Platform Runtime Mechanism

Version: 1.1

Date: November 2022

Disclaimers

The material Contained herein is not a license, either expressly or impliedly, to any intellectual property owned or Controlled By any of the authors or developers of this material or to any Contribution thereto. The material Contained herein is provided on an "AS IS"Basis and, to the maximum extent permitted By applicable law, this information is provided AS IS AND WITH ALL FAULTS, and the authors and developers of this material hereby disclaim all other warranties and Conditions, either express, implied or statutory, including, But not limited to, any (if any) implied warranties, duties or Conditions of merchantability, of fitness for a particular purpose, of accuracy or Completeness of responses, of results, of workmanlike effort, of lack of viruses and of lack of negligence, all with regard to this material and any Contribution thereto. Designers must not rely on the absence or Characteristics of any features or instructions marked "reserved" or "undefined." The Unified EFI Forum, Inc. reserves any features or instructions so marked for future definition and shall have no responsibility whatsoever for Conflicts or incompatibilities arising from future Changes to them. ALSO, THERE IS NO WARRANTY ORCONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION,CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THE SPECIFICATION AND ANYCONTRIBUTION THERETO.

IN NO EVENT WILL ANY AUTHOR OR DEVELOPER OF THIS MATERIAL OR ANYCONTRIBUTION THERETOBE LIABLE TO ANY OTHER PARTY FOR THECOST OF PROCURING SUBSTITUTE GOODS OR SERVICES, LOST PROFITS, LOSS OF USE, LOSS OF DATA, OR ANY INCIDENTAL,CONSEQUENTIAL, DIRECT, INDIRECT, OR SPECIAL DAMAGES WHETHER UNDERCONTRACT, TORT, WARRANTY, OR OTHERWISE, ARISING IN ANY WAY OUT OF THIS OR ANY OTHER AGREEMENT RELATING TO THIS DOCUMENT, WHETHER OR NOT SUCH PARTY HAD ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

TERMINOLOGY

Term	Definition
ACPI	Advanced Configuration and Power Interface
AP	Application Processor. All the processor threads other than Startup-BSP
APIC	Advanced Programmable Interrupt Controller residing in the processor agents to generate and accept interrupt messages from other processor and I/O agents. Sometimes it is referred as Local APIC.
ARS	Address Range Scrubbing (ARS) allows platform to communicate persistent memory errors to system software. See pmem.io
BMC	IPMI compliant Board Management Controller.
BWG	BIOS Writer's Guide
BSP	Bootstrap Processor
CSP	Cloud Service Provider
DDDC	Double Device Data Correction
DMI	Proprietary data path between IIO and South bridge.
DPC	DIMMs Per Channel
EL	Exception Level
ESRT	The EFI System Resource Table (ESRT) provides a mechanism for identifying integrated device and system firmware resources for the purposes of targeting firmware updates to those resources.
GPIO	General Purpose Input/Output
GUID	Globally Unique Identifier
HPC	High Performance Computing
HVC	Hypervisor Call, an instruction in the Arm A-profile architecure which induces an exception taken to EL2.
IIO	Integrated IO – PCI Host Bridge logic that has been integrated in the CPU complex.
IPMI	Intelligent Platform Management Interface
KSR	Kernel Soft Reboot
KTI	Keizer Technology Interface
Legacy Processor	The processor that is connected to PCH via DMI link and manages the boot devices
LPC Bus	"Low Pin Count" connection used to connect to the BMC, super I/O device, and TPM.
MCA	Machine Check Architecture
Microcode	CPU's firmware coordinating CPU's actions, sometimes abbreviated as uCode
MMIO	Memory Mapped IO
NFV	Network Function Virtualization
NMI	Non-maskable Interrupts
OS	Operating System
OOB	Out of Band (such as via a BMC)
OSPM	Operating System Directed Power Management
PCH	Platform Controller Hub. Some might call this a Peripheral Controller Hub

Term	Definition
Pcode	Code executing on PCU
PCU	Power Control Unit, a controller inside the processor that performs power and other power management functions
PMEM	Persistent Memory (Such as Intel 3D XPoint / Non-Volatile DIMMs)
Powergood Reset	A reset during which power is removed. E.g. using a power-button. RTC power and Standby power is not affected.
QoS	Quality of Service
RAS	Reliability-Availability-Serviceabilty
SCI	System Control Interrupt as defined in ACPI Specification
SDDC	Single Device Data Correction
SiP	Silicon Provider
SMBIOS	System Management BIOS
SMC	Secure Monitor Call, an instruction in the Arm A-profile architecure which induces an exception taken to EL3.
SMI	System Management Interrupt
SMM	System Management Mode
SoC	System on Chip
SW	Software
TCB	Trusted Computing Base
UEFI	Unified Extensible Firmware Interface

References

- ACPI Specification
- UEFI Specification
- http://pmem.io/documents/NVDIMM_DSM_Interface-V1.6.pdf
- [SMCCC] DEN0028.E “SMC calling convention” (https://developer.arm.com/documentation/den0028/latest)
- [A-profile] DDI0487H.a “Arm® Architecture Reference Manual for A-profile architecture”

1. Introduction

Computer architectures have execution contexts of different privilege.

The properties of the execution context design and privilege/trust relations are architecture specific.

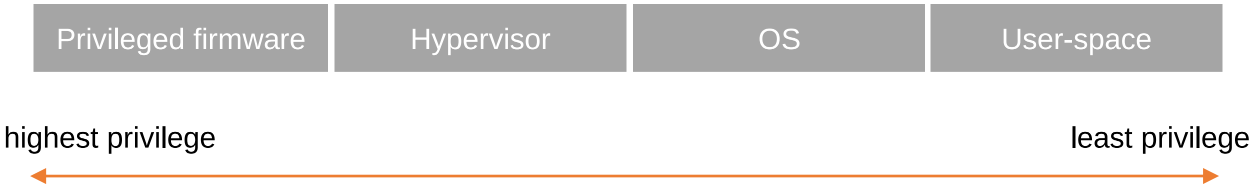


Figure 1-1: Privilege levels in computer architectures

Examples of privileged execution contexts are the System Management Mode (SMM) in the Intel Architecture and the Secure state in the Arm A-profile Architecture. These are placed in the left-most area of the diagram in Figure 1-1.

Resorting to a privileged firmware execution context, for functionality that does not require such privilege, can have detrimental consequences to the system performance and security guarantees.

This document describes the Platform Runtime Mechanism (PRM) as a means to move a certain classes of firmware functionality to a less privileged execution context. PRM, in combination with native OS drivers and offload to other hardware engines such as a Baseboard Management Controller (BMC), offers a path to reduce reliance on privileged firmware contexts.

1.1 x86 System Mangement Mode (SMM)

System Management Mode (SMM) is an operating mode in the x86 computer architecture. SMM code is written within the context of the system firmware and typically used for system-wide functions which are highly platform and silicon specific. Some examples would be RAS (e.g. Hardware Errors such as memory ECC, or to take corrective hardware actions to extend system uptime), power management, execution of OEM proprietary code, platform hardware events and implementation of hardware workarounds.

In order to enter SMM, a System Management Interrupt (SMI) must be generated by a platform events. SMI is a high priority, non-maskable, broadcast interrupt. On receipt of this interrupt, the processors in the system save their context in SMRAM and transition to SMM to execute the corresponding SMI Handler for the given event.

The SMI handler code then sets up its own environment (page tables, IDTs etc.), identifies the source of SMI and handles that event accordingly. The SMI handler is placed by the platform BIOS during BOOT to a special area of memory called System Management RAM (SMRAM).

SMRAM is invisible to OS/VMM. A processor executing in the 0-3 ring privilege levels will not be able to read from or write to SMRAM space. When the processor switches to SMM on receipt of an SMI, then processor executes code out of the SMRAM area.

There are two different categories of SMI sources, asynchronous and synchronous. SMIs due to platform and hardware events are asynchronous in nature. Software SMIs are invoked by writing to the SMI Command Port (0xB2 in Intel® Architecture platforms) and are synchronous in nature.

1.1.1 Issues with SMM – Problem Statement

As described in the previous section, SMM mode of operation has the following key attributes:

1. SMRAM (a memory space where SMI Handlers reside) is a Black Box to OS/VMM.
2. SMI Handlers run with their own page tables and accessibility to all system resources (higher privileged than ring-0).
3. SMI is non-maskable, broadcast and opaque to the OS/VMM.
4. Once in SMM Mode, all other interrupts are pended.

1.1.1.1 SMI and Perf/QoS degradation

SMI is a global / broadcast event which stalls all system processors. On receipt of a SMI, all the CPU threads in the system enter SMM mode immediately after completing their current instruction. This leads to unpredictable performance jitters.

Once inside the SMM environment threads are not available for OS use, and their execution is stalled until the SMI handler relinquishes control back to the previously executing context. The amount of execution time in SMM is called SMM Latency. In a typical 4-socket (4S) server class system, the latency can vary between ~300us to 1ms depending on core/thread count, the nature of the event being handled, and other factors.

1.1.1.2 SMI and Firmware complexity

SMM was never designed to handle so many asynchronous events in many-core environment. In reality, SMM handler has to deal with potential scenarios such as the following:

1. Some threads in a blocked state (WFS, VMX shutdown, LTS)
2. Some threads in the middle of executing a long flow instruction (wbinvd, ucode patch load) or in C6 state and will respond much later
3. Generation of more than one SMI in close proximity whereby some threads will observe a merged SMI (single SMI) while other observe multiple SMIs, leading to out-of-sync SMI scenarios.
4. Distribution of SMM sources that do not correspond to a single hierarchy
5. A narrow complexity threshold – solutions to address problems are difficult to adopt due to complexity analysis.

1.1.2 Summary

In conclusion, SMI is a very powerful mechanism for invoking runtime platform firmware, that has complete access to system memory and system hardware resources. SMI enables a large number of technologies to be employed in scenarios when system software is unavailable. For example, during an OS has crashed (for error harvesting) or on AC power-failure (to ensure data persistence for Non-Volatile DIMMs).

Although, this power comes with some notable downsides:

- Unpredictable performance jitters, as all the threads in the system are stalled for simple error collection, for example.
- Corner cases and race conditions, such as SMI Merge / out-of-sync SMI and OS kernel panics

There is a massive industry wide push to move away or reduce SMM footprint. The goal of this document is to provide a mechanism to reduce and eventually eliminate SMM usages that result in unpredictable performance jitters in the platform in OS runtime. There could still be usage models based on Planned SMI events, or SMI events during End-of-life of the boot, but those don’t give rise to unpredictable performance impact and is out of scope of this specification.

1.2 Arm A-profile Privileged Firmware

The A-class Arm® architecture defines the Exception Level (EL) and Security State [A-profile] concepts. The different exception levels and security states have a trust/privilege relationship, as shown in Figure 1-2.

Firmware executing in the Secure state can map and access pages that belong to the Host OS (Non-secure EL2 or EL1).

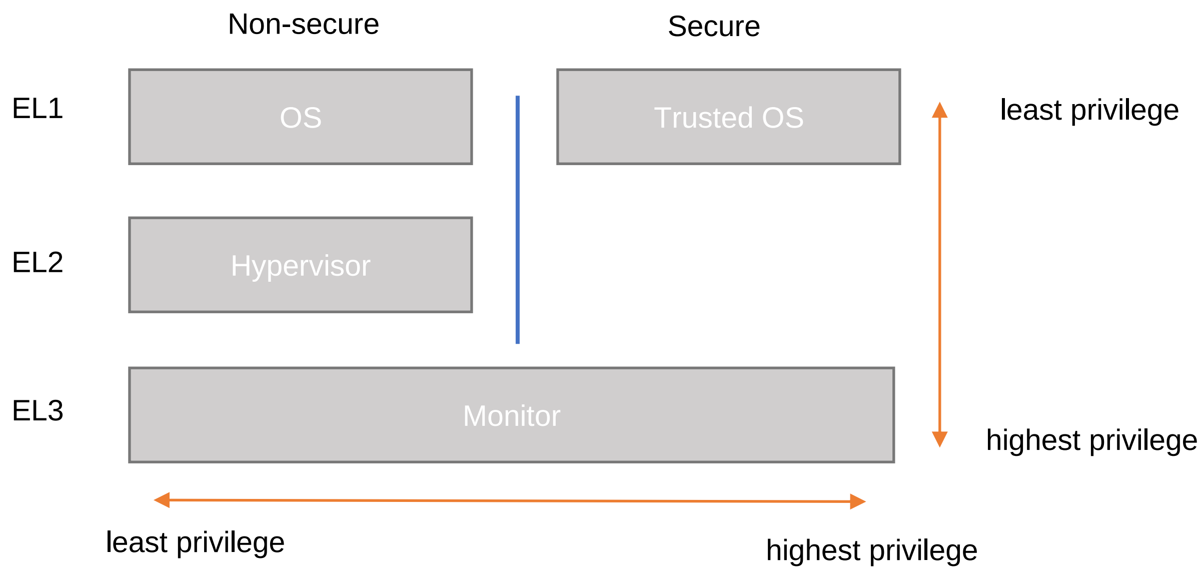


Figure 1-2: Arm A-profile Exception Level and Security State privilege relationships

The firmware that executes in the Secure state is said to run on a high privileged execution context compared to the Host OS.

Examples of firmware that is deployed in the Secure state are:

Trusted OS – used in mobile devices to handle use-cases like secure payments or media protection.

Standalone MM – used in the server ecosystem to deploy Tianocore MM drivers which communicate via the MM communicate with firmware executing in the Non-secure state.

Note: On Armv9, EL3 is in the Root security state. The Root security state is also considered privileged firmware.

1.2.1 Privileged Firmware Execution Context Properties

On the Arm A-profile architecture, each core on a multi-core processor can be in a different Exception Level and Security state at a given time instant.

Privileged firmware execution contexts may mask interrupts and thus be un-interruptible by a lesser privileged execution context (e.g. the Host OS), which may excessively limit the core availability to the Host OS.

1.3 ACPI and Privileged Firmware

Advanced Configuration and Power Interface (ACPI) is an open standard that defines a mechanism for operating systems to discover and configure hardware components, and actively perform device and platform power management. ACPI code is written in the ACPI Source Language (ASL) and typically shipped as a binary component in the form of a bytecode called ACPI Machine Language (AML) in the platform firmware.

Today’s ACPI firmware can call services implemented by privileged firmware by using a software defined invocation mechanism.

2. Privileged Firmware Usages

Understanding today’s privileged firmware use cases is essential for exploring potential alternatives. This section classifies privileged firmware applications today and corresponding mechanisms to eliminate privileged firmware usages for some of those usage models.

2.1 Privileged Firmware Invocation

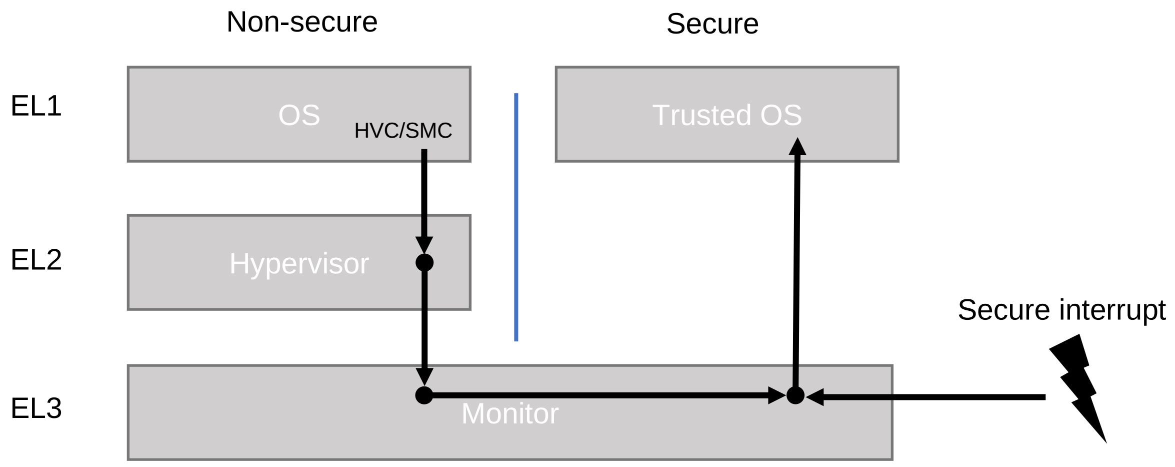
Privileged firmware execution can be triggered synchronously by software, termed SW invocation (e.g. a software triggered SMI in the Intel Architecture), or asynchronously by an interrupt or hardware generated exception, termed interrupt invocation (e.g. a Hardware SMI in the Intel Architecture). The actual mechanisms are architecture specific.

2.1.1 x86 Architecture Privileged Firmware Invocation

SMIs can be triggered either via software means or by the platform hardware. On x86 systems, a write to IO command port (e.g. 0xB2) will trigger a SMI. Software uses this path to trigger a SMI in order to invoke BIOS/Platform firmware services during system runtime. Hardware SMIs, on the other hand, are triggered by the platform hardware in response to system events such as errors, GPIO events etc.

2.1.2 Arm A-profile Privileged Firmware Invocation

- On the Arm A-profile architecture, privileged firmware is invoked by SW via the execution of a Secure Monitor Call (SMC) or a Hypervisor Call (HVC) instruction [SMCCC], or by HW via a secure interrupt.
-
-
-



-
-

Figure 2-1: High-level example of the invocation of privileged firmware on the Arm A-profile architecture

-

2.2 Privileged Firmware Usage Models

SW Invocation Usage Model:

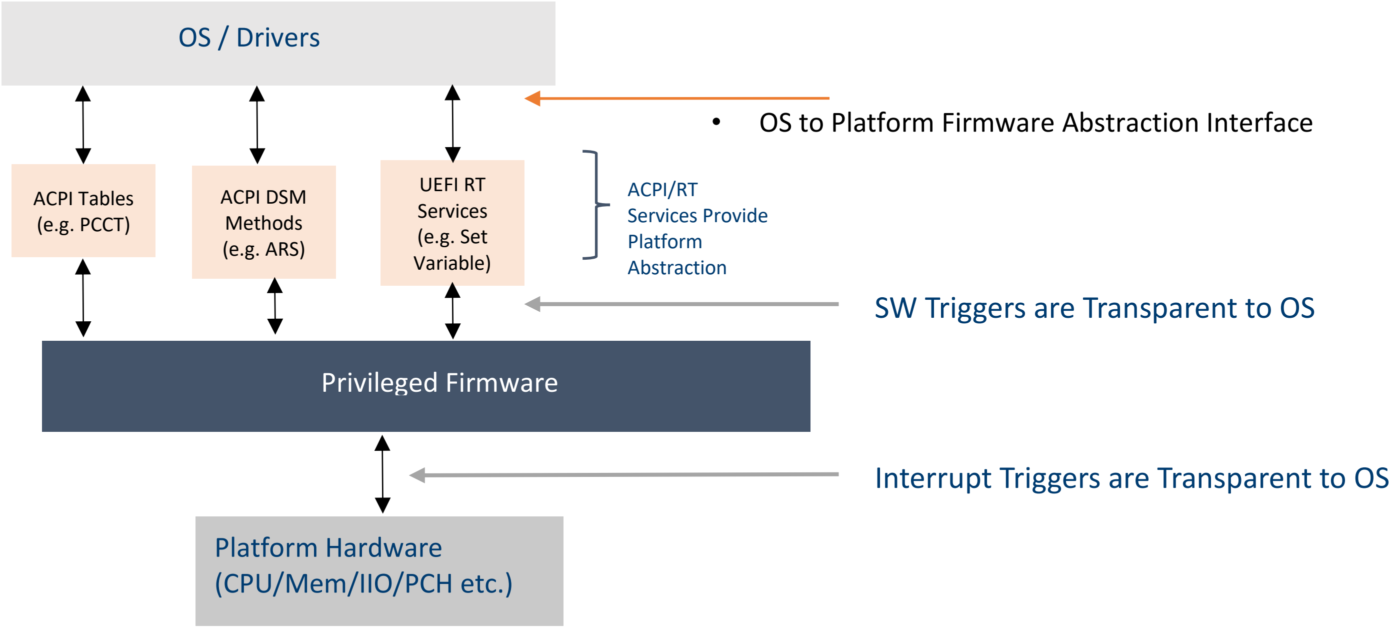
As shown in Figure 2 below, OS level entities typically use ACPI and UEFI interfaces as an abstraction to invoke runtime platform firmware services. These OS and BIOS interfaces then invoke privileged firmware internally, if native code execution is required. In other words, the fact that privileged firmware was invoked is kept transparent to the OS by these abstraction interfaces.

A key to reduce the privileged execution context footprint with compatibility to existing software is to the retain the same software interface to OS entities but provide an alternate means for invoking platform code execution context from ACPI. Platform Runtime Mechanism (PRM), as explained in subsequent sections provides such an alternative for certain cases.

Interrupt Usage Model:

Interrupts are events triggered by the platform hardware in response to platform events such as memory and other system errors, thermal events, GPIOs etc. These are transparent to the OS as well. Migrating some of these usages out of privileged firmware could involve a combination of PRM and assistance from an Out-Of-Band agent, such as BMC.

Figure 2-2 Privileged Firmware Triggers


OS / Drivers
OS to Platform Firmware Abstraction Interface
ACPI Tables (e.g. PCCT)
ACPI DSM Methods (e.g. ARS)
UEFI RT Services (e.g. Set Variable)
ACPI/RT Services Provide Platform Abstraction
SW Triggers are Transparent to OS
Privileged Firmware
Interrupt Triggers are Transparent to OS
Platform Hardware (CPU/Mem/IIO/PCH etc.)

As shown in Figure 2-2, OS/VMM entities invoke platform functionalities in runtime for a plurality of reasons. One of the main factors is that, it is the platform firmware that has intimate knowledge of the silicon and platform features and configurations and carrying this knowledge as part of the OS entity is a logistical challenge for broader enabling of Off-the-shelf operating systems. Hence OS entities rely on platform abstractions such as Advanced Configuration and Power Management Interface (ACPI).

Though ACPI Source Language (ASL) provides runtime space for handling platform events, development and debug of ASL poses a special challenge due to the interpreted and highly restrictive nature of ASL language and runtime environment. Also, being architecture neutral, executing ISA specific instructions is not possible in ASL context. To overcome the restrictive environment of ASL, BIOS developers often resort to tricks like dropping into a privileged firmware handler to carry out BIOS tasks.

By providing a mechanism to transition to an environment wherein ASL code can invoke platform runtime native code at the same privilege level, we alleviate the need to drop into privileged firmware handler only for the sole purpose of executing native code.

Example 1: Address translation from System Physical Address (SPA) to DIMM Address (DA).

Linux distros have an EDAC driver for error handling, and hitherto carried the knowledge of doing the address translation as well (e.g. translating a given Physical Address to a Socket/Memory Controller/Channel/Rank/Bank/Column/Row). Address translation is a feature that is highly silicon dependent and varies between generations of silicon. It might also depend on third party silicon such as xNC (Node controllers) that some OEMs use, and in the future will depend on the CXL devices populated in the platform. Hence an ACPI _DSM was created as the abstraction interface.

Example 2: PSHED Plug-in

PSHED drivers are WHEA/APEI OS drivers for error handling. Plug-in model was created to enable the driver to cater to platform and silicon variances. But this proved to be a challenge for wide deployment, and hence ACPI Tables were created (EINJ, ERST, HEST etc.) using which the platform firmware is invoked to handle these variances.

Example 3: NVDIMMs

NVDIMMs have introduced a new set of ACPI _DSM interfaces (http://pmem.io/documents/NVDIMM_DSM_Interface-V1.6.pdf) as a way to abstract the platform and NVDIMM technology variances from the OS/VMM. These _DSM drop into SMI to be able to handle the tasks.

Example 4: Arm OEM/SiP Specific Runtime Functionality

Arm vendors deploy functionality in the Secure state to configure and control IP which may not be part of the platform TCB. This functionality does not require a high privileged execution context.

Examples of functionality deployed by the OEM/SiP are:

- PCI quirks to control hardware
- Control peripherals via their MMIO regions.

There are more such examples of the OS entities using platform abstraction. The SMM elimination strategy should ensure compatibility with the existing abstraction interfaces.

2.3 Categories of firmware handlers

Privileged firmware handlers can be broadly classified as:

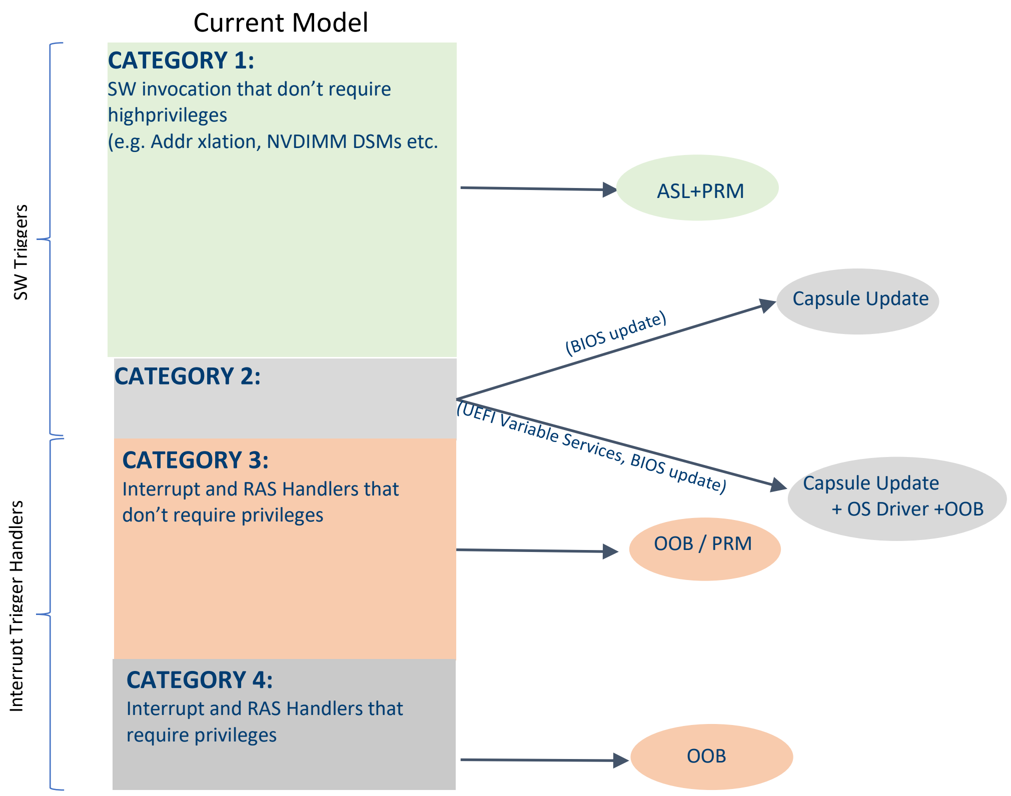
- Category 1: SW triggered firmware handlers that don’t require high privileges
- Category 2: SW triggered firmware handlers that require high privileges
- Category 3: Interrupt firmware handlers that don’t require high privileges
- Category 4: Interrupt firmware handler the require high privileges

Note: high *privileges means that there are certain hardware resources (e.g. registers) that can be written to only in a high privilege execution context.*

SW triggered firmware handlers (Category 1 and 2) are invoked by software. Interrupt triggered handlers (Category 3 and 4) are invoked by platform hardware events such as system Errors.

Platform Runtime Mechanism (PRM) provides a means to eliminate Category 1 SMM handlers and in some cases can be used to reduce Category 3 handlers as well.

Figure 2-3 Categories of Privileged Firmware Handlers


Current Model
CATEGORY 1:
SW invocation that don't require highprivileges
(e.g. Addr xlation, NVDIMM DSMs etc.
CATEGORY 2:
CATEGORY 3:
Interrupt and RAS Handlers that don't require privileges
CATEGORY 4:
Interrupt and RAS Handlers that require privileges
SW Triggers
Interrupt Trigger Handlers
ASL+PRM
(BIOS update)
Capsule Update
(UEFI Variable Services, BIOS update)
Capsule Update + OS Driver +OOB
OOB / PRM
OOB

Category 1 handlers will be migrated to use PRM.

Category 2 handlers are mainly related to UEFI authenticated variable services. Not in scope for this Specification

Certain Category 3 handlers can be handled by PRM as explained in later sections.

Category 4 handlers are mainly related to Uncorrectable Hardware Errors and advanced RAS features. Not in scope for this Specification

2.4 Category 1 Usages

These are SW triggered from an abstraction interface such as ACPI _DSM methods. There is a plethora of such _DSM methods that today invoke a SW trigger so that complex algorithms and tasks can be handled in a native code execution context. Providing an alternate means of executing native code using PRM alleviates the need to invoke high privileged firmware for this category of handlers. Examples include DSMs for RAS (such as address translation) and DSMs for supporting Non-Volatile DIMMs.

2.5 Category 3 Usages

Interrupts can be generated for asynchronous platform events such as memory and IO errors. In response, the handlers in high privileged firmware collect more information about the errors and surface them to the OS or log them to a BMC. In addition to the above, the handlers in high privileged firmware can trigger RAS events to remediate or mitigate the errors that caused the initial interrupt.

Though PRM is mainly designed with Category 1 SW triggered functionality in mind, Category 3 Interrupt handlers can be migrated to PRM, if so desired by the platform vendor / OEM.

Category 3 handlers are commonly used for correctable error harvesting and reporting. By generating an interrupt delivered to the Host OS instead of privileged interrupt for these error conditions, ASL code can be invoked which can utilize PRM for error harvesting and reporting.

3. Platform Runtime Mechanism Overview

Platform Runtime Mechanism (PRM) introduces the capability of transitioning certain usages that were hitherto executed out of privileged firmware, to a code that executes with the OS/VMM context. Such usages are those that don't require higher privileges (Category 1) and a sub-set of interrupt handlers that don't require higher privileges (Category 3) . This eliminates many of the cons present when executing the same code within a privileged firmware execution environment. The code can also be updated within the Host OS context through targeted online servicing of specific sets of functionalities

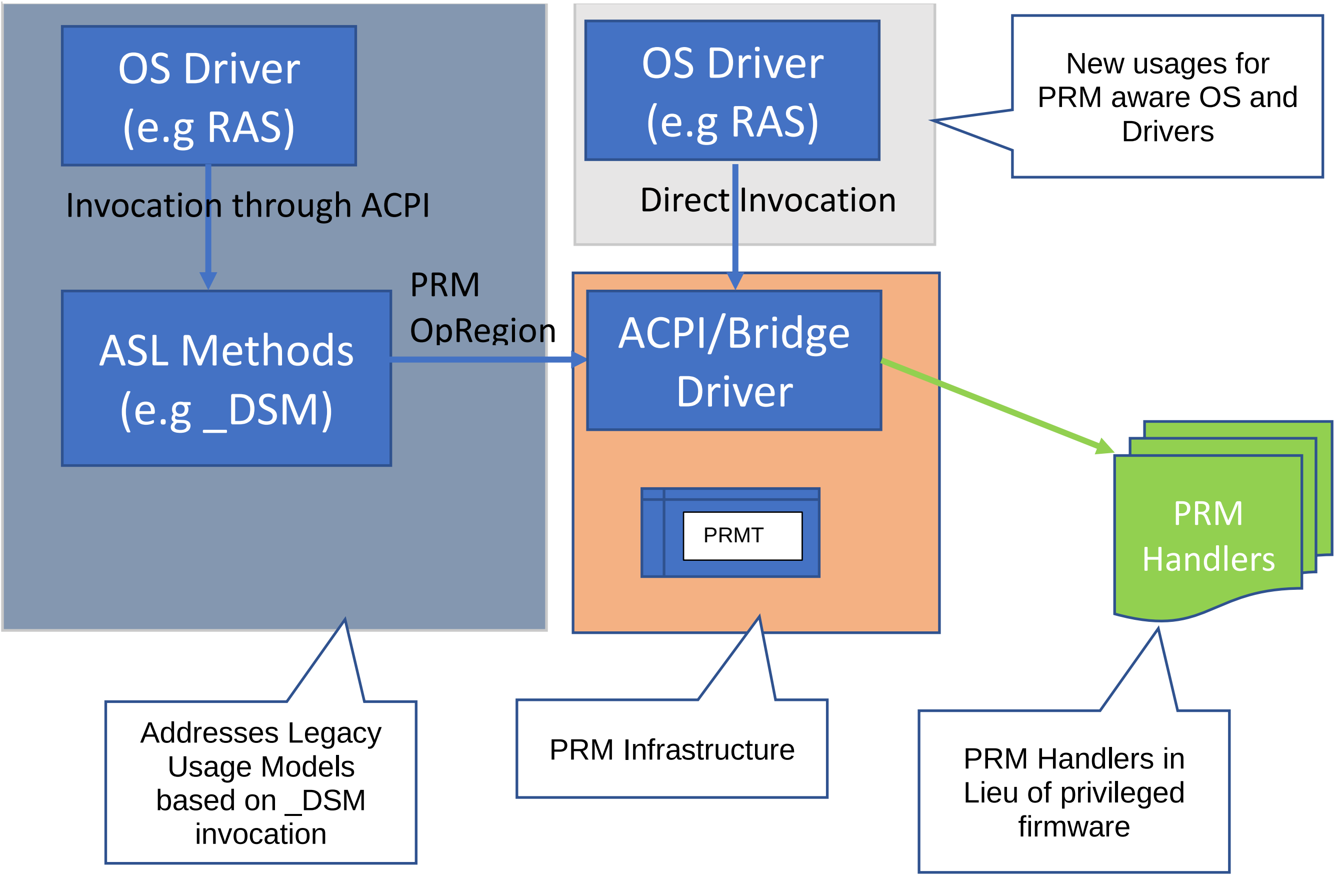
As shown in the figure below (Figure 3-1), Platform Runtime Mechanism provides an ACPI Interpreter based infrastructure to invoke runtime platform firmware handlers. These runtime handlers are called PRM Handlers and are placed by the BIOS during boot (and updatable in OS runtime) in a runtime area reserved for firmware usage (such as UEFI Runtime area).

***NOTE**: The ACPI Interpreter based PRM infrastructure is the PRM OpRegion Handler in ACPICA/ACPI subsystem and can also logically be implemented as an independent driver (Bridge Driver) in certain implementations. <u>This Specification interchangeably uses the terms 'Bridge Driver', 'PRM OpRegion Handler', 'ACPI Driver', ACPI Interpreter or 'ACPICA' to mean the same thing.</u>*

PRM handlers can be invoked by two means (detailed in Section 5)

1. Directly from an OS driver - if the OS driver and the OS ACPI subsystem is PRM aware.
2. From ASL context – if the OS driver is not PRM aware and uses _DSM instead, or platform events that trigger an interrupt to the Host OS.

Figure 3-1 PRM Overview



3.1 PRM Requirements

1. PRM handlers must be code capable of executing within the context of a runtime OS.
2. PRM handlers loaded at boot time should be part of the firmware boot chain of trust.
3. PRM handler's internal pointers should be fixed-up, if needed, during boot based on the OS virtual address.
4. PRM handlers should be OS agnostic and not dependent on any OS provided support APIs.
5. PRM handler should be securely replaceable/over rideable in runtime without resetting the system.
6. PRM handlers should be executable by the OS, interruptible and single threaded.
7. PRM handlers shall only access MMIO registers that are listed in the handler's parent module's RuntimeMmioPages field in the PRMT.
8. PRM handlers must not contain any privileged instructions.

3.2 Architecture-specific PRM requirements

3.2.1 Arm A-profile architecture (AArch64)

9. PRM handlers must respect the detailed calling convention specified for AArch64 platforms in the UEFI specification.

Any platform firmware / BIOS environment that satisfies the above requirement can make use of PRM. The UEFI environment is able to support PRM with minimal overhead.

3.3 PRM and UEFI

PRM is not confined to UEFI boot, however, the above requirements are largely supported with functionality in place for UEFI Runtime Services.

1. UEFI Runtime Services are an industry standard way of publishing code from firmware that is executable in OS runtime. The Runtime Services definition provides an Application Binary Interface (ABI) for PRM handlers and pre-existing requirements for executing conditions such as available stack size for PRM handler invocation.
2. The UEFI Secure Boot chain-of-trust already provides a mechanism to authenticate PRM modules that are included as components in the firmware boot image.
3. Runtime virtual address fixups are commonly performed in runtime driver code so they can access resources at OS runtime. Firmware support is already available to map a given physical address to its virtual address.

PRM Requirement	**UEFI Based Boot**	**Non-UEFI Boot**
PRM handlers execute at OS runtime and are published by firmware in pre-OS boot	UEFI Runtime Services are an industry standard way for firmware to publish OS runtime code	Need to build support to publish runtime code by the BIOS, that is OS visible
Chain of Trust	UEFI Secure Boot	No standard mechanism
Pointer Fix-ups	Built in support	No standard mechanism
OS-Independent	Yes	Implementation specific

3.4 PRM Loading and Invocation

1. During boot, the firmware discovers PRM modules included in the platform firmware flash image.
2. During boot, the firmware publishes the PRM ACPI table (PRMT) to describe the PRM modules, handlers, and related structures such as context buffers for the given boot.
3. During boot, firmware allocates any required buffers and, in some cases, populates the buffer contents as is the case with the static data buffer.

4. During OS runtime, OS code invokes PRM handlers via the direct call mechanism or with a _DSM.

3.5 PRMT Table Overview

The PRMT table is an ACPI table published by the BIOS during boot, which advertises the pointers to the PRM handlers. This information is then used by the ACPI Interpreter to invoke the PRM handler(s). The PRMT table exposes a hierarchical structure.

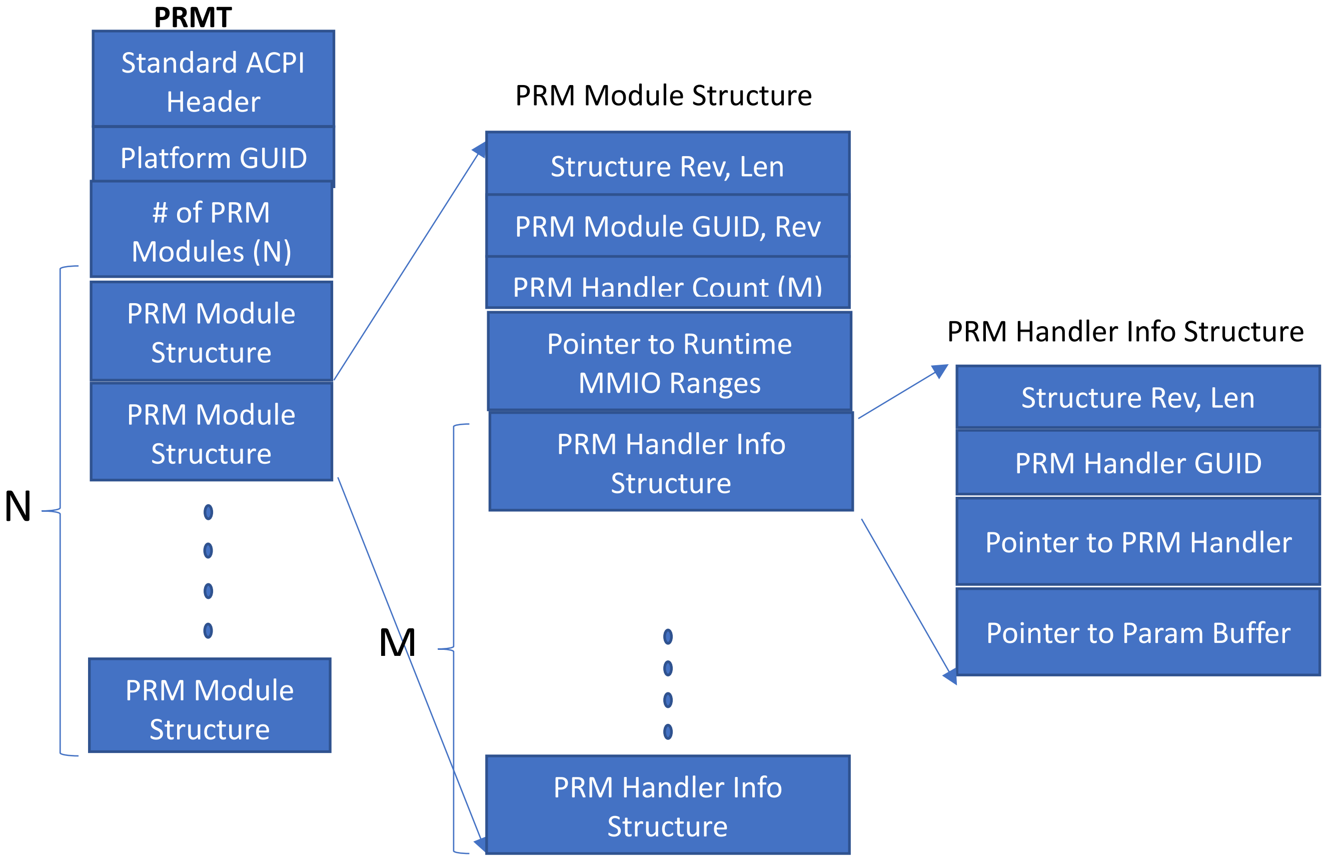
A PRM module consists of a set of PRM handlers. A PRM module is based on a feature that it supports. For example, there could be a RAS module, or a NVDIMM Module etc., with each module containing multiple handlers.

As shown in Figure 3-2 below, the PRMT table consists of an array of PRM Module Structures.

- Each PRM Module Structure will have a pointer to MMIO Ranges that the PRM handlers use in runtime.
- Each PRM Module Structure will have an array of PRM Handler Info Structures
    - Each Handler Info Structure will have a GUID Identifying the handler, and the corresponding pointer to the handler.
    - Each Handler Info Structure may optionally have a pointer to an ACPI Parameter Buffer. The ACPI Parameter Buffer is a BIOS reserved range of memory during boot, which is used by the invoker (e.g. ASL Code) and the PRM Handler for parameter passing. The format of the data is a contract between the invoker and the handler.

- For Direct Invocation, the invoker will allocate the parameter buffer (explained in section 5)

Figure 3-2 PRMT Topology



4. ACPI Tables

PRM ACPI tables are used to communicate PRM information between the firmware and operating system. The ACPI table format is standardized as described in this section. The system BIOS (e.g. UEFI firmware) is expected to construct the ACPI tables during the boot services portion of the boot flow and then populate the tables before loading the OS boot loader.

4.1 Platform Runtime Mechanism Table (PRMT)

Table 4-1 PRMT Top-Level Table

Field	Byte length	Byte offset	Description
Header			
Signature	4	0	'PRMT'. Signature of the PRM ACPI table.
Length	4	4	Length, in bytes, of the entire PRM ACPI table.
Revision	1	8	For this version, the value is 0.
Checksum	1	9	The checksum is computed when the table is installed in the firmware boot environment.
OEM ID	6	10	Original equipment manufacturer (OEM) ID.
OEM Table ID	8	16	The system firmware OEM Table ID.
OEM Revision	4	24	OEM revision of PRMT for the supplied OEM Table ID.
Creator ID	4	28	Vendor ID of the utility that created the table.
Creator Revision	4	32	Revision of the entity that created the table.
PrmPlatformGuid	16	36	A GUID that uniquely identifies the current platform, to assist OSPM in platform targeting for runtime PRM Module updates. NOTE: Some OSPMs might use proprietary mechanisms for targeting instead of this field.
PrmModuleInfoOffset	4	52	Offset, in bytes, from the beginning of this table to the first PRM Module Information entry.
PrmModuleInfoCount	4	56	The number of PRM Module Information entries.
PrmModuleInfoStructure [PrmModuleInfoCount]	Variable	Prm Module Info Offset	An array of PRM Module Information entries for this platform.

4.1.1 PRM Module Information Structure

Table 4-2 PRM Module Information Structure (PrmModuleInfoStructure)

Field	Byte length	Byte offset	Description
StructureRevision	2	0	Revision of this PRM Module Information Structure.
StructureLength	2	2	Length, in bytes, of this structure, including the variable length PRM Handler Information Structure array.
Identifier	16	4	The GUID for the PRM module represented by this PRM Module Information Structure.

MajorRevision	2	20	The major revision of the PRM module represented by this PRM Module Information Structure.
MinorRevision	2	22	The minor revision of the PRM module represented by this PRM Module Information Structure.
HandlerCount	2	24	Indicates the number of PRM Handler Information Structure entries that are present in the PrmHandlerInformationStructure[] field of this structure.
HandlerInfoOffset	4	26	Offset, in bytes, from the beginning of this structure to the first PRM Handler Information Structure.
RuntimeMmioPages (PRM_RUNTIME_MMIO _RANGES *) (physical address)	8	30	A pointer to a PRM_RUNTIME_MMIO_RANGES structure. The structure is used to describe MMIO ranges that need to be mapped to virtual memory addresses for access at OS runtime. This pointer may be NULL if runtime memory ranges are not needed.
HandlerInfoStructure [HandlerCount]	(Prm Handler Count) * sizeof (Prm Handler Info Structure)	38	An array of PRM Handler Info Structures. Each structure represents a PRM Handler present in the PRM Module represented by this structure.

4.1.2 PRM Handler Information Structure

Table 4-3 PRM Handler Information Structure (*HandlerInfoStructure*)

Field	Byte length	Byte offset	Description
StructureRevision	2	0	Revision of this PRM Handler Information Structure.
StructureLength	2	2	Length, in bytes, of this structure.
Identifier	16	4	The GUID for the PRM Handler represented by this PRM Handler Information Structure.
PhysicalAddress	8	20	The address of the PRM Handler represented by this PRM Handler Information Structure.
StaticDataBuffer (PRM_DATA_BUFFER *) (physical address)	8	28	A physical address pointer to the static data buffer allocated for this PRM handler. The static buffer is intended to be populated in the firmware boot environment. This pointer may be NULL if a static data buffer is not needed.

AcpiParameterBuffer (PRM_DATA_BUFFER *) (physical address)	8	36	A physical address of a parameter buffer for this PRM handler that is only used in case of ASL invocation of the handler. The buffer is allocated in the firmware boot environment and typically updated at runtime by ASL. The pointer may be if a parameter buffer is not required in case of ASL invocation, or if ASL invocation is not used.

4.2 Explanation of Buffers Used

This section explains the usages of various buffers and data structures mentioned in PRMT

4.2.1 Static Data Buffer

The static data buffer is a data buffer allocated in the BIOS boot phase whose contents (and size) are implementation specific. The Boot BIOS is also responsible for populating this static data buffer, from various implementation-specific data sources. For example, BIOS setup menu options, board straps, SOC fuse values, etc.

While this is a per PRM specific data buffer as defined, some implementations might choose to optimize by placing one instance of the structure in memory and have all the PRM entries in the module point to this same structure.

While the contents are arbitrary, the buffer header is standardized below.

A pointer to this *StaticDataBuffer* is passed to the PRM Handler during invocation.

Table 4-4 PRM Static Data Buffer Structure *(StaticDataBuffer)*

Field	Byte length	Byte offset	Description
Header			
Signature	4	0	'PRMS'. Signature of a PRM Static Data Buffer Header structure.
Length	4	4	The total length in bytes of this PRM data buffer including the size of the PRM_DATA_BUFFER_HEADER.
Data	Varies	8	The variable length data specific to a PRM module the constitutes the data in the buffer.

4.2.2 ACPI Parameter Buffer

The *AcpiParameterBuffer* is a data buffer allocated in the BIOS boot phase that is only used in the ASL invocation path.

The buffer is used for passing parameters between the ASL based caller and the PRM handler. The internal data format of the ParameterBuffer is a contract between the caller and the PRM handler and outside the scope of this document. If the ParameterBuffer is not provided, NULL will be passed as this argument.

While the contents are arbitrary, the buffer header is standardized below.

A pointer to this *AcpiParameterBuffer* is passed to the PRM Handler during invocation.

Table 4-5 PRM ACPI Data Buffer Structure *(AcpiParameterBuffer)*

Field	Byte length	Byte offset	Description
Header			
Signature	4	0	'PRMP'. Signature of a PRM ACPI Parameter Data Buffer Header structure.
Length	4	4	The total length in bytes of this PRM data buffer including the size of the PRM_DATA_BUFFER_HEADER.
Data	Varies	8	The variable length data specific to a PRM module the constitutes the data in the buffer.

4.2.3 Module Runtime MMIO Ranges

A PRM module is responsible for creating an array of MMIO range descriptors using the structures below to describe ranges that may be accessed by a PRM handler in the module. The OS is responsible for populating the VirtualBaseAddress and ensuring that memory is marked as a memory space type that allows firmware to retrieve the virtual memory mapping for the address range.

A pointer to this *RuntimeMmioPages* is passed to the PRM Handler during invocation.

4.2.3.1 PRM_MODULE_RUNTIME_MMIO_RANGE

This structure describes a single runtime MMIO range that a PRM module declares may be used by a PRM handler in the module.

Table 4-6 PRM_MODULE_RUNTIME_MMIO_RANGE Structure

Field	Byte length	Byte offset	Description
PhysicalBaseAddress	8	0	Physical base address of the MMIO range.
VirtualBaseAddress	8	8	Virtual address of the MMIO range.
Length	4	16	Length of the MMIO range in bytes.

4.2.3.2 PRM_MODULE_RUNTIME_MMIO_RANGES

This structure describes an array of PRM_MODULE_RUNTIME_MMIO_RANGE structures declared by a PRM module that may be used by a PRM handler in the module.

Table 4-7 PRM_MODULE_RUNTIME_MMIO_RANGES Structure

Field	Byte length	Byte offset	Description
Count	8	0	The number of PRM_MODULE_RUNTIME_MMIO_RANGE elements that follow.
RuntimeMmioRange [Count]	8	8	Array of PRM ModuleRuntime MMIO Range Structures. Each structure represents a MMIO range used by the PRM Module represented by this structure.

5. Invocation of PRM Handlers

As described earlier, PRM handlers can be invoked by two means

1. Directly from an OS driver - if the OS driver and the OS ACPI subsystem is PRM aware.
2. From ASL context – if the OS driver is not PRM aware and uses _DSM instead, or platform events that trigger an interrupt to the Host OS invoking _Lxx methods.

5.1 Direct Call vs ASL Based Invocation

For PRM aware OS and OS drivers, a direct call is recommended and preferred for at least the following reasons:

1. _DSM implementation brings an programming dependency for PRM into the system ACPI FW (as opposed to only a declarative table). This code is required to act in lieu of the OS device driver to update the AcpiParameterBuffer for the active PRM handler. This requires an AML debugger to debug and if a bug is present, a full system reboot is needed to update the ASL code logic loaded by system firmware.
2. _DSM constrains the OS driver's ability to interact with PRM. For example, in the case of direct call, the OS device driver can directly call into PRM module update lock and unlock APIs around the PRM calls that need to be protected (see section 7). In _DSM invocation, this is outside the control of the device driver and must be handled internally within the corresponding _DSM.

   As another example, in the case of direct call, the OS device driver can directly allocate and populate a buffer of information shared with a PRM handler. In _DSM invocation, data can only be shared using a fixed buffer allocated by firmware that is populated at runtime by AML code loaded during boot. If during a runtime PRM update, a PRM handler depends upon a parameter buffer that did not previously or the layout of the buffer changes, the corresponding ASL must be modified which requires a system reboot. In direct call, ASL does not need to be modified.
3. _DSM invocation requires more overhead to execute AML bytecode in the ACPI interpreter.

On the other hand, there is a significant install base in the industry that relies on _DSM mechanisms as an abstraction to invoke platform firmware services from OS drivers. To maintain compatibility with the installed base until they are deprecated, the _DSM invocation path provides a mechanism to invoke PRM handlers from ASL context. In addition, certain hardware events can generate an interrupt to the Host OS which will enter ACPI context via a _Lxx method, from which PRM handlers can be invoked.

5.2 Invocation Mechanism - Overview

The caller (either from ASL for Direct Call from OS Driver) passes the following information to the ACPI Bridge Driver

1. GUID of the PRM handler to be invoked
2. In the case of Direct call, the pointer to a *ParameterBuffer* (allocated by the caller)

The ACPI Bridge Driver then

1. Identifies the PRM Handler pointer corresponding to the GUID that was passed
   a. Convert the PRM Handler Pointer from a Physical Address to a Virtual Address.
2. Extracts the Static Data Buffer Pointer and the Runtime MMIO Ranges Pointer and create a *ContextBuffer* (see
3. Table **5-1**), which is passed to the PRM Handler.
4. In the case of ASL call, extracts the *AcpiParameterBuffer* pointer from PRMT. In the case of direct call, the *ParameterBuffer* pointer is passed by the caller and *AcpiParameterBuffer* is ignored.
5. Invokes the PRM handler with the following calling convention

```
EFI_STATUS
PRM_EXPORT_API
(EFIAPI *PRM_HANDLER) (
    IN  VOID                        *ParameterBuffer OPTIONAL,
    IN  PRM_MODULE_CONTEXT_BUFFER  *ContextBuffer   OPTIONAL
    );
```

5.3 Direct Invocation

ACPI Bridge Driver exposes an IOCTL that can be invoked by a PRM aware OS driver. In the case of Direct Invocation, the PRM aware OS driver calls into this IOCTL, by passing the GUID of the PRM handler to be invoked, and a pointer to the *ParameterBuffer*.

NOTE: Direct Invocation is a mechanism that is intended for future use, in an environment where the OS ACPI subsystem, OS Drivers and BIOS are PRM compatible.

5.4 ASL (_DSM) Based Invocation

To be able to invoke runtime code from ASL, a bridging mechanism needs to be in place. ASL provides for an OpRegion handler that is synchronous in nature. The PRM extends this by introducing a new ‘PRM’ OpRegion Type. Further details regarding ACPI-specific structures introduced for PRM support are described in the _DSM invocation section in the appendix.

5.5 Context Buffer

The Context Buffer is a well-defined buffer per PRM handler that describes resources available to the handler during its execution. This buffer is allocated within the OS and the OS is responsible for converting physical addresses to virtual addresses if applicable.

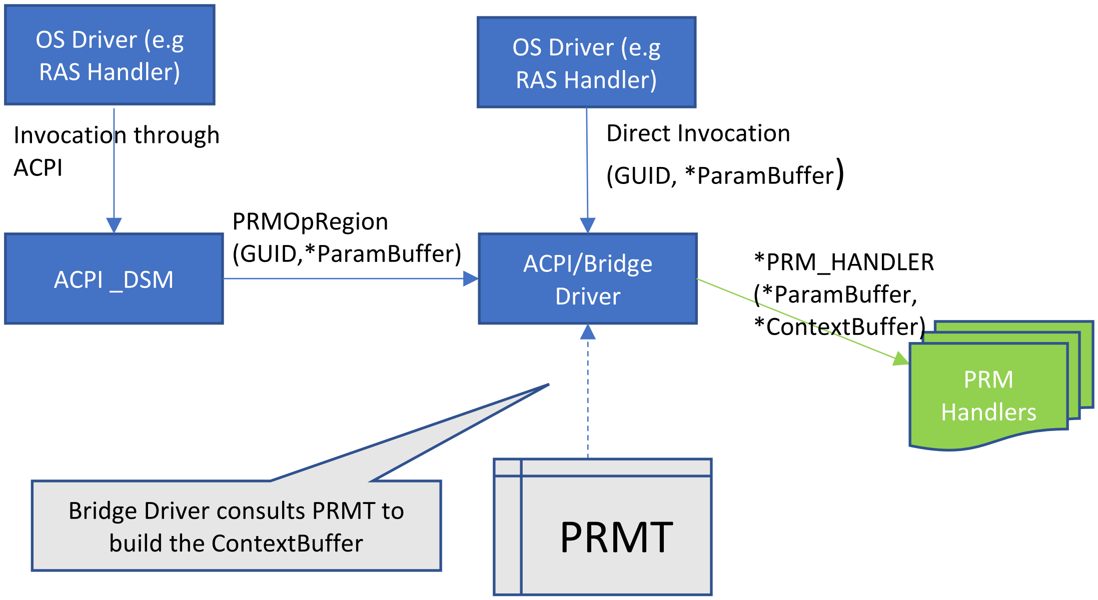
Table 5-1 Context Buffer Structure (*ContextBuffer)*

Field	Byte length	Byte offset	Description
Signature	4	0	‘PRMC’. Signature of the PRM Module Context Buffer structure.

Revision	2	4	Revision of this PRM Module Context Buffer structure.
Reserved	2	6	Reserved
Identifier	16	8	The GUID of the PRM handler represented by this structure.
StaticDataBuffer **(PRM_DATA_** **BUFFER)** **(virtual address)**	8	24	A virtual address pointer to the static data buffer allocated for the PRM handler represented by this context instance. The static buffer is intended to be populated in the firmware boot environment. This pointer may be NULL if a static data buffer is not needed.
RuntimeMmio **Ranges** **(PRM_MODULE_** **CONFIG_RUNTIM_** **MMIO_RANGES)** **(virtual address)**	8	32	A virtual address pointer to an array of PRM_RUNTIME_MMIO_RANGE structures that describe MMIO physical address ranges mapped to virtual memory addresses for access at OS runtime. The MMIO ranges are intended to be populated in the firmware boot environment. The virtual address pointer should also be set in the firmware boot environment. This pointer may be NULL if runtime memory ranges are not needed.

The Context Buffer is allocated by the OS Bridge Driver. This is constructed using data discovered in the PRMT ACPI table (*StaticDataBuffer* and *RuntimeMmioPages*) and passed as an argument to PRM handlers. For any pointer that is NULL in the ACPI table, a NULL pointer may be passed to PRM handlers. PRM handler code should expect and handle this case.

Figure 5-1 Invocation Summary


OS Driver (e.g RAS Handler)
Invocation through ACPI
ACPI _DSM
PRMOpRegion (GUID,*ParamBuffer)
OS Driver (e.g RAS Handler)
Direct Invocation
(GUID, *ParamBuffer)
ACPI/Bridge Driver
*PRM_HANDLER (*ParamBuffer, *ContextBuffer)
PRM Handlers
Bridge Driver consults PRMT to build the ContextBuffer
PRMT

6. PRM Software Organization

At a high-level, PRM collateral can be viewed as three levels of increasing granularity:

1. PRM interface – A software interface that encompasses the entirety of firmware functionalities available to OS runtime
2. PRM module – An independently updatable package of PRM handlers. The PRM interface can be composed of one or more updatable PRM modules. This requirement allows for independent authoring and packaging of OEM and IHV PRM code.
3. PRM handler – The implementation of a single piece of PRM functionality as identified by a GUID.

6.1 PRM Module Image Format

The PRM module format is designed to be loaded during boot by the BIOS (Baseline PRM), and to be replaced in OS runtime without needing a reboot, if so desired.

A PRM module is composed of a PE/COFF binary image with certain characteristics that uniquely identify the image as a PRM module. These characteristics are described in this section.

A PRM-compliant PE/COFF image contains the following notable sections:

- An Optional header with the MajorImageVersion and MinorImageVersion fields set to appropriate value for the PRM module.
    - In most environments, this allows the image version to be obtained using filesystem APIs. For example, an OS loader could determine whether a given binary version is greater than the current version without needing to load the binary into memory and computing an address to an object using a relative virtual address.
- An .edata section that contains references to the following elements:
    - PRM Module Export Descriptor - A structure that describes the PRM Module and contains an array of PRM Handler Export Descriptors to identify the PRM Handlers present in the PRM Module. The PRM Module identifier (a 128-bit GUID) is included in the metadata to uniquely identify the module.
        - PRM Handler Export Descriptor - A structure that describes a given PRM Handler. Each entry in the structure associates a PRM Handler with a GUID.
    - An Export Address Table, Name Pointer Table, and Ordinal Table that contain an entry to the PRM Module Export Descriptor and each PRM Handler.
- A .text section that contains executable PRM Handler code. The RVAs to each PRM Handler are computed at compile-time and placed into image export table.

The PRM module PE/COFF image is required to have a valid relocation table so the PRM loader software can load the image at a dynamic base address.

6.1.1 Export Descriptor Structures

The export data section is defined in the PE/COFF format as a section that contains information about symbols in the code image that other images can access through dynamic linking. PRM makes use of the export section to pass PRM module metadata known at build-time to the PRM loader.

The export descriptor structures are architecturally defined in in this section to contain metadata describing the host PRM module and by extension its PRM Handlers. A single PRM Module Export Descriptor Structure is required to be present in each PRM Module export table. If the PRM Module Export Descriptor is not present, the PE/COFF image is not considered a PRM module. The Signature field in the PRM Export Descriptor Structure must also be valid for the PRM module to be recognized appropriately.

6.1.1.1 PRM Module Export Descriptor Structure

Field	**Byte length**	**Byte offset**	**Description**
Signature	8	0	‘PRM_MEDT’. Signature of the PRM Module Export Descriptor Table.
Revision	2	8	Revision of this PRM Module Export Descriptor Table structure.
HandlerCount	2	10	Indicates the number of PRM Handler Information Structure entries that are present in the HandlerExportDescriptorStructure[] field of this structure.
PlatformGuid	16	12	The GUID that uniquely identifies the platform targeted by this PRM module instance. This GUID is used to determine if a given PRM module is valid for a platform during PRM module update.
Identifier	16	28	The GUID of this PRM module.
HandlerExportDescriptor **Structure** **[HandlerCount]**	Varies	44	An array of PrmHandlerExportDescriptors that describes the PRM handler GUID to PRM handler ordinal mapping for this PRM module.

Table 6-1 PRM Module Export Descriptor Structure

The PRM Export Descriptor Structure is required:

- To be present in a PRM module export table
- To have only a single instance per PRM module
- To be named “PrmModuleExportDescriptor”

6.1.1.2 PRM Handler Export Descriptor Structure

Field	**Byte length**	**Byte offset**	**Description**

HandlerGuid	16	0	A PRM handler GUID that maps to the PRM handler name specified in this descriptor.
HandlerName	128	16	A PRM handler name that maps to the PRM handler GUID specified in this descriptor.

Table 6-2 PRM Handler Export Descriptor Structure

6.2 PRM Module Loader

The PRM loader is a software component that is responsible for the following actions:

1. Authenticating PRM module binary images
2. Validating compliance of the image to the requirements in this document
3. Loading the PRM module into a valid memory address range that is executable by the host OS
4. Performing any updates to system data structures necessary to make the PRM module available for use

6.2.1 Firmware PRM Loader

The baseline PRM module is distributed within the platform firmware image and the PRM loader for that image will be a BIOS boot time component (such as an UEFI DXE driver). In this case, the image will typically be loaded from the non-volatile storage device that stores the system boot firmware. Though it is certainly possible and valid to load the image from other storage media. A firmware loader also has the special responsibility to produce and publish the PRMT ACPI tables based on the PRM modules it discovers.

6.2.2 OS PRM Loader

PRM updates at OS runtime allows for modification of PRM functionality without rebooting the platform. In the case of OS runtime PRM updates, an OS software component acts as the PRM loader. The OS PRM loader is required to ensure:

1. PRM updates are always applied in monotonically increasing fashion. For instance, a PRM update with version number smaller than the current PRM module should never be applied.
2. PRM update sequencing minimizes the downtime of PRM functionalities available to OS components.

A OS PRM Loader can only replace existing PRM module that is already published as part of the BIOS boot process and part of the PRMT table. Such newly loaded PRM Module can only replace the functionalities of existing PRM handlers, but will not be able to add a new PRM handler.

6.3 PRM Handler

A PRM handler is a function in a PRM module.

6.3.1 Overview

Each PRM handler must be assigned a GUID by the PRM module author and each PRM handler GUID and corresponding function name must be described as a pair in the PRM Module Export Descriptor.

The PRM module loader resolves the PRM handler GUID to PRM handler physical address mapping.

6.3.2 Function Signature

All PRM handlers are required to follow the architecture-specific calling convention defined for UEFI Runtime services in the UEFI specification. The standard PRM handler function signature is defined below.

```
EFI_STATUS
PRM_EXPORT_API
(EFIAPI *PRM_HANDLER) (
    IN  VOID                         *ParameterBuffer OPTIONAL,
    IN  PRM_MODULE_CONTEXT_BUFFER  *ContextBuffer   OPTIONAL
    );
```

Parameters

ParameterBuffer	A virtual address pointer to a caller allocated buffer that may be consumed by the PRM handler. The internal data format of the ParameterBuffer is a contract between the caller and the PRM handler and outside the scope of this document. If the ParameterBuffer is not provided, NULL will be passed as this argument.
ContextBuffer	A virtual address pointer to a PRM_MODULE_CONTEXT_BUFFER. All addresses referenced in the buffer must be virtual addresses. The ContextBuffer may be NULL if no context information is available and the handler must check for this condition.

The EFI_STATUS and EFI ABI (designated with the EFIAPI modifier in the signature) defined in the UEFI specification are adopted for PRM handlers. The PRM_EXPORT_API includes the appropriate keyword to add the data or function to the export directive in the PRM module object file.

The following requirements are applied to PRM handlers:

- The PRM handler function must use PRM_EXPORT_API to be placed into the image’s export table.
- The maximum name length of a PRM handler function is 128 bytes.

- All PRM handlers must have an entry in the PRM Export Descriptor Table to be recognized as a valid PRM handler.
- Functions in the PRM module binary image that are not exposed as PRM handlers are considered private to the PRM module. Private functions should not have entries in the PRM module’s export table.

7. Servicable PRM

Over time, a PRM handler might need to be updated for a variety of reasons such as bug fixes, workarounds or to enhance the runtime capability or the feature set. PRM updates occur at the module level. It is not possible to update a handler without updating the whole PRM module. For this reason, PRM versioning is applied at the module level. Conventionally, such a PRM module update would require a system reboot that updates the firmware code allowing the new code to be loaded in a future boot.

In a cloud services environment, rebooting the system is not a viable solution and is reserved as a last resort. Hence we need an alternate means to update PRM modules at OS runtime and activate them without a system reboot.

This document describes a generic framework for such an update, by enlightening the ACPI Bridge driver for a mechanism to switch to a new PRM Module image.

7.1 High-Level Flows

If a new PRM Module update is desired, the system BIOS build process generates a new PRM Module image as described in Section 6, or in an OS Specific format from a repository. The generation and delivery of this image is implementation specific and beyond the scope of this specification.

Generically, during OS Runtime, an OS updater consumes a newly delivered PRM Module

1. Parses the PE/COFF Export Descriptor structure
   a. to identify it as a PRM Module
   b. Ensure that the right platform is targeted by matching the PlatformGuid to the PrmPlatformGuid in the PRMT Table *(NOTE: Some implementations might choose the ESRT mechanism for platform targeting, or any other proprietary mechanism)*
2. The updater loads the PRM module into memory and performs the fix-ups
3. Sends request to ACPI to update its PRM handler pointers.

The ACPI Subsystem, on receipt of the request does the following:

1. Checks if the updates are locked or allowed (See section 7.1.1)
2. If locked, then stages the new PRM image until updates are unlocked
3. If unlocked, then switches the pointers to the new PRM Module.

7.1.1 Update Lock/Unlock

Most PRM Handler invocations are considered stateless and hence a PRM Module update can be applied in-between PRM invocations. But this specification allows for a mechanism to lock an update of a PRM Module under certain circumstances, as described below.

If an operation requires a sequence of PRM invocations (via _DSM or via Direct call), then an runtime update of the PRM handler should be blocked until this sequence is complete.

An example of such operation is Address Range Scrub (ARS) for persistent memory (https://pmem.io/documents/NVDIMM_DSM_Interface_Example.pdf) which requires a sequence of _DSM calls (which in-turn invoke the corresponding PRM Handlers), then a PRM Module update request need to be pended until this sequence is complete.

Expanding on the ARS example above, this operation contains invocation of ARS _DSM Method with the following functions.

1. Query ARS Capabilities (Function Index 1)

2. Start ARS (Function Index 2)

3. Get ARS Status (Function Index 3)

The PMEM Driver will start the sequence by querying the ARS capabilities and invoking the Start ARS function. Since the ARS is a long latency operation, the Start ARS function will start the ARS process and return back. The PMEM driver can then poll for the ARS status by invoking the GetARSStatus function.

Each of these _DSM functions will in turn invoke the corresponding PRM Handler to accomplish the task.

During an update flow, care must be taken to ensure that a PRM Module that is in a middle of such sequence is not updated, until the sequence is complete.

To enable this, a Lock/Unlock semantics is provided (see section 8.1.2 and Table 8-1) as part of ACPI sub-system.

A _DSM Method which is start of a such a sequence should invoke a Lock request (see Section 8.2.1) first before starting the sequence of operations, and should invoke an Unlock request (see Section 8.2.2) at the end of the sequence.

Going back to the ARS example above,

1. once Query ARS Capabilities is invoked, the ASL code should first invoke a ‘Lock’ by passing the PRM Handler GUID corresponding to this _DSM (which is the _DSM UUID). The ACPI Interpreter will then find the module that this GUID is part of, and will ‘lock’ this Module from updates. The PRM Handler will NOT be invoked as part of the ‘Lock’ invocation.
2. After this point, any update request received by ACPI will be staged but will not be activated until the update is Unlocked.
3. Once the GetARSStatus PRM calls returns as ARS Complete, the ASL code then calls the Unlock Method by passing the PRM Hander GUID to the ACPI Interpreter, which will then ‘unlock’ this Module to allow for updates.
    a. If a previous update is staged (step 2 above), then the ACPI might choose to switch the pointers to the staged PRM Module at the Unlock call.

The onus of taking the Lock and Unlock is left to the invoker (_DSM, in the case of ASL based invocation, or an OS driver in the case of direct invocation), as the invoker will have the visibility as to if this is start of a sequence or an one-off stateless invocation.

An OS may choose to not support runtime update at all leveraging only a firmware update path or provide a robust framework around updates to minimize system downtime. Since this is OS dependent, this section cannot provide many generic details to describe how PRM serviceability should be implemented in a given OS. However, it does describe the runtime update process used in Microsoft Windows to serve as an example of how such a process can be performed.

7.2 Installation in Windows

1. During OS runtime, an OS-owned updater validates and writes the PRM module update to disk in a well- known location (e.g. in Windows: \System32\Prm\Modules\{Guid}).
2. The updater will parse information from the update and persist the following metadata to registry (to be used across reboots):
    1. Full file path of the PRM module.
    2. The PRM module version number.
    3. The list of PRM handler GUIDs included in the PRM module.
3. The updater loads the PRM module into memory and sends request to ACPI to update its PRM handler pointers.

7.2.1 Persisting PRM Module Updates Across Reboot/KSR

After reboot, winload will read the system hive to see if any PRM module updates have been updated from the firmware's base image. For each PRM module, winload will load the latest version (as indicated by the system hive) from the on-disk location to memory and describe the instance in the boot start driver list. This is the scheme used for boot start drivers, for which existing MM support (relocation of the drivers) exists.

As ACPI.sys reinitializes post boot, it will consume the information from the loader block and reconstruct an up-to-date view of PRM handlers. For both KSR and cold boot scenarios, the ACPI interpreter will be paused up until all PRM updates have been processed.

7.3 Rollback

It is imperative that the platforms implementing PRM functionalities support the rolling back of updates in the event of problematic updates. This is a similar requirement to that now being mandated for microcode updates. To simply the update process, rollbacks will be modeled as an update (increment the module version number) that reverts the behavior to a previous version.

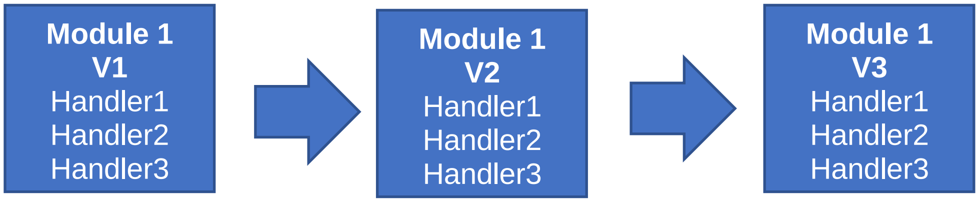


Figure 7-1 PRM Module Versioning Update Example

Note: This assumes stateless behavior in hardware. Specifically, if a PRM update causes reserved bits to be set in HW, downgrading PRM behavior (moving to V3 in the diagram above) needs to ensure the corresponding bits reverted to a known good state or that the presence of the set bits do not adversely affect the behavior of the down-level PRM module.

8. Appendix A: PRM Handler _DSM Invocation

There is a significant install base in the industry that relies on _DSM mechanisms as an abstraction to invoke platform firmware services. In addition to device hardware interrupts, category 3 can generate a SCI event which will enter ACPI context via a _Lxx method. Hence, it is essential to provide a mechanism to bridge the ASL code to the PRM handler to address these cases.

In essence:

- PRM provides a mechanism to invoke native code from ACPI context
- ASL can be the landing point for SW or HW based runtime events
- ASL will invoke PRM if required (ASL serves as a PRM invocation proxy)

8.1 PRM OpRegion Definition

The syntax for the OperationRegion term is described below:

```
OperationRegion (
RegionName, // NameString
RegionSpace, // RegionSpaceKeyword
Offset, // TermArg=>Integer
Length // TermArg=>Integer
)
```

Thus, the PRM Operation Region term in ACPI namespace will be defined as follows:

```
OperationRegion ([subspace-name], PlatformRtMechanism, 0, 1)
```

Where:

- *RegionName* is set to `[subspace-name]`, which is a unique name for this PRM subspace.
- *RegionSpace* must be set to `PlatformRtMechanism`, operation region type **`0x0B`**
- *Offset* must be set to 0.
- *Length* must be set to 1.

The `PlatformRtMechanism` operation region has a single access type allowed.

Address Space	Permitted Access Type(s)	Description
`PlatformRtMechanism`	`BufferAcc`	Reads and writes to this operation region involve the use of a region specific data buffer.

8.1.1 Declaring Fields in the PRM Operation Region

For all `PlatformRtMechanism` OperationRegion definitions, the field definition format must comply with the syntax for the Field as follows:

```
Field (
RegionName,
AccessType,
LockRule,
UpdateRule
) {FieldUnitList}
```

For PRM Operation Regions:

- *RegionName* specifies the name of the operation region, declared above the field term.
- *AccessType* must be set to `BufferAcc`.
- *LockRule* indicates if access to this operation region requires acquisition of the Global Lock for synchronization. This field must be set to `NoLock`.
- *UpdateRule* is not applicable to PRM operation region accesses since each access is performed in its entirety.

The `FieldUnitList` specifies a single field unit of 8 bits. The PRM handler is invoked by writing data to this field unit. The following is an example of an OperationRegion and a Field declaration using the `PlatformRtMechanism` subtype.

```
OperationRegion (PRMR, PlatformRtMechanism, 0x0, 0x1)
Field (PRMR, BufferAcc, NoLock, Preserve)
{
        PRMF, 8
}
```

In order to invoke the PRM OperationRegion handler, a buffer object of 26 bytes must be written to the field unit. Similar to SMBus, IPMI, and Generic Serial bus, this input buffer will also serve as the output buffer. The buffer format and its use will be described in the following sections.

8.1.2 Declaring and Using a PRM Data Buffer

A PRM data buffer is an ASL buffer object that is used as a request and a response buffer for the PRM handler. Writing the PRM data buffer to the PRM field unit will result in the invocation of the PRM OperationRegion where the result of the handler is stored to the PRM field unit. This bidirectionality allows ASL to capture the status of the transaction so that it may perform error handling if necessary.

The format of the PRM data buffer are defined as follows:

Table 8-1 PRM Data Buffer (ASL Buffer Object)

Byte offset	Byte length	Description
0	1	Data buffer status value. This value is populated by the PRM OperationRegion handler. The following are valid status values: 0x0 – success 0x1 – The PRM handler returned an error (only valid for command value 0) 0x2 – Invalid command value 0x3 - Invalid GUID 0x4 – back to back lock command 0x5 – unlock command called without calling lock 0x6 – back to back call to unlock command 0x7-0xff - reserved
1	8	PRM handler status value. This value is populated by the PRM OperationRegion handler only when command value 0. Otherwise, this field is invalid.
9	1	Command value. This value is populated by the caller. The supported command values are as follows: 0x0 – run the PRM service associated with the GUID parameter. 0x1 – start a sequence of PRM calls. When the sequence has been started for a GUID, the PRM module containing the GUID must not be updated until the terminate command for this GUID has been called. This command does not run the actual PRM service. It is a way to communicate the start of a sequence of PRM calls to the OperationRegion handler. 0x2 – terminate a sequence of PRM calls. This command should be called after the start sequence has been called. This tells the PRM OperationRegion that the sequence of PRM calls has ended and that it is safe to update the PRM handlers. This command does not run the actual PRM service. It is a way to communicate the end of a sequence to the PRM OperationRegion handler. 0x3-0xff - reserved

10	16	_DSM GUID. This value is populated by the caller. This GUID must be present in the list of available handlers published by the PRMT table.

The above byte fields can be manipulated using CreateByteField, CreateQWordField, and CreateField operators. By doing so, ASL can read and write values from this buffer using a single store operator.

8.2 PRM Invocation Example

The following is an example of how data is written to the PRM data buffer:

```
/*
 * Control method to Run PRM service
 * Arg0 contains a buffer of a _DSM GUID
 */
Method (RUNS, 1)
{
    /* Local0 is the PRM data buffer */
    Local0 = buffer (26){}

    /* Create byte fields */
    CreateByteField (Local0, 0x0, PSTA)
    CreateQWordField (Local0, 0x1, USTA)
    CreateByteField (Local0, 0x9, CMD)
    CreateField (Local0, 0x50, 0x80, DATA)

    /* Fill in the command and data fields of the data buffer */
    CMD = 0
    DATA = Arg0
    ...
}
```

In order to invoke the PRM OperationRegion Handler, the contents of Local0 need to be written to a PRM OperationRegion FieldUnit. The result of the handler can be acquired by storing the contents of the field unit back to Local0. The following example defines a PRM OperationRegion and FieldUnit and a function that will tell the PRM OperationRegion Handler to run the PRM service described by Arg0.

```
OperationRegion (PRMR, PlatformRtMechanism, 0x0, 0x1)
Field (PRMR, BufferAcc, NoLock, Preserve)
{
    PRMF, 8
```

```
}
/*
 * Control method to invoke PRM OperationRegion handler
 * Arg0 contains a buffer representing a _DSM GUID
 */
Method (RUNS, 1)
{
    /* Local0 is the PRM data buffer */
    Local0 = buffer (26){}

    /* Create byte fields over the buffer */
    CreateByteField (Local0, 0x0, PSTA)
    CreateQWordField (Local0, 0x1, USTA)
    CreateByteField (Local0, 0x9, CMD)
    CreateField (Local0, 0x50, 0x80, GUID)

    /* Fill in the command and data fields of the data buffer */
    CMD = 0 // run command
    GUID = Arg0

    /* Invoke PRM OperationRegion Handler and store the result into Local0 */
    Local0 = (PRMF = Local0)

    /* PSTA and USTA now contains the status returned by running the handler */
    If (!PSTA)
    {
        /* do error handling here */
        ...
        If (!USTA)
        {
            /* Optionally handle status returned by the PRM service */
            ...
        }
    }

    /* Return status */
    Return (PSTA)
}
```

8.2.1 Example ASL Code for Locking Updates

The following is an example that will lock the PRM transaction using the OperationRegion and Field defined in the previous example:

```
/*
 * Control method to lock a PRM transaction
 * Arg0 contains a buffer representing a _DSM GUID
 */
Method (LOCK, 1)
{
    /* Local0 is the PRM data buffer */
    Local0 = buffer (26){}

    /* Create byte fields over the buffer */
    CreateByteField (Local0, 0x0, STAT)
    CreateByteField (Local0, 0x9, CMD)
    CreateField (Local0, 0x50, 0x80, GUID)
    CMD = 1 // Lock command
    GUID = Arg0
    Local0 = (PRMF = Local0)

    /* Note STAT contains the return status */
    Return (STAT)
}
```

8.2.2 Example ASL Code for Unlocking Updates

The following is an example that will unlock the PRM transaction using the same OperationRegion and Field definitions:

```
/*
 * Control method to unlock a PRM transaction
 * Arg0 contains a buffer representing a _DSM GUID
 */
Method (ULCK, 1)
{
        /* Local0 is the PRM data buffer */
        Local0 = buffer (26){}

        /* Create byte fields over the buffer */
        CreateByteField (Local0, 0x0, STAT)
        CreateByteField (Local0, 0x9, CMD)
        CreateField (Local0, 0x50, 0x80, GUID)
        CMD = 2 // Unlock command
        GUID = Arg0
        Local0 = (PRMF = Local0)

        /* Note STAT contains the return status */
        Return (STAT)
    }
```

9. Appendix B: _OSC and OpRegion

9.1 Platform-Wide OSPM Capabilities

A new _OSC capabilities bit (BIT 21) will be used to indicate OS support of Platform Runtime Mechanism.

Based on this indication, BIOS can choose switch from legacy handling (such as SMI) to using PRM

The ACPI ECR for this bit is shown here for completeness.

Add a new bit at the end of the table as follows:

Platform-Wide _OSC Capabilities DWORD 2

Capabilities DWORD	Interpretation
21	Reserved for future use - The OS sets this bit to indicate support for Platform Runtime Mechanism (PRM).
31:22	Reserved (must be 0)

9.2 PRM Operation Region

A new Operation Region Address space identifier is defined for *PlatformRtMechanism* and the ACPI ECR is shown here for completeness.

Operation Region Address Space Identifiers Value

Value	Name (RegionSpace Keyword)
0x0B	PlatformRtMechanism (Reserved for future use by a mechanism developed in the code-first approach)