



Intel VROC 9.1 UEFI specification

Revision 1.1

January 2025

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

No computer system can provide absolute security. Requires an enabled Intel® processor, enabled chipset, firmware and/or software optimized to use the technologies. Consult your system manufacturer and/or software vendor for more information.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer to learn more at [intel.com](https://www.intel.com).

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

All products, computer systems, dates, and figures specified are preliminary based on current expectations, and are subject to change without notice.

Intel and the Intel logo are trademarks of Intel Corporation in the United States and/or other countries.

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

Copyright © 2024, Intel Corporation. All rights reserved

Intel Confidential

Contents

1	Introduction.....	6
1.1	Scope	6
1.2	Intended Audience	6
1.3	Terminology.....	6
1.4	References.....	6
2	Architectural Overview	7
2.1	Intel VROC UEFI drivers.....	7
3	Intel VROC PEI modules	9
3.1	VROC SATA PEIM.....	9
3.1.1	Provided PPIs.....	10
3.1.2	Required PPIs.....	10
3.2	VROC VMD PEIM	10
3.2.1	Provided PPIs.....	11
3.2.2	Required PPIs.....	11
3.2.2.1	VMD_HOST_CONTROLLER_PPI.....	12
3.2.2.1.1	VMD_HOST_CONTROLLER_PPI.GetVmdHcBdf.....	12
3.2.2.1.2	VMD_HOST_CONTROLLER_PPI.GetVmdHcDevicePath	13
3.3	Block I/O PPI interfaces.....	13
3.4	PPIs table for VROC PEI modules.....	14
4	Intel VROC DXE drivers.....	15
4.1	VMD DXE driver	15
4.2	VROC VMD DXE driver	16
4.3	VROC SATA DXE drivers.....	17
5	Protocols.....	20
5.1	Clarification for UEFI standard protocols in VROC	20
5.1.1	EFI_DRIVER_BINDING_PROTOCOL.....	20
5.1.2	EFI_COMPONENT_NAME2_PROTOCOL	20
5.1.3	EFI_STORAGE_SECURITY_COMMAND_PROTOCOL	20
5.1.4	EFI_FIRMWARE_MANAGEMENT_PROTOCOL	21
5.1.4.1	EFI_FIRMWARE_MANAGEMENT_PROTOCOL provided by VROC VMD DXE driver	22
5.1.4.1.1	GetImageInfo.....	22
5.1.4.1.2	SetImageInfo	23
5.1.4.2	EFI_FIRMWARE_MANAGEMENT_PROTOCOL provided by VROC SATA DXE drivers.....	23

5.1.4.2.1	GetImageInfo.....	23
5.1.4.2.2	SetImageInfo	24
5.2	VROC (non-standard) protocols and variables specification	25
5.2.1	VROC_DEVICE_INFO_PROTOCOL.....	25
5.2.2	VROC_VOLUME_INFO_PROTOCOL	29
5.2.3	VrocPhysicalLocation variable	32

Figures

Figure 1. Intel VROC UEFI drivers.....	8
Figure 2. VROC SATA PEIM components diagram	10
Figure 3. VROC VMD PEIM components diagram.....	11
Figure 4. Block devices indexing by VROC PEI module	14
Figure 5. VMD DXE driver UEFI handles.....	15
Figure 6. VROC VMD DXE driver UEFI handles.....	16
Figure 7. VROC SATA DXE driver UEFI handles	18
Figure 8. Example NVMe SSDs unlocking flow using SSCP	21

Tables

Table 1. Terminology	6
Table 2. Reference Documents and Resources	6
Table 3. Status codes returned by VMD_HOST_CONTROLLER_PPI.GetVmdHcBdf function	13
Table 4. Status codes returned by VMD_HOST_CONTROLLER_PPI.GetVmdHcDevicePath function.....	13
Table 5. PPIs table for VROC PEI modules	14
Table 6. VMD DXE driver UEFI handles and protocols.....	16
Table 7. VROC VMD DXE driver UEFI handles and protocols.....	17
Table 8. VROC SATA DXE driver UEFI handles and protocols	19
Table 9. VROC UEFI driver version encoding in EFI_DRIVER_BINDING_PROTOCOL.....	20
Table 10. GetDriverName and GetControllerName values returned by VROC UEFI drivers	20
Table 11. VROC EFI Firmware Management Protocol functions	22
Table 12. GetImageInfo values returned by VROC VMD DXE driver	22
Table 13. EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC VMD DXE driver.....	22
Table 14. GetImageInfo values returned by VROC SATA DXE drivers.....	24
Table 15. EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC SATA DXE drivers.....	24
Table 16. ATA DOWNLOAD MICROCODE subcommand decision matrix in VROC SATA DXE drivers.....	25

VROC version changes summary

VROC Version	Description	Affected content	Date
9.1	Remove VROC Physical Location version 2 (not released in 9.1).	Section 5.2.3	11/15/2024
9.1	Merged Intel VROC UEFI Physical Drive Location into this document. Added new section for VrocPhysicalLocation variable.	Section 5.2.3	11/15/2024
9.1	Removed the VROC_REENUMERATION_PROTOCOL, since VROC 9.1 automatically detects the change of drive's security state and automatically invokes internally re-enumeration.	Section 5.1.3, Section 5.2	07/24/2024
9.1	Changed the handle, where VROC_DEVICE_INFO_PROTOCOL is installed from NVMe namespace to drive handle	Section 4.1, Section 4.2	07/05/2024
9.0	Added Intel On Demand DXE driver description.	Section 2.1	07/23/2024
8.6	Initial version of the document.	N/A	01/30/2024

1 Introduction

This document describes the architecture for the Intel® Virtual RAID on CPU (VROC) for Unified Extensible Firmware Interface (UEFI). Intel VROC is a software RAID solution for NVMe and SATA drives, which also provides manageability features in UEFI.

1.1 Scope

This document contains information required to understand what the VROC 9.1 UEFI components are and how they work.

1.2 Intended Audience

The primary audiences of this document are OEMs who would like to use the Intel VROC family of products included in their platform. This document is also intended to support BIOS firmware engineers who may be developing appropriate changes in the BIOS.

1.3 Terminology

Table 1. Terminology

Term	Definition
API	Application Program Interface
BDF	Bus, Device, Function for the PCIe device
DXE	Drive Execution Environment
NVMe	Non-volatile Memory Express
OEM	Original Equipment Manufacturer
OOB	Out-of-Band
PCIe	Peripheral Component Interconnect express
PEI	Pre-EFI Initialization
PEIM	Pre-EFI Initialization Module
SATA	Serial Advanced Technology Attachment
VMD	Volume Management Device
VROC	Virtual RAID on CPU
UEFI	Unified Extensible Firmware Interface

1.4 References

Table 2. Reference Documents and Resources

Title	Location
UEFI Specification 2.10	https://uefi.org/sites/default/files/resources/UEFI_Spec_2_10_Aug29.pdf
UEFI Platform Initialization Specification 1.8	https://uefi.org/sites/default/files/resources/UEFI_PI_Spec_1_8_March3.pdf

2 Architectural Overview

Intel VROC delivers multiple UEFI drivers for PEI and DXE phase, to be integrated with the BIOS.

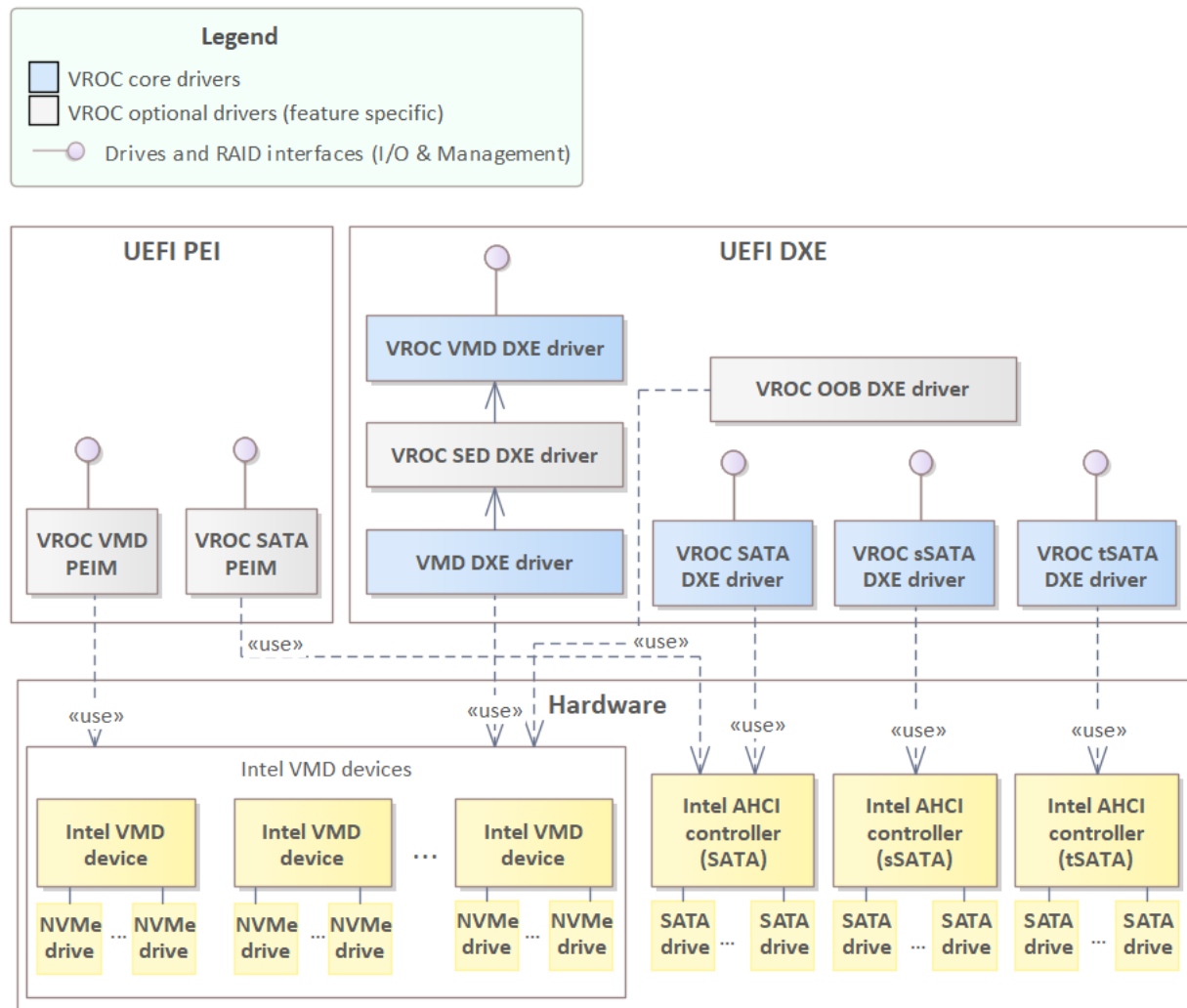
2.1 Intel VROC UEFI drivers

There are following Intel VROC UEFI drivers that are part of Intel VROC solution:

1. UEFI PEI modules:
 - a. *VROC VMD PEIM* – implements Intel VMD hardware initialization and software RAID solution for NVMe SSDs in PEI phase
 - b. *VROC SATA PEIM* – implements Intel AHCI controller initialization and software RAID solution for SATA drives in PEI phase
2. UEFI DXE drivers:
 - a. *VMD DXE driver* – initializes VMD hardware and enumerates NVMe drives
 - b. *VROC VMD DXE driver* – implements software RAID solution for NVMe drives connected behind VMD
 - c. *VROC SED DXE driver* – implements automatic provisioning and unlocking of NVMe SED SSDs, by using EFI Key Management Service Protocol
 - d. *VROC OOB DXE driver* – implements Out-of-Band (OOB) management interface
 - e. *VROC SATA DXE drivers* – implements software RAID solution for SATA drives, connected behind Intel AHCI controller. There are three *VROC SATA DXE drivers*, separate for each Intel AHCI controller:
 - i. The first – called SATA controller
 - ii. The second – called sSATA controller (secondary SATA)
 - iii. The third – called tSATA controller (tertiary SATA)The number of available Intel AHCI controllers depends on the Intel chipset model and Intel motherboard design and generation, it may vary from zero up to three Intel AHCI controllers in the system.

Figure 1 shows Intel VROC UEFI drivers, their dependencies and hardware usage.

Figure 1. Intel VROC UEFI drivers



3 Intel VROC PEI modules

Intel VROC solution comprises VROC PEI modules for the following features:

- Firmware Update Recovery solution, which ensures that the system is not left inoperable in the event of a firmware update that is interrupted by an intermittent cause of failure. The typical example and solution scenario is power loss during firmware update. Firmware Update Recovery (a.k.a. seamless recovery, automated recovery, or Fault Tolerant Update) is intended to restart a power-loss interrupted firmware update process without end user involvement, continue the firmware update, and complete it. VROC UEFI PEI modules provide access to data (stored on drive or RAID volume) used in that process, but the recovery flow itself is implemented outside VROC UEFI PEI modules, by the OEM.
- Initialization of Opal-enabled NVMe SSDs during system boot – for example to send Block SID command to prevent OS-present malware from maliciously performing Take Ownership of Security Identifier of the drive. VROC VMD PEIM provides interface for sending security commands to individual SSDs.
- Unlocking the SED drive, when resuming from S3 power state. User credentials are not eradicated on sleep mode entry and thus the user is not de-authenticated from the system. Opal management software will typically keep the user password, or the Authentication Credential stored in system nonvolatile memory. Upon system resumption from Sleep mode, the Host Opal management software automatically replays the authentication sequence to the SED using the cached Authentication Credential(s). This results in the SED drive being unlocked without the user having to manually supply their authentication value again. VROC VMD PEIM provides interface for sending security commands in the PEI phase.

The Intel VROC 9.1 package contains two VROC PEI modules:

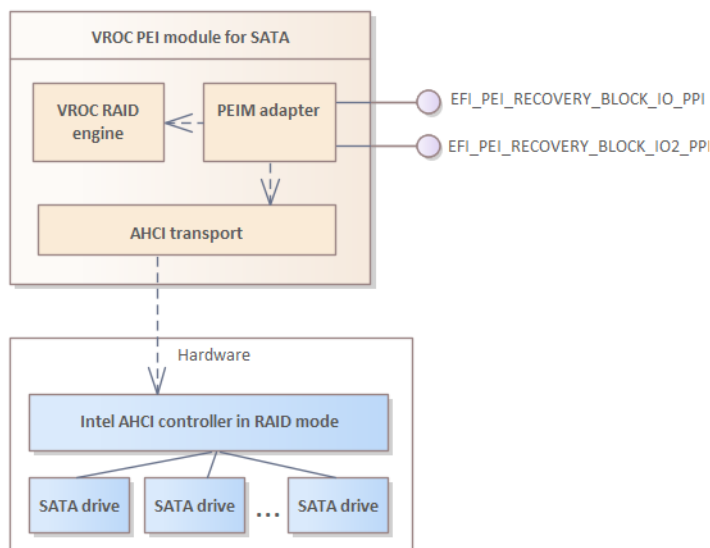
1. VROC PEI module for SATA (*VROC SATA PEIM*)
2. VROC PEI module for VMD (*VROC VMD PEIM*)

Both modules have common code base with PEI interface implementation (PEIM adapter) and RAID logic. The main difference between them is different transport implementations: AHCI and VMD with NVMe.

3.1 VROC SATA PEIM

The *VROC SATA PEIM* provides RAID functions and interfaces for SATA drives connected to Intel AHCI controller on Intel Platform Control Hub (PCH) in Pre-EFI Initialization Phase. The Intel AHCI controller must be configured in RAID mode. On the Figure 2, one can see *VROC SATA PEIM* components diagram.

Figure 2. VROC SATA PEIM components diagram



3.1.1 Provided PPIs

VROC SATA PEIM installs two PEIM-to-PEIM interfaces (PPI), defined in PI Specification:

1. EFI_PEI_RECOVERY_BLOCK_IO_PPI – single instance of PPI
2. EFI_PEI_RECOVERY_BLOCK_IO2_PPI – single instance of PPI

For more details about Block I/O PPI interfaces, see section 3.3.

3.1.2 Required PPIs

The VROC SATA PEIM shall be dispatched after SATA Init Done PPI installation.

SATA Init Done PPI is the PPI to assure proper order of loading PEI modules. It has following GUID:

```
{ 0xbcc5d5cb, 0x828c, 0x4fa1, {0x8a, 0x2a, 0xbf, 0x31, 0x1c, 0xe2, 0x61, 0x89} }
```

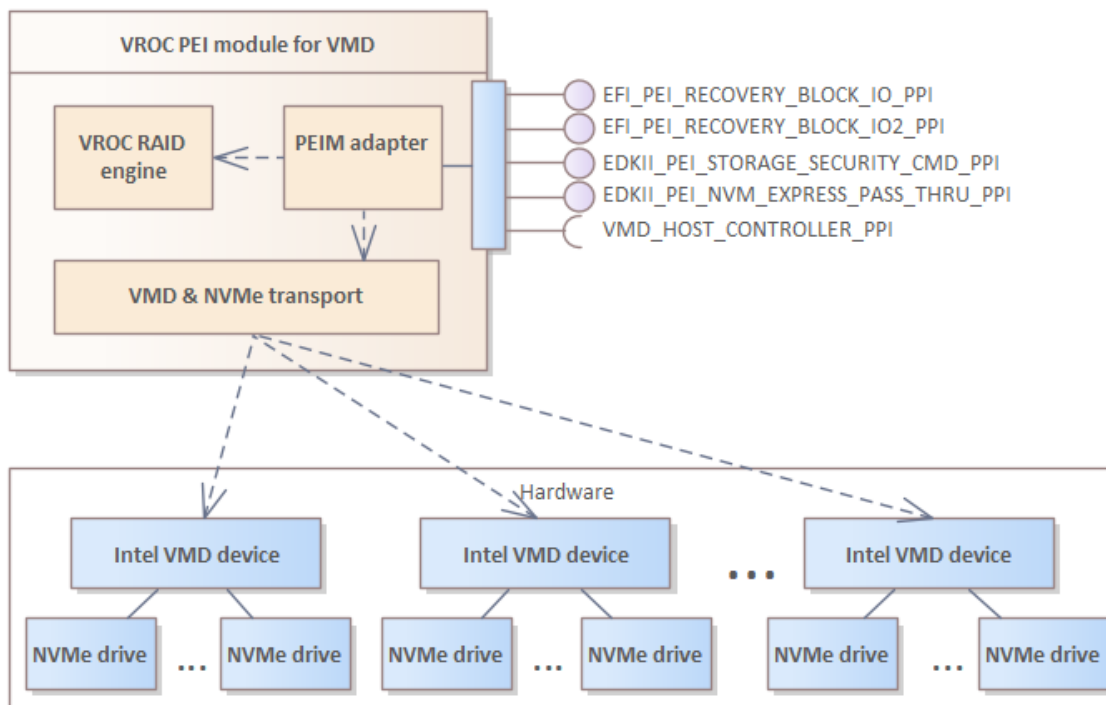
The structure of this PPI is not defined.

SATA Init Done PPI shall be installed after PCH SATA controller initialization, and it shall be installed only when SATA controller is set to RAID mode.

3.2 VROC VMD PEIM

The VROC VMD PEIM provides RAID functions and interfaces for NVMe drives connected to Intel VMD controllers in Pre-EFI Initialization Phase. On the Figure 3, one can see VROC VMD PEIM components diagram.

Figure 3. VROC VMD PEIM components diagram



3.2.1 Provided PPIs

VROC VMD PEIM installs following PEIM-to-PEIM interfaces (PPI):

1. EFI_PEI_RECOVERY_BLOCK_IO_PPI – single instance of PPI
2. EFI_PEI_RECOVERY_BLOCK_IO2_PPI – single instance of PPI
3. EDKII_PEI_STORAGE_SECURITY_CMD_PPI – single instance of PPI
4. EDKII_PEI_NVM_EXPRESS_PASS_THRU_PPI – one instance per NVMe drive discovered by VROC VMD PEIM.

For more details about Block I/O PPI interfaces, see section 3.3. If there is at least one NVMe SSD discovered under VMD, the rest of PPIs are always installed by VROC VMD PEIM.

3.2.2 Required PPIs

The VROC VMD PEIM shall be dispatched after VMD Host Controller PPI installation. The PPI is to provide the list of VMD controllers from BIOS to VROC VMD PEIM. Thus, in PEI phase, the only PPI which could be used by VROC VMD PEIM for VMD devices discovery is PCI Configuration PPI. VROC VMD PEIM would require significant amount of time to discover VMD devices manually, because it would require trying to read PCI configuration space of all possible PCI device address combinations (PCI Bus, Device, Function), which results in 65536 iterations. This PPI shall be installed after BIOS initialization of all VMD controllers in the system

3.2.2.1 VMD_HOST_CONTROLLER_PPI

Summary

The protocol for getting information about Intel VMD controllers.

GUID

```
#define VMD_HOST_CONTROLLER_PPI_GUID \
    {0x51f28c3b, 0xcfa, 0x4308, { 0xb7, 0xdb, 0xc6, 0x82, 0xac, 0xcd, 0xe0, 0x8a }}
```

Protocol Interface Structure

```
typedef struct _VMD_HOST_CONTROLLER_PPI
{
    VMD_HC_GET_BDF          GetVmdHcBdf;
    VMD_HC_GET_DEVICE_PATH  GetVmdHcDevicePath;
} VMD_HOST_CONTROLLER_PPI;
```

Parameters

- GetVmdHcBdf – function for getting the PCI address of VMD host controller
- GetVmdHcDevicePath – function for getting the device path of VMD host controller

3.2.2.1.1 VMD_HOST_CONTROLLER_PPI.GetVmdHcBdf

Summary

Gets the PCI address of VMD host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *VMD_HC_GET_BDF)(
    IN  VMD_HOST_CONTROLLER_PPI *This,
    IN  UINT8                    ControllerId,
    OUT UINT16                   *Segment,
    OUT UINT8                    *Bus,
    OUT UINT8                    *Device,
    OUT UINT8                    *Function
);
```

Parameters

- This – the PPI instance pointer
- ControllerId – the ID of the VMD host controller. Because the provider of VMD Host Controller PPI will manage multiple VMD devices, the *VROC VMD PEIM*, which wants to get address of single VMD device, must specify the device index that was assigned during the enumeration process. This index is a number from zero to number of VMD devices in the system. All VMD devices shall be indexed continuously (e.g., 0, 1, 2, 3), without gaps in numbering. *VROC VMD PEIM* issues GetVmdHcBdf function multiple times, starting from ControllerId equal to zero, increasing ControllerId one by one, until EFI_NOT_FOUND is returned.
- Segment – the PCI bus segment of the controller
- Bus – the PCI bus number of the controller
- Device – the PCI device number of the controller
- Function – the PCIe function number of the controller

Description

Gets the PCI address of VMD host controller.

Status Codes Returned

Table 3. Status codes returned by VMD_HOST_CONTROLLER_PPI.GetVmdHcBdf function

Status code	Description
EFI_SUCCESS	The operation succeeds
EFI_INVALID_PARAMETER	The parameters are invalid
EFI_NOT_FOUND	The specified VMD host controller not found

3.2.2.1.2 VMD_HOST_CONTROLLER_PPI.GetVmdHcDevicePath

Summary

Gets the device path of VMD host controller.

Prototype

```
typedef
EFI_STATUS
(EFI_API *VMD_HC_GET_DEVICE_PATH)(
    IN  VMD_HOST_CONTROLLER_PPI *This,
    IN  UINT8                    ControllerId,
    OUT UINTN                    *DevicePathLength,
    OUT EFI_DEVICE_PATH_PROTOCOL **DevicePath
);
```

Parameters

- This – the PPI instance pointer
- ControllerId – the ID of the VMD host controller. It must be consistent with ControllerId parameter in GetVmdHcBdf function.
- DevicePathLength – the length of the device path in bytes specified by DevicePath
- DevicePath – the device path of VMD host controller. This field re-uses EFI Device Path Protocol as defined by Section 10.2 EFI Device Path Protocol of UEFI specification mentioned in 1.4 References.

Description

Gets the device path of VMD host controller.

Status Codes Returned

Table 4. Status codes returned by VMD_HOST_CONTROLLER_PPI.GetVmdHcDevicePath function

Status code	Description
EFI_SUCCESS	The operation succeeds
EFI_INVALID_PARAMETER	The parameters are invalid
EFI_NOT_FOUND	The specified VMD host controller not found
EFI_OUT_OF_RESOURCES	The operation fails due to lack of resources

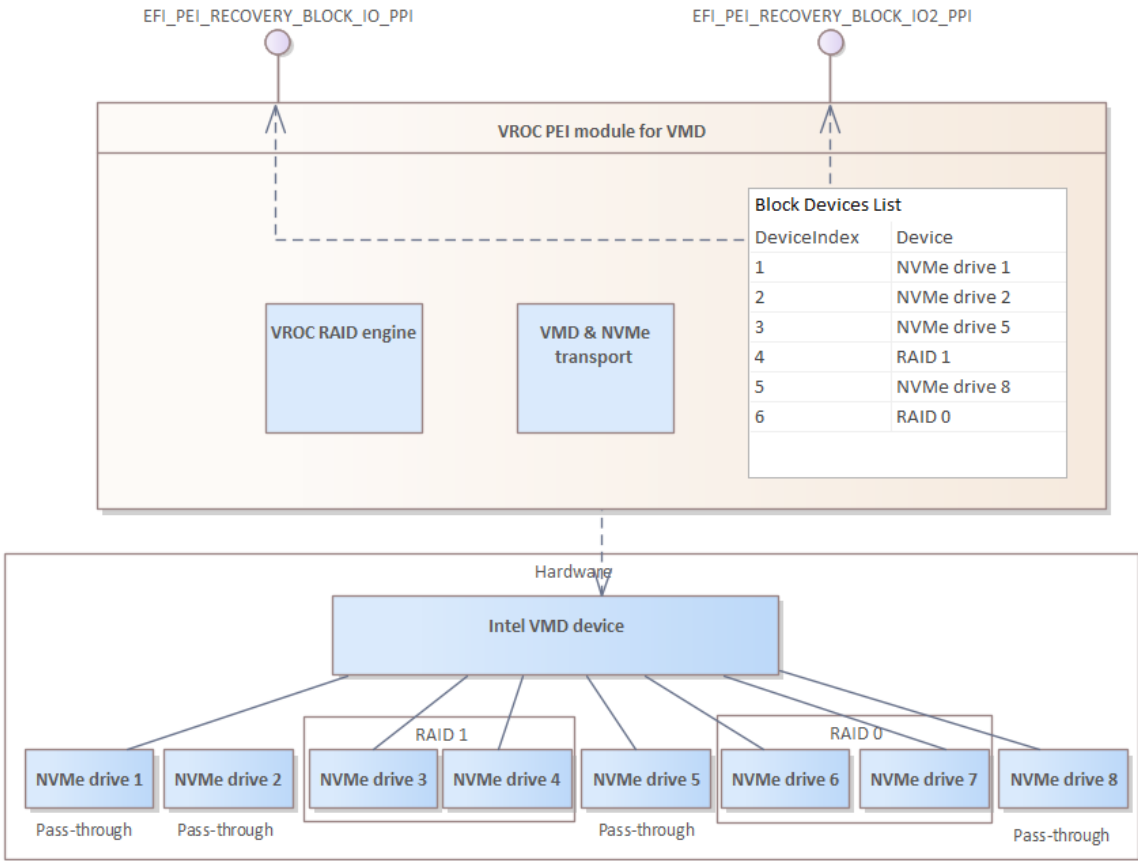
3.3 Block I/O PPI interfaces

EFI_PEI_RECOVERY_BLOCK_IO_PPI and EFI_PEI_RECOVERY_BLOCK_IO2_PPI are used to read data from pass-through drives and RAID volumes, enumerated by VROC PEI modules. Each VROC PEI module installs one instance of EFI_PEI_RECOVERY_BLOCK_IO_PPI and one instance of EFI_PEI_RECOVERY_BLOCK_IO2_PPI, regardless of number of drives or RAID volumes discovered by each module. For pass-through drives and RAID volumes present

behind single AHCI/VMD controller, the block device selection must be done by *DeviceIndex* argument of *GetBlockDeviceMediaInfo* and *ReadBlocks* functions.

Figure 4 presents the example drives configuration connected behind single VMD controller.

Figure 4. Block devices indexing by VROC PEI module



The order of indexing RAID volumes and drives is not set by any rule in VROC PEI module. The internal enumeration procedure discovers devices one by one, and any specific order cannot be guaranteed by VROC PEI module. The BIOS implementation must not assume any ordering scheme.

3.4 PPIs table for VROC PEI modules

Table 5 shows the summary of installed PPIs and BIOS dependencies for both VROC PEI modules.

Table 5. PPIs table for VROC PEI modules

	VROC SATA PEIM	VROC VMD PEIM
Provided PPIs	EFI_PEI_RECOVERY_BLOCK_IO_PPI EFI_PEI_RECOVERY_BLOCK_IO2_PPI	EFI_PEI_RECOVERY_BLOCK_IO_PPI EFI_PEI_RECOVERY_BLOCK_IO2_PPI EDKII_PEI_STORAGE_SECURITY_CMD_PPI EDKII_PEI_NVM_EXPRESS_PASS_THRU_PPI
Required PPIs	SATA Init Done PPI (see 3.1.2)	VMD_HOST_CONTROLLER_PPI (see 3.2.2.1)

4 Intel VROC DXE drivers

Intel VROC uses two UEFI DXE drivers to provide RAID functionality for NVMe SSDs connected behind VMD controller:

1. *VMD DXE driver*, which is used for VMD hardware initialization and NVMe drives enumeration. This driver implements VMD PCIe bus scanning to discover NVMe controllers attached to VMD. The driver enumerates all NVMe namespaces and exposes them to *VROC VMD DXE driver* through private protocols.
2. *VROC VMD DXE driver*, which implements RAID features on top of NVMe drives. This driver consumes NVMe namespaces exposed by *VMD DXE driver* and uses those as drives for RAID logic.

4.1 VMD DXE driver

VMD DXE driver handles all VMD devices in the system, by consuming `EFI_PCI_IO_PROTOCOL`.

Figure 5. VMD DXE driver UEFI handles

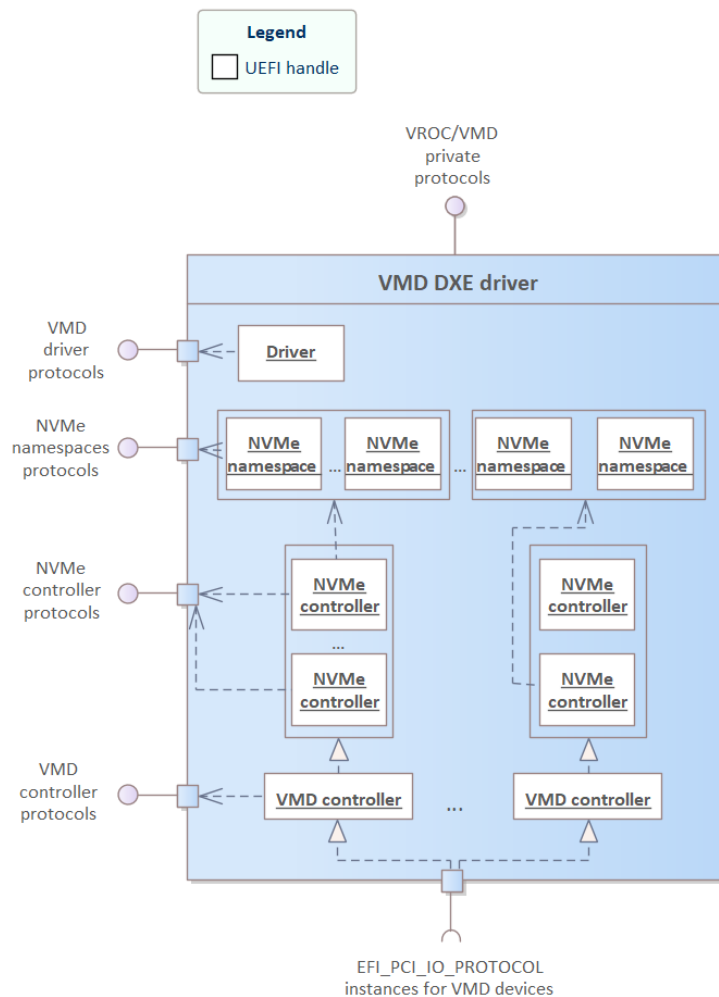


Table 6 lists UEFI handles and provided protocols for each handle installed by VMD DXE driver.

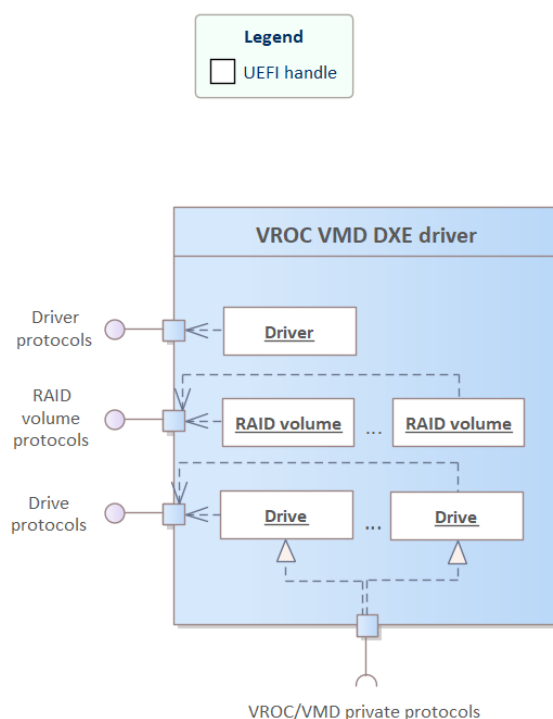
Table 6. VMD DXE driver UEFI handles and protocols

UEFI Handle	Description	Provided protocols
Driver	Single handle for <i>VMD DXE driver</i>	EFI_DRIVER_BINDING_PROTOCOL EFI_COMPONENT_NAME2_PROTOCOL
NVMe namespace	Handle for each NVMe namespace discovered by <i>VMD DXE driver</i>	
NVMe controller	Handle for each NVMe drive (controller) discovered by <i>VMD DXE driver</i>	EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL
VMD controller	Handle for each Intel VMD device in the system.	EFI_DRIVER_HEALTH_PROTOCOL

4.2 VROC VMD DXE driver

VROC VMD DXE driver consumes private protocols provided by *VMD DXE driver* and creates UEFI handles for driver, drives and RAID volumes as shown on Figure 6.

Figure 6. VROC VMD DXE driver UEFI handles



Each drive in the system has their UEFI handle, created by *VROC VMD DXE driver*. There are two types of drives managed by *VROC VMD DXE driver*:

- Pass-through drives, which are not part of any RAID volume, and used as individual drives in the system
- RAID drives, which could be RAID member drives, or reserved for RAID usage, like spare drives or journaling drives.

Depending on the drive type, there are different set of protocols installed for drive's UEFI handle.

Table 7. VROC VMD DXE driver UEFI handles and protocols

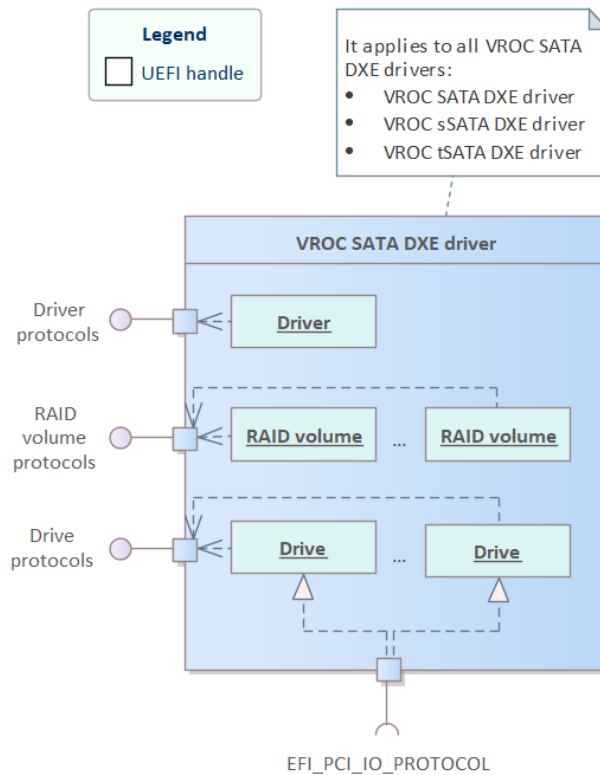
UEFI Handle	Description	Provided protocols
Driver	Single handle for <i>VROC VMD DXE driver</i>	EFI_DRIVER_BINDING_PROTOCOL (see 5.1.1) EFI_DRIVER_HEALTH_PROTOCOL EFI_ADAPTER_INFORMATION_PROTOCOL EFI_COMPONENT_NAME2_PROTOCOL (see 5.1.2) EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL
RAID volume	Handle for each RAID volume exposed by <i>VROC VMD DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL EFI_BLOCK_IO_PROTOCOL VROC_VOLUME_INFO_PROTOCOL (see 5.2.2)
Drive (pass-through)	Handle for each pass-through drive exposed by <i>VROC VMD DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL EFI_BLOCK_IO_PROTOCOL VROC_DEVICE_INFO_PROTOCOL (see 5.2.1) EFI_STORAGE_SECURITY_COMMAND_PROTOCOL (see 5.1.3) EFI_FIRMWARE_MANAGEMENT_PROTOCOL (see 5.1.4)
Drive (RAID drive)	Handle for each RAID drive exposed by <i>VROC VMD DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL VROC_DEVICE_INFO_PROTOCOL (see 5.2.1) EFI_STORAGE_SECURITY_COMMAND_PROTOCOL (see 5.1.3) EFI_FIRMWARE_MANAGEMENT_PROTOCOL (see 5.1.4)

In addition to protocols, *VROC VMD DXE driver* offers custom reporting the physical location of the drives. For more information see section 5.2.3.

4.3 VROC SATA DXE drivers

This section describes *VROC SATA DXE drivers*. All three drivers have exactly the same functionality, but handles different instances of Intel AHCI controller (SATA, sSATA and tSATA).

Figure 7. VROC SATA DXE driver UEFI handles



Each drive in the system has their UEFI handle, created by *VROC SATA DXE driver*. There are two types of drives managed by *VROC SATA DXE driver*:

- Pass-through drives, which are not part of any RAID volume, and used as individual drives in the system
- RAID drives, which could be RAID member drives, or reserved for RAID usage, like spare drives or journaling drives.

Depending on the drive type, there are different set of protocols installed for drive's UEFI handle.

Table 8. VROC SATA DXE driver UEFI handles and protocols

UEFI Handle	Description	Provided protocols
Driver	Single handle for <i>VROC SATA DXE driver</i>	EFI_DRIVER_BINDING_PROTOCOL (see 5.1.1) EFI_DRIVER_HEALTH_PROTOCOL EFI_ADAPTER_INFORMATION_PROTOCOL EFI_COMPONENT_NAME2_PROTOCOL (see 5.1.2) EFI_DRIVER_SUPPORTED_EFI_VERSION_PROTOCOL
RAID volume	Handle for each RAID volume exposed by <i>VROC SATA DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL EFI_BLOCK_IO_PROTOCOL VROC_VOLUME_INFO_PROTOCOL (see 5.2.2)
Drive (pass-through)	Handle for each pass-through drive exposed by <i>VROC SATA DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL EFI_BLOCK_IO_PROTOCOL EFI_ATA_PASS_THRU_PROTOCOL EFI_EXT_SCSI_PASS_THRU_PROTOCOL EFI_STORAGE_SECURITY_COMMAND_PROTOCOL (see 5.1.3) EFI_FIRMWARE_MANAGEMENT_PROTOCOL (see 5.1.4)
Drive (RAID drive)	Handle for each RAID drive exposed by <i>VROC SATA DXE driver</i>	EFI_DEVICE_PATH_PROTOCOL EFI_ATA_PASS_THRU_PROTOCOL EFI_EXT_SCSI_PASS_THRU_PROTOCOL EFI_STORAGE_SECURITY_COMMAND_PROTOCOL (see 5.1.3) EFI_FIRMWARE_MANAGEMENT_PROTOCOL (see 5.1.4)

In addition to protocols, *VROC SATA DXE driver* offers custom reporting the physical location of the drives. For more information see section 5.2.3.

5 Protocols

5.1 Clarification for UEFI standard protocols in VROC

5.1.1 EFI_DRIVER_BINDING_PROTOCOL

Version parameter value of EFI_DRIVER_BINDING_PROTOCOL installed by VROC UEFI drivers is encoded according to Table 9.

Table 9. VROC UEFI driver version encoding in EFI_DRIVER_BINDING_PROTOCOL

Bits	Value
31:24	Major version
23:20	Minor version
19:16	Maintenance version
15:0	Build version

For example, version 8.2.0.1132 is reported as 0x0820046C.

5.1.2 EFI_COMPONENT_NAME2_PROTOCOL

VROC VMD DXE driver and VROC SATA DXE drivers provides EFI_COMPONENT_NAME2_PROTOCOL with the “en-US” language supported. The GetDriverName and GetControllerName functions returns values as defined in Table 10.

Table 10. GetDriverName and GetControllerName values returned by VROC UEFI drivers

VROC Driver	GetDriverName	GetControllerName
VROC VMD DXE	“Intel(R) VROC <version> VMD Driver” Where version corresponds to VROC VMD DXE driver version, for example: “Intel(R) VROC 8.2.0.1132 VMD Driver”	“Intel(R) Virtual RAID on CPU”
VROC SATA DXE	“Intel(R) VROC <version> SATA Driver” Where version corresponds to VROC SATA DXE driver version, for example: “Intel(R) VROC 8.2.0.1132 SATA Driver”	“Intel(R) VROC SATA Controller”
VROC sSATA DXE	“Intel(R) VROC <version> sSATA Driver” Where version corresponds to VROC sSATA DXE driver version, for example: “Intel(R) VROC 8.2.0.1132 sSATA Driver”	“Intel(R) VROC sSATA Controller”
VROC tSATA DXE	“Intel(R) VROC <version> tSATA Driver” Where version corresponds to VROC tSATA DXE driver version, for example: “Intel(R) VROC 8.2.0.1132 tSATA Driver”	“Intel(R) VROC tSATA Controller”

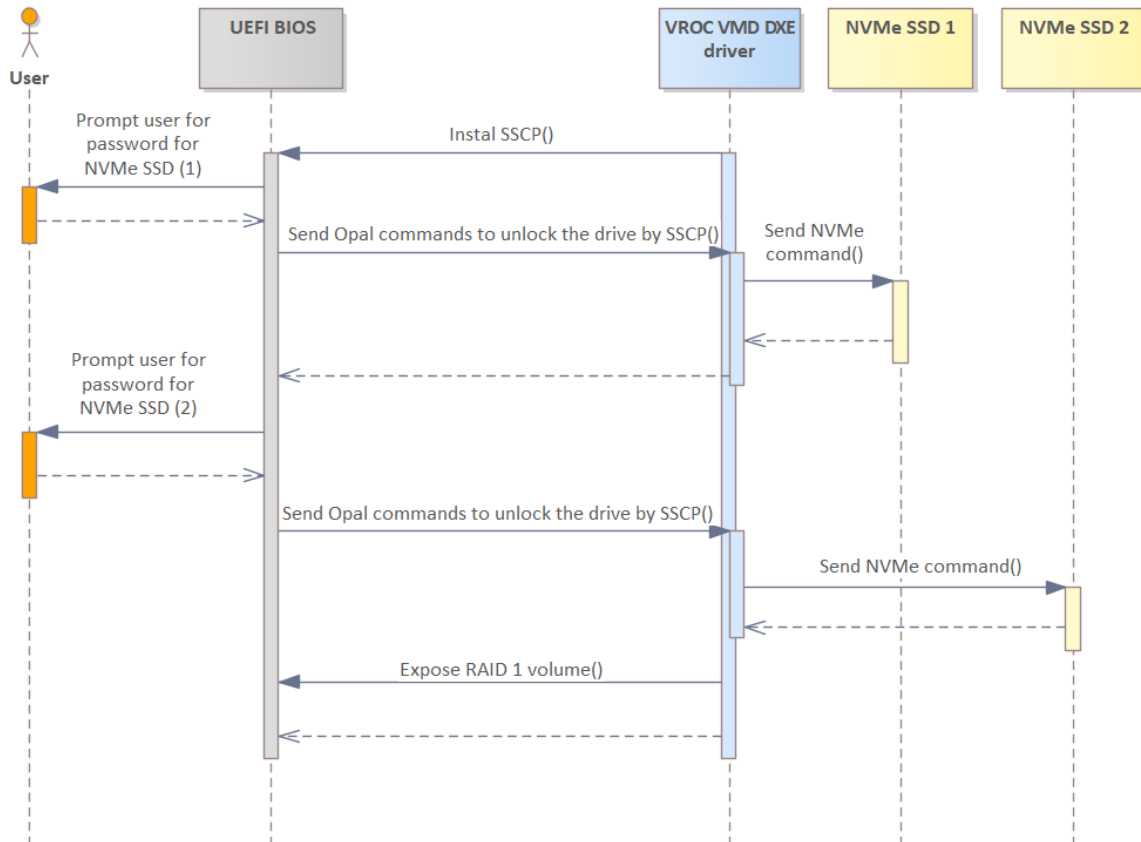
5.1.3 EFI_STORAGE_SECURITY_COMMAND_PROTOCOL

VROC VMD DXE driver and VROC SATA DXE drivers implement EFI_STORAGE_SECURITY_COMMAND_PROTOCOL to manage security of each drive by the BIOS. They support both ReceiveData and SendData functions.

Since security for each drive is managed separately, the security state of the drive may be changed by external (to VROC UEFI driver) entity. There might be cases where RAID volume consists of multiple SED drives. For example,

when RAID 1 volume consists of two NVMe SED drives with security enabled, then during the boot, the BIOS shall firstly unlock both drives to access RAID 1 volume data. After unlocking of both drives, the *VROC VMD DXE driver* assembles RAID 1 volume and exposes as a block device (by installing Block I/O protocols). *VROC VMD DXE driver* and *VROC SATA DXE drivers* automatically detects a drive's security state change and exposes block device(s). The example flow is presented on Figure 8.

Figure 8. Example NVMe SSDs unlocking flow using SSCP



Once all RAID member drives in the system are unlocked, the VROC VMD DXE driver exposes RAID 1 volume, by creating RAID volume handle and installing protocols as listed in Table 7. The properties of drives and RAID volumes are also automatically updated in VROC HII interface.

5.1.4 EFI_FIRMWARE_MANAGEMENT_PROTOCOL

VROC UEFI drivers implement subset of `EFI_FIRMWARE_MANAGEMENT_PROTOCOL` functions defined in UEFI Specification, which is reflected in Table 11.

Table 11. VROC EFI Firmware Management Protocol functions

EFI Firmware Management Protocol functions	VROC UEFI implementation
GetImageInfo	Supported
GetImage	Not supported
SetImage	Supported
CheckImage	Not supported
GetPackageInfo	Not supported
SetPackageInfo	Not supported

5.1.4.1 *EFI_FIRMWARE_MANAGEMENT_PROTOCOL* provided by VROC VMD DXE driver

5.1.4.1.1 GetImageInfo

VROC VMD DXE driver implements GetImageInfo function to return information about the current firmware image(s) of the NVMe drive. Table 12 specifies values returned by VROC VMD DXE driver in GetImageInfo function.

Table 12. GetImageInfo values returned by VROC VMD DXE driver

GetImageInfo field	VROC VMD DXE driver returned value
ImageInfoSize	As specified in UEFI Specification
ImageInfo	See
DescriptorVersion	3
DescriptorCount	Number of NVMe Firmware Slots with non-zero Firmware Revision
DescriptorSize	As specified in UEFI Specification
PackageVersion	0xFFFFFFFF
PackageVersionName	NULL

Table 13 specifies values of EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC VMD DXE driver.

Table 13. EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC VMD DXE driver

EFI_FIRMWARE_IMAGE_DESCRIPTOR member	VROC VMD DXE driver returned value
ImageIndex	Firmware Slot Index (1..N)
ImageTypeId	Preset GUID for NVMe (the same for all NVMe drives) 25D4CA70-05E8-44CB-B13D-ABA064717432
ImageId	Firmware Slot Index (1..N)
ImageIdName	"Drive model: <ModelNumber>, Firmware Slot: <FirmwareSlotIndex>"
Version	0
VersionName	Firmware Slot Information Log Page -> Firmware Revision for Slot [ImageIndex]
Size	1
AttributesSupported	IMAGE_ATTRIBUTE_IN_USE IMAGE_ATTRIBUTE_IMAGE_UPDATEABLE
AttributesSetting	IMAGE_ATTRIBUTE_IN_USE: If this Firmware Slot is active (pointed by AFI 2:0), then attribute set to 1, otherwise attribute cleared to 0 IMAGE_ATTRIBUTE_IMAGE_UPDATEABLE: If this is the first Firmware Slot and FRMW bit 0 is set, then attribute cleared to 0, otherwise attribute set to 1
Compatibilities	0

LowestSupportedImageVersion	3	
LastAttemptVersion	0	
LastAttemptStatus	Status of last SetImage command if called, otherwise LAST_ATTEMPT_STATUS_SUCCESS	
HardwareInstance	Bits	Value (physical location of NVMe drive)
	63:32	0
	31:24	CPU Socket Number
	23:16	VMD Controller Number
	15:8	Root Port Offset
	7:0	PCIe Slot Number

N in Table 13 represents the number of Firmware Slots supported by the NVMe controller, indicated by FRMW (bits 3:1) in NVMe specification.

Since, firmware management capabilities in NVMe specification are limited:

- Version and LastAttemptVersion are always equal to 0
- Size is always equal to 1

5.1.4.1.2 SetImageInfo

VROC VMD DXE driver implements SetImage function but does not support specifying firmware image update policy for NVMe drive. Thus, VendorCode argument of SetImage function shall be equal to NULL. The rest of SetImage arguments shall be provided according to the UEFI specification. If Progress function is provided by the caller, then *VROC VMD DXE driver* reports progress of firmware update process. Arguments This, ImageIndex, Image, ImageSize and AbortReason are mandatory and shall be non-zero values.

When SetImage function is called on EFI_FIRMWARE_IMAGE_PROTOCOL installed by *VROC VMD DXE driver*, the *VROC VMD DXE driver* firstly validates input. While input is valid, the driver checks if Firmware Commit and Firmware Image Download commands are supported by the NVMe drive. If the commands are supported, the driver sends image in chunks using Firmware Image Download NVMe command to the drive. While sending all chunks succeeds, the driver follows one of two branches:

1. If the drive does not support new firmware activation without reset, the *VROC VMD DXE driver* sends Firmware Commit NVMe command with Commit Action equal to 001b, specifying the new firmware is activated at the next Controller Level Reset. While command successfully completes, the driver issues Controller Level Reset.
2. If drive supports new firmware activation without reset, the *VROC VMD DXE driver* sends Firmware Commit NVMe command with Commit Action equal to 011b, specifying the new firmware is activated immediately. If the firmware activation was not successful because reset is required to activate firmware, the *VROC VMD DXE driver* issues Controller Level Reset.

Note that Controller Level Reset does not guarantee, the new firmware is activated. The drive may require NVM Subsystem Reset or Conventional Reset. These operations are not supported by *VROC VMD DXE driver*. Then, the caller's responsibility is to check whether new firmware is active or not, by calling GetImageFunction.

As final step, the driver reports 100% progress and returns EFI_SUCCESS.

5.1.4.2 EFI_FIRMWARE_MANAGEMENT_PROTOCOL provided by VROC SATA DXE drivers

5.1.4.2.1 GetImageInfo

VROC SATA DXE drivers implement GetImageInfo function to return information about the current firmware image(s) of the SATA drive. Table 14 specifies values returned by *VROC VMD DXE driver* in GetImageInfo function.

Table 14. GetImageInfo values returned by VROC SATA DXE drivers

GetImageInfo field	VROC SATA DXE drivers returned value
ImageInfoSize	As specified in UEFI Specification
ImageInfo	See
DescriptorVersion	3
DescriptorCount	1
DescriptorSize	As specified in UEFI Specification
PackageVersion	0xFFFFFFFF
PackageVersionName	NULL

Table 15 specifies values of EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC SATA DXE drivers.

Table 15. EFI_FIRMWARE_IMAGE_DESCRIPTOR structure returned by VROC SATA DXE drivers

EFI_FIRMWARE_IMAGE_DESCRIPTOR member	VROC SATA DXE drivers returned value
ImageIndex	1
ImageTypeId	Preset GUID for SATA 9C9A55B0-A5CD-456F-9AC3-76765D68733D
ImageId	1
ImageIdName	"Drive model: <ModelNumber>"
Version	0
VersionName	IDENTIFY DEVICE DATA Firmware Revision
Size	1
AttributesSupported	IMAGE_ATTRIBUTE_IN_USE IMAGE_ATTRIBUTE_IMAGE_UPDATEABLE
AttributesSetting	IMAGE_ATTRIBUTE_IN_USE: Always set to 1 IMAGE_ATTRIBUTE_IMAGE_UPDATEABLE: If DOWNLOAD MICROCODE or DOWNLOAD MICROCODE DMA is supported, then attribute set to 1, otherwise attribute cleared to 0
Compatibilities	0
LowestSupportedImageVersion	3
LastAttemptVersion	0
LastAttemptStatus	Status of last SetImage command if called, otherwise LAST_ATTEMPT_STATUS_SUCCESS
HardwareInstance	AHCI Port Number

Since, firmware management capabilities in ATA specification are limited:

- Version and LastAttemptVersion are always equal to 0
- Size is always equal to 1

5.1.4.2.2 SetImageInfo

VROC SATA DXE drivers implement SetImage function and allows to optionally specify firmware image update policy for SATA drive. Thus, VendorCode argument of SetImage function shall be NULL or shall be a pointer to the structure, defined as the following:

```

#define VROC_FW_UPDATE_ATA_SUBCOMMAND_03      0x03
#define VROC_FW_UPDATE_ATA_SUBCOMMAND_0E_0F   0x0E

typedef struct _VROC_FW_UPDATE_POLICY
{
    UINT8 VrocFwUpdatePolicyVersion; //Version of this structure
    UINT8 VrocFwUpdateAtaSubcommand; //ATA subcommand used to update SATA drive's FW
} VROC_FW_UPDATE_POLICY;

```

VrocFwUpdatePolicyVersion shall be set to 1, because it is the first version of the structure.

VrocFwUpdateAtaSubcommand shall be set to one of the defined values (VROC_FW_UPDATE_ATA_SUBCOMMAND_03 or VROC_FW_UPDATE_ATA_SUBCOMMAND_0E_0F)

The VendorCode argument is optional, when SetImage function is called with VendorCode equal to NULL, the *VROC SATA DXE driver* applies default policy. The rest of SetImage arguments must be provided according to the UEFI specification. If *Progress* function is provided by the caller, then *VROC SATA DXE driver* reports progress of firmware update process. Arguments This, ImageIndex, Image, ImageSize and AbortReason are mandatory and shall be non-zero values.

When SetImage function is called on EFI_FIRMWARE_IMAGE_PROTOCOL installed by *VROC SATA DXE driver*, the *VROC SATA DXE driver* firstly validates input. While input is valid, *VROC SATA DXE driver* selects ATA subcommand based on user input (VendorCode) and drive capabilities. Then, the driver follows one of two branches:

1. If subcommand 03h is selected, the driver sends firmware in chunks to the drive and report progress after sending every chunk.
2. If subcommands 0Eh and 0Fh are selected, the driver sends firmware in chunks to the drive using subcommand 0Eh and report progress after sending every chunk (up to 90%), and then activates new firmware using subcommand 0Fh and reports 100% progress.

While all commands are completed with success, the driver returns EFI_SUCCESS.

The selection policy for ATA DOWNLOAD MICROCODE subcommand is presented in Table 16 as decision matrix, where rows represent SATA drive capabilities, columns represent preferences provided by user (by VendorCode argument) and elements show one of the *VROC SATA DXE driver* flows:

- UNS – *VROC SATA DXE driver* returns EFI_UNSUPPORTED status
- 03h – *VROC SATA DXE driver* updates firmware using ATA DOWNLOAD MICROCODE subcommand 03h
- 0Eh, 0Fh – *VROC SATA DXE driver* updates firmware using ATA DOWNLOAD MICROCODE subcommands 0Eh and 0Fh

Table 16. ATA DOWNLOAD MICROCODE subcommand decision matrix in VROC SATA DXE drivers

Supported subcommands by drive	VendorCode argument value		
	NULL	SUBCOMMAND 03	SUBCOMMAND 0E 0F
None	UNS	UNS	UNS
03h	03h	03h	UNS
0Eh, 0Fh	0Eh, 0Fh	UNS	0Eh, 0Fh
03h, 0Eh, 0Fh	0Eh, 0Fh	03h	0Eh, 0Fh

5.2 VROC (non-standard) protocols and variables specification

5.2.1 VROC_DEVICE_INFO_PROTOCOL

Summary

Used to retrieve information about NVMe devices behind VMD controllers.

GUID

```
#define VROC_DEVICE_INFO_PROTOCOL_GUID \
    {0xef104449, 0xa5cd, 0x44ff, {0x9a,0x65,0x15,0xd,0xfd,0xa5,0x27,0xee}}
```

Protocol Interface Structure

```
#define VROC_OEM_SERVICE_LABEL_SZ 31          ///< Size of service label including null

typedef struct _VROC_DEVICE_INFO_PROTOCOL
{
    UINTN          SocketNumber;              ///< CPU Socket number.
    UINTN          VmdControllerNumber;      ///< VMD domain number per CPU.
    UINTN          RootPortNumber;          ///< Root port.
    UINT32         SlotNumber;              ///< PCI slot number.
    UINTN          BusNumber;               ///< PCI Bus number.
    UINTN          DeviceNumber;            ///< Device number.
    UINTN          FunctionNumber;          ///< Function number.
    UINT16         VendorId;               ///< Vendor ID i.e. 8086.
    EFI_STRING     SerialNumber;            ///< Device serial number.
    EFI_STRING     ModelNumber;            ///< Model number.
    UINT64         TotalBlocks;            ///< Total blocks number.
    UINT32         BlockSize;              ///< Block size.
    BOOLEAN        RaidDeviceMember;        ///< Determines if device is a RAID member.
    UINT16         RootPortOffset;          ///< Root port offset number.
    EFI_HANDLE     BlockDeviceHandle;       ///< Returns associated block device handle.
    UINT16         DeviceId;
    UINT16         SubsystemVendorId;
    UINT16         SubsystemId;
    UINT8          ClassCode[3];
    UINT8          RevisionId;
    UINT8          FirmwareRev[8];
    BOOLEAN        OptionROMBar;
    UINT8          RootPortBusNum;
    UINT8          RootPortDeviceNum;
    UINT8          RootPortFunctionNum;
    UINT8          SegmentNum;
    UINT8          Reserved;
    UINT8          ServiceLabel[VROC_OEM_SERVICE_LABEL_SZ]; ///< service label
    UINT8          LocationType;           ///< location type
    UINT32         LocationOrdinalValue;   ///< ordinal value connected with location type
} VROC_DEVICE_INFO_PROTOCOL;
#pragma pack()
```

Parameters

- SocketNumber – CPU socket number where NVMe device is connected to.
- VmdControllerNumber – Identifier of the VMD controller associated to NVMe device per each CPU. This value is enumerated for each CPU separately.
- RootPortNumber – Number of the PCI root port where NVMe device is connected.
- SlotNumber – NVMe device slot number.
- BusNumber – NVMe device's PCI bus number.
- DeviceNumber – NVMe device's PCI device number.
- FunctionNumber – NVMe device's PCI function number.
- VendorId – NVMe device's vendor ID, i.e. for Intel it is 0x8086
- SerialNumber – Null terminated string with NVMe device's serial number.
- ModelNumber – Null terminated string with NVMe device's model number.
- TotalBlocks – Total size of NVMe device, in blocks.

- BlockSize – Logical block size in bytes.
- RaidDeviceMember – If TRUE, this disk is a member of RAID device, otherwise it is a pass-thru disk.
- RootPortOffset – In case of direct attached NVMe drives, offset is a 0-based number of the slot, where the drive is attached, within a given VMD domain. In case of switch attached drives, this field will be equal to switch slot number and a SlotNumber field has to be used to identify the drives instead.
- BlockDeviceHandle – EFI handle on which EFI_BLOCK_IO_PROTOCOL was installed for this device (available only for pass thru devices, for RAID members this field is NULL).
- DeviceId – NVMe device's PCI device ID
- SubsystemVendorId – NVMe device's Subsystem vendor ID
- Class Code – NVMe Class Code
- RevisionId – NVMe Firmware Revision ID
- FirmwareRev – NVMe Firmware Version ID
- OptionROMBar – False when Intel VMD is enabled. This is because the Option ROM on a single NVMe SSD controller firmware will not be exposed on devices on VMD-enabled domains.
- RootPortBusNum – Bus Number of the Intel VMD Root Port
- RootPortDeviceNum – Device Number of the Intel VMD Root Port
- RootPortFunctionNum – Function Number of the Intel VMD Root Port
- SegmentNum – Segment Number of the Intel VMD Root Port
- Reserved – depreciated field
- ServiceLabel – The label of the part (drive) location, for example "Port=1A, Box=3, Bay=4"
- LocationType – The type of locations for a part (drive) within an enclosure. One of the following values:

Value	Description
0	Slot
1	Bay
2	Connector
3	Socket
4	Backplane
5	Embedded

- LocationOrdinalValue – The number that represents the location of the part (drive). For example, if LocationType is "Slot" and this unit is in slot 2, the LocationOrdinalValue is `2`.

Description

This protocol is installed on each handle of NVMe drive that is managed by *VROC VMD DXE driver*. Caller should not try to deallocate or change values of the memory under returned pointer as this can lead to undefined behavior. This is *VROC VMD DXE driver* responsibility to allocate and free this memory area whenever it is necessary.

If caller service intends to keep device information i.e. for a lifetime of a global structure instance, a copy of interface memory should be made, and protocol should be closed. This is dictated by fact, that protocol data can change in time if device enumeration is done for i.e. create or delete volume.

To calculate drive capacity in bytes, one can multiply the TotalBlocks and BlockSize parameters.

Example usage

```
EFI_GUID gEfiVROCDeviceInfoProtocolGuid = VROC_DEVICE_INFO_PROTOCOL_GUID;

EFI_STATUS EFIAPI UefiMain(IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE* SystemTable)
{
    EFI_HANDLE* pHandleBuf = NULL;
    UINTN nHandles = 0;
    VROC_DEVICE_INFO_PROTOCOL* result = NULL;
    EFI_BOOT_SERVICES* gBS = SystemTable->BootServices;
    Print(L"Searching for VROC_DEVICE_INFO_PROTOCOL instances...\n");
}
```

```

EFI_STATUS nStatus = gBS->LocateHandleBuffer(
    ByProtocol,
    &gEfiVROCDeviceInfoProtocolGuid,
    NULL,
    &nHandles,
    &pHandleBuf
);

if (nStatus != EFI_SUCCESS) {
    return nStatus;
}

Print(L"Found %d instances.\n\n", nHandles);
for (UINTN nIdx = 0; nIdx < nHandles; nIdx++) {
    if (pHandleBuf[nIdx]) {
        Print(L"%d] Opening VROC_DEVICE_INFO_PROTOCOL.\n", nIdx);
        nStatus = gBS->OpenProtocol(
            pHandleBuf[nIdx],
            &gEfiVROCDeviceInfoProtocolGuid,
            (void**)&result,
            ImageHandle,
            NULL,
            EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
        );
        if (nStatus != EFI_SUCCESS) {
            Print(L"%d] ERROR: %d\n", nIdx, nStatus);
            continue;
        }

        if (result) {
            UINT64 sizeInBytes = result->TotalBlocks * result->BlockSize;
            Print(L"%d] %s - %s\n", nIdx, result->ModelNumber, result->SerialNumber);
            Print(L"    VMD Controller: %d\n", result->VmdControllerNumber);
            Print(L"    Socket Number: %d\n", result->SocketNumber);
            Print(L"    Slot Number: %d\n", result->SlotNumber);
            Print(L"    Root Port: %d\n", result->RootPortNumber);
            Print(L"    Root Port Offset: %d\n", result->RootPortOffset);
            Print(L"    PCI Bus:Device.Function: %02x:%02x.%x\n",
                result->BusNumber, result->DeviceNumber, result->FunctionNumber);
            Print(L"    Total Blocks: %ld\n", result->TotalBlocks);
            Print(L"    Block Size: %d\n", result->BlockSize);
            Print(L"    Size in Bytes: %ld\n", sizeInBytes);
            Print(L"    RAID Member: %s\n", result->RaidDeviceMember ? "TRUE" : "FALSE");
            Print(L"    Block Device Handle: %x\n", result->BlockDeviceHandle);
            Print(L"    Vendor ID: 0x%x\n", result->VendorId);
            Print(L"    Device ID: 0x%x\n", result->DeviceId);
            Print(L"    Subsystem Vendor ID: 0x%x\n", result->SubsystemVendorId);
            Print(L"    Subsystem ID: 0x%x\n", result->SubsystemId);
            Print(L"    Device Class Codes: %02x:%02x:%02x\n",
                result->ClassCode[0], result->ClassCode[1], result->ClassCode[2]);
            Print(L"    Revision ID: 0x%x\n", result->RevisionId);
            Print(L"    Firmware Revision: %a\n", result->FirmwareRev);
            Print(L"    Option ROM Bar: %s\n", result->OptionROMBar ? "TRUE" : "FALSE");
            Print(L"    RootPortBusNum: %d\n", result->RootPortBusNum);
            Print(L"    RootPortDeviceNum: %d\n", result->RootPortDeviceNum);
            Print(L"    RootPortFunctionNum: %d\n", result->RootPortFunctionNum);
            Print(L"    SegmentNum: %d\n", result->SegmentNum);

            if (result->BlockDeviceHandle == NULL) {
                Print(L"%d] No block IO protocol associated to given handle.\n", nIdx);
            }
            else {
                EFI_DEVICE_PATH_PROTOCOL* pdpDev = DevicePathFromHandle(result->BlockDeviceHandle);
                if (!pdpDev) {
                    Print(L"ERROR: Could not get device path from handle!\n");
                }
                else {
                    Print(L"%d] Device path text: %s\n",

```

```

        nIdx, ConvertDevicePathToText(pdpDev, TRUE, TRUE));
    }
}
else {
    Print(L "[%d] ERROR: No interface returned.\n", nIdx);
}
Print(L "[%d] Closing VROC_DEVICE_INFO_PROTOCOL.\n\n", nIdx);
gBS->CloseProtocol(
    pHandleBuf[nIdx],
    &gEfiVROCDeviceInfoProtocolGuid,
    ImageHandle,
    NULL
);
}
}
gBS->FreePool(pHandleBuf);
return EFI_SUCCESS;
}

```

5.2.2 VROC_VOLUME_INFO_PROTOCOL

Summary

VROC_VOLUME_INFO_PROTOCOL can be used to retrieve information for all RAID volumes created through Intel VROC. The information can also be seen using the Human Interface Infrastructure (HII) and Intel VROC UEFI tools.

GUID

```

#define VROC_VOLUME_INFO_PROTOCOL_GUID \
    {0x6139796a, 0x3df0, 0x4838, {0x82,0x7b,0x42,0x7e,0x7d,0xc8,0x28,0x1c}}

```

Protocol Interface Structure

```

typedef struct _VROC_VOLUME_INFO_PROTOCOL
{
    EFI_STRING VendorID;
    EFI_STRING ProductID;
    EFI_STRING VolumeName;
    UINT64      TotalBlocks;
    UINTN       BlockSize;
    UINT8       RaidLevel;
    UINT8       VolumeType;
} VROC_VOLUME_INFO_PROTOCOL;

```

Parameters

- VendorId – RAID volume vendor identification string – “Intel”
- ProductId – RAID volume product identification string, constructed as following:

Driver	ProductId
VROC VMD DXE driver	“VROC <RaidLevel> Volume”, for example: “VROC 1 Volume”
VROC SATA DXE drivers	“RAID <RaidLevel> Volume”, for example: “RAID 1 Volume”

- VolumeName – name of a RAID volume, set during RAID volume creation
- TotalBlocks – total size of RAID volume, in blocks
- BlockSize – logical block size of RAID volume, in bytes
- RaidLevel – RAID volume level, one of the following values:

```
#define VROC_VOLUME_RAID0 0
#define VROC_VOLUME_RAID1 1
#define VROC_VOLUME_RAID5 5
#define VROC_VOLUME_RAID10 10
```

- VolumeType – RAID volume type, which determines the VROC UEFI driver, where RAID volume originates from, one of the following values:

```
#define VROC_VOLUME_SATA 0
#define VROC_VOLUME_sSATA 1
#define VROC_VOLUME_VMD 2
#define VROC_VOLUME_tSATA 3
```

Description

This protocol is installed on each handle of RAID volume created by VROC UEFI drivers. Caller should not try to deallocate or change values of the memory under returned pointer as this can lead to undefined behavior. This is VROC UEFI driver responsibility to allocate and free this memory area whenever it is necessary.

If caller service intends to keep device information i.e. for a lifetime of a global structure instance, a copy of interface memory should be made, and protocol should be closed. This is dictated by fact, that protocol data can change in time if device enumeration is done for i.e. create or delete volume.

To calculate RAID volume capacity in bytes, one can multiply the TotalBlocks and BlockSize parameters.

Example usage

```
EFI_GUID gEfiVROCVolumeInfoProtocolGuid = VROC_VOLUME_INFO_PROTOCOL_GUID;

EFI_STATUS
EFIAPI
UefiMain(
    IN EFI_HANDLE      ImageHandle,
    IN EFI_SYSTEM_TABLE *SystemTable
)
{
    EFI_HANDLE      * pHandleBuf = NULL;
    UINTN           nHandles = 0;
    VROC_VOLUME_INFO_PROTOCOL * result = NULL;

    Print(L"\nIntel(R) VROC Volume Info Protocol Example.\n\n");

    Print(L"Searching for RSTE_VOLUME_INFO_PROTOCOL instances...\n");

    EFI_STATUS nStatus = gBS->LocateHandleBuffer(
        ByProtocol,
        &gEfiVROCVolumeInfoProtocolGuid,
        NULL,
        &nHandles,
        &pHandleBuf
    );

    if (nStatus != EFI_SUCCESS) {
        return nStatus;
    }

    Print(L"Found %d instances.\n\n", nHandles);
    for (UINTN nIdx = 0; nIdx < nHandles; nIdx++) {
        if (pHandleBuf[nIdx]) {
            Print(L"[%d] Opening RSTE_VOLUME_INFO_PROTOCOL.\n", nIdx);

            nStatus = gBS->OpenProtocol(
                pHandleBuf[nIdx],
                &gEfiVROCVolumeInfoProtocolGuid,
                (void**)&result,
                ImageHandle,
```

```

        NULL,
        EFI_OPEN_PROTOCOL_BY_HANDLE_PROTOCOL
    );

    if (nStatus == EFI_INVALID_PARAMETER) {
        Print(L"%d] ERROR: One of the parameters has an invalid value.\n", nIdx);
        continue;
    }

    if (nStatus == EFI_ACCESS_DENIED || nStatus == EFI_ALREADY_STARTED) {
        Print(L"%d] ERROR: Already opened with BY_DRIVER or EXCLUSIVE attribute.\n", nIdx);
        continue;
    }

    if (result) {
        UINT64 sizeInBytes = result->TotalBlocks * result->BlockSize;

        Print(L"%d] %s %s (%s)\n",
            nIdx,
            result->VendorID,
            result->ProductID,
            result->VolumeName
        );

        Print(L"        Total blocks: %d\n", result->TotalBlocks);
        Print(L"        Block size: %d\n", result->BlockSize);
        Print(L"        Size in bytes: %d\n", sizeInBytes);
        Print(L"        Raid Level: %d\n", result->RaidLevel);

        switch (result->VolumeType)
        {
        case VROC_VOLUME_SATA:
            Print(L"        Volume Type: SATA\n");
            break;
        case VROC_VOLUME_sSATA:
            Print(L"        Volume Type: sSATA\n");
            break;
        case VROC_VOLUME_VMD:
            Print(L"        Volume Type: VMD\n");
            break;
        case VROC_VOLUME_tSATA:
            Print(L"        Volume Type: tSATA\n");
            break;
        default:
            break;
        }
    }
    else {
        Print(L"%d] ERROR: No interface returned.\n", nIdx);
    }

    Print(L"%d] Closing VROC_VOLUME_INFO_PROTOCOL.\n\n", nIdx);

    gBS->CloseProtocol(
        pHandleBuf[nIdx],
        &gEfiVROCVolumeInfoProtocolGuid,
        ImageHandle,
        NULL
    );
}

gBS->FreePool(pHandleBuf);
return EFI_SUCCESS;
}

```

5.2.3 VrocPhysicalLocation variable

This section outlines the architecture enabling Intel VROC to report the physical location of the drives directly attached to the platform. Intel VROC reports the physical location of the platform drives in the UEFI HII user interface. For example, in the Intel VROC HII, the drives are reported, to the user, as "Port 5:0, Slot 13, CPU0, VMD4, BDF 81:00.0". This information may not be very user-friendly.

For platforms that do not produce VrocPhysicalLocation variable, Intel VROC reports the location of the drives based on how the Intel VMD controller discovers and reports the drives.

The challenge is that there is no way for Intel VROC to know how a particular manufacturer has designed and laid out the drive connections.

To make this information more user-friendly, the VrocPhysicalLocation variable provides an interface for the manufacturer to provide that physical location information to Intel VROC. The user will see provided information in the Intel VROC HII.

Summary

The VrocPhysicalLocation UEFI variable will hold a predefined structure with all the information required for the manufacturer to implement the physical drive location functionality. VROC UEFI driver consumes the variable and uses the provided data to build the device Redfish repository. **The OEM must ensure this UEFI variable is available before starting the VROC UEFI driver.**

Name

```
#define VROC_CONF_VAR_NAME L"VrocPhysicalLocation"
```

GUID

```
#define VROC_CONF_GUID \
    {0x193dfefa, 0xa445, 0x4302, {0x99, 0xd8, 0xef, 0x3a, 0xad, 0x1a, 0x04, 0xc6}}
```

Note: The Intel VROC driver already uses this GUID to expose other UEFI variables.

Variable Structure

```
/* SATA controllers (SATA, sSATA, tSATA) */
#define VROC_SATA_CTRL_LIMIT    3

/*
 * Should fit the OEM Part Number used for VROC license type
 * Shall end with the NULL(0x00) character.
 */
#define VROC_SKU_LABEL_STR_LEN    17

#pragma pack(1)
typedef union {
    struct {
        UINT32 Version;
        VROC_NVME_CTRL NvmeCtrl;
        VROC_SATA_CTRL SataCtrls[VROC_SATA_CTRL_LIMIT];
        /*
         * The OEM "Part Number" - defined for the VROC license type.
         * Shall end with the NULL(0x00) character.
         */
        CHAR8 OemSkuLabel[VROC_SKU_LABEL_STR_LEN];
    } v1;
} VROC_PHYSICAL_LOCATION;
#pragma pack()
```

Parameters

- Version – the structure version of VrocPhysicalLocation UEFI variable.

```
#define VROC_PHYSICAL_LOCATION_V1 1
```

- NvmeCtrl – the structure describing the physical location of NVMe controllers in the system, defined as follows:

```
#define VROC_NVME_DRIVE_LIMIT 96
#define VROC_NVME_PORTS_LIMIT VROC_NVME_DRIVE_LIMIT

#pragma pack(1)
typedef struct {
    VROC_PORT NvmePorts[VROC_NVME_PORTS_LIMIT];
    VROC_DRIVE NvmeDrives[VROC_NVME_DRIVE_LIMIT];
    VROC_PHYSICAL_LOCATION_INFO CtrlPhysicalLocation;
} VROC_NVME_CTRL;
#pragma pack()
```

Please refer to related definitions section below for the structure definitions.

- SataCtrls – array of VROC_SATA_CTRL structures, where each VROC_SATA_CTRL structure describes the physical location of SATA drives, defined as follows:

```
#define VROC_SATA_DRIVE_LIMIT 8
#define VROC_SATA_PORTS_LIMIT VROC_SATA_DRIVE_LIMIT

#pragma pack(1)
typedef struct {
    VROC_PORT SataPorts[VROC_SATA_PORTS_LIMIT];
    VROC_DRIVE SataDrives[VROC_SATA_DRIVE_LIMIT];
    VROC_PHYSICAL_LOCATION_INFO CtrlPhysicalLocation;
} VROC_SATA_CTRL;
#pragma pack()
```

Please refer to related definitions section below for the structure definitions.

Related definitions

The structures used in the VrocPhysicalLocation UEFI variable are defined as follows:

```
/* VROC controller location ordinal values */
#define VROC_CTRL_LOCATION_ORDINAL_NVME 0
#define VROC_CTRL_LOCATION_ORDINAL_SATA 1
#define VROC_CTRL_LOCATION_ORDINAL_SSATA 2
#define VROC_CTRL_LOCATION_ORDINAL_TSATA 3

#pragma pack(1)
/*
 * Defined in Redfish schema for LocationType (Resource.v1_13_0.json).
 */
typedef enum {
    SLOT,
    BAY,
    CONNECTOR,
    SOCKET,
    BACKPLANE,
    EMBEDDED
}
```

```

} VROC_LOCATION_TYPE;

/*
 * The location of a resource.
 */
typedef struct {
    /*
     * The label of the part location, for example "Port=1A, Box=3, Bay=4".
     * Shall end with the NULL(0x00) character.
     */
    CHAR8 ServiceLabel[VROC_SERVICE_LABEL_STR_LEN];
    /*
     * The types of locations for a part within an enclosure.
     */
    VROC_LOCATION_TYPE LocationType;
    /*
     * The number that represents the location of the part.
     * For example, if LocationType is `Slot` and this unit is in slot 2, the
     * LocationOrdinalValue is `2`.
     */
    UINT32 LocationOrdinalValue;
} VROC_PHYSICAL_LOCATION_INFO;

/*
 * Controller types supported by Intel® VROC.
 */
typedef enum {
    /*
     * NONE implies that the drive or port slot in the table is not used.
     */
    NONE,
    NVME,
    SATA
} VROC_CTRL_TYPE;

/*
 * The address below are used to match VROC ports with the OEM's provided location
 * information.
 */
typedef struct {
    VROC_CTRL_TYPE DriveType; // Indicates if the port is used.
    UINT8 CpuId;
    UINT8 RootPort;
} VROC_NVME_PORT_ADDRESS;

typedef struct {
    VROC_CTRL_TYPE DriveType; // Indicates if the port is used.
    UINT8 SataId;
    UINT8 SataPort;
} VROC_SATA_PORT_ADDRESS;

/*
 * The union is convenient to abstract interface in the VROC_PORT struct.
 */
typedef union {
    VROC_NVME_PORT_ADDRESS NvmePort;
    VROC_SATA_PORT_ADDRESS SataPort;
} VROC_PORT_ADDRESS;

/*
 * The address below are used to match VROC drives with the OEM's provided location

```

```

    * information.
    */
typedef struct {
    VROC_NVME_PORT_ADDRESS Port;
    /*
     * It is used to describe the drive PCIe port. For example, if the drive
     * PCIe port equals "1C," the offset will have a value of 2. The letter
     * at the end of the port address should be mapped to a number
     * value (0-based).
     */
    UINT8 RootPortOffset;
} VROC_NVME_DRIVE_ADDRESS;

/*
 * The union is convenient to abstract interface in the VROC_DRIVE struct.
 */
typedef union {
    VROC_SATA_PORT_ADDRESS Sata;
    VROC_NVME_DRIVE_ADDRESS Nvme;
} VROC_DRIVE_ADDRESS;

/*
 * For NVMe, Port is equal VMD domain. For SATA, it is a single SATA controller
 * port.
 */
typedef struct {
    VROC_PORT_ADDRESS PortAddress;
    VROC_PHYSICAL_LOCATION_INFO PortPhysicalLocation;
} VROC_PORT;

/*
 * Used for NVMe and SATA drives.
 */
typedef struct {
    VROC_DRIVE_ADDRESS DriveAddress;
    /*
     * Indicates if the drive was available on the boot time.
     * Note: the UEFI EDK2 environment defines BOOLEAN as uint8_t
     */
    BOOLEAN DrivePresent;
    VROC_PHYSICAL_LOCATION_INFO DrivePhysicalLocation;
} VROC_DRIVE;
#pragma pack()

```

Notes:

1. For NVMe drive service label, Intel VROC will take the physical drive location value directly from the `NvmeCtrl.NvmeDrives[].DrivePhysicalLocation.ServiceLabel` field in `VROC_PHYSICAL_LOCATION` structure. There is no additional concatenation required of the Service Label values for the connected Port and Controllers.
2. For SATA drive service label, Intel VROC will take the physical drive location value directly from the `SataCtrls[].SataDrives[].DrivePhysicalLocation.ServiceLabel` field in `VROC_PHYSICAL_LOCATION` structure. There is no additional concatenation required of the Service Label values for the connected Port and Controllers.
3. Intel VROC uses the following physical location values for the NVMe storage controller resource:
 - `NvmeCtrl.CtrlPhysicalLocation.ServiceLabel` = "Embedded=NVMe"
 - `NvmeCtrl.CtrlPhysicalLocation.LocationType` = 5 (Embedded)
 The `NvmeCtrl.CtrlPhysicalLocation.LocationOrdinalValue` is not used.

Examples

Example 1 (NVMe)

Below is an example with the data in the JSON format to illustrate how OEMs/ODMs can map Intel VMD addressing to the required physical location format for a scenario with three drives:

- drive connected to the PCIe port "0C" (0-based) on CPU 0 - connected to the drive enclosure (Box=1, Bay=4),
- drive connected to the PCIe port "2A" (0-based) on CPU 0 - add-in-card SSD.
- drive connected to the PCH and port 9 - connected to the internal M.2 slot (e.g. "Embedded:Port=9A")

Implementation:

```

{
  "NvmeCtrl": {
    "NvmePorts": [
      {
        "PortAddress": {
          "NvmePort": {
            "CpuId": 0,
            "DriverType": 1, // Nvme
            "RootPort": 0 // 0-based, related to "0C"
          }
        },
        "PortPhysicalLocationInfo": {
          "LocationOrdinalValue": 1,
          "LocationType": 2, // Connector
          "ServiceLabel": "Port=0"
        }
      },
      {
        "PortAddress": {
          "NvmePort": {
            "CpuId": 0,
            "DriverType": 1, // Nvme
            "RootPort": 2 // 0-based, related to "2A"
          }
        },
        "PortPhysicalLocationInfo": {
          "LocationOrdinalValue": 2,
          "LocationType": 2, // Connector
          "ServiceLabel": "Port=2"
        }
      },
      {
        "PortAddress": {
          "NvmePort": {
            "CpuId": 15, // value used for the PCH
            "DriverType": 1, // Nvme
            "RootPort": 0 // value used for the PCH
          }
        },
        "PortPhysicalLocationInfo": {
          "LocationOrdinalValue": 3,
          "LocationType": 2, // Connector
          "ServiceLabel": "Embedded"
        }
      }
    ],
    "NvmeDrives": [
      {
        "DriveAddress": {
          "Nvme": {
            "Port": {
              "CpuId": 0,
              "DriverType": 1, // Nvme
              "RootPort": 0 // 0-based, related to "0C"
            },
            "RootPortOffset": 2 // related to "0C"
          }
        },
        "DrivePresent": true,
        "DrivePhysicalLocation": {

```

```

        "LocationOrdinalValue": 3,
        "LocationType": 1, // Bay
        "ServiceLabel": "Port=0C:Box=1:Bay=3"
    },
    {
        "DriveAddress": {
            "Nvme": {
                "Port": {
                    "CpuId": 0,
                    "DriverType": 1, // Nvme
                    "RootPort": 2 // 0-based, related to "2A"
                },
                "RootPortOffset": 0 // related to "2A"
            }
        },
        "DrivePresent": true,
        "DrivePhysicalLocation": {
            "LocationOrdinalValue": 2,
            "LocationType": 2, // Connector
            "ServiceLabel": "Port=2A:Box=1:Bay=3"
        }
    },
    {
        "DriveAddress": {
            "Nvme": {
                "Port": {
                    "CpuId": 15, // value used for the PCH
                    "DriverType": 1, // Nvme
                    "RootPort": 0 // value used for the PCH
                },
                "RootPortOffset": 9 // related to port 9
            }
        },
        "DrivePresent": true,
        "DrivePhysicalLocation": {
            "LocationOrdinalValue": 3,
            "LocationType": 2, // Connector
            "ServiceLabel": "Embedded:Port=9A"
        }
    }
],
"CtrlPhysicalLocation": {
    "LocationOrdinalValue": 0,
    "LocationType": 5, // Embedded
    "ServiceLabel": "Embedded=NVMe"
},
"Version": 0
}

```

Example 2 (SATA)

Below is an example with the data to illustrate how OEMs/ODMs can map SATA addressing to the required physical location format for a drive connected to the sSATA controller on port 4.

Implementation:

```

{
  "SataCtrls": [
    {
      "CtrlPhysicalLocation": {
        "LocationOrdinalValue": 2,
        "LocationType": 5, // Embedded
        "ServiceLabel": "Embedded=SSATA"
      },
      "SataDrives": [
        {
          "DriveAddress": {
            "Sata": {
              "DriveType": 2, // Sata
              "SataId": 1,
              "SataPort": 4 // related to port 4
            }
          },
          "DrivePresent": true,
          "DrivePhysicalLocation": {
            "LocationOrdinalValue": 4,
            "LocationType": 1, // Bay
            "ServiceLabel": "Port=4:Box=1:Bay=4"
          }
        }
      ],
      "SataPorts": [
        {
          "PortAddress": {
            "SataPort": {
              "DriveType": 2, // Sata
              "SataId": 1,
              "SataPort": 4 // related to port 4
            }
          },
          "PortPhysicalLocationInfo": {
            "LocationOrdinalValue": 4,
            "LocationType": 2, // Connector
            "ServiceLabel": "Port=4"
          }
        }
      ]
    },
    {
      "Version": 0
    }
  ]
}

```