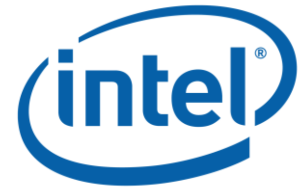


Persistent Memory Programming – Application Handling of Dirty Shutdowns

White Paper

January 2020

Revision V1.0

Notices

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

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

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

© 2020 Intel Corporation.

Contents

Contents

1 Introduction 5

2 Dirty Shutdown Count Architecture.. 6

2.1 DSC Overview.. 6
2.2 The importance of utilizing a counter 7
2.3 Application utilization of the DSC.. 9
2.4 Limitations of the architecture 11
3 OS specific implementation details 13

3.1 ESX 13
3.2 Linux 13
3.3 Windows 14
4 Intel DCPMM specific implementation details.. 15

Revision History

DOC#	REV#	Revision Description	Date
XXXXXX	1.0	**Initial public version**	January 2020

1 Introduction

Persistent memory (pmem) is presented to applications in various ways. For legacy applications that are not pmem-aware, the traditional storage interfaces provide access, allowing them to use pmem without modification. Alternatively, the application may be *pmem-aware*, modified to get the full benefit of byte-addressable persistence. These pmem-aware applications get direct access to the pmem, completely bypassing the kernel once the pmem is memory-mapped by the application. This allows direct load/store access between the application and the pmem. Along with this direct access comes several application-level responsibilities: flushing changes to persistence, handling memory errors, and detecting *dirty shutdowns*. This document focuses on dirty shutdowns, specifically the mechanism used to detect them: the *dirty shutdown count* (DSC).

This document assumes a general background on persistent memory programming, the SNIA NVM Programming Model, and the mechanisms used by applications to memory-map pmem. For more background on these topics, the web site http://pmem.io provides documentation, including an online copy of the Persistent Memory Programming book. The Persistent Memory Development Kit (PMDK), a suite of libraries to make pmem programming easier, is also available on the pmem.io web site. Although the goal of this document is to describe the dirty shutdown handling for application developers, the PMDK libraries already handle dirty shutdowns and our recommendation is to use those libraries rather than re-inventing the code to deal with them.

Section 2 of this document goes into the details of dirty shutdowns and the steps an application takes to handle them. Those steps can be summarized at a high-level as:

- Tracking the last known DSC and volume UUID, typically by storing it in headers of persistent memory data files and detecting when the DSC has changed on start-up.
- Avoiding false positives by tracking clean/dirty information in persistent memory data files, so that files that were cleanly-closed are not considered corrupted when the volume's DSC increases.
- Forcing the appropriate service action to happen when affected by a dirty shutdown, typically by failing to start the application, requiring any corrupt data files to be restored from backup copies.

Sections 3 and 4 contain deeper details on operating system support and Intel persistent memory specifics, respectively.

The Intel architecture and the Intel® Optane™ DC Persistent Memory module (DCPMM) are the focus of this paper, however the programming model and the need to detect dirty shutdowns is not specific to those products. The PMDK libraries are designed to use the techniques described in this paper to detect dirty shutdowns on any product that supports them.

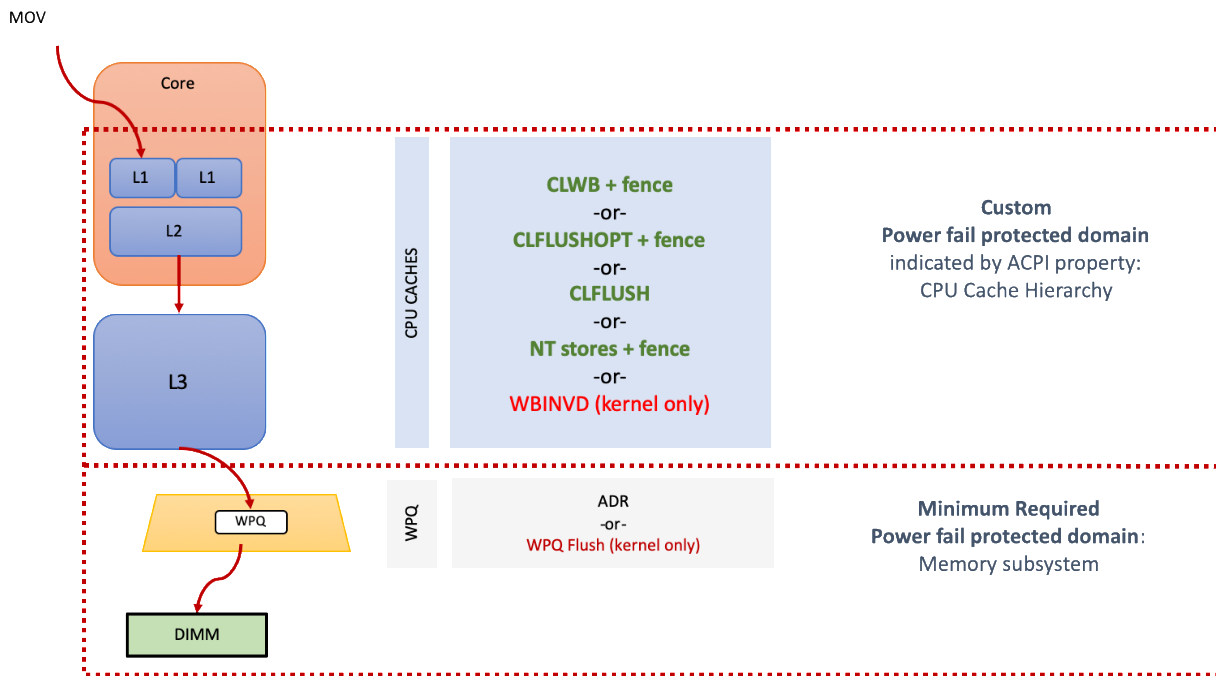
2 Dirty Shutdown Count Architecture

This white paper provides background information on the application level persistent memory (PMEM) Dirty Shutdown Count (DSC), a feature that is available with current PMEM technology. This functionality is optionally implemented by a platform and provides an indicator to any SW component that wishes to make use of persistent memory. While this paper is written with the Intel® DCPMM implementation specifically in mind, the functionality put forward here is an application level feature that, depending on NVDIMM vendor implementation, can make use of vendor agnostic ACPI 6.3 interfaces.

2.1 DSC Overview

Consider the path that a store takes on a typical Intel x86 server, as shown in Figure 2-1. The store, starting in the upper left, will traverse the CPU caches, eventually making its way to the memory controller (shown as a yellow trapezoid), where it is queued in a *write pending queue* (WPQ) to be sent to the persistent memory DIMM.

Figure 2-1. The Power Fail Protected Domains



Now consider the steps that software must take to ensure the store is persistent, even in the face of a power failure. There are two cases to consider. First, the large red box represents a system where the CPU caches are automatically flushed on power failure, so that stores sitting in the CPU caches are considered persistent by applications. In the second, more common case, the smaller red box on the bottom of the figure represents the power fail protected domain and stores are only considered persistent when they have been accepted by the memory subsystem.

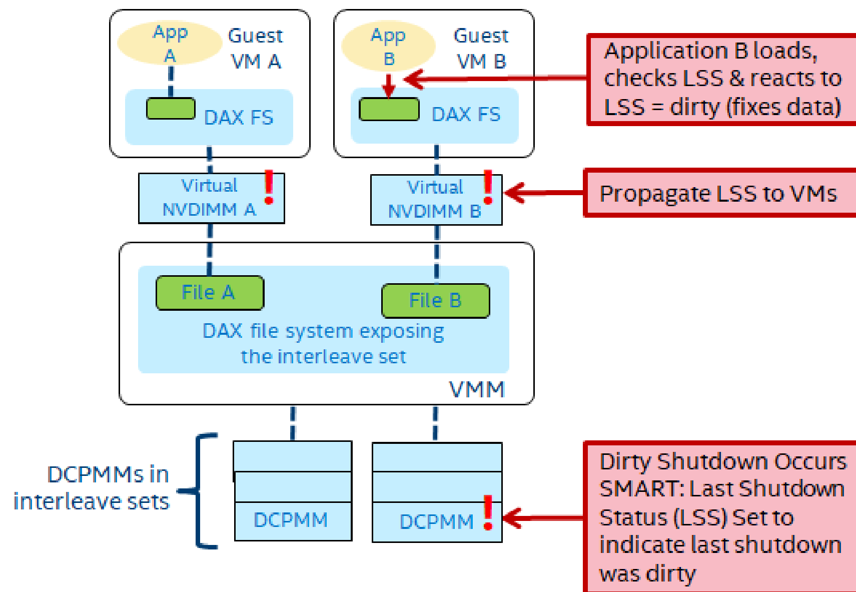
As shown in the figure, x86 platforms that support pmem **may** support the larger persistence domain, but they **must** support the smaller persistence domain. A required platform feature known as *asynchronous DRAM refresh* (ADR) ensures that any stores in the memory controller are flushed all the way to the DIMM in the face of unexpected power loss. This *flush-on-fail* operation is made possible by stored energy in the system power supply. All ADR-capable systems support this additional stored energy, as well as a hardware signal sent by the power supply to indicate power has failed.

What happens if something prevents the flush-on-fail from working correctly? The result is that an unknown number of stores in the write pending queue did not make it to the DIMM. Since the number and destination of the stores is unknown, and any ordering assumptions made by software is lost, the result is that the data on the DIMM is in an unknown state. It is possible, however, to reduce the impact of this failure by keeping track of which files were potentially being written when the system failed. Files that were not open or that were only open for reading could not be impacted by this failure, only files that were open for writing (more on this in section 2.3 below). The first step to handling this failure case is to indicate the failure to software. The mechanism for doing this is the *dirty shutdown count* (DSC).

2.2 The importance of utilizing a counter

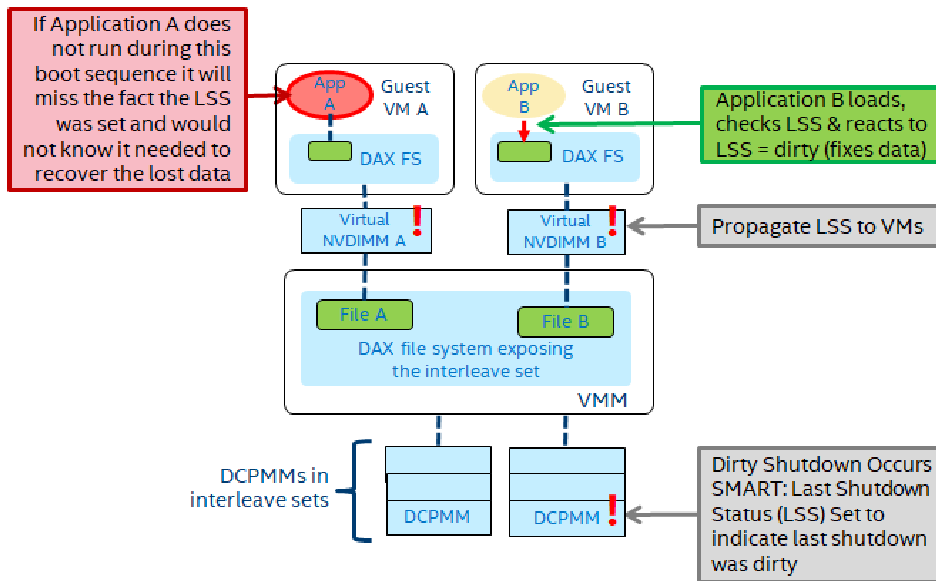
Some background on the design of the DSC may help understand how it works. During the early design of the Intel® persistent memory, known as Optane™ Data Center Persistent Memory modules (DPCMM), a single bit was thought to be sufficient for detecting dirty shutdowns. This bit is known as the *last shutdown state* (LSS) and was latched so that a dirty shutdown was indicated to the OS until the OS specifically cleared the state on the DCPMM. This way, if a dirty shutdown was followed by several interrupted reboots, or by clean shutdowns, the error condition persisted until software “consumed” it. This LSS flow is shown in Figure 2-2 below.

Figure 2-2. Utilizing the Last Shutdown Status state



To illustrate why a single flag like the LSS is too simplistic, imagine that the server has multiple virtual machines running on it, as shown in the above figure. Both virtual machines (shown as "Guest VM A" and "Guest VM B") are storing persistent memory files on the DCPMM attached to the machine. The interleave set on the lower right experiences a dirty shutdown and indicates this using the LSS. When does the LSS get cleared in this situation? The figure shows Guest VM B detecting the dirty shutdown and reacting by clearing the LSS on the DIMM. Later, Guest VM A starts up, as shown in Figure 2-3 below.

Figure 2-3. Limitations with the Last Shutdown Status state



Guest VM A doesn't see a dirty shutdown because the other guest has cleared the condition. Adding more virtual machines, considering some of them to be running and some stopped, and considering multiple clean and dirty shutdowns causes this solution to fall apart in complexity.

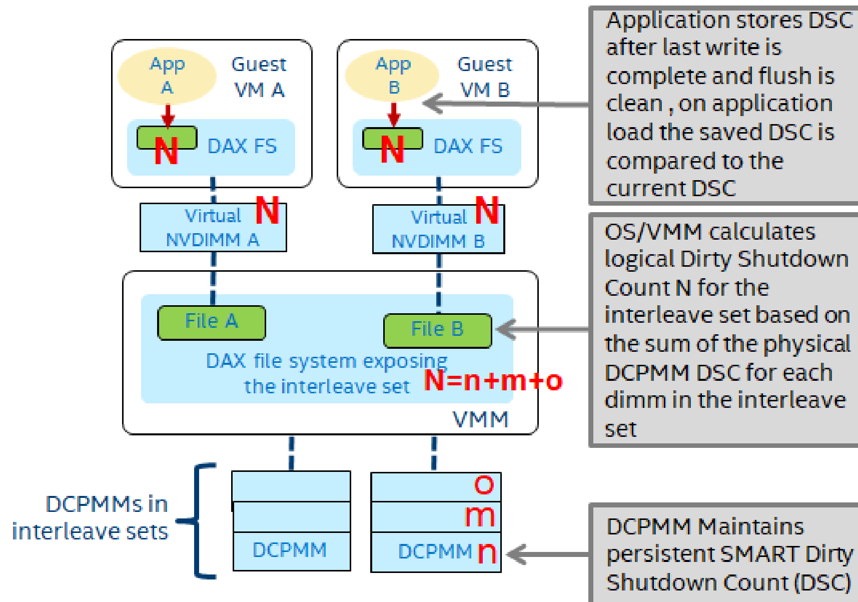
Since the initial design for detecting dirty shutdown was the LSS, the Intel DCPMM implementation provides support for a LSS bit that is used by the BIOS for accurately reporting the NFIT Memory Device State flag Bit 2, as required by ACPI. The recommendation at the OS and application level is to use the dirty shutdown count mechanism instead. Instead of a single bit, the DSC is incremented every time the DIMM experiences a dirty shutdown. As far as the application is concerned, no latching is done and no "re-arming" of the counter by software is required. The way an application uses the DSC is described in the next section.

2.3 Application utilization of the DSC

Making use of the DSC in a pmem-aware application requires some additional start-up logic in the application, or in libraries like those provided by PMDK. The operating systems provide the basic infrastructure for finding the DSC for the physical DIMM devices, and when multiple DIMMs are interleaved together by the memory controller, a logical DSC is created by adding the physical DSCs together, so that any number of DIMMs in an interleave set can experience a dirty shutdown and the DSC for the entire interleave set will increase as a result. The calculation of the logical DSC happens at the application or library level, not at the OS level. See the OS specific implementation section for more specifics.

The following figure outlines the basic high-level SW stack for the DSC feature.

Figure 2-4. SW Stack for Dirty Shutdown Count



The following outlines the application responsibilities and possible implementation suggestions:

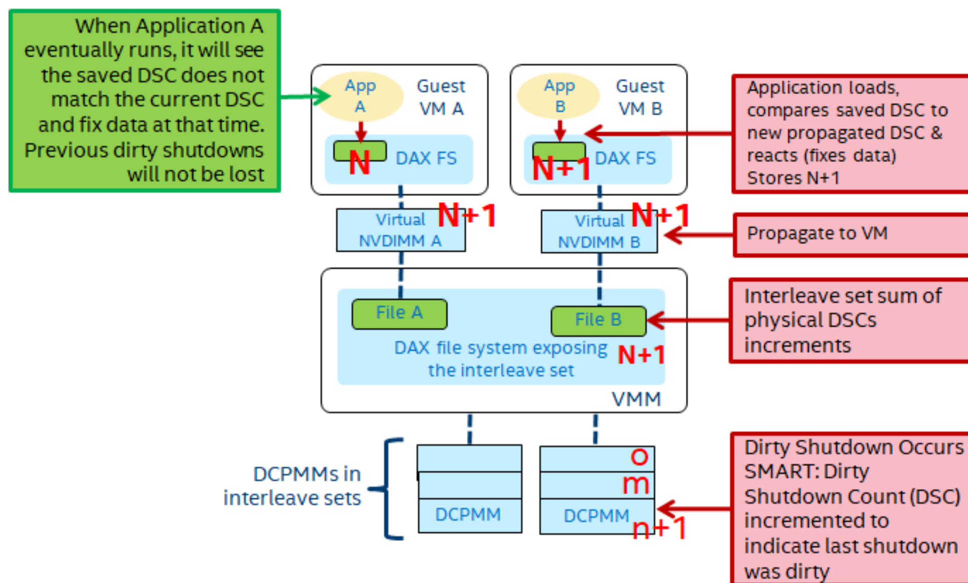
1. The application first creates its pmem file with some metadata stored in the header:
    a. The application retrieves the current DSC for each physical DIMM being used, using an OS-specific method such as a /sys file, or an IOCTL.
    b. The application calculates the current *logical dirty shutdown count* (LDSC) as the sum of the DSC for all physical DIMMs that make up the logical volume the pmem file resides on.
    c. The application stores the current LDSC in its metadata in the pmem file
    d. The application determines GUID or UUID for the logical volume the pmem file resides on and stores this in its metadata in the pmem file.
    e. The application flushes this information to ensure it is persistent. If supported by the OS, the application can use a *deep flush* operation here so that the metadata used to detect dirty shutdowns is itself more resilient against dirty shutdowns. (Deep flush is part of the pmem programming model and flushes stores to smallest failure domain available to software.)

2. Each time application runs & retrieves its metadata from the pmem file:
    a. The application checks the UUID saved in its metadata with the current UUID for the logical volume the pmem file metadata resides. If they match, then the LDSC is describing the same logical volume the app was utilizing and can proceed to checking the LDSC. If they do not match, then the LDSC is for some other logical volume and no longer applies (i.e. the file was copied to a different volume), so no further action is required since there’s no information available to detect a dirty shutdown in this case.
    b. The application calculates the LDSC.

c. The application compares the current LDSC calculated with the saved LDSC retrieved from its metadata in the pmem file.
d. If the current LDSC != saved LDSC then the DIMM has detected a dirty shutdown and possible data loss. It is up to the application to determine if it is possible writes to the file were outstanding. For example, the application can maintain a “clean” flag in its metadata in the pmem file that the above checks only happen if the file was open for writing during the last system crash.
e. If the application determines a dirty shutdown happened, it performs an application-specific recovery action, such as exiting with a fatal error and telling the user the file needs to be restored from backup. Of course, a more sophisticated application could repair or recover the pmem file from redundant information itself.

The following figure outlines the application starting, determining the current LDSC has incremented beyond its saved LDSC.

Figure 2-5. Utilizing the Dirty Shutdown Count



2.4 Limitations of the architecture

Here are the known limitations of the optional DSC architecture and current implementations. This section is likely less interesting to application developers and more interesting to OS and platform vendors.

1. While the DSC mechanism can be utilized for OS kernel components, file systems, and applications, the intended architecture described here focuses on detection of a dirty shutdown for an application. OS kernel and file system components currently do not utilize the DSC to harden themselves to recover if they were writing when a DSC increment occurs. If

those components do not utilize persistent memory themselves, there is no reason those components should look at the DSC. If kernel components are utilizing persistent memory, they can optionally elect to store their own version of the DSC and utilize it as suggested here for applications.

2. While the ACPI specification covers the reporting of an unlatched **DataLossCount** through the _NBS interface, for any NVDIMM in the ACPI0012 device tree, the consumption and use of the **DataLossCount** by the OS is optional and not standardized within the ACPI spec. The lack of a common OS response to the DSC can be considered a limitation of the ACPI specification that will be addressed in future ACPI specification updates, driven through the NVST ACPI NVDIMM work group. The use of DSC in the OS kernel varies across OS vendor so please see the following **OS specific implementation details** section for OS specific limitations. Please contact your OS vendor directly for the latest details on their specific implementation.
3. Logical storage volumes made up of interleave sets of NVDIMMs that have no files on them (and thus no application use of the storage) or files that were copied to a temporary storage location (and no longer associated with an application), will not have an entity checking the DSC on that volume. If the OS kernel components do not consume the DSC then this limitation exists. If the DSC were incremented, no one would be listening for it. Any consumer of the DSC, whether application or kernel component can protect itself from movement of its DSC metadata to another location by also tracking the location (i.e. volume UUID) the DSC was calculated for.

3 OS specific implementation details

The following section outlines the differences in the OS implementations of DSC and is considered accurate at the time of this document update.

3.1 ESX

VMware ESX will make use of the ACPI _NBS interface which internally still utilizes the DCPMM SMART LDSC. As far as the OS is concerned, this is an unlatched DSC mechanism. Please contact VMware for more information on the ESX implementation of DSC.

3.2 Linux

The Linux kernel treats the dirty shutdown counter similar to how it handles "deep flush", namely a mechanism outside the data consistency model of a typical Linux filesystem. It assumes that the sources of ADR failure need to be accounted for in the platform thermal and power design not actively mitigated in the OS kernel. To that end the dirty-shutdown count is not consumed by the Linux kernel and only exported to applications. Specifically, the kernel is more likely to disable the optimized persistent memory programming model as a mitigation for frequent dirty-shutdowns than add filesystem consideration of the DSC.

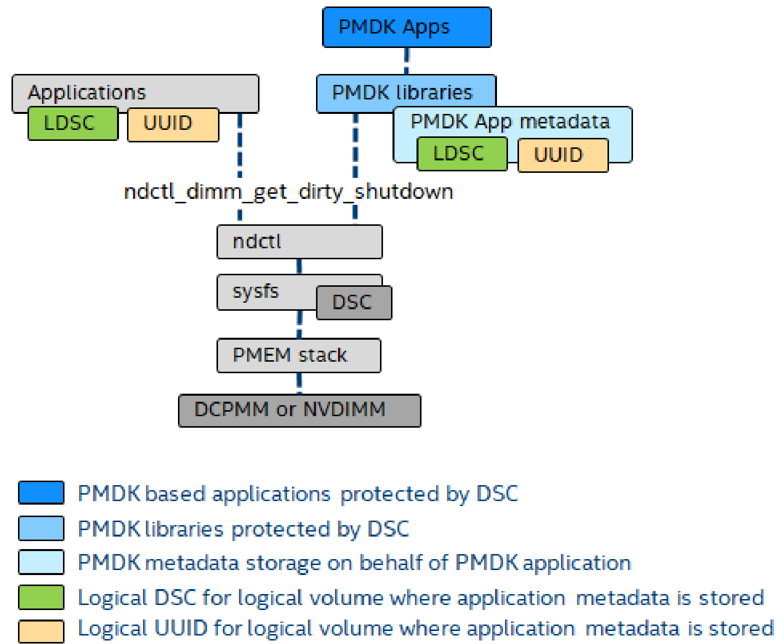
Setting this consideration of kernel's handling, or non-handling as it were, to the side, here are the details of how the counter is exposed to applications and how it is consumed by helper libraries like PMDK to let applications validate data consistency:

At driver initialization time the kernel will harvest the raw SMART LDSC from the BIOS DSM.GetSmartHealthInfo API (an Intel-specific DSM, other platforms may differ) for each NVDIMM and store a cached copy in sysfs for use by libndctl. The ndctl library provides an interface, ndctl_dimm_get_dirty_shutdown(), that allows applications to retrieve the cached DSC for an NVDIMM. It is left up to the application to determine the logical count associated with its pmem file location, determine when the DSC increment is important to the application, and how to handle the data recovery when the application was writing during a DSC increment. The kernel provides no other support for DSC handling.

PMDK library users will get the DSC checking as part of the core error handling of the PMDK libraries. There is no need for applications utilizing PMDK to also look at the DSC on their own. The DSC and UUID of the pmem files are protected by the PMDK DSC implementation as outlined in the application sequences above.

The following figure outlines this Linux specific DSC support.

Figure 3-1. Linux DSC implementation



3.3 Windows

The Windows persistent memory stack will harvest the raw SMART LDSC from the BIOS DSM.GetSmartHealthInfo API for each NVDIMM. If the DSC has incremented, the physical NVDIMM is marked as Unhealthy. Any logical disks associated with that NVDIMM are also marked as Unhealthy, with an operational status reason indicating a potential data loss. Windows places those logical disks in read-only mode to warn the user about a possible issue with the NVDIMM. If the user wants to write to the logical disk, they must run the “Reset-PhysicalDisk” PowerShell cmdlet on the affected logical disk, and then offline and online the volumes on the disk. The logical disk remains read-only until the user runs that cmdlet, even after reboots.

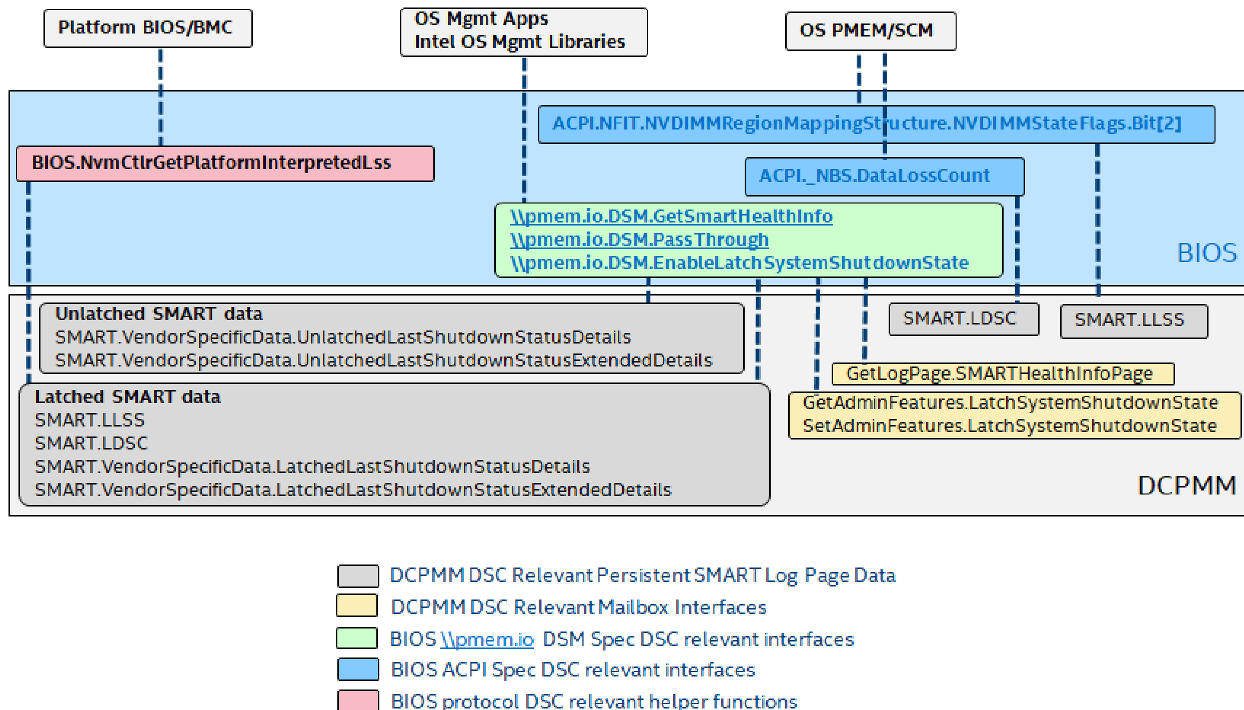
Applications can query the logical disk’s DSC by sending IOCTL_STORAGE_QUERY_PROPERTY with StorageDeviceUnsafeShutdownCount (defined in ntddstor.h) to the logical disk or volume stack.

At the time of this writing, the DSC information is not yet exposed to user space in Windows, so the PMDK checking of DSC is not yet available. This is expected to change in a future version of Windows.

4 Intel DCPMM specific implementation details

The following figure outlines the interfaces and data provided by Intel's DCPMM and the BIOS that are relevant to DSC.

Figure 4-1. Intel DCPMM specific support of DSC



DCPMM Interfaces

The Intel DCPMM implementation is broken into specific interfaces for supporting DSC include the following:

1. **Latched LSS and DSC implementation:**
   a. The Intel DCPMM supports a full synchronous latching mechanism to allow an OS to precisely control when the DCPMM is enabled for latching of the LSS and DSC. If the OS does not enable the latch, the current state of the LSS and DSC does not change. The interface to this data on the Intel DCPMM is through the GetLogPage.SMARTHealthInfoPage and GetAdminFeatures.LatchSystemShutdownState NVDIMM mailbox commands.
   b. The support for LSS is only utilized by the BIOS ACPI NFIT NVDIMM State Flags Bit 2 implementation where a simple state of the last shutdown is reported. Because of the limitations of using a flag instead of a count, it is not intended for use by applications.
2. **Unlatched DSC implementation:**

a. The Intel DCPMM support an unlatched DSC that always increments on any dirty shutdown without any latching required. Due to internal differences in how the DCPMM handles the unlatched shutdown relative to latched shutdowns, it is not utilized in any native interfaces for the OS.

b. **WARNING**: Intel does ***not*** advise using the *DCPMM.SMARTHealthInfoLogPage.IntelVendorSpecificData.**Unlatched**DirtyShutdownCount* to make platform shutdown clean/dirty decisions. There are cases where this count can increment on the next boot, resulting in a false positive. If power is removed from the DCPMM before it has reached MRC & training complete, the unlatched DSC will increment on the next boot. This data is often valuable for debug and triage so the unlatched data can be logged but the BIOS and OS components should not be making platform shutdown clean/dirty decisions based on this count. This window at the beginning of boot, where a false positive can occur, is not possible with the latched variant of the DSC since the latch is not enabled until the DCPMM has reached training/MRC complete. See the **BIOS Enabling of the Latch** details below on when the latch is enabled in the Intel implementation.

BIOS Interfaces

The Intel BIOS implementation is broken into specific interfaces for supporting DSC include the following:

1. **Latched LSS and DSC implementation:**
   a. The Intel BIOS supports reporting the state of the DCPMM SMART.LSS in the ACPI.NFIT.NVDIMMRegionMappingStructure.NVDIMMStateFlags.Bit[2]. This is a static value written into the NFIT at BIOS initialization time and does not get updated until the next system reset or reboot.
   b. The Intel BIOS supports the following pmem.io specified DSMs for reporting the DSC to an OS component using **DSM.GetSmartHealthInfo**, **DSM.PassThrough** (requires building a GetLogPage.SMARTHealthInfoPage mailbox command) and **DSM.EnableLatchSystemShutdownState.**
   c. The Intel BIOS will enable the Latched LSS and DSC implementation by enabling the LSS latch on every boot. This is to cover the Linux OS which does not enable the latch at the beginning of time and VMware ESX which only supports an unlatched DSC implementation and is expecting the platform to enable the latch at the appropriate time.
   d. The Intel BIOS will report the *latched* DSC through the ACPI specified _NBS.DataLossCount interface. See the ACPI V6.3 for the interface details. Since the BIOS is enabling the Latch for the OS, it is safe for the same latched DSC to be utilized in reporting _NBS.DataLossCount. It is not possible for the _NBS API to be executed until after the BIOS has enabled the latch. This allows all OS implementations to utilize the same heavily verified latched DSC path, independent of whether the OS supports a latched DSC or unlatched DSC mechanism.
2. **Unlatched DSC implementation:**
   a. There are no native BIOS ACPI interfaces that expose the unlatched DSC. While intended to be an unlatched interface, the ACPI specified _NBS.DataLossCount interface

returns latched DSC content, in the Intel implementation, and is covered in the latched DSC discussion above.

b. The **BIOS.NvmCtlrGetPlatformInterpretedUnlatchedDsc** interface was inspecting only unlatched SMART data so cannot reliably report a clean shutdown, exposing the caller to false positive count increment. *This API is not present in the DCPMM UEFI Protocol.*

BIOS Enabling of the Dirty Shutdown Count (DSC) Latch:
The Intel BIOS implementation requires the enabling of the LSS Latch to cover VMware ESX and Linux OS, which do not enable the latch on their own. Intel recommends the following for the BIOS implementation that enables the latch:

- The BIOS shall enable the latch AFTER the DCPMM has reached MRC complete phase and BEFORE handing off boot execution to the OS Boot Loader.
- The latch must be enabled before any OS/host/application write transactions can occur.
- MRC complete is reached at the point the BIOS has initialized the DCPMM and DDRT training has been completed. This occurs once the **SetAdminFeatures/DDRTIoInitInfo** mailbox command has executed successfully and DCPMM has returned successful status. The latch must be enabled after the DCPMM has completed training. Enabling the latch before this point could lead to false incrementing of the Latched DSC.

BIOS Disabling of the Dirty Shutdown Count (DSC) Latch:
The Intel DCPMM products require the latch to be disabled during the Warm Reset restart flows to prevent the LDSC incrementing as a false positive.

- A false positive increment in the LDSC can occur if the DCPMMs have initialized but the IMC memory controller is still initializing. During this 5-10 second window where the IMC is not ready, any power transitions that require ADR will result in a dirty shutdown count increment because the IMC ADR signals will not come through to the DIMM.
- To prevent the incorrect increment in the DSC during this window, the Intel BIOS will disable the latch when coming back up, after handling a CF9 slow or fast warm reset flow. The BIOS CF9 handler has executed the HW CF9 reset flow, HW has completed handling of the reset, and the reset vector will allow the preMRC BIOS to disable the latch using the SMBUS mailbox command.
    - By waiting for the CF9 HW flow to complete, all previous writes from the OS and Applications have been completely flushed from the IMC's WPQs to the DCPMMs persistence domain while the latch was enabled. This insures that any failure in the WPQ flush path would cause an increment in the DSC on the next boot.
    - By disabling the latch as early as possible in the BIOS warm boot path, this will insure that the latch is disabled during the critical window where the IMC is not ready, preventing a false positive increment of the DSC.