.sqlite.org/wal.html); when the application starts, recovery logic uses the write-ahead log to determine whether store operations completed and modifies data to achieve consistency. Similarly, durability objectives vary with applications. For database software, durability typically means that once a transaction has been committed it will remain so, even in the event of unexpected restarts. Other applications use a checkpoint mechanism other than transactions to define durable data states.

When persistence behavior is ignored, memory-mapped PM is expected to operate like volatile memory. Compiled code without durability expectations is expected to continue to run correctly.

This includes the following:

- Accessible through load, store, and atomic read/modify/write instructions
- Subject to existing processor cache coherency and “uncacheable” models (uncacheable models do not require a cache flush instruction to assure data is written to memory)
- Load, store, and atomic read/modify/write instructions retain their current semantics
    - Even when accessed from multiple threads
    - Even if locks or lock-protected data live in PM
- Able to use existing code (e.g., sort function) on PM data
- Applies for all data producers: CPU and, where relevant, I/O
- “Execute In Place” capability

- Supports pointers to PM data structures

At the implementation level, the behavior for fence instructions in libraries and thread visibility behavior is the same for data in PM as for data in volatile memory.

Two properties assure data is consistent and durable even after failures:

- Atomicity: some stores can’t be partly visible even after a failure
- Strict write ordering

EXAMPLE - This is a pseudo C language example of atomicity and strict ordering. In this example, msync implements NVM.PM.FILE.SYNC:

```
// a, a_end in PM
a[0] = foo();
msync(&(a[0]), …);
a_end = 0;
msync(&a_end, …);
. . .
n = a_end + 1;
a[n] = foo();
msync(&(a[n]), …);
a_end = n;
msync(&a_end, …);
```

For correctness of this example, the following assertions apply:

- a[0 .. a_end] always contains valid data, even after a failure in this code.
- a_end is written atomically to PM, so that the second store to a_end occurs no earlier than the store to a[n].

To achieve failure atomicity, aligned stores of fundamental data types (see 6.10) reach PM atomically. After a failure (allowed by the failure model), each such store is fully reflected in the resulting PM state or not at all.

At least two facilities are useful to achieve strict ordering:

- msync: Wait for all writes in a range to complete
- optimization using an intra-cache-line ordering guarantee.

To elaborate on these, `msync(address_range)` ensures that if any effects from code following the call are visible, then so are all stores to `address_range` (from any thread) which precede the call to msync .

With intra-cache-line ordering, thread-ordered stores to a single cache line become visible in PM in the order in which they are issued. The term “thread-ordered” refers to certain stores that are already known in today’s implementations to reach coherent cache in order, such as:

- x86 MOV
- some C11, C++11 atomic stores

- Java & C# volatile stores.

The CPU core and compiler do not reorder these. Within a single cache line, this order is normally preserved when the lines are evicted to PM. This last point is a critical consideration as the preservation of thread-ordered stores during eviction to PM is sometimes not guaranteed.

10.2 Actions

The following actions are mandatory for compliance with the NVM Programming Model NVM.PM.FILE mode.

10.2.1 Actions that apply across multiple modes

The following actions apply to NVM.PM.FILE mode as well as other modes.

NVM.COMMON.GET_ATTRIBUTE (see 6.11.1)
NVM.COMMON.SET_ATTRIBUTE (see 6.11.2)

10.2.2 Native file system actions

Native actions shall apply with unmodified syntax and semantics provided that they are compatible with programming model specific actions. This is intended to support traditional file operations allowing many applications to use PM without modification. This specifically includes mandatory implementation of the native synchronization of mapped files. As always, specific implementations may choose whether or not to implement optional native operations.

10.2.3 NVM.PM.FILE.MAP

Requirement: mandatory

The mandatory form of this action shall have the same syntax found in a pre-existing file system, preferably the operating system's native file map call. The specified subset of a PM file is added to application's address space for load/store access. The semantics of this action are unlike the native MAP action because NVM.PM.FILE.MAP causes direct load/store access. For example, the role of the page cache might be reduced or eliminated. This reduces or eliminates the consumption of volatile memory as a staging area for non-volatile data. In addition, by avoiding demand paging, direct access can enable greater uniformity of access time across volatile and non-volatile data.

PM mapped file operation may not provide the access time and modify time behavior typical of native file systems.

PM mapped file operation may not provide the normal semantics for the native file synchronization actions (e.g., POSIX fsync and fdatasync and Win32 FlushFileBuffers). If a file is mapped at the time when the native file synchronization action is invoked, the normal semantics apply. However if the file had been mapped, data had been written to the file through the map, the data had not been synchronized by use of the NVM.PM.FILE.SYNC action, the NVM.PM.FILE.OPTIMIZED_FLUSH action, the

NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY action, or the native mapped file sync action, and the mapping had been removed prior to the execution of the native file synchronization action, the action is not required to synchronize the data written to the map.

Requires NVM.PM.FILE.OPEN

Inputs: align with native operating system's map

Outputs: align with native operating system's map

Relevant Options:

All of the native file system options should apply.

> NVM.PM.FILE.MAP_SHARED (Mandatory) – This existing native option shall be supported by the NVM.PM.FILE.MAP action. This option indicates that user space processes other than the writer can see any changes to mapped memory immediately.

> NVM.PM.FILE.MAP_COPY_ON_WRITE (Optional)– This existing native option indicates that any write after mapping will cause a copy on write to volatile memory, or PM that is discarded during any type of restart. The copy is only visible to the writer. The copy is not folded back into PM during the sync command.

Relevant Attributes:

> NVM.PM.FILE.MAP_COPY_ON_WRITE_CAPABLE (see 10.3.2) - Native operating system map commands make a distinction between MAP_SHARED and MAP_COPY_ON_WRITE. Both are supported with native semantics under the NVM Programming Model. This attribute indicates whether the MAP_COPY_ON_WRITE mapping mode is supported. All NVM.PM.FILE.MAP implementations shall support the MAP_SHARED option.

Error handing for mapped ranges of persistent memory is unlike I/O, in that there is no acknowledgement to a load or store instruction. Instead processors equipped to detect memory access failures respond with machine checks. These can be routed to user threads as asynchronous events. With memory-mapped PM, asynchronous events are the primary means of discovering the failure of a load to return good data. Please refer to NVM.PM.FILE.GET_ERROR_INFO (section 10.2.6) for more information on error handling behavior.

Depending on memory configuration, CPU memory write pipelines may effectively preclude application level error handling during memory accesses that result from store instructions. For example, errors detected during the process of flushing the CPU’s write pipeline are more likely to be associated with that pipeline than the NVM itself. Errors that arise within the CPU’s write pipeline generally do not enable application level recovery at the point of the error. As a result application processes may be forced to restart when these errors occur (see PM Error Handling Annex B). Such errors should appear in CPU event logs, leading to an administrative response that is outside the scope of this specification.

Applications needing timely assurance that recently stored data is recoverable should use the NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY action to verify data from NVM after it is flushed (see 10.2.7). Errors during verify are handled in the manner described in this annex.

10.2.4 NVM.PM.FILE.SYNC

Requirement: mandatory

The purpose of this action is to synchronize persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain.

The native file system sync action may be supported by implementations that also support NVM.PM.FILE.SYNC. The intent is that the semantics of NVM.PM.FILE.SYNC match native sync operation on memory-mapped files however because persistent memory is involved, NVM.PM.FILE implementations need not flush full pages. Note that writes may still be subject to functionality that may mask whether stored data has reached the persistence domain (such as caching or buffering within processors or memory controllers). NVM.PM.FILE.SYNC is responsible for insuring that data within the processor or memory buffers reaches the persistence domain.

A number of boundary conditions can arise regarding interoperability of PM and non-PM implementation components. The following limitations apply:

- The behavior of an NVM.PM.FILE.SYNC action applied to a range in a file that was not mapped using NVM.PM.FILE.MAP is unspecified.
- The behavior of NVM.PM.FILE.SYNC on non-persistent memory is unspecified.

In both the PM and non-PM modes, updates to ranges mapped as shared can and may become persistent in any order before a sync requires them all to become persistent. The sync action applied to a shared mapping does not guarantee write atomicity. The byte range referenced by the sync parameter may have reached a persistence domain prior to the sync command. The sync action guarantees only that the range referenced by the sync action will reach the persistence domain before the successful completion of the sync action. Any atomicity that is achieved is not caused by the sync action itself.

Requires: NVM.PM.FILE.MAP

Inputs: Align with native operating system's sync with the exception that alignment restrictions are relaxed.

Outputs: Align with native operating system's sync with the addition that it shall return an error code.

Users of the NVM.PM.FILE.SYNC action should be aware that for files that are mapped as shared, there is no requirement to buffer data on the way to the persistence domain. Although data may traverse a processor’s write pipeline and other buffers within memory controllers these are more transient than the disk I/O buffering that is common in NVM.FILE implementations.

10.2.5 Error handling related to this action is expected to be derived from ongoing work that begins with Annex B (Informative) PM error handling.NVM.PM.FILE.OPTIMIZED_FLUSH

Requirement: mandatory if NVM.PM.OPTIMIZED_FLUSH_CAPABLE is set.

The purpose of this action is to synchronize multiple ranges of persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain. This action has the same effect as NVM.PM.FILE.SYNC however it is intended to allow additional implementation optimization by excluding options supported by sync and by allowing multiple byte ranges to be synchronized during a single action. Page oriented alignment constraints imposed by the native definition are lifted. Because of this, implementations might be able to use underlying persistent memory more optimally than they could with the native sync. In addition some implementations may enable this action to avoid context switches into kernel space. With the exception of these differences all of the content of the NVM.PM.FILE.SYNC action description also applies to NVM.PM.FILE.OPTIMIZED_FLUSH.

Requires: NVM.PM.FILE.MAP

Inputs: Identical to NVM.PM.FILE.SYNC except that an array of byte ranges is specified and options are precluded. A reference to the array and the size of the array are input instead of a single address and length. Each element of the array contains an address and length of a range of bytes to be synchronized.

Outputs: Align with native OS's sync with the addition that it shall return an error code.

Relevant attributes: NVM.PM.FILE.OPTIMIZED_FLUSH_CAPABLE – Indicates whether this action is supported by the NVM.PM.FILE implementation (see 10.3.5).

NVM.PM.FILE.OPTIMIZED_FLUSH provides no guarantee of atomicity within or across the synchronized byte ranges. Neither does it provide any guarantee of the order in which the bytes within the ranges of the action reach a persistence domain.

In the event of failure the progress of the action is indeterminate. Various byte ranges may or may not have reached a persistence domain. There is no indication as to which byte ranges may have been synchronized.

10.2.6 NVM.PM.FILE.GET_ERROR_EVENT_INFO

Requirement: mandatory if NVM.PM.ERROR_EVENT_CAPABLE is set.

The purpose of this action is to provide a sufficient description of an error event to enable recovery decisions to be made by an application. This action is intended to originate during an application event handler in response to a persistent memory error. In some implementations this action may map to the delivery of event description information to the application at the start of the event handler rather than a call made by the event handler. The error information returned is specific to the memory error that caused the event.

Inputs: It is assumed that implementations can extract the information output by this action from the event being handled.

Outputs:

1 – An indication of whether or not execution of the application can be resumed from the point of interruption. If execution cannot be resumed then the process running the application should be restarted for full recovery.

2 – An indication of error type enabling the application to determine whether an address is provided and the direction of data flow (load/verify vs. store) when the error was detected.

3 – The memory mapped address and length of the byte range where data loss was detected by the event.

Relevant attributes:

NVM.PM.FILE.ERROR_EVENT_CAPABLE – Indicates whether load error event handling and this action are supported by the NVM.PM.FILE implementation (see 10.3.6).

This action is used to obtain information about an error that caused a machine check involving memory mapped persistent memory. This is necessary because with persistent memory there is no opportunity to provide error information as part of a function call or I/O. The intent is to allow sophisticated error handling and recovery to occur before the application sees the event by allowing the NVM.PM.FILE implementation to handle it first. It is expected that after NVM.PM.FILE has completed whatever recovery is possible, the application error handler will be called and use the error information described here to stage subsequent recovery actions, some of which may occur after the application's process is restarted.

In some implementations the same event handler may be used for many or all memory errors. Therefore this action may arise from memory accesses unrelated to NVM. It is the application event handler's responsibility to determine whether the memory range indicated is relevant for recovery. If the memory range is irrelevant then the event should be ignored other than as a potential trigger for a restart.

In some systems, errors related to memory stores may not provide recovery information to the application unless and until load instructions attempt to access the memory locations involved. This can be accomplished using the NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY action (section 10.2.7).

For more information on the circumstances which may surround this action please refer to PM Error Handling Annex B.

10.2.7 NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY

Requirement: mandatory if NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY_CAPABLE is set.

The purpose of this action is to synchronize multiple ranges of persistent memory content to assure durability and enable recovery by forcing data to reach the persistence domain. Furthermore, this action verifies that data was written correctly by verifying it. The intent is to supply a mechanism whereby the application can receive data integrity assurance on writes to memory-mapped PM prior to completion of this action. This is the PM equivalent to the POSIX definition of synchronized I/O which clarifies that the intent of synchronized I/O data integrity completion is "so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices”.

Except for the additional verification of flushed data, this action has the same effect as NVM.PM.FILE.OPTIMIZED_FLUSH.

Requires: NVM.PM.FILE.MAP

Inputs: Identical to NVM.PM.FILE.OPTIMIZED_FLUSH.

Outputs: Align with native OS's sync with the addition that it shall return an error code. The error code indicates whether or not all data in the indicated range set is readable.

Relevant attributes:

NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY_CAPABLE – Indicates whether this action is supported by the NVM.PM.FILE implementation (see 10.3.7).

OPTIMIZED_FLUSH_AND_VERIFY shall assure that data has been verified to be readable. Any errors discovered during verification should be logged for administrative attention. Verification shall occur across all data ranges specified in the action regardless of when they were actually flushed. Verification shall complete prior to completion of the action.

In the event of failure the progress of the action is indeterminate.

10.3 Attributes

10.3.1 Attributes that apply across multiple modes

The following attributes apply to NVM.PM.FILE mode as well as other modes.

NVM.COMMON.SUPPORTED_MODES (see 6.12.1)

NVM.COMMON.FILE_MODE (see 6.12.2)

10.3.2 NVM.PM.FILE.MAP_COPY_ON_WRITE_CAPABLE

Requirement: mandatory

This attribute indicates that MAP_COPY_ON_WRITE option is supported by the NVM.PM.FILE.MAP action.

10.3.3 NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY

Requirement: mandatory

INTERRUPTED_STORE_ATOMICITY indicates whether the volume supports power fail atomicity of aligned store operations on fundamental data types. To achieve failure atomicity, aligned operations on fundamental data types reach NVM atomically. Formally “aligned operations on fundamental data types” is implementation defined. See 6.10.

A value of true indicates that after an aligned store of a fundamental data type is interrupted by reset, power loss or system crash; upon restart the contents of persistent memory reflect either the state before the store or the state after the completed store. A value of false indicates that after a store interrupted by reset, power loss or system crash, upon restart the contents of memory may be such that subsequent loads may create exceptions. A value of false also indicates that after a store interrupted by reset, power loss or system crash; upon restart the contents of persistent memory may not reflect either the state before the store or the state after the completed store.

The value of this attribute is true only if it’s true for all ranges in the file system.

10.3.4 NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE

Requirement: mandatory

FUNDAMENTAL_ERROR_RANGE is the number of bytes that may become unavailable due to an error on an NVM device.

An application may organize data in terms of FUNDAMENTAL_ERROR_RANGE to assure two key data items are not likely to be affected by a single error.

Unlike NVM.PM.VOLUME (see 9), NVM.PM.FILE does not associate an offset with the FUNDAMENTAL_ERROR_RANGE because the file system is expected to handle any volume mode offset transparently to the application. The value of this attribute is the maximum of the values for all ranges in the file system.

10.3.5 NVM.PM.FILE.OPTIMIZED_FLUSH_CAPABLE

Requirement: mandatory

This attribute indicates that the OPTIMIZED_FLUSH action is supported by the NVM.PM.FILE implementation.

10.3.6 NVM.PM.FILE.ERROR_EVENT_CAPABLE

Requirement: mandatory

This attribute indicates that the NVM.PM.FILE implementation is capable of handling error events in such a way that, in the event of data loss, those events are subsequently delivered to applications. If error event handling is supported then NVM.PM.FILE.GET_ERROR_INFO action shall also be supported.

10.3.7 NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY_CAPABLE

Requirement: mandatory

This attribute indicates that the OPTIMIZED_FLUSH_AND_VERIFY action is supported by the NVM.PM.FILE implementation.

10.4 Use cases

10.4.1 Update PM File Record

Update a record in a PM file.

Purpose/triggers:

An application using persistent memory updates an existing record. For simplicity, this application uses fixed size records. The record size is defined by application data considerations.

Scope/context:

Persistent memory context; this use case shows basic behavior.

Preconditions:

- The administrator created a PM file and provided its name to the application; this name is accessible to the application – perhaps in a configuration file
- The application has populated the PM file contents
- The PM file is not in use at the start of this use case (no sharing considerations)

Inputs:

The content of the record, the location (relative to the file) where the record resides

Success scenario:

1) The application uses the native OPEN action, passing in the file name
2) The application uses the NVM.PM.FILE.MAP action, passing in the file descriptor returned by the native OPEN. Since the records are not necessarily page aligned, the application maps the entire file.
3) The application registers for memory hardware exceptions
4) The application stores the new record content to the address returned by NVM.PM.FILE.MAP offset by the record's location
5) The application uses NVM.PM.FILE.SYNC to flush the updated record to the persistence domain
    a. The application may simply sync the entire file

    b. Alternatively, the application may limit the range to be sync'd

6) The application uses the native UNMAP and CLOSE actions to clean up.

Failure Scenario:

While reading PM content (accessing via a load operation), a memory hardware exception is reported. The application's event handler is called with information about the error as described in NVM.PM.FILE.GET_ERROR_INFO. Based on the information provided, the application records the error for subsequent recovery and determines whether to restart or continue execution.

Postconditions:

The record is updated.

10.4.2 Direct load access

Purpose/triggers:

An application developer wishes to retrieve data from a persistent memory-mapped file using direct memory load instruction access with error handling for uncorrectable errors.

Scope/context:

NVM.PM.FILE

Inputs:

- Virtual address of the data.

Outputs:

- Data from persistent memory if successful
- Error code if an error was detected within the accessed memory range.

Preconditions:

- The persistent memory file must be mapped into a region of virtual memory.
- The virtual address must be within the mapped region of the file.

Postconditions:

- If an error was returned, the data may be unreadable. Future load accesses may continue to return an error until the data is overwritten to clear the error condition
- If no error was returned, there is no postcondition.

Success and Failure Scenarios:

Consider the following fragment of example source code, which is simplified from the code for the function that reads SQLite's transaction journal:

```
retCode = pread(journalFD, magic, 8, off);
if (retCode != SQLITE_OK) return retCode;

```

```
    if (memcmp(magic, journalMagic, 8) != 0)
      return SQLITE_DONE;
```

This example code reads an eight-byte magic number from the journal header into an eight-byte buffer named *magic* using a standard file *read* call. If an error is returned from the *read* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, it then compares the contents of the *magic* buffer against the expected magic number constant named *journalMagic*. If the contents of the buffer do not match the expected magic number, the function exits with an error return code.

An equivalent version of the function using direct memory load instruction access to a mapped file is:

```
    volatile siginfo_t errContext;
    ...
    int retCode = SQLITE_OK;

    TRY
    {
        if (memcmp(journalMmapAddr + off, journalMagic, 8) != 0)
            retCode = SQLITE_DONE;
    }
    CATCH(BUS_MCEERR_AR)
    {
        if ((errContext.si_code == BUS_MCEERR_AR) &&
            (errContext.si_addr >= journalMmapAddr) &&
            (errContext.si_addr < (journalMmapAddr + journalMmapSize))){
            retCode = SQLITE_IOERR;
        } else {
            signal(errContext.si_signo, SIG_DFL);
            raise(errContext.si_signo);
        }
    }
    ENDTRY;

    if (retCode != SQLITE_OK) return retCode;
```

The mapped file example compares the magic number in the header of the journal file against the expected magic number using the *memcmp* function by passing a pointer containing the address of the magic number in the mapped region of the file. If the contents of the magic number member of the file header do not match the expected magic number, the function exits with an error return code.

The application-provided TRY/CATCH/ENDTRY macros implement a form of exception handling using POSIX *sigsetjmp* and *siglongjmp* C library functions. The TRY macro initializes a *sigjmp_buf* by calling *sigsetjmp*. When a SIGBUS signal is raised, the signal handler calls *siglongjmp* using the *sigjmp_buf* set by the *sigsetjmp* call in the TRY macro. Execution then continues in the CATCH clause. (Caution: the code in the TRY block should not call library functions as they are not likely to be exception-safe.) Code for the Windows platform would be similar except that it would use the standard Structured Exception Handling *try-except* statement catching the EXCEPTION_IN_PAGE_ERROR exception rather than application-provided TRY/CATCH/ENDTRY macros.

If an error occurs during the read of the magic number data from the mapped file, a SIGBUS signal will be raised resulting in the transfer of control to the CATCH clause. The address of the error is compared against the range of the memory-mapped file. In this example the error address is assumed to be in the process's logical address space. If the error address is within the range of the memory-mapped file, the function returns an error code indication that an I/O error occurred. If the error address is outside the range of the memory-mapped file, the error is assumed to be for some other memory region such as the program text, stack, or heap, and the signal or exception is re-raised. This is likely to result in a fatal error for the program.

See also:

- Microsoft Corporation, Reading and Writing From a File View (Windows), available from http://msdn.microsoft.com/en-us/library/windows/desktop/aa366801.aspx

10.4.3 Direct store access

Purpose/triggers:

An application developer wishes to place data in a persistent memory-mapped file using direct memory store instruction access.

Scope/context:

NVM.PM.FILE

Inputs:

- Virtual address of the data.
- The data to store.

Outputs:

- Error code if an error occurred.

Preconditions:

- The persistent memory file must be mapped into a region of virtual memory.
- The virtual address must be within the mapped region of the file.

Postconditions:

- If an error was returned, the state of the data recorded in the persistence domain is indeterminate.
- If no error was returned, the specified data is either recorded in the persistence domain or an undiagnosed error may have occurred.

Success and Failure Scenarios:

Consider the following fragment of example source code, which is simplified from the code for the function that writes to SQLite's transaction journal:

```
ret = pwrite(journalFD, dbPgData, dbPgSize, off);
if (ret != SQLITE_OK) return ret;
ret = write32bits(journalFD, off + dbPgSize, cksum);
```

```
    if (ret != SQLITE_OK) return ret;
    ret = fdatasync(journalFD);
    if (ret != SQLITE_OK) return ret;
```

This example code writes a page of data from the database cache to the journal using a standard file *write* call. If an error is returned from the *write* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the function then appends the checksum of the data, again using a standard file *write* call. If an error is returned from the *write* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the function then invokes the *fdatasync* system call to flush the written data from the file system buffer cache to the persistence domain. If an error is returned from the *fdatasync* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the written data has been recorded in the persistence domain.

An equivalent version of the function using direct memory store instruction access to a memory-mapped file is:

```
    memcpy(journalMmapAddr + off, dbPgData, dbPgSize);
    PM_track_dirty_mem(dirtyLines, journalMmapAddr + off, dbPgSize);

    store32bits(journalMmapAddr + off + dbPgSize, cksum);
    PM_track_dirty_mem(dirtyLines, journalMmapAddr + off + dbPgSize, 4);

    ret = PM_optimized_flush(dirtyLines, dirtyLinesCount);

    if (ret == SQLITE_OK) dirtyLinesCount = 0;

    return ret;
```

The memory-mapped file example writes a page of data from the database cache to the journal using the *memcpy* function by passing a pointer containing the address of the page data field in the mapped region of the file. It then appends the checksum using direct stores to the address of the checksum field in the mapped region of the file.

The code calls the application-provided *PM_track_dirty_mem* function to record the virtual address and size of the memory regions that it has modified. The *PM_track_dirty_mem* function constructs a list of these modified regions in the *dirtyLines* array.

The function then calls the *PM_optimized_flush* function to flush the written data to the persistence domain. If an error is returned from the *PM_optimized_flush* call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the written data is either recorded in the persistence domain or an undiagnosed error may have occurred. Note that this postcondition is weaker than the guarantee offered by the *fdatasync* system call in the original example.

See also:

- Microsoft Corporation, Reading and Writing From a File View (Windows), available from http://msdn.microsoft.com/en-us/library/windows/desktop/aa366801.aspx

10.4.4 Direct store access with synchronized I/O data integrity completion

Purpose/triggers:

An application developer wishes to place data in a persistent memory-mapped file using direct memory store instruction access with synchronized I/O data integrity completion.

Scope/context:

NVM.PM.FILE

Inputs:

- Virtual address of the data.
- The data to store.

Outputs:

- Error code if an error occurred.

Preconditions:

- The persistent memory file must be mapped into a region of virtual memory.
- The virtual address must be within the mapped region of the file.

Postconditions:

- If an error was returned, the state of the data recorded in the persistence domain is indeterminate.
- If no error was returned, the specified data is recorded in the persistence domain.

Success and Failure Scenarios:

Consider the following fragment of example source code, which is simplified from the code for the function that writes to SQLite’s transaction journal:

```
ret = pwrite(journalFD, dbPgData, dbPgSize, off);
if (ret != SQLITE_OK) return ret;
ret = write32bits(journalFD, off + dbPgSize, cksum);
if (ret != SQLITE_OK) return ret;

ret = fdatasync(journalFD);
if (ret != SQLITE_OK) return ret;
```

This example code writes a page of data from the database cache to the journal using a standard file *write* call. If an error is returned from the *write* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the function then appends the checksum of the data, again using a standard file *write* call. If an error is returned from the *write* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the function then invokes the *fdatasync* system call to flush the written data from the file system buffer cache to the persistence domain. If an error is returned from the *fdatasync* system call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the written data has been recorded in the persistence domain.

An equivalent version of the function using direct memory store instruction access to a memory-mapped file is:

```
    memcpy(journalMmapAddr + off, dbPgData, dbPgSize);
    PM_track_dirty_mem(dirtyLines, journalMmapAddr + off, dbPgSize);

    store32bits(journalMmapAddr + off + dbPgSize, cksum);
    PM_track_dirty_mem(dirtyLines, journalMmapAddr + off + dbPgSize, 4);

    ret = PM_optimized_flush_and_verify(dirtyLines, dirtyLinesCount);

    if (ret == SQLITE_OK) dirtyLinesCount = 0;

    return ret;
```

The memory-mapped file example writes a page of data from the database cache to the journal using the *memcpy* function by passing a pointer containing the address of the page data field in the mapped region of the file. It then appends the checksum using direct stores to the address of the checksum field in the mapped region of the file.

The code calls the application-provided *PM_track_dirty_mem* function to record the virtual address and size of the memory regions that it has modified. The *PM_track_dirty_mem* function constructs a list of these modified regions in the *dirtyLines* array.

The function then calls the *PM_optimized_flush_and_verify* function to flush the written data to the persistence domain. If an error is returned from the *PM_optimized_flush_and_verify* call, the function exits with an error return code indicating that an I/O error occurred. If no error occurs, the written data has been recorded in the persistence domain. Note that this postcondition is equivalent to the guarantee offered by the *fdatasync* system call in the original example.

See also:

- Microsoft Corp, FlushFileBuffers function (Windows), http://msdn.microsoft.com/en-us/library/windows/desktop/aa364439.aspx
- Oracle Corp, Synchronized I/O section in the Programming Interfaces Guide, available from http://docs.oracle.com/cd/E19683-01 /816-5042/chap7rt-57/index.html
- The Open Group, “The Open Group Base Specification Issue 6”, section 3.373 “Synchronized Input and Output”, available from http://pubs.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap03.html#tag_03_373

10.4.5 Persistent Memory Transaction Logging

Purpose/Triggers:

An application developer wishes to implement a transaction log that maintains data integrity through system crashes, system resets, and power failures. The underlying storage is byte-granular persistent memory.

Scope/Context:

NVM.PM.VOLUME and NVM.PM.FILE

For notational convenience, this use case will use the term “file” to apply to either a file in the conventional sense which is accessed through the NVM.PM.FILE interface, or a specific subset of memory ranges residing on an NVM device which are accessed through the NVM.BLOCK interface.

Inputs:

- A set of changes to the persistent state to be applied as a single transaction.
- The data and log files.

Outputs:

- An indication of transaction commit or abort.

Postconditions:

- If an abort indication was returned, the data was not committed and the previous contents have not been modified.
- If a commit indication was returned, the data has been entirely committed.
- After a system crash, reset, or power failure followed by system restart and execution of the application transaction recovery process, the data has either been entirely committed or the previous contents have not been modified.

Success Scenario:

The application transaction logic uses a log file in combination with its data file to atomically update the persistent state of the application. The log may implement a before-image log or a write-ahead log. The application transaction logic should configure itself to handle torn or interrupted writes to the log or data files.

Since persistent memory may be byte-granular, torn writes may occur at any point during a series of stores. The application should be prepared to detect a torn write of the record and either discard or recover such a torn record during the recovery process. One common way of detecting such a torn write is for the application to compute a hash of the record and record the hash in the record. Upon reading the record, the application re-computes the hash and compares it with the recorded hash; if they do not match, the record has been torn.

10.4.5.1 NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY is true

If the NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY is true, then writes which are interrupted by a system crash, system reset, or power failure occur atomically. In other words, upon restart the contents of persistent memory reflect either the state before the store or the state after the completed store.

In this case, the application need not handle interrupted writes to the log or data files.

10.4.5.2 NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY is false

NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY is false, then writes which are interrupted by a system crash, system reset, or power failure do not occur atomically. In other words, upon restart the contents of persistent memory may be such that subsequent loads may create exceptions depending on the value of the FUNDAMENTAL_ERROR_RANGE attribute.

In this case, the application should be prepared to handle an interrupted write to the log or data files.

10.4.5.2.1 NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE > 0

If the NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE is greater than zero, the application should align the log or data records with the NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE and pad the record size to be an integral multiple of NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE. This prevents more than one record from residing in the same fundamental error range. The application should be prepared to discard or recover the record if a load returns an exception when subsequently reading the record during the recovery process. (See also SQLite.org, *Powersafe Overwrite*, http://www.sqlite.org/psow.html.)

10.4.5.2.2 NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE = 0

If the NVM.PM.FILE.FUNDAMENTAL_ERROR_RANGE is zero, the application lacks sufficient information to handle interrupted writes to the log or data files.

Failure Scenarios:

Consider the recovery of an error resulting from an interrupted write on a persistent memory volume or file system where the NVM.PM.FILE.INTERRUPTED_STORE_ATOMICITY is false. This error may be persistent and may be returned whenever the affected fundamental error range is read. To repair this error, the application should be prepared to overwrite such a range.

One common way of ensuring that the application will overwrite a range is by assigning it to the set of internal free space managed by the application, which is never read and is available to be allocated and overwritten at some point in the future. For example, the range may be part of a circular log. If the range is marked as free, the transaction log logic will eventually allocate and overwrite that range as records are written to the log.

Another common way is to record either a before-image or after-image of a data range in a log. During recovery after a system crash, system reset, or power failure, the application replays the records in the log and overwrites the data range with either the before-image contents or the after-image contents.

See also:

- SQLite.org, *Atomic Commit in SQLite*, http://www.sqlite.org/atomiccommit.html
- SQLite.org, *Powersafe Overwrite*, http://www.sqlite.org/psow.html
- SQLite.org, *Write-Ahead Logging*, http://www.sqlite.org/wal.html

Annex A (Informative) PM pointers

Pointers are data types that hold virtual addresses of data in memory. When applications use pointers with volatile memory, the value of the pointer must be re-assigned each time the program is run (a consequence of the memory being volatile). When applications map a file (or a portion of a file) residing in persistent memory to virtual addresses, it may or may not be assigned the same virtual address. If not, then pointers to values in that mapped memory will not reference the same data. There are several possible solutions to this problem:

1) Relative pointers
2) Regions are mapped at fixed addresses
3) Pointers are relocated when region is remapped

All three approaches are problematic, and involve different challenges that have not been fully addressed.

None, except perhaps the third one, handles C++ vtable pointers inside persistent memory, or pointers to string constants, where the string physically resides in the executable, and not the memory-mapped file. Both of those issues are common.

Option (1) implies that no existing pointer-containing library data structures can be stored in PM, since pointer representations change. Option (2) requires careful management of virtual addresses to ensure that memory-mapped files that may need to be accessed simultaneously are not assigned to the same address. It may also limit address space layout randomization. Option (3) presents challenges in, for example, a C language environment in which pointers may not be unambiguously identifiable, and where they may serve as hash table indices or the like. Pointer relocation would invalidate such hash tables. It may be significantly easier in the context of a Java-like language.

Annex B (Informative) PM error handling

Persistent memory error handing for NVM.PM.FILE.MAP ranges is unique because unlike I/O, there is no acknowledgement to a load or store instruction. Instead processors equipped to detect memory access failures respond with machine checks. In some cases these can be routed to user threads as asynchronous events.

This annex only describes the handling of errors resulting from load instructions that access memory. As will be described later in this annex, no application level recovery is enabled at the point of a store error. These errors should appear in CPU event logs, leading to an administrative response that is outside the scope of this annex.

Applications needing timely assurance that recently stored data is recoverable should use the NVM.PM.FILE.OPTIMIZED_FLUSH_AND_VERIFY (see 10.2.7) action to read data back from NVM after it is flushed. Errors during verify are handled in the manner described in this annex.

There are several scenarios that can arise in the handling of machine checks related to persistent memory errors while reading data from memory locations such as can occur during “load” instructions. Concepts are introduced here in an attempt to advance the state of the art in persistent memory error handling. The goal is to provide error reporting and recovery capability to applications that is equivalent to the current practice for I/O.

We need several definitions to assist in reasoning about asynchronous events.

- Machine check: an interrupt. In this case interrupts that result from memory errors are of specific interest.
- Precise machine check – an interrupt that allows an application to resume at the interrupted instruction
- Error containment – this is an indication of how well the system can determine the extent of an error. This enables a range of memory affected by an error that caused an interrupt to be returned to the application.
- Real time error recovery – This refers to scenarios in which the application can continue execution after an error as opposed to being restarted.
- Asynchronous event handler – This refers to code provided by an application that runs in response to an asynchronous event, in this case an event that indicates a memory error. An application’s event handler uses information about the error to determine whether execution can safely continue from within the application or whether a partial or full restart of the application is required to recover from the error.

The ability to handle persistent memory errors depends on the capability of the processor and memory system. It is useful to categorize error handling capability into three levels:

- No memory error detection – the lowest end systems have little or no memory error detection or correction capability such as ECC, CRC or parity.
- Non-precise or uncontained memory error detection – these systems detect memory errors but they do not provide information about the location of the error and/or fail to offer enough information to resume execution from the interrupted instruction.

- Precise, contained memory error detection – these systems detect memory errors and report their locations in real time. These systems are also able to contain many errors more effectively. This increases the range of errors that allowing applications to continue execution rather than resetting the application or the whole system. This capability is common when using higher RAS processors.

Only the last category of systems can, with appropriate operating system software enhancement, meet the error reporting goal stated above. The other two categories of systems risk scenarios where persistent memory errors are forced to repeatedly reset threads or processors, rendering them incapable of doing work. Unrecovered persistent memory errors are more problematic than volatile memory errors because they are less likely to disappear during a processor reset or application restart.

Systems with precise memory error detection capability can experience a range of scenarios depending on the nature of the error. These can be categorized into three types.

- Platform can't capture error
    - Perhaps application or operating system dies
    - Perhaps hardware product include diagnostic utilities
- Platform can capture error, considered fatal
    - Operating system crashes
    - Address info potentially stored by operating system or hardware/firmware
    - Application could use info on restart
- Platform can capture error & deliver to application
    - Reported to application using asynchronous "event"
    - Example: SIGBUS on UNIX w/address info

If the platform can't capture the error then no real time recovery is possible. The system may function intermittently or not at all until diagnostics can expose the problem. The same thing happens whether the platform lacks memory error detection capability or the platform has the capability but was unable to use it due to a low probability error scenario.

If the platform can capture the error but it is fatal then real time recovery is not possible, however then the system may make information about the error available after system or application restart. For this scenario, actions are proposed below to obtain error descriptions.

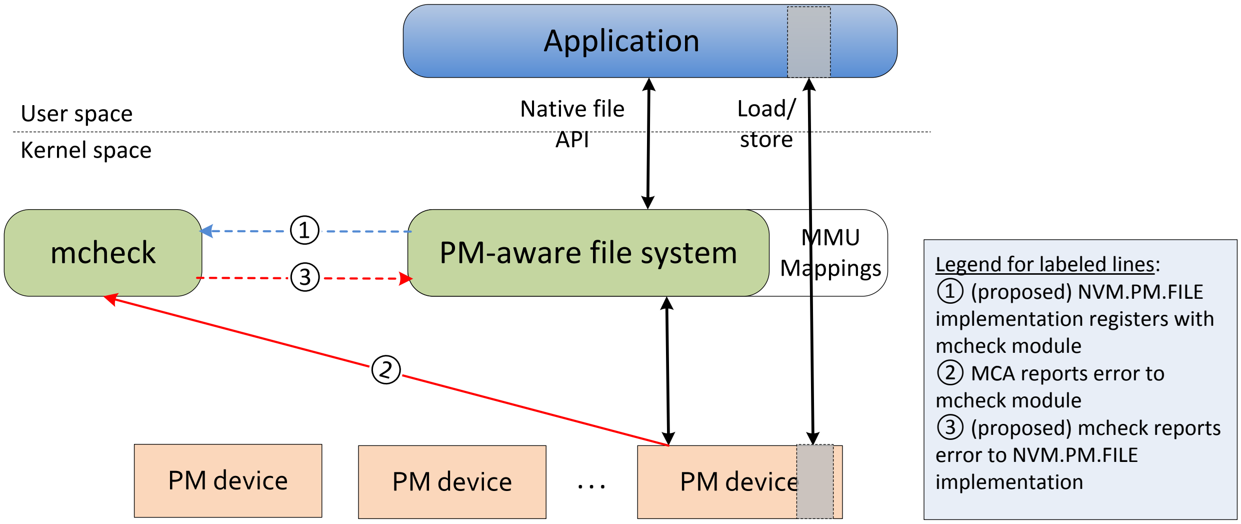
If the platform can deliver the error to the application then real time recovery may be possible. An action is proposed below to represent the means that the application uses to obtain error information immediately after the failure.

As stated at the beginning of this annex, only errors during load are addressed by this annex. As with other storage media, little or no error checking occurs during store instructions (aka writes). In addition, memory write pipelines within CPU's effectively preclude error handling during memory accesses that result from store instructions. For example, errors detected during the process of flushing the CPU's write pipeline are more likely to be associated with that pipeline than the NVM itself. Errors that arise within the CPU's write pipeline are generally not contained so no application level recovery is enabled at the point of the error.

Continuing to analyze the real time error delivery scenario, the handling of errors on load instructions is sufficient in today's high RAS systems to avoid the consumption of erroneous data by the application. Several enhancements are required to meet the goal of I/O-like application recoverability.

Using Linux running on the Intel architecture as an example, memory errors are reported using Intel's Machine Check Architecture (MCA). When the operating system enables this feature, the error flow on an uncorrectable error is shown by the solid red arrow (labeled ②) in Figure 15 Linux Machine Check error flow with proposed new interface, which depicts the mcheck component getting notified when the bad location in PM is accessed.

Figure 15 Linux Machine Check error flow with proposed new interface



As mentioned above, sending the application a SIGBUS (a type of asynchronous event) allows the application to decide what to do. However, in this case, remember that the NVM.PM.FILE manages the PM and that the location being accessed is part of a file on that file system. So even if the application gets a signal preventing it from using corrupted data, a method for recovering from this situation must be provided. A system administrator may try to back up rest of the data in the file system before replacing the faulty PM, but with the error mechanism we've described so far, the backup application would be sent a SIGBUS every time it touched the bad location. What is needed in this case is a way for the NVM.PM.FILE implementation to be notified of the error so it can isolate the affected PM locations and then continue to provide access to the rest of the PM file system. The dashed arrows in Figure 15 show the necessary modification to the machine check code in Linux. On start-up, the NVM.PM.FILE implementation registers with the machine code to show it has responsibility for certain ranges of PM. Later, when the error occurs, NVM.PM.FILE gets called back by the mcheck component and has a chance to handle the error.

This suggested machine check flow change enables the file system to participate in recovery while not eliminating the ability to signal the error to the application. The application view of errors not corrected by the file system depends on whether the error handling was precise and contained. Imprecise error handling precludes resumption of the application, in which case the one recovery method available besides restart is a non-local go-to. This resumes execution at an application error handling routine which, depending on the design of the application, may be able to recover from the error without resuming from the point in the code that was interrupted.

Taking all of this into account, the proposed application view of persistent memory errors is as described by the NVM.PM.FILE.MAP action (section 10.2.3) and the NVM.PM.FILE.GET_ERROR_INFO action (section 10.2.6).

The following actions have been proposed to provide the application with the means necessary to obtain error information after a fatal error.

- PM.FILE.ERROR_CHECK(file, offset, length): Discover if range has any outstanding errors. Returns a list of errors referenced by file and offset.
- PM.FILE.ERROR_CLEAR(file, offset, length): Reset error state (and data) for a range: may not succeed

The following attributes have been proposed to enable application to discover the error reporting capabilities of the implementation.

- NVM.PM.FILE.ERROR_CHECK_CAPABLE - System supports asking if range is in error state

Annex C (Informative) Deferred behavior

This annex lists some behaviors that are being considered for future specifications.

D.1 Remote sharing of NVM

This version of the specification talks about the relationship between DMA and persistent memory (see 6.6 Interaction with I/O devices) which should enable a network device to access NVM devices. But no comprehensive approach to remote share of NVM is addressed in this version of the specification.

D.2 MAP_CACHED OPTION FOR NVM.PM.FILE.MAP

This would enable memory mapped ranges to be either cached or uncached by the CPU.

D.3 NVM.PM.FILE.DURABLE.STORE

This might imply that through this action things become durable and visible at the same time, or not visible until it is durable. Is there a special case for atomic write that, by the time the operation completes, it is both visible and durable? The prospective use case is an opportunity for someone with a hardware implementation that does not require separation of store and sync. This is not envisioned as the same as a file system write. It still implies a size of the store. The use case for NVM.FILE.DURABLE.STORE is to force access to the persistence domain.

D.4 Enhanced NVM.PM.FILE.WRITE

Add an NVM.PM.FILE.WRITE action where the only content describes error handling.

D.5 Management-only behavior

Several management-only behaviors have been discussed, but deferred to a future revision; including:

- Secure Erase
- Behavior enabling management application to discover PM devices (and behavior to fill gaps in the discovery of block NVM attributes)
- Attribute exposing flash erase block size for management of disk partitions

D.6 Access hints

Allow applications to suggest how data is placed on storage

D.7 Multi-device atomic multi-write action

Perform an atomic write to multiple extents in different devices.

D.8 NVM.BLOCK.DISCARD_IF_YOU_MUST action

The text below was partially developed, before being deferred to a future revision.

10.4.6 NVM.BLOCK.DISCARD_IF_YOU_MUST

Proposed new name MARK_DISCARDABLE

Purpose - discard blocks to prevent write amplification

This action notifies the NVM device that some or all of the blocks which constitute a volume are no longer needed by the application, but the NVM device should defer changes to the blocks as long as possible. This action is a hint to the device.

If the data has been retained, a subsequent read shall return “success” along with the data. Otherwise, it shall return an error indicating the data does not exist (and the data buffer area for that block is undefined).

Inputs: a range of blocks (starting LBA and length in logical blocks)

Status: Success indicates the request is accepted but not necessarily acted upon.

Existing implementations of TRIM may work this way.

10.4.7 DISCARD_IF_YOU_MUST use case

Purpose/triggers:

An NVM device may allocate blocks of storage from a common pool of storage. The device may also allocate storage through a thin provisioning mechanism. In each of these cases, it is useful to provide a mechanism which allows an application or NVM user to notify the NVM storage system that some or all of the blocks which constitute the volume are no longer needed by the application. This allows the NVM device to return the memory allocated for the unused blocks to the free memory pool and make the unused blocks available for other consumers to use.

DISCARD_IF_YOU_MUST operation informs the NVM device that that the specified blocks are no longer required. DISCARD_IF_YOU_MUST instructs the NVM device to release previously allocated blocks to the NVM device’s free memory pool. The NVM device releases the used memory to the free storage pool based on the specific implementation of that device. If the device cannot release the specified blocks, the DISCARD_IF_YOU_MUST operation returns an error.

Scope/context:

This use case describes the capabilities of an NVM device that the NVM consumer can determine.

Inputs:
The range to be freed.

Success scenario:
The operation succeeds unless an invalid region is specified or the NVM device is unable to free the specified region.

Outputs:

The completion status.

Postconditions:

The specified region is erased and released to the free storage pool.

See also:

DISCARD_IF_YOU_CAN

EXISTS

D.9 Atomic write action with Isolation

Offer alternatives to ATOMIC_WRITE and ATOMIC_MULTIWRITE that also include isolation with respect to other atomic write actions. Issues to consider include whether order is required, whether isolation applies across multiple paths, and how isolation applies to file mapped I/O.

D.10 Atomic Sync/Flush action for PM

The goal is a mechanism analogous to atomic writes for persistent memory. Since stored memory may be implicitly flushed by a file system, defining this mechanism may be more complex than simply defining an action.

D.11 Hardware-assisted verify

Future PM device implementations may provide a capability to perform the verify step of OPTIMIZED_FLUSH_AND_VERIFY without requiring an explicit load instruction. This capability may require the addition of actions and attributes in NVM.PM.VOLUME mode; this change is deferred until we have examples of this type of device.