VOLUME 5: Platform Initialization Specification

Standards

Version 1.1 Errata B

July 1, 1010

Revision History

Revision	Revision History	Date
1.0	Initial public release.	8/21/06
1.0 errata	Mantis tickets: • M47 dxe_dispatcher_load_image_behavior • M48 Make spec more consistent GUID & filename. • M155 FV_FILE and FV_ONLY: Change subtype number back to th:e original one. • M171Remove 10 us lower bound restriction for the TickPeriod in the Metronome • M178 Remove references to tail in file header and made file checksum for the data • M183 Vol 1-Vol 5: Make spec more consistent. • M192 Change PAD files to have an undefined GUID file name and update all FV	10/29/07
1.1	Mantis tickets: • M39 (Updates PCI Hostbridge & PCI Platform) • M41 (Duplicate 167) • M42 Add the definition of theDXE CIS Capsule AP & Variable AP • M43 (SMbios) • M46 (SMM error codes) • M163 (Add Volume 4--SMM • M167 (Vol2: adds the DXE Boot Services Protocols--new Chapter 12) • M179 (S3 boot script) • M180 (PMI ECR) • M195 (Remove PMI references from SMM CIS) • M196 (disposable-section type to the FFS)	11/05/07
1.1 correction	Restore (missing) MP protocol	03/12/08

1.1 Errata	• 230 Updated to Volume 4, section 4.2, ReportStatusCode • 231Parameter/description updates for Volume 4, section 4.3, ReadSaveState() & WriteSaveState(), Parameters • 232 SMM I/O Protocol Updates • 233 Volume 4, Section 5.2 & 5.3 Updates • 234 Volume 4, Section 5.5 Misc. Errata • 235Volume 4, Chapter 8 Should Be Integrated Into Volume 3, Section 2.1.4.1, 2.1.5.1 and 3.2.5 • 236 Volume 4, Section 9.5.1, 9.6, 9.7, 9.8 and 9.9 Formatting • 238 CpuSaveStateFormat deprecated in Vol4 of SMM PI1.1 draft • 239 rename EFI_SMM_HANDLER_ENTRY_POINT to be EFI_SMM_HANDLER_ENTRY_POINT2 in Vol4 SMM of PI1.1 • 240 PI1.1 Vol4 typos • 244 Replace EFI_FIRMWARE_VOLUME_INFO_PPI with EFI_PEI_FIRMWARE_VOLUME_INFO_PPI • 250 PEI_SPECIFICATION_MINOR_REVISION should be 10 • 251 Firmware File Type Table (Volume 3, 2.1.4.1, Table 1) Should Not Contain Section Information • 252 Volume 3, Table 2 (2.1.5.1) does not contain EFI_SECTION_DISPOSABLE • 253 EFI_SECTION_PIC has incorrect typedef • 254 ReInstallPpi() has incorrect prototype • 255 NotifyPpi() has the incorrect prototype • 256 CreateHob() has incorrect prototype • 257 PEI Specification, Section 4.2.1 and Section 4.2.2 should be peers of 4.1, 4.3, etc. • 258 CreateHob() refers to non-existent specification. • 259 FfsFindNextFile() Parameters Are Incorrect • 260 FfsFindSectionData() has incorrect parameter description • 261 AllocatePages() (PEI) refers to a non-existent specification and non-existent function. • 262 FfsGetVolumeInfo() missing return status codes • 263 EFI_PEI_NOTIFY_DESCRIPTOR and EFI_PEI_PPI_DESCRIPTOR prototypes are incorrect • 264 EFI_PEI_SERVICES: Remove references to "future installed services" from prototype • 265 EFI_FV_BLOCK_MAP definition does not exist • 267 Invalid References To the PI Firmware Storage Specification • 268 GUIDED_SECTION_EXTRACTION_PROTOCOL missing 'EFI_' prefix • 269 References to EFI_FIRMWARE_VOLUME_PROTOCOL should be EFI_FIRMWARE_VOLUME2_PROTOCOL • 272 Various fixes for Communicate() in PI 1.1, Volume 4 • 273 EFI_SMM_CONTROL2_PROTOCOL Errata • 274 Miscellanous SMST Errata from Volume 4, Section 3.2 • 275 Chapter heading for DXE ReportStatusCode function • 276 EFI_STATUS_CODE_RUNTIME_PROTOCOL_GUID has extra ';' • 277 Remove references to "Framework" and "Framework-based" in Volume 5	04/25/08

1.1 Errata	Mantis tickets • 204 Stack HOB update 1.1errata • 225 Correct references from EFI_FIRMWARE_VOLUME_PROTOCOL to EFI_FIRMWARE_VOLUME2_PROTOCOL • 226 Remove references to Framework • 227 Correct protocol name GUIDED_SECTION_EXTRACTION_PROTOCOL • 228 insert"typedef" missing from some typedefs in Volume 3 • 243 Define interface "EFI_PEI_FV_PPI" declaration in PI1.0 FfsFindNextVolume() • 285 Time quality of service in S3 boot script poll operation • 287 Correct MP spec, PIVOLUME 2:Chapter 13.3 and 13.4 - return error language • 290 PI Errata • 305 Remove Datahub reference • 336 SMM Control Protocol update • 345 PI Errata • 353 PI Errata • 360 S3RestoreConfig description is missing • 363 PI Volume 1 Errata • 367 PCI Hot Plug Init errata • 369 Volume 4 Errata • 380 SMM Development errata • 381 Errata on EFI_SMM_SAVE_STATE_IO_INFO	01/13/09
1.1 Errata	• 247 Clarification regarding use of dependency expression section types with firmware volume image files • 399 SMBIOS Protocol Errata • 405 PIWG Volume 5 incorrectly refers to EFI_PCI_OVERRIDE_PROTOCOL • 422 TEMPORARY_RAM_SUPPORT_PPI is misnamed • 428 Volume 5 PCI issue • 430 Clarify behavior w/ the FV extended header	02/23/09
1.1 Errata	• 407 Add LMA Pseudo-Register to SMM Save State Protocol • 455 Clarify InstallPeiMemory() • 465 Correct PMI Interface • 466 Add EXTENDED_SAL_PROC definition, etc • 467 Vol2 & Vol3 Errata	05/22/09

1.1 errata	• 345 PI1.0 errata • 468 Issues on proposed PI1.2 ACPI System Description Table Protocol • 492 Add Resource HOB Protectability Attributes • 494 Vol. 2 Appendix A Clean up • 495 Vol 1: update HOB reference • 380 • 501 Clean Up SetMemoryAttributes() language Per Mantis 489 (from USWG) • 502 Disk info • 503 typo • 504 remove support for fixed address resources • 509 PCI errata – execution phase • 510 PCI errata - platform policy • 511 PIC TE Image clarification/errata • 520 PI Errata • 521 Add help text for EFI_PCD_PROTOCOL for GetNextTokenSpace • 525 Itanium ESAL, MCA/INIT/PMI errata • 526 PI SMM errata • 529 PCD issues in Volume 3 of the PI1.2 Specification • 541 Volume 5 Typo • 543 Clarification around usage of FV Extended header • 550 Naming conflicts w/ PI SMM	12/16/09
1.1 Errata B	• 363 PI volume 1 errata • 365 UEFI Capsule HOB • 381 PI1.1 Errata on EFI_SMM_SAVE_STATE_IO_INFO • 482 One other naming inconsistency in the PCD PPI declaration • 483 PCD Protocol / PPI function name synchronization..... • 496 Boot mode description • 497 Status Code additions • 548 Boot firmware volume clarification • 552 MP services • 553 Update text to PEI • 554 update return code from PEI AllocatePages • 555 Inconsistency in the S3 protocol • 561 Minor update to PCD->SetPointer • 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4) • 581 EFI_HOB_TYPE_LOAD_PEIM ambiguity • 591ACPI Protocol Name collision • 592 More SMM name conflicts • 593 A couple of ISA I/O clarifications • 595 SMM driver entry point clarification • 596 Clarify ESAL return codes • 602 SEC->PEI hand-off update • 604 EFI_NOT_SUPPORTED versus EFI_UNSUPPORTED	(2/24/10) 5/27/10

1.1 Errata B	• 628 ACPI SDT protocol errata • 629 Typos in PCD GetSize()	5/27/10

Specification Volumes

The **Platform Initialization Specification** is divided into volumes to enable logical organization, future growth, and printing convenience. The **Platform Initialization Specification** consists of the following volumes:

VOLUME 1: Pre-EFI Initialization Core Interface

VOLUME 2: Driver Execution Environment Core Interface

VOLUME 3: Shared Architectural Elements

VOLUME 4: System Management Mode

VOLUME 5: Standards

Each volume should be viewed in the context of all other volumes, and readers are strongly encouraged to consult the entire specification when researching areas of interest. Additionally, a single-file version of the **Platform Initialization Specification** is available to aid search functions through the entire specification.

Contents

Figures

Tables

1 Platform Initialization Standards Introduction

1.1 Overview

These sections define the core code and services that are required for an implementation of the System Management Bus (SMBus) Host Controller Protocol and System Management Bus (SMBus) PEIM-to-PEIM Interface (PPI).

The SMBus Host Controller Protocol is used by code, typically early chipset drivers, and SMBus bus drivers that are running in the UEFI Boot Services environment to perform data transactions over the SMBus. This specification does the following:

- Describes the basic components of the SMBus Host Controller Protocol
- Provides code definitions for the SMBus Host Controller Protocol and the SMBus-related type definitions that are architecturally required.

The SMBus PPI is used by other Pre-EFI Initialization Modules (PEIMs) to control an SMBus host controller.

This specification does the following:

- Describes the basic components of the PEI SMBus PPI
- Provides code definitions for the PEI SMBus PPI and SMBus-related type definitions that are architecturally required.

1.2 Terms Used in this Document

16-bit PC Card

Legacy cards that follow the *PC Card Standard* and operate in 16-bit mode.

CardBay PC Card

32-bit PC Cards that follow the high-performance serial *PC Card Standard.* After initialization, these devices appear as standard PCI devices.

CardBus bridge

A hardware controller that produces a CardBus bus. A CardBus bus can accept a CardBus PC Card as well as legacy 16-bit PC Cards. CardBus PC Cards appear just like PCI devices to the configuration software.

CardBus PC Card

32-bit PC Cards that follow the *PC Card Standard.*

HB

Host bridge. See PCI host bridge.

HPB

Hot Plug Bus.

HPC

Hot Plug Controller. A generic term that refers to both a PHPC and a CardBus bridge.

HPRT

Hot Plug Resource Table.

incompatible PCI device

A PCI device that does not fully comply with the PCI Specification. Typically, this kind of device has a special requirement for Base Address Register (BAR) allocation. Some devices may want a special resource length or alignment, while others may want fixed I/O or memory locations.

JEITA

Japan Electronics and Information Technology Association.

legacy PHPC

PCI devices that can be identified by their class code but were defined prior to the *PCI Standard Hot-Plug Controller and Subsystem Specification,* revision 1.0. These devices have a base class of 0x6, subclass of 0x4, and programming interface of 0.

MWI

Memory Write and Invalidate. See the *PCI Local Bus Specification,* revision 2.3, for more information.

PC Card

Integrated circuit cards that follow the PC Card Standard. "PC Card" is a generic term that is used to refer to 16-bit PC Cards, 32-bit CardBus PC Cards, and high-performance CardBay PC Cards.

PC Card Standard

Refers to the set of specifications that are produced jointly by the PCMCIA and JEITA. This standard was defined to promote interchangeability among mobile computers.

PCI bus

A generic term used to describe any PCI-like buses, including conventional PCI, PCI-X*, and PCI Express*. From a software standpoint, a PCI bus is collection of up to 32 physical PCI devices that share the same physical PCI bus.

PCI bus driver

Software that creates a handle for every PCI controller in the system and installs both the PCI I/O Protocol and the Device Path Protocol onto that handle. It may optionally perform PCI enumeration if resources have not already been allocated to all the PCI controllers. It also loads and starts any EFI drivers that are found in any PCI option ROMs that were discovered during PCI enumeration.

PCI configuration space

The configuration channel that is defined by PCI to configure PCI devices into the resource domain of the system. Each PCI device must produce a standard set of registers in the form of a PCI configuration header and can optionally produce device-specific registers. The registers are addressed via Type 0 or Type 1 PCI configuration cycles as described by the *PCI Specification.* The PCI configuration space can be shared across multiple PCI buses. On Intel® architecture-based systems, the PCI configuration space is accessed via I/O ports 0xCF8 and 0xCFC. The PCI Express configuration space is accessed via a memory-mapped aperture.

PCI controller

A hardware components that is discovered by a PCI bus driver and is managed by a PCI device driver. This document uses the terms "PCI function" and "PCI controller" equivalently.

PCI device

A collection of up to 8 PCI functions that share the same PCI configuration space. A PCI device is physically connected to a PCI bus.

PCI enumeration

The process of assigning resources to all the PCI controllers on a given PCI host bridge. This process includes the following:

- Assigning PCI bus numbers and PCI interrupts
- Allocating PCI I/O resources, PCI memory resources, and PCI prefetchable memory resources
- Setting miscellaneous PCI DMA values

Typically, PCI enumeration is to be performed only once during the boot process.

PCI function

A controller that provides some type of I/O services. It consumes some combination of PCI I/O, PCI memory, and PCI prefetchable memory regions and the PCI configuration space. The PCI function is the basic unit of configuration for PCI.

PCI host bridge

The software abstraction that produces one or more PCI root bridges. All the PCI buses that are produced by a host bus controller are part of the same coherency domain. A PCI host bridge is an abstraction and may be made up of multiple hardware devices. Most systems can be modeled as having one PCI host bridge. This software abstraction is necessary while dealing with PCI resource allocation because resources that are assigned to one PCI root bridge depend on another and all the "related" PCI root bridges must be considered together during resource allocation.

PCI root bridge

A PCI root bridge that produces a root PCI bus. It bridges a root PCI bus and a bus that is not a PCI bus (e.g., processor local bus, InfiniBand* fabric). A PCI host bridge may have one or more root PCI bridges. Configurations of a root PCI bridge within a host bridge can have dependencies upon other root PCI bridges within the same host bridge.

PCI segment

A collection of up to 256 PCI buses that share the same PCI configuration space. A PCI segment is defined in section 6.5.6 of the *ACPI 2.0 Specification* (also *ACPI 3.0*) as the _SEG object. If a system supports only a single PCI segment, the PCI segment number is defined to be zero.The existence of PCI segments enables the construction of systems with greater than 256 PCI buses.

PEC

Packet Error Code. It is similar to a checksum data of the data coming across the SMBus wire.

PCI-to-CardBus bridges

A PCI device that produces a CardBus bus. The PCI-to-CardBus bridge has a PEI

Pre-EFI Initialization.

PEIM

Pre-EFI Initialization Module.

greater than 256 PCI buses.

PERR

Parity Error.

type 2 PCI configuration header and has a class code of 0x070600.

PHPC

PCI Hot Plug* Controller. A hardware component that controls the power to one or more conventional PCI slots or PCI Express slots.

PPI

PEIM-to-PEIM Interface.

RB

Root bridge. See PCI root bridge.

resource padding

Also known as resource overallocation. System resources are said to be overallocated if more resources are allocated to a PCI bus than are required. Resource padding allows a limited number of add-in cards to be hot added to a PCI bus without disturbing allocation to the rest of the buses.

root HPC

Root Hot Plug Controller. An HPC that gets reset only when the entire system is reset. Such HPCs can depend upon the system firmware to initialize them because system firmware is guaranteed to run after a system reset. An HPC that is embedded in the PCI root bridge is considered a root HPC bridge. Some platform chipsets include special-purpose PCI-to-PCI bridges. They appear like a PCI-to-PCI bridge to the configuration software, but their primary bus interface is not a PCI bus. Such a component can be considered a root HPC if it is not subordinate to an HPC. Legacy HPCs may be implemented as chipset devices that appear to be behind a special-purpose PCI-to-PCI bridge, but these HPCs are not reset when the secondary

bus reset bit in the parent PCI-to-PCI bridge is set. Such HPCs are considered as root HPCs as well.

An HPC that is a child of a PCI-to-PCI bridge is not a root HPC. Such an HPC can be reset if the secondary bus reset bit in the PCI-to-PCI bridge is set by an operating system. Because the initialization code in the system firmware may not be executed during this case, such an HPC must initialize itself using hardware mechanisms, without any firmware intervention. An HPC that is a child of another HPC is not a root HPC. See section 3.5.1.3 in the *PCI Standard Hot-Plug Controller and Subsystem Specification,* revision 1.0, for details regarding this term.

root PCI bus

A PCI bus that is not a child of another PCI bus. For every root PCI bus, there is an object in the ACPI name space with a Plug and Play (PNP) ID of "PNP0A03." Typical desktop systems include only one root PCI bus.

SERR

System error.

SHPC

Standard (PCI) Hot Plug Controller. A PCI Hot Plug controller that conforms to the *PCI Standard Hot-Plug Controller and Subsystem Specification,* revision 1.0. This specification is published by the PCI Special Interest Group (PCI-SIG). An SHPC can either be embedded in a PCI root bridge or a PCI-to-PCI bridge.coherency domain

The address resources of a system as seen by a processor. It consists of both system memory and I/O space.

SMBus

System Management Bus.

SMBus host controller

Provides a mechanism for the processor to initiate communications with SMBus slave devices. This controller can be connected to a main I/O bus such as PCI.

SMBus master device

Any device that initiates SMBus transactions and drives the clock.

SMBus PPI

A software interface that provides a method to control an SMBus host controller and access the data of the SMBus slave devices that are attached to it.

SMBus slave device

The target of an SMBus transaction, which is driven by some master.

UDID

Unique Device Identifier. A 128-bit value that a device uses during the Address Resolution Protocol (ARP) process to uniquely identify itself.

1.3 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.3.1 Data Structure Descriptions

Supported processors are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest address. Some supported processors may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

In some memory layout descriptions, certain fields are marked *reserved*. Software must initialize such fields to zero and ignore them when read. On an update operation, software must preserve any reserved field.

The data structures described in this document generally have the following format:

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

Prototype: A “C-style” type declaration for the data structure.

Parameters: A brief description of each field in the data structure prototype.

Description: A description of the functionality provided by the data structure, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this data structure.

1.3.2 Protocol Descriptions

The protocols described in this document generally have the following format:

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

GUID: The 128-bit Globally Unique Identifier (GUID) for the protocol interface.

Protocol Interface Structure: A “C-style” data structure definition containing the procedures and data fields produced by this protocol interface.

Parameters: A brief description of each field in the protocol interface structure.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used in the protocol interface structure or any of its procedures.

1.3.3 Procedure Descriptions

The procedures described in this document generally have the following format:

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

Prototype: A "C-style" procedure header defining the calling sequence.

Parameters: A brief description of each field in the procedure prototype.

Description: A description of the functionality provided by the interface, including any limitations and caveats of which the caller should be aware.

Related Definitions: The type declarations and constants that are used only by this procedure.

Status Codes Returned: A description of any codes returned by the interface. The procedure is required to implement any status codes listed in this table. Additional error codes may be returned, but they will not be tested by standard compliance tests, and any software that uses the procedure cannot depend on any of the extended error codes that an implementation may provide.

1.3.4 Pseudo-Code Conventions

Pseudo code is presented to describe algorithms in a more concise form. None of the algorithms in this document are intended to be compiled directly. The code is presented at a level corresponding to the surrounding text.

In describing variables, a *list* is an unordered collection of homogeneous objects. A *queue* is an ordered list of homogeneous objects. Unless otherwise noted, the ordering is assumed to be First In First Out (FIFO).

Pseudo code is presented in a C-like format, using C conventions where appropriate. The coding style, particularly the indentation style, is used for readability and does not necessarily comply with an implementation of the *Unified Extensible Firmware Interface Specification* (UEFI 2.0 specification).

1.3.5 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.
Bold	In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.
Italic	In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.
`BOLD Monospace`	Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.
`Bold Monospace`	In the online help version of this specification, words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **`Bold Monospace`** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).
`Plain Monospace`	In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

1.4 Requirements

This document is an architectural specification that is part of the Platform Initialization Architecture (PI Architecture) family of specifications defined and published by the Unified EFI Forum. The primary intent of the PI Architecture is to present an interoperability surface for firmware components that may originate from different providers. As such, the burden to conform to this specification falls both on the producer and the consumer of facilities described as part of the specification.

In general, it is incumbent on the producer implementation to ensure that any facility that a conforming consumer firmware component might attempt to use is present in the implementation. Equally, it is incumbent on a developer of a firmware component to ensure that its implementation relies only on facilities that are defined as part of the PI Architecture. Maximum interoperability is assured when collections of conforming components are designed to use only the required facilities defined in the PI Architecture family of specifications.

As this document is an architectural specification, care has been taken to specify architecture in ways that allow maximum flexibility in implementation for both producer and consumer. However, there are certain requirements on which elements of this specification must be implemented to ensure a consistent and predictable environment for the operation of code designed to work with the architectural interfaces described here.

For the purposes of describing these requirements, the specification includes facilities that are required, such as interfaces and data structures, as well as facilities that are marked as optional.

In general, for an implementation to be conformant with this specification, the implementation must include functional elements that match in all respects the complete description of the required facility descriptions presented as part of the specification. Any part of the specification that is not explicitly marked as “optional” is considered a required facility.

Where parts of the specification are marked as “optional,” an implementation may choose to provide matching elements or leave them out. If an element is provided by an implementation for a facility, then it must match in all respects the corresponding complete description.

In practical terms, this means that for any facility covered in the specification, any instance of an implementation may only claim to conform if it follows the normative descriptions completely and exactly. This does not preclude an implementation that provides additional functionality, over and above that described in the specification. Furthermore, it does not preclude an implementation from leaving out facilities that are marked as optional in the specification.

By corollary, modular components of firmware designed to function within an implementation that conforms to the PI Architecture are conformant only if they depend only on facilities described in this and related PI Architecture specifications. In other words, any modular component that is free of any external dependency that falls outside of the scope of the PI Architecture specifications is conformant. A modular component is not conformant if it relies for correct and complete operation upon a reference to an interface or data structure that is neither part of its own image nor described in any PI Architecture specifications.

It is possible to make a partial implementation of the specification where some of the required facilities are not present. Such an implementation is non-conforming, and other firmware components that are themselves conforming might not function correctly with it. Correct operation of non-conforming implementations is explicitly out of scope for the PI Architecture and this specification.

2
SMBus Host Controller Design Discussion

2.1 SMBus Host Controller Overview

These section describe the System Management Bus (SMBus) Host Controller Protocol. This protocol provides an I/O abstraction for an SMBus host controller. An SMBus host controller is a hardware component that interfaces to an SMBus. It moves data between system memory and devices on the SMBus by processing data structures and generating transactions on the SMBus. The following use this protocol:

- An SMBus bus driver to perform all data transactions over the SMBus
- Early chipset drivers that need to manage devices that are required early in the Driver Execution Environment (DXE) phase, before the Boot Device Selection (BDS) phase

This protocol should be used only by drivers that require direct access to the SMBus.

Considerable discussion has been done to understand the usage model of the UEFI Driver Model in the SMBus. Although, the UEFI Driver Model concepts can be applied to SMBus, only the SMBus Host Controller Protocol was created for now for the following reasons:

- The UEFI Driver Model is designed primarily for boot devices. Boot devices are unlikely to be connected to the SMBus because of SMBus-intrinsic capability. They are slow and not enumerable.
- The current usage model of SMBus is to enable and configure devices early during the boot phase, before BDS.

A DXE driver that publishes this protocol will either support Execute, ArpDevice, GetArpMap, and Notify; alternatively, a driver will support only Execute and return “not supported” for the latter 3 services.

If some of these assumptions become obsolete and require being revisited in the future, this specification is extensible to convert to the UEFI Driver Model.

2.2 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

Industry Specifications

- *System Management Bus (SMBus) Specification,* version 2.0, SBS Implementers Forum, August 3, 2000: http://www.smbus.org
- *PCI Local Bus Specification,* revision 3.0, PCI Special Interest Group.

2.3 SMBus Host Controller Protocol Terms

The following terms are used throughout this document to describe the model for constructing SMBus Host Controller Protocol instances in the DXE environment.

PEC

Packet Error Code. It is similar to a checksum data of the data coming across the SMBus wire.

SMBus

System Management Bus.

SMBus host controller

Provides a mechanism for the processor to initiate communications with SMBus slave devices. This controller can be connected to a main I/O bus such as PCI.

SMBus master device

Any device that initiates SMBus transactions and drives the clock.

SMBus slave device

The target of an SMBus transaction, which is driven by some master.

UDID

Unique Device Identifier. A 128-bit value that a device uses during the Address Resolution Protocol (ARP) process to uniquely identify itself.

2.4 SMBus Host Controller Protocol Overview

The interfaces that are provided in the **EFI_SMBUS_HC_PROTOCOL** are used to manage data transactions on the SMBus. The **EFI_SMBUS_HC_PROTOCOL** is designed to support SMBus 1.0– and 2.0–compliant host controllers.

Each instance of the **EFI_SMBUS_HC_PROTOCOL** corresponds to an SMBus host controller in a platform. To provide support for early drivers that need to communicate on the SMBus, this protocol is available before the Boot Device Selection (BDS) phase. During BDS, this protocol can be attached to the device handle of an SMBus host controller that is created by a device driver for the SMBus host controller's parent bus type. For example, an SMBus controller that is implemented as a PCI device would require a PCI device driver to produce an instance of the **EFI_SMBUS_HC_PROTOCOL**.

See "SMBus Host Controller Protocol" on page 14 for the definition of this protocol.

3
SMBus Host Controller Code Definitions

3.1 Introduction

This section contains the basic definitions of the SMBus Host Controller Protocol. The following protocol is defined in this section:

- **EFI_SMBUS_HC_PROTOCOL**

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- **EFI_SMBUS_NOTIFY_FUNCTION**

3.2 SMBus Host Controller Protocol

EFI_SMBUS_HC_PROTOCOL

Summary

Provides basic SMBus host controller management and basic data transactions over the SMBus.

GUID

```
#define EFI_SMBUS_HC_PROTOCOL_GUID  \
{0xe49d33ed, 0x513d, 0x4634, 0xb6, 0x98, 0x6f, 0x55, 0xaa, 0x75,
0x1c, 0x1b}
```

Protocol Interface Structure

```
typedef struct _EFI_SMBUS_HC_PROTOCOL {
  EFI_SMBUS_HC_EXECUTE_OPERATION     Execute;
  EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE   ArpDevice;
  EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP  GetArpMap;
  EFI_SMBUS_HC_PROTOCOL_NOTIFY       Notify;
} EFI_SMBUS_HC_PROTOCOL;
```

Parameters

Execute

Executes the SMBus operation to an SMBus slave device. See the **Execute()** function description.

ArpDevice

Allows an SMBus 2.0 device(s) to be Address Resolution Protocol (ARP). See the **ArpDevice()** function description.

GetArpMap

Allows a driver to retrieve the address that was allocated by the SMBus host controller during enumeration/ARP. See the **GetArpMap()** function description.

Notify

Allows a driver to register for a callback to the SMBus host controller driver when the bus issues a notification to the bus controller driver. See the **Notify()** function description.

Description

The **EFI_SMBUS_HC_PROTOCOL** provides SMBus host controller management and basic data transactions over SMBus. There is one **EFI_SMBUS_HC_PROTOCOL** instance for each SMBus host controller.

Early chipset drivers can communicate with specific SMBus slave devices by calling this protocol directly. Also, for drivers that are called during the Boot Device Selection (BDS) phase, the device driver that wishes to manage an SMBus bus in a system retrieves the **EFI_SMBUS_HC_PROTOCOL** instance that is associated with the SMBus bus to be managed. A device handle for an SMBus host

controller will minimally contain an **EFI_DEVICE_PATH_PROTOCOL** instance and an **EFI_SMBUS_HC_PROTOCOL** instance.

EFI_SMBUS_HC_PROTOCOL.Execute()

Summary

Executes an SMBus operation to an SMBus controller. Returns when either the command has been executed or an error is encountered in doing the operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_EXECUTE_OPERATION) (
  IN      CONST EFI_SMBUS_HC_PROTOCOL   *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS      SlaveAddress,
  IN      EFI_SMBUS_DEVICE_COMMAND      Command,
  IN      EFI_SMBUS_OPERATION           Operation,
  IN      BOOLEAN                       PecCheck,
  IN OUT  UINTN                         *Length,
  IN OUT  VOID                          *Buffer
  );
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

The SMBus slave address of the device with which to communicate. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*

Command

This command is transmitted by the SMBus host controller to the SMBus slave device and the interpretation is SMBus slave device specific. It can mean the offset to a list of functions inside an SMBus slave device. Not all operations or slave devices support this command's registers. Type **EFI_SMBUS_DEVICE_COMMAND** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*

Operation

Signifies which particular SMBus hardware protocol instance that it will use to execute the SMBus transactions. This SMBus hardware protocol is defined by the *SMBus Specification* and is not related to PI Architecture. Type **EFI_SMBUS_OPERATION** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*

PecCheck

Defines if Packet Error Code (PEC) checking is required for this operation.

Length

Signifies the number of bytes that this operation will do. The maximum number of bytes can be revision specific and operation specific. This field will contain the actual number of bytes that are executed for this operation. Not all operations require this argument.

Buffer

Contains the value of data to execute to the SMBus slave device. Not all operations require this argument. The length of this buffer is identified by *Length*.

Description

The **Execute()** function provides a standard way to execute an operation as defined in the *System Management Bus (SMBus) Specification.* The resulting transaction will be either that the SMBus slave devices accept this transaction or that this function returns with error.

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	Checksum is not correct (PEC is incorrect).
EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_DEVICE_ERROR	The request was not completed because a failure that was reflected in the Host Status Register bit. Device errors are a result of a transaction collision, illegal command field, unclaimed cycle (host initiated), or bus errors (collisions).
EFI_INVALID_PARAMETER	*Operation* is not defined in **EFI_SMBUS_OPERATION**.
EFI_INVALID_PARAMETER	*Length*/*Buffer* is **NULL** for operations except for **EfiSmbusQuickRead** and **EfiSmbusQuickWrite**. *Length* is outside the range of valid values.
EFI_UNSUPPORTED	The SMBus operation or PEC is not supported.
EFI_BUFFER_TOO_SMALL	*Buffer* is not sufficient for this operation.

EFI_SMBUS_HC_PROTOCOL.ArpDevice()

Summary

Sets the SMBus slave device addresses for the device with a given unique ID or enumerates the entire bus.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE) (
  IN      CONST EFI_SMBUS_HC_PROTOCOL    *This,
  IN      BOOLEAN                        ArpAll,
  IN      EFI_SMBUS_UDID                 *SmbusUdid,    OPTIONAL
  IN OUT  EFI_SMBUS_DEVICE_ADDRESS       *SlaveAddress OPTIONAL
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

ArpAll

A Boolean expression that indicates if the host drivers need to enumerate all the devices or enumerate only the device that is identified by *SmbusUdid*. If *ArpAll* is **TRUE**, *SmbusUdid* and *SlaveAddress* are optional. If *ArpAll* is **FALSE**, *ArpDevice* will enumerate *SmbusUdid* and the address will be at *SlaveAddress*.

SmbusUdid

The Unique Device Identifier (UDID) that is associated with this device. Type **EFI_SMBUS_UDID** is defined in **EFI_PEI_SMBUS_PPI.ArpDevice()** in the *Platform Initialization SMBus PPI Specification.*

SlaveAddress

The SMBus slave address that is associated with an SMBus UDID. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*

Description

The **ArpDevice()** function provides a standard way for a device driver to enumerate the entire SMBus or specific devices on the bus.

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	Checksum is not correct (PEC is incorrect).

EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_DEVICE_ERROR	The request was not completed because a failure was reflected in the Host Status Register bit. Device Errors are a result of a transaction collision, illegal command field, unclaimed cycle (host initiated), or bus errors (collisions).
EFI_UNSUPPORTED	*ArpDevice, GetArpMap*, and *Notify* are not implemented by this driver.

EFI_SMBUS_HC_PROTOCOL.GetArpMap()

Summary

Returns a pointer to the Address Resolution Protocol (ARP) map that contains the ID/address pair of the slave devices that were enumerated by the SMBus host controller driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP) (
  IN      CONST EFI_SMBUS_HC_PROTOCOL     *This,
  IN OUT  UINTN                           *Length,
  IN OUT  EFI_SMBUS_DEVICE_MAP            **SmbusDeviceMap
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

Length

Size of the buffer that contains the SMBus device map.

SmbusDeviceMap

The pointer to the device map as enumerated by the SMBus controller driver. Type **EFI_SMBUS_DEVICE_MAP** is defined in **EFI_PEI_SMBUS_PPI.GetArpMap()** in the *Platform Initialization SMBus PPI Specification.*

Description

The **GetArpMap()** function returns the mapping of all the SMBus devices that were enumerated by the SMBus host driver.

Status Codes Returned

EFI_SUCCESS	The SMBus returned the current device map.
EFI_UNSUPPORTED	*ArpDevice*, *GetArpMap*, and *Notify* are not implemented by this driver.

EFI_SMBUS_HC_PROTOCOL.Notify()

Summary

Allows a device driver to register for a callback when the bus driver detects a state that it needs to propagate to other drivers that are registered for a callback.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_HC_PROTOCOL_NOTIFY) (
  IN      CONST EFI_SMBUS_HC_PROTOCOL   *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS      SlaveAddress,
  IN      UINTN                         Data,
  IN      EFI_SMBUS_NOTIFY_FUNCTION     NotifyFunction
);
```

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

Address that the host controller detects as sending a message and calls all the registered function. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*

Data

Data that the host controller detects as sending a message and calls all the registered function.

NotifyFunction

The function to call when the bus driver detects the *SlaveAddress* and *Data* pair. Type **EFI_SMBUS_NOTIFY_FUNCTION** is defined in "Related Definitions" below.

Description

The **Notify()** function registers all the callback functions to allow the bus driver to call these functions when the *SlaveAddress*/*Data* pair happens.

Related Definitions

```
//*******************************************************
// EFI_SMBUS_NOTIFY_FUNCTION
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_SMBUS_NOTIFY_FUNCTION) (
  IN      EFI_SMBUS_DEVICE_ADDRESS            SlaveAddress,
  IN      UINTN                               Data
);
```

SlaveAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *Platform Initialization SMBus PPI Specification.*.

Data

Data of the SMBus host notify command that the caller wants to be called.

Status Codes Returned

EFI_SUCCESS	*NotifyFunction* was registered.
EFI_UNSUPPORTED	*ArpDevice*, *GetArpMap*, and *Notify* are not implemented by this driver.

4
SMBus PPI Design Discussion

4.1 Introduction

These sections describe the System Management Bus (SMBus) PEIM-to-PEIM Interfaces (PPIs). This document provides enough material to implement an SMBus Pre-EFI Initialization Module (PEIM) that can control transactions between an SMBus host controller and its slave devices.

The material that is contained in this document is designed to support communication via the SMBus. These extensions are provided in the form of SMBus-specific protocols. This document provides the information that is required to implement an SMBus PEIM in the Pre-EFI Initialization (PEI) portion of system firmware.

A full understanding of the *Unified Extensible Firmware Interface Specification* (UEFI specification) and the *System Management Bus (SMBus) Specification* is assumed throughout this document. See "Related Information," below, for the URL for the *System Management Bus (SMBus) Specification.*

4.2 Target Audience

This document is intended for the following readers:

- Original equipment manufacturers (OEMs) who will be creating platforms that are intended to boot shrink-wrap operating systems.
- BIOS developers, either those who create general-purpose BIOS and other firmware products, or those who modify these productss.
- Operating system developers who will be creating and/or adapting their shrink-wrap operating system products.

4.3 Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

Industry Specifications

- *System Management Bus (SMBus) Specification,* version 2.0, SBS Implementer's Forum, August 3, 2000:
  http://www.smbus.org
- *PCI Local Bus Specification,* revision 3.0, PCI Special Interest Group.

4.4 PEI SMBus PPI Overview

The PEI SMBus PPI is used by code, typically other PEIMs, that is running in the PEI environment to access data on an SMBus slave device via the SMBus host controller. In particular, functions for managing devices on SMBus buses are defined in this specification.

The interfaces that are provided in the **EFI_PEI_SMBUS2_PPI** are for performing basic operations to an SMBus slave device. The system provides abstracted access to basic system resources to allow a PEIM to have a programmatic method to access these basic system resources. The main goal of this PPI is to provide an abstraction that simplifies the writing of PEIMs for SMBus slave devices. This goal is accomplished by providing a standard interface to the SMBus slave devices that does not require detailed knowledge about the particular hardware implementation or protocols of the SMBus.

Certain implentations of the module may omit Arp capabilities. Specifically, a module will either support Execute, ArpDevice, GetArpMap, and Notify; alternatively, a module will support only Execute and return “not supported” for the latter 3 services.

See “PEI SMBus PPI” on page 26 for the definition of **EFI_PEI_SMBUS2_PPI**. This PPI is produced by each of the SMBus host controllers in the system.

5
SMBus PPI Code Definitions

5.1 Introduction

This section contains the basic definitions for PEIMs and SMBus devices to use during the PEI phase. The following PPI is defined in this section:

- **EFI_PEI_SMBUS2_PPI**

This section also contains the definitions for additional SMBus-related data types and structures that are subordinate to the structures in which they are called. All of the data structures below except for **EFI_PEI_SMBUS_NOTIFY_FUNCTION** can be used in the DXE phase as well. The following types or structures can be found in "Related Definitions" of the parent function definition:

- **EFI_SMBUS_DEVICE_ADDRESS**
- **EFI_SMBUS_DEVICE_COMMAND**
- **EFI_SMBUS_OPERATION**
- **EFI_SMBUS_UDID**
- **EFI_SMBUS_DEVICE_MAP**
- **EFI_PEI_SMBUS_NOTIFY_FUNCTION**

5.2 PEI SMBus PPI

EFI_PEI_SMBUS2_PPI

Summary

Provides the basic I/O interfaces that a PEIM uses to access its SMBus controller and the slave devices attached to it.

GUID

```
#define EFI_PEI_SMBUS2_PPI_GUID  \
{ 0x9ca93627, 0xb65b, 0x4324, \
{ 0xa2, 0x2, 0xc0, 0xb4, 0x61, 0x76, 0x45, 0x43 } }
```

PPI Interface Structure

```
typedef struct _EFI_PEI_SMBUS2_PPI {
  EFI_PEI_SMBUS2_PPI_EXECUTE_OPERATION  Execute;
  EFI_PEI_SMBUS2_PPI_ARP_DEVICE         ArpDevice;
  EFI_PEI_SMBUS2_PPI_GET_ARP_MAP        GetArpMap;
  EFI_PEI_SMBUS2_PPI_NOTIFY             Notify;
  EFI_GUID                              Identifier
} EFI_PEI_SMBUS2_PPI;
```

Parameters

Execute

Executes the SMBus operation to an SMBus slave device. See the **Execute()** function description.

ArpDevice

Allows an SMBus 2.0 device(s) to be Address Resolution Protocol (ARP). See the **ArpDevice()** function description.

GetArpMap

Allows a PEIM to retrieve the address that was allocated by the SMBus host controller during enumeration/ARP. See the **GetArpMap()** function description.

Notify

Allows a PEIM to register for a callback to the SMBus host controller PEIM when the bus issues a notification to the bus controller PEIM. See the **Notify()** function description.

Identifier

Identifier which uniquely identifies this SMBus controller in a system.

Description

The **EFI_PEI_SMBUS2_PPI** provides the basic I/O interfaces that are used to abstract accesses to SMBus host controllers. There is one **EFI_PEI_SMBUS2_PPI** instance for each SMBus host controller in a system. A PEIM that wishes to manage an SMBus slave device in a system will have to retrieve the **EFI_PEI_SMBUS2_PPI** instance that is associated with its SMBus host controller.

EFI_PEI_SMBUS2_PPI.Execute()

Summary

Executes an SMBus operation to an SMBus controller. Returns when either the command has been executed or an error is encountered in doing the operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SMBUS2_PPI_EXECUTE_OPERATION) (
  IN     CONST EFI_PEI_SMBUS2_PPI       *This,
  IN     EFI_SMBUS_DEVICE_ADDRESS       SlaveAddress,
  IN     EFI_SMBUS_DEVICE_COMMAND       Command,
  IN     EFI_SMBUS_OPERATION            Operation,
  IN     BOOLEAN                        PecCheck,
  IN OUT UINTN                          *Length,
  IN OUT VOID                           *Buffer
);
```

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in "Related Definitions" below.

Command

This command is transmitted by the SMBus host controller to the SMBus slave device and the interpretation is SMBus slave device specific. It can mean the offset to a list of functions inside an SMBus slave device. Not all operations or slave devices support this command's registers. Type **EFI_SMBUS_DEVICE_COMMAND** is defined in "Related Definitions" below.

Operation

Signifies which particular SMBus hardware protocol instance that it will use to execute the SMBus transactions. This SMBus hardware protocol is defined by the *System Management Bus (SMBus) Specification* and is not related to UEFI. Type **EFI_SMBUS_OPERATION** is defined in "Related Definitions" below.

PecCheck

Defines if Packet Error Code (PEC) checking is required for this operation.

Length

Signifies the number of bytes that this operation will do. The maximum number of bytes can be revision specific and operation specific. This parameter will contain the

actual number of bytes that are executed for this operation. Not all operations require this argument.

Buffer

Contains the value of data to execute to the SMBus slave device. Not all operations require this argument. The length of this buffer is identified by *Length*.

Description

The **Execute()** function provides a standard way to execute an operation as defined in the *System Management Bus (SMBus) Specification.* The resulting transaction will be either that the SMBus slave devices accept this transaction or that this function returns with error.

Related Definitions

```
//*******************************************************
// EFI_SMBUS_DEVICE_ADDRESS
//*******************************************************
typedef struct _EFI_SMBUS_DEVICE_ADDRESS {
  UINTN   SmbusDeviceAddress:7;
} EFI_SMBUS_DEVICE_ADDRESS;
```

SmbusDeviceAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated.

```
//*******************************************************
// EFI_SMBUS_DEVICE_COMMAND
//*******************************************************
typedef UINTN EFI_SMBUS_DEVICE_COMMAND;


//*******************************************************
// EFI_SMBUS_OPERATION
//*******************************************************
typedef enum _EFI_SMBUS_OPERATION {
  EfiSmbusQuickRead,
  EfiSmbusQuickWrite,
  EfiSmbusReceiveByte,
  EfiSmbusSendByte,
  EfiSmbusReadByte,
  EfiSmbusWriteByte,
  EfiSmbusReadWord,
  EfiSmbusWriteWord,
  EfiSmbusReadBlock,
  EfiSmbusWriteBlock,
  EfiSmbusProcessCall,
  EfiSmbusBWBRProcessCall
```

```
} EFI_SMBUS_OPERATION;
```

See the *System Management Bus (SMBus) Specification* for descriptions of the fields in the above definition.

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	The checksum is not correct (PEC is incorrect).
EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_DEVICE_ERROR	The request was not completed because a failure reflected in the Host Status Register bit. Device errors are a result of a transaction collision, illegal command field, unclaimed cycle (host initiated), or bus errors (collisions).
EFI_INVALID_PARAMETER	*Operation* is not defined in **EFI_SMBUS_OPERATION**.
EFI_INVALID_PARAMETER	*Length*/*Buffer* is **NULL** for operations except for **EfiSmbusQuickRead** and **EfiSmbusQuickWrite**. *Length* is outside the range of valid values.
EFI_UNSUPPORTED	The SMBus operation or PEC is not supported.
EFI_BUFFER_TOO_SMALL	*Buffer* is not sufficient for this operation.

EFI_PEI_SMBUS2_PPI.ArpDevice()

Summary

Sets the SMBus slave device addresses for the device with a given unique ID or enumerates the entire bus.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SMBUS2_PPI_ARP_DEVICE) (
  IN     CONST EFI_PEI_SMBUS2_PPI       *This,
  IN     BOOLEAN                        ArpAll,
  IN     EFI_SMBUS_UDID                 *SmbusUdid,     OPTIONAL
  IN OUT EFI_SMBUS_DEVICE_ADDRESS       *SlaveAddress   OPTIONAL
);
```

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

ArpAll

A Boolean expression that indicates if the host PEIMs need to enumerate all the devices or enumerate only the device that is identified by *SmbusUdid*. If *ArpAll* is **TRUE**, *SmbusUdid* and *SlaveAddress* are optional. If *ArpAll* is **FALSE**, *ArpDevice* will enumerate *SmbusUdid* and the address will be at *SlaveAddress*.

SmbusUdid

The targeted SMBus Unique Device Identifier (UDID). The UDID may not exist for SMBus devices with fixed addresses. Type **EFI_SMBUS_UDID** is defined in "Related Definitions" below.

SlaveAddress

The new SMBus address for the slave device for which the operation is targeted. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS2_PPI.Execute()**.

Description

The **ArpDevice()** function enumerates the entire bus or enumerates a specific device that is identified by *SmbusUdid*.

Related Definitions

```
//*******************************************************
// EFI_SMBUS_UDID
//*******************************************************
typedef struct _EFI_SMBUS_UDID {
  UINT32 VendorSpecificId;
  UINT16 SubsystemDeviceId;
  UINT16 SubsystemVendorId;
  UINT16 Interface;
  UINT16 DeviceId;
  UINT16 VendorId;
  UINT8  VendorRevision;
  UINT8  DeviceCapabilities;
} EFI_SMBUS_UDID;
```

VendorSpecificId

A unique number per device.

SubsystemDeviceId

Identifies a specific interface, implementation, or device. The subsystem ID is defined by the party that is identified by the *SubsystemVendorId* field.

SubsystemVendorId

This field may hold a value that is derived from any of several sources:

- The device manufacturer's ID as assigned by the SBS Implementer's Forum or the PCI SIG.
- The device OEM's ID as assigned by the SBS Implementer's Forum or the PCI SIG.
- A value that, in combination with the *SubsystemDeviceId*, can be used to identify an organization or industry group that has defined a particular common device interface specification.

Interface

Identifies the protocol layer interfaces that are supported over the SMBus connection by the device. For example, Alert Standard Format (ASF) and IPMI.

DeviceId

The device ID as assigned by the device manufacturer (identified by the *VendorId* field).

VendorId

The device manufacturer's ID as assigned by the SBS Implementer's Forum or the PCI SIG.

VendorRevision

UDID version number and a silicon revision identification.

DeviceCapabilities

Describes the device's capabilities.

Status Codes Returned

EFI_SUCCESS	The SMBus slave device address was set.
EFI_INVALID_PARAMETER	*SlaveAddress* is **NULL**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.
EFI_TIMEOUT	The SMBus slave device did not respond.
EFI_DEVICE_ERROR	The request was not completed because the transaction failed. Device errors are a result of a transaction collision, illegal command field, or unclaimed cycle (host initiated).
EFI_UNSUPPORTED	*ArpDevice*, *GetArpMap*, and *Notify* are not implemented by this PEIM.

EFI_PEI_SMBUS2_PPI.GetArpMap()

Summary

Returns a pointer to the Address Resolution Protocol (ARP) map that contains the ID/address pair of the slave devices that were enumerated by the SMBus host controller PEIM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SMBUS2_PPI_GET_ARP_MAP) (
  IN     CONST EFI_PEI_SMBUS2_PPI      *This,
  IN OUT UINTN                         *Length,
  IN OUT EFI_SMBUS_DEVICE_MAP          **SmbusDeviceMap
);
```

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

Length

Size of the buffer that contains the SMBus device map.

SmbusDeviceMap

The pointer to the device map as enumerated by the SMBus controller PEIM. Type **EFI_SMBUS_DEVICE_MAP** is defined in "Related Definitions" below.

Description

The **GetArpMap()** function returns the mapping of all the SMBus devices that are enumerated by the SMBus host PEIM.

Related Definitions

```
//*******************************************************
// EFI_SMBUS_DEVICE_MAP
//*******************************************************
typedef struct _EFI_SMBUS_DEVICE_MAP {
  EFI_SMBUS_DEVICE_ADDRESS  SmbusDeviceAddress;
  EFI_SMBUS_UDID            SmbusDeviceUdid;
} EFI_SMBUS_DEVICE_MAP;
```

SmbusDeviceAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS2_PPI.Execute()**.

SmbusDeviceUdid

The SMBUS Unique Device Identifier (UDID) as defined in **EFI_SMBUS_UDID**. Type **EFI_SMBUS_UDID** is defined in **EFI_PEI_SMBUS2_PPI.ArpDevice()**.

Status Codes Returned

EFI_SUCCESS	The device map was returned correctly in the buffer.
EFI_UNSUPPORTED	*ArpDevice*, *GetArpMap*, and *Notify* are not implemented by this PEIM.

EFI_PEI_SMBUS2_PPI.Notify()

Summary

Allows a PEIM to register for a callback when the PEIM detects a state that it needs to propagate to other PEIMs that are registered for a callback.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SMBUS2_PPI_NOTIFY) (
  IN     CONST EFI_PEI_SMBUS2_PPI        *This,
  IN     EFI_SMBUS_DEVICE_ADDRESS        SlaveAddress,
  IN     UINTN                           Data,
  IN     EFI_PEI_SMBUS_NOTIFY2_FUNCTION  NotifyFunction
);
```

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

Address that the host controller detects as sending a message and calls all the registered functions. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS2_PPI.Execute()**.

Data

Data that the host controller detects as sending a message and calls all the registered functions.

NotifyFunction

The function to call when the PEIM detects the *SlaveAddress* and *Data* pair. Type **EFI_PEI_SMBUS_NOTIFY2_FUNCTION** is defined in "Related Definitions" below.

Description

The **Notify()** function registers all the callback functions to allow the PEIM to call these functions when the *SlaveAddress*/*Data* pair happens.

Related Definitions

```
//*******************************************************
// EFI_PEI_SMBUS_NOTIFY2_FUNCTION
//*******************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SMBUS_NOTIFY2_FUNCTION) (
  IN       CONST_EFI_PEI_SMBUS2_PPI     *SmbusPpi,
  IN       EFI_SMBUS_DEVICE_ADDRESS     SlaveAddress,
  IN       UINTN                        Data
);
```

SmbusPpi

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

The SMBUS hardware address to which the SMBUS device is preassigned or allocated. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS2_PPI.Execute()**.

Data

Data of the SMBus host notify command that the caller wants to be called.

Status Codes Returned

EFI_SUCCESS	*NotifyFunction* has been registered.
EFI_UNSUPPORTED	*ArpDevice*, *GetArpMap*, and *Notify* are not implemented by this PEIM.

6
SMBIOS Protocol

EFI_SMBIOS_PROTOCOL

Summary

Allows consumers to log SMBIOS data records, and enables the producer to create the SMBIOS tables for a platform.

GUID

```
#define EFI_SMBIOS_PROTOCOL_GUID \
 { 0x3583ff6, 0xcb36, 0x4940, \
 { 0x94, 0x7e, 0xb9, 0xb3, 0x9f, 0x4a, 0xfa, 0xf7 } }
```

Protocol Interface Structure

```
typedef struct _EFI_SMBIOS_PROTOCOL {
  EFI_SMBIOS_ADD           Add;
  EFI_SMBIOS_UPDATE_STRING UpdateString;
  EFI_SMBIOS_REMOVE        Remove;
  EFI_SMBIOS_GET_NEXT      GetNext;
  UINT8                    MajorVersion;
  UINT8                    MinorVersion;
} EFI_SMBIOS_PROTOCOL;
```

Member Description

Add

Add an SMBIOS record including the formatted area and the optional strings that follow the formatted area.

UpdateString

Update a string in the SMBIOS record.

Remove

Remove an SMBIOS record.

GetNext

Discover all SMBIOS records.

MajorVersion

The major revision of the SMBIOS specification supported.

MinorVersion

The minor revision of the SMBIOS specification supported.

Description

This protocol provides an interface to add, remove or discover SMBIOS records. The driver which produces this protocol is responsible for creating the SMBIOS data tables and installing the pointer to the tables in the EFI System Configuration Table.

The caller is responsible for only adding SMBIOS records that are valid for the SMBIOS *MajorVersion* and *MinorVersion*. When an enumerated SMBIOS field's values are controlled by the DMTF, new values can be used as soon as they are defined by the DMTF without requiring an update to *MajorVersion* and *MinorVersion*.

The SMBIOS protocol can only be called a **TPL < TPL_NOTIFY**.

EFI_SMBIOS_PROTOCOL.Add()

Summary

Add an SMBIOS record.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_ADD) (
  IN     CONST EFI_SMBIOS_PROTOCOL *This,
  IN     EFI_HANDLE                ProducerHandle, OPTIONAL
  IN OUT EFI_SMBIOS_HANDLE         *SmbiosHandle,
  IN     EFI_SMBIOS_TABLE_HEADER   *Record
  );
```

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

ProducerHandle

The handle of the controller or driver associated with the SMBIOS information. **NULL** means no handle.

SmbiosHandle

On entry, if non-zero, the handle of the SMBIOS record. If zero, then a unique handle will be assigned to the SMBIOS record. If the SMBIOS handle is already in use **EFI_ALREADY_STARTED** is returned and the SMBIOS record is not updated.

Record

The data for the fixed portion of the SMBIOS record. The format of the record is determined by **EFI_SMBIOS_TABLE_HEADER.***Type*. The size of the formatted area is defined by **EFI_SMBIOS_TABLE_HEADER.***Length* and either followed by a double-null (0x0000) or a set of null terminated strings and a null.

Description

This function allows any agent to add SMBIOS records. The caller is responsible for ensuring *Record* is formatted in a way that matches the version of the SMBIOS specification as defined in the *MajorRevision* and *MinorRevision* fields of the **EFI_SMBIOS_PROTOCOL**.

Record must follow the SMBIOS structure evolution and usage guidelines in the SMBIOS specification. Record starts with the formatted area of the SMBIOS structure and the length is defined by **EFI_SMBIOS_TABLE_HEADER.***Length*. Each SMBIOS structure is terminated by a double-null (0x0000), either directly following the formatted area (if no strings are present) or directly following the last string. The number of optional strings is not defined by the formatted area, but is fixed by the call to *Add()*. A string can be a place holder, but it must not be a **NULL** string as two **NULL** strings look like the double-null that terminates the structure.

Related Definitions

```
typedef UINT8  EFI_SMBIOS_TYPE;
typedef UINT16 EFI_SMBIOS_HANDLE;

typedef struct {
  EFI_SMBIOS_TYPE   Type;
  UINT8             Length;
  EFI_SMBIOS_HANDLE Handle;
} EFI_SMBIOS_TABLE_HEADER;

#define EFI_SMBIOS_TYPE_BIOS_INFORMATION                0
#define EFI_SMBIOS_TYPE_SYSTEM_INFORMATION              1
#define EFI_SMBIOS_TYPE_BASEBOARD_INFORMATION           2
#define EFI_SMBIOS_TYPE_SYSTEM_ENCLOSURE                3
#define EFI_SMBIOS_TYPE_PROCESSOR_INFORMATION           4
#define EFI_SMBIOS_TYPE_MEMORY_CONTROLLER_INFORMATION   5
#define EFI_SMBIOS_TYPE_MEMORY_MODULE_INFORMATON        6
#define EFI_SMBIOS_TYPE_CACHE_INFORMATION               7
#define EFI_SMBIOS_TYPE_PORT_CONNECTOR_INFORMATION      8
#define EFI_SMBIOS_TYPE_SYSTEM_SLOTS                    9
#define EFI_SMBIOS_TYPE_ONBOARD_DEVICE_INFORMATION      10
#define EFI_SMBIOS_TYPE_OEM_STRINGS                     11
#define EFI_SMBIOS_TYPE_SYSTEM_CONFIGURATION_OPTIONS    12
#define EFI_SMBIOS_TYPE_BIOS_LANGUAGE_INFORMATION       13
#define EFI_SMBIOS_TYPE_GROUP_ASSOCIATIONS              14
#define EFI_SMBIOS_TYPE_SYSTEM_EVENT_LOG                15
#define EFI_SMBIOS_TYPE_PHYSICAL_MEMORY_ARRAY           16
#define EFI_SMBIOS_TYPE_MEMORY_DEVICE                   17
#define EFI_SMBIOS_TYPE_32BIT_MEMORY_ERROR_INFORMATION  18
#define EFI_SMBIOS_TYPE_MEMORY_ARRAY_MAPPED_ADDRESS     19
#define EFI_SMBIOS_TYPE_MEMORY_DEVICE_MAPPED_ADDRESS    20
#define EFI_SMBIOS_TYPE_BUILT_IN_POINTING_DEVICE        21
#define EFI_SMBIOS_TYPE_PORTABLE_BATTERY                22
#define EFI_SMBIOS_TYPE_SYSTEM_RESET                    23
#define EFI_SMBIOS_TYPE_HARDWARE_SECURITY               24
#define EFI_SMBIOS_TYPE_SYSTEM_POWER_CONTROLS           25
#define EFI_SMBIOS_TYPE_VOLTAGE_PROBE                   26
#define EFI_SMBIOS_TYPE_COOLING_DEVICE                  27
#define EFI_SMBIOS_TYPE_TEMPERATURE_PROBE               28
#define EFI_SMBIOS_TYPE_ELECTRICAL_CURRENT_PROBE        29
#define EFI_SMBIOS_TYPE_OUT_OF_BAND_REMOTE_ACCESS       30
#define EFI_SMBIOS_TYPE_BOOT_INTEGRITY_SERVICE          31
#define EFI_SMBIOS_TYPE_SYSTEM_BOOT_INFORMATION         32
#define EFI_SMBIOS_TYPE_64BIT_MEMORY_ERROR_INFORMATION  33
#define EFI_SMBIOS_TYPE_MANAGEMENT_DEVICE               34
#define EFI_SMBIOS_TYPE_MANAGEMENT_DEVICE_COMPONENT     35
#define EFI_SMBIOS_TYPE_MANAGEMENT_DEVICE_THRESHOLD_DATA 36
```

```
#define EFI_SMBIOS_TYPE_MEMORY_CHANNEL                  37
#define EFI_SMBIOS_TYPE_IPMI_DEVICE_INFORMATION         38
#define EFI_SMBIOS_TYPE_SYSTEM_POWER_SUPPLY             39
#define EFI_SMBIOS_TYPE_INACTIVE                        126
#define EFI_SMBIOS_TYPE_END_OF_TABLE                    127
#define EFI_SMBIOS_OEM_BEGIN                            128
#define EFI_SMBIOS_OEM_END                              255

typedef UINT8 EFI_SMBIOS_STRING;
```

Status Codes Returned

EFI_SUCCESS	*Record* was added.
EFI_OUT_OF_RESOURCES	*Record* was not added.
EFI_ALREADY_STARTED	The *SmbiosHandle* passed in was already in use.

EFI_SMBIOS_PROTOCOL.UpdateString()

Summary

Update the string associated with an existing SMBIOS record.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_UPDATE_STRING) (
  IN CONST EFI_SMBIOS_PROTOCOL   *This,
  IN EFI_SMBIOS_HANDLE           *SmbiosHandle,
  IN UINTN                       *StringNumber,
  IN CHAR8                       *String
  );
```

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

SMBIOS Handle of structure that will have its string updated.

StringNumber

The non-zero string number of the string to update

String

Update the *StringNumber* string with *String*.

Description

This function allows the update of specific SMBIOS strings. The number of valid strings for any SMBIOS record is defined by how many strings were present when *Add()* was called.

Status Codes Returned

EFI_SUCCESS	*SmbiosHandle* had its *StringNumber* *String* updated.
EFI_INVALID_PARAMETER	*SmbiosHandle* does not exist.
EFI_UNSUPPORTED	String was not added since it's longer than 64 significant characters.
EFI_NOT_FOUND	The *StringNumber*.is not valid for this SMBIOS record.

EFI_SMBIOS_PROTOCOL.Remove()

Summary

Remove an SMBIOS record.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_REMOVE) (
  IN CONST EFI_SMBIOS_PROTOCOL *This,
  IN       EFI_SMBIOS_PROTOCOL SmbiosHandle
  );
```

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

The handle of the SMBIOS record to remove.

Description

This function removes an SMBIOS record using the handle specified by SmbiosHandle.

Status Codes Returned

EFI_SUCCESS	SMBIOS record was removed.
EFI_INVALID_PARAMETER	*SmbiosHandle* does not specify a valid SMBIOS record.

EFI_SMBIOS_PROTOCOL.GetNext()

Summary

Allow the caller to discover all or some of the SMBIOS records.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMBIOS_GET_NEXT) (
  IN CONST EFI_SMBIOS_PROTOCOL         *This,
  IN OUT   EFI_SMBIOS_HANDLE           *SmbiosHandle,
  IN       EFI_SMBIOS_TYPE             *Type,         OPTIONAL
  OUT      EFI_SMBIOS_TABLE_HEADER     **Record,
  OUT      EFI_HANDLE                  *ProducerHandle  OPTIONAL
  );
```

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

On entry, points to the previous handle of the SMBIOS record. On exit, points to the next SMBIOS record handle. If it is zero on entry, then the first SMBIOS record handle will be returned. If it returns zero on exit, then there are no more SMBIOS records.

Type

On entry, it points to the type of the next SMBIOS record to return. If NULL, it indicates that the next record of any type will be returned. *Type* is not modified by the this function.

Record

On exit, points to a pointer to the the SMBIOS Record consisting of the formatted area followed by the unformatted area. The unformatted area optionally contains text strings.

ProducerHandle

On exit, points to the *ProducerHandle* registered by *Add()*. If no *ProducerHandle* was passed into *Add()* **NULL** is returned. If a **NULL** pointer is passed in no data will be returned.

Description

This function allows all of the SMBIOS records to be discovered. It's possible to find only the SMBIOS records that match the optional *Type* argument.

Status Codes Returned.

EFI_SUCCESS	.SMBIOS record information was successfully returned in *Record*. *SmbiosHandle* is the handle of the current SMBIOS record
EFI_NOT_FOUND	The SMBIOS record with *SmbiosHandle* was the last available record.

7
S3 Resume

7.1 S3 Overview

This specification defines the core code and services that are required for an implementation of the S3 resume boot path in the PI. The S3 resume boot path is a special boot path that causes the system to take actions different from those in the normal boot path. In this special path, the system derives pre-saved data about the platform's configuration from persistent storage and configures the platform before jumping to the operating system's waking vector.

This specification does the following:

- Describes the basic components of the S3 resume boot path, how it relates to a normal boot path, and how it interacts with other PI phases and code
- Provides code definitions for the S3-related protocols and PPIs that are architecturally required by the *PI Specification.*

7.2 Goals

The PI S3 resume boot path design has the following goals:

Extensibility:

The PI S3 resume boot path should easily adapt to different platforms by replacing only a few platform-specific modules.

High performance:

The performance of the PI S3 resume boot path is highly visible to end users and must be optimized.

7.3 Requirements

All aspects of this PI S3 resume boot path design must comply with the *Advanced Configuration and Power Interface Specification* (hereafter referred to as the "ACPI specification"), revision 2.0.

The design should emphasize size efficiency, code reuse and maintainability.

7.4 Assumptions

7.4.1 Multiple Phases of Platform Initialization

The PI Architecture consists of multiple phases. For example:

- Pre-EFI Initialization (PEI)
- Driver Execution Environment (DXE)

- SMM (System Management Mode)

The PEI phase is responsible for initializing enough of the platform's resources to enable the execution of the DXE phase, which is where the majority of platform configuration is performed by different DXE drivers.

Initialization that is done in PEI is not necessarily preserved in DXE. In other words, a DXE driver can override the configuration settings that were derived from PEI. In light of this fact, the preboot platform state that the S3 resume boot path needs to restore is the DXE snapshot of the platform state, rather than the PEI snapshot of the platform state.

7.4.2 Process of Platform Initialization

Platform initialization can be viewed as a flow of the following:

- I/O operations
- Memory operations
- Accessing the PCI configuration space
- A collection of platform-specific actions that can be abstracted by Pre-EFI Initialization Module (PEIM) PEIM-to-PEIM Interfaces (PPIs)

The process of restoring hardware settings in different platforms involves different actions or even different instruction sets. These differences, however, can be abstracted behind PEIM PPIs.

7.5 Restoring the Platform

The goal of the S3 resume process is to restore the platform to its preboot configuration. However, it is impossible to restore the platform in only one step, without going through all the PI initialization phases, because the PI Architecture cannot have a priori knowledge of the following:

- Preboot configuration that is introduced by various PEIMs
- Drivers provided by different vendors

As a result, the PI Architecture still needs to restore the platform in a phased fashion as it does in a normal boot path. The figure below shows the phases in an S3 resume boot path. See the following subsections for details of each phase.

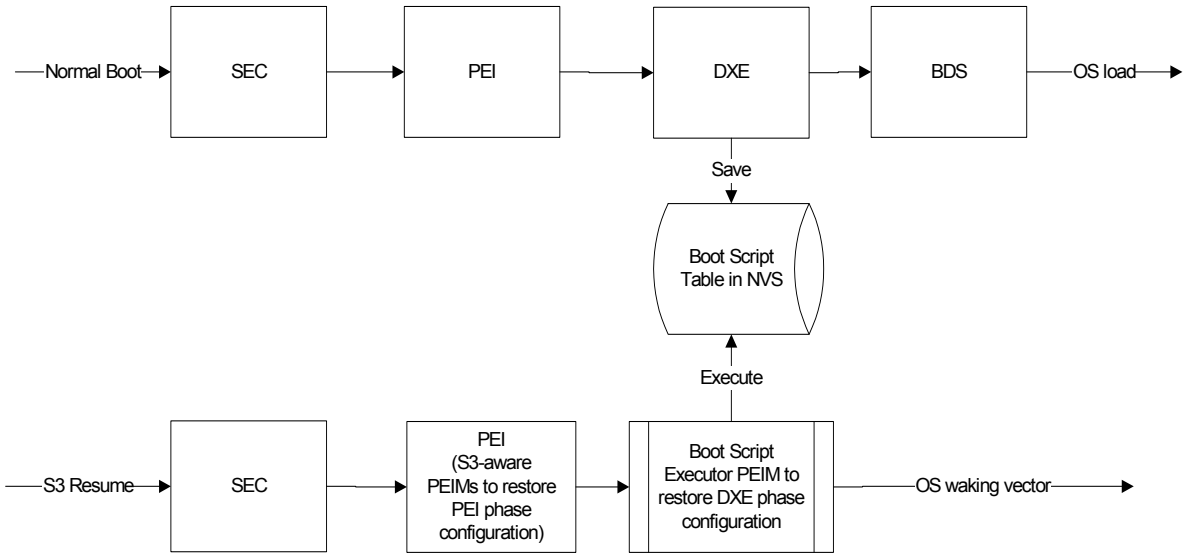

Figure 1. Framework S3 Resume Boot Path

7.5.1 Phases in the S3 Resume Boot Path

7.5.1.1 SEC and the S3 Resume Boot Path

The Security (SEC) phase is the first architectural phase in the PI Architecture. It builds the root of trust for the entire system. As such, the SEC phase remains intact in the S3 resume boot path.

7.5.1.2 PEI

7.5.1.2.1 PEI and the S3 Resume Boot Path

The PEI phase initializes the platform with the minimum configuration needed to enable the execution of the DXE phase. During the S3 resume boot path, the Framework still needs to restore the PEI portion of configuration.

Each PEIM is "boot path aware" in that the PEIM can call the appropriate PEI service to find out what the current boot path is. This awareness enables the platform to restore more efficiently because the same PEIM can save the configuration during a normal boot path and take advantage of that configuration in the S3 resume boot path. The figure below shows how the PEI phase works in a normal boot path and in an S3 resume boot path.

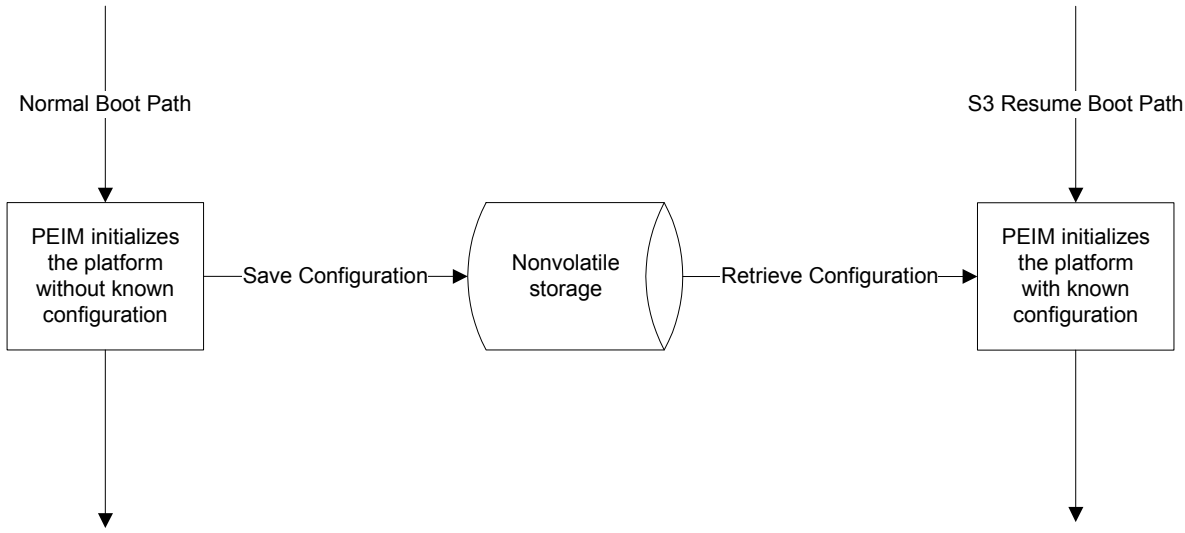

Figure 2. PEI Phase in S3 Resume Boot Path

7.5.1.2.2 Saving Configuration Data in PEI

There are different ways to save configuration data, such as the firmware volume variable, for the PEI phase in nonvolatile storage (NVS). One way is to save the data directly in the PEI phase. However, if the PEI phase does not implement the capability to write to a firmware volume, a PEIM can choose to pass the configuration data to the DXE phase using a Hand-Off Block (HOB). The PEIM's DXE counterpart or another appropriate DXE component can then save the configuration data. The figure below illustrates this mechanism to save the configuration data. See the PI Specification for more details on HOBs.

To achieve higher performance, it is recommended to implement the latter mechanism because code running in the PEI phase is more time consuming than code running in the DXE phase. Note that the way to save the configuration data during the PEI phase is outside the scope of this document.

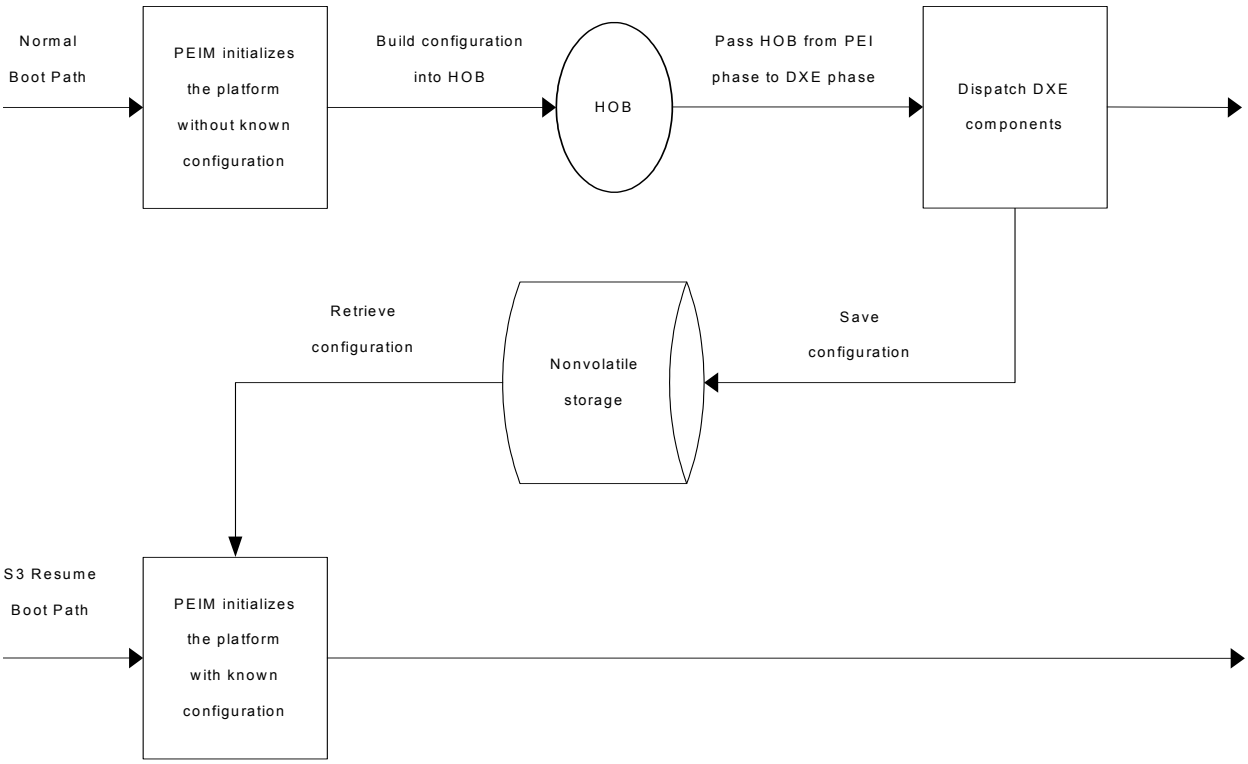

Figure 3. Configuration Save for PEI Phase

7.5.1.3 DXE

7.5.1.3.1 DXE and the S3 Resume Boot Path

In the DXE phase during a normal boot path, various DXE drivers collectively bring the platform to the preboot state. However, bringing DXE into the S3 resume boot path and making a DXE driver boot-path aware is very risky for the following reasons:

- The DXE phase hosts numerous services, which makes it rather large.
- Loading DXE from flash is very time consuming.

Even if DXE could be relocated into NVS during a normal boot, the large amount of memory that DXE consumes and the complexity of executing the DXE phase do not justify doing so.

Instead, the Framework provides a boot script that lets the S3 resume boot path avoid the DXE phase altogether, which helps to maximize optimum performance. During a normal boot, DXE drivers record the platform's configuration in the boot script, which is saved in NVS. During the S3 resume boot path, a boot script engine executes the script, thereby restoring the configuration.

The ACPI specification only requires the BIOS to restore chipset and processor configuration. The chipset configuration can be viewed as a series of memory, I/O, and PCI configuration operations, which DXE drivers record in the Framework boot script. During an S3 resume, a boot script engine executes the boot script to restore the chipset settings. Processor configuration involves the following:

- "Basic setup for System Management Mode (SMM)
- "Microcode updates
- "Processor-specific initialization
- "Processor cache setting

DXE drivers register a pointer to a function in the boot script to restore processor configuration. During the S3 resume boot path, the boot script engine can jump to execute the registered code to restore all processor-related configurations.

7.5.1.3.2 S3 Resume PPI and DXE IPL PPI

The DXE Initial Program Load (IPL) PPI is architecturally the last PPI that is executed in the PEI phase. It is also made aware of the exact boot path that the Framework is currently using. It discovers the boot mode and initiates the process of restoring the pre-boot platform state and jumping to the operating system (OS) waking vector. The DXE phase is not entered, as it would be during a normal boot.

When resuming from S3, the DXE IPL PEIM will transfer control to the S3 Resume PPI, which is responsible for restoring the platform configuration and jumping to the waking vector.

7.5.1.4 SMM

The **EFI_S3_SMM_SAVE_STATE_PROTOCOL** publishes the PI SMM boot script abstractions

In an S3 resume boot path the data stored via this protocol is replayed in the order it was stored.

The order of replay is the order either of the S3 Save State Protocol or S3 SMM Save State Protocol *Write()* functions were called during the boot process. **Insert()**, **Label()**, and **Compare()** operations are ordered relative other S3 SMM Save State Protocol **Write()** operations and the order relative to S3 State Save **Write()** operations is not defined. Due to these ordering restrictions it is recommended that the S3 State Save Protocol be used during the DXE phase when every possible.

The **EFI_S3_SMM_SAVE_STATE_PROTOCOL** can be called at runtime and **EFI_OUT_OF_RESOURCES** may be returned from a runtime call. It is the responsibility of the platform to ensure enough memory resource exists to save the system state. It is recommended that runtime calls be minimized by the caller.3

7.6 PEI Boot Script Executer PPI

EFI_PEI_S3_RESUME2_PPI

Summary

This PPI produces functions to interpret and execute the PI boot script table.

GUID

```
#define EFI_PEI_S3_RESUME2_PPI_GUID  \
  {0x6d582dbc, 0xdb85, 0x4514, \
  0x8f, 0xcc, 0x5a, 0xdf, 0x62, 0x27, 0xb1,0x47}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_S3_RESUME2_PPI {
  EFI_PEI_S3_RESUME_PPI_RESTORE_CONFIG2 S3RestoreConfig2;
} EFI_PEI_S3_RESUME2_PPI;
```

Parameters

S3RestoreConfig2

Perform S3 resume operation.

Description

This PPI is published by a PEIM and provides for the restoration of the platform's configuration when resuming from the ACPI S3 power state. The ability to execute the boot script may depend on the availability of other PPIs. For example, if the boot script includes an SMBus command, this PEIM looks for the relevant PPI that is able to execute that command.

7.7 S3 Save State Protocol

This section defines how a DXE PI module can record IO operations to be performed as part of the S3 resume. This is done via the **EFI_S3_SAVE_STATE_PROTOCOL** and this allows the implementation of the S3 resume boot path to be abstracted from DXE drivers.

EFI_S3_SAVE_STATE_PROTOCOL

Summary

Used to store or record various IO operations to be replayed during an S3 resume.

GUID

```
#define EFI_S3_SAVE_STATE_PROTOCOL_GUID \
    { 0xe857caf6, 0xc046, 0x45dc, \
    { 0xbe, 0x3f, 0xee, 0x7, 0x65, 0xfb, 0xa8, 0x87 } }
```

Protocol Interface Structure

```
typedef struct _EFI_S3_SAVE_STATE_PROTOCOL
  EFI_S3_SAVE_STATE_PROTOCOL;

typedef struct _EFI_S3_SAVE_STATE_PROTOCOL {
  EFI_S3_SAVE_STATE_WRITE           Write;
  EFI_S3_SAVE_STATE_INSERT          Insert;
  EFI_S3_SAVE_STATE_LABEL           Label;
  EFI_S3_SAVE_STATE_COMPARE         Compare;
} EFI_S3_SAVE_STATE_PROTOCOL;
```

Parameters

Write

Write an opcode at the end of the boot script table. See the **Write()** function description.

Insert

Write an opcode at the specified position in the boot script table. See the **Insert()** function description.

Label

Find an existing label in the boot script table or, if not present, create it. See the **Label()** function description.

Compare

Compare two positions in the boot script table to determine their relative location. See the **Compare()** function description.

Description

The **EFI_S3_SAVE_STATE_PROTOCOL** publishes the PI boot script abstractions. This protocol is not required for all platforms.

On an S3 resume boot path the data stored via this protocol is replayed in the order it appears in the boot script table.

EFI_S3_SAVE_STATE_PROTOCOL.Write()

Summary

Record operations that need to be replayed during an S3 resume .

Prototype

```
typedef
EFI_STATUS
 (EFIAPI *EFI_S3_SAVE_STATE_WRITE) (
  IN CONST EFI_S3_SAVE_STATE_PROTOCOL      *This,
  IN UINT16                                OpCode,
  ...
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

The operation code (opcode) number. See "Related Definitions" below for the defined opcode types.

…

Argument list that is specific to each opcode. See the following subsections for the definition of each opcode.

Description

This function is used to store an OpCode to be replayed as part of the S3 resume boot path. It is assumed this protocol has platform specific mechanism to store the OpCode set and replay them during the S3 resume.

Note: *The opcode is inserted at the end of the boot script table.*

This function has a variable parameter list. The exact parameter list depends on the *OpCode* that is passed into the function. If an unsupported OpCode or illegal parameter list is passed in, this function returns **EFI_INVALID_PARAMETER**.

If there are not enough resources available for storing more scripts, this function returns **EFI_OUT_OF_RESOURCES**.

OpCode values of 0x80 - 0xFE are reserved for implementation-specific functions.

Related Definitions

```
//*******************************************
// EFI Boot Script Opcode definitions
//*******************************************

#define EFI_BOOT_SCRIPT_IO_WRITE_OPCODE                0x00
#define EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE           0x01
#define EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE               0x02
#define EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE          0x03
#define EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE        0x04
#define EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE   0x05
#define EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE           0x06
#define EFI_BOOT_SCRIPT_STALL_OPCODE                   0x07
#define EFI_BOOT_SCRIPT_DISPATCH_OPCODE                0x08
#define EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE              0x09
#define EFI_BOOT_SCRIPT_INFORMATION_OPCODE             0x0A
#define EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE       0x0B
#define EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE  0x0C
#define EFI_BOOT_SCRIPT_IO_POLL_OPCODE                 0x0D
#define EFI_BOOT_SCRIPT_MEM_POLL_OPCODE                0x0E
#define EFI_BOOT_SCRIPT_PCI_CONFIG_POLL_OPCODE         0x0F
#define EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE        0x10

//*******************************************
// EFI_BOOT_SCRIPT_WIDTH
//*******************************************

typedef enum {
  EfiBootScriptWidthUint8,
  EfiBootScriptWidthUint16,
  EfiBootScriptWidthUint32,
  EfiBootScriptWidthUint64,
  EfiBootScriptWidthFifoUint8,
  EfiBootScriptWidthFifoUint16,
  EfiBootScriptWidthFifoUint32,
  EfiBootScriptWidthFifoUint64,
  EfiBootScriptWidthFillUint8,
  EfiBootScriptWidthFillUint16,
  EfiBootScriptWidthFillUint32,
  EfiBootScriptWidthFillUint64,
  EfiBootScriptWidthMaximum
} EFI_BOOT_SCRIPT_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The operation succeeded. A record was added into the specified script table.
EFI_INVALID_PARAMETER	The parameter is illegal or the given boot script is not supported.
EFI_OUT_OF_RESOURCES	There is insufficient memory to store the boot script.

7.7.1 Opcodes for Write()

This section contains the prototypes for variations of the **Write()** function, based on the *Opcode* parameter.

EFI_BOOT_SCRIPT_IO_WRITE_OPCODE

Summary

Adds a record for an I/O write operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL      *This,
  IN  UINT16                                OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                 Width,
  IN  UINT64                                Address,
  IN  UINTN                                 Count,
  IN  VOID                                  *Buffer
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_IO_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_IO_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the I/O operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the I/O operations.

Count

The number of I/O operations to perform. The number of bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

The source buffer from which to write data. The buffer size is *Width* size * *Count*.

Description

This function adds an I/O write record into a specified boot script table. On script execution, this operation writes the presaved value into the specified I/O ports.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE

Summary

Adds a record for an I/O modify operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the I/O operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the I/O operations.

Data

A pointer to the data to be OR-ed.

DataMask

A pointer to the data mask to be AND-ed with the data read from the register.

Description

This function adds an I/O read and write record into the specified boot script table. When the script is executed, the register at Address is read, AND-ed with DataMask, and OR-ed with Data, and finally the result is written back.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_IO_POLL_OPCODE

Summary

Adds a record for I/O reads the I/O location and continues when the exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask,
  IN  UINT64                                    Delay
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the I/O operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the I/O operations.

Data

The comparison value used for the polling exit criteria.

DataMask

Mask used for the polling criteria. The bits in the bytes below *Width* which are zero in *Data* are ignored when polling the memory address.

Delay

The number of 100ns units to poll. Note that timer available may be of poorer granularity so the delay may be longer.

Description

This function adds a delay to the boot script table. The I/O read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Data* & *DataMask*) is equal to *Data*. At least one I/O access is always performed regardless of the value of *Delay*.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE

Summary

Adds a record for a memory write operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT64                                    Address,
  IN  UINTN                                     Count,
  IN  VOID                                      *Buffer
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE.** Type **EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the memory operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the memory operations. Address needs alignment if required.

Count

The number of memory operations to perform. The number of bytes moved is Width size * Count, starting at Address.

Buffer

The source buffer from which to write the data. The buffer size is *Width* size * *Count*.

Description

This function adds a memory write record into a specified boot script table. When the script is executed, this operation writes the presaved value into the specified memory location.

Status Codes Returned

See "Status Codes Returned" in **`Write()`**.

EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE

Summary

Adds a record for a memory modify operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                   Width,
  IN  UINT64                                  Address,
  IN  VOID                                    *Data,
  IN  VOID                                    *DataMask
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the memory operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH.** Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the memory operations. *Address* needs alignment if required.

Data

A pointer to the data to be **OR**-ed.

DataMask

A pointer to the data mask to be **AND**-ed with the data read from the register.

Description

This function adds a memory read and write record into a specified boot script table. When the script is executed, the memory at *Address* is read, **AND**-ed with *DataMask*, and **OR**-ed with *Data*, and finally the result is written back.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_MEM_POLL_OPCODE

Summary

Adds a record for memory reads of the memory location and continues when the exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                   Width,
  IN  UINT64                                  Address,
  IN  VOID                                    *Data,
  IN  VOID                                    *DataMask,
  IN  UINT64                  Delay
  );
```

Parameters

This

A pointer to the EFI_S3_SAVE_STATE_PROTOCOL instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the memory operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The base address of the memory operations. *Address* needs alignment if required.

Data

The comparison value used for the polling exit criteria.

DataMask

Mask used for the polling criteria. The bits in the bytes below Width which are zero in Data are ignored when polling the memory address.

Delay

The number of 100ns units to poll. Note that timer available may be of poorer granularity so the delay may be longer.

Description

This function adds a delay to the boot script table. The memory read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Data* & *DataMask*) is equal to *Data*. At least one I/O access is always performed regardless of the value of *Delay*.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE

Summary

Adds a record for a PCI configuration space write operation into a specified boot script table.

```
Prototype
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                   Width,
  IN  UINT64                                  Address,
  IN  UINTN                                   Count,
  IN  VOID                                    *Buffer
  )
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The address within the PCI configuration space. See Table 12-1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Count

The number of PCI operations to perform. The number of bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

The source buffer from which to write the data. The buffer size is *Width* size * **Count**.

Description

This function adds a PCI configuration space write record into a specified boot script table. When the script is executed, this operation writes the presaved value into the specified location in PCI configuration space.

Status Codes Returned

See "Status Codes Returned" in **`Write()`**.

EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE

Summary

Adds a record for a PCI configuration space modify operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The address within the PCI configuration space. See Table 12 1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Data

A pointer to the data to be **OR**-ed. The size depends on *Width*.

DataMask

A pointer to the data mask to be **AND**-ed.

Description

This function adds a PCI configuration read and write record into a specified boot script table. When the script is executed, the PCI configuration space location at *Address* is read, **AND**-ed with *DataMask*, and **OR**-ed with, and finally the result is written back.

Status Codes Returned

See "Status Codes Returned" in **`Write()`**.

EFI_BOOT_SCRIPT_PCI_CONFIG_POLL_OPCODE

Summary

Adds a record for PCI configuration space reads and continues when the exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask,
  IN  UINT64                                    Delay
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Address

The address within the PCI configuration space. See Table 12 1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Data

The comparison value used for the polling exit criteria.

DataMask

Mask used for the polling criteria. The bits in the bytes below *Width* which are zero in *Data* are ignored when polling the memory address.

Delay

The number of 100ns units to poll. Note that timer available may be of poorer granularity so the delay may be longer.

Description

This function adds a delay to the boot script table. The PCI configuration read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Data* & *DataMask*) is equal to *Data*. At least one PCI configuration access is always performed regardless of the value of *Delay*.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE

Summary

Adds a record for a PCI configuration space write operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST   EFI_S3_SAVE_STATE_PROTOCOL      *This,
  IN  UINT16                                  OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                   Width,
  IN  UINT16                                  Segment,
  IN  UINT64                                  Address,
  IN  UINTN                                   Count,
  IN  VOID                                    *Buffer
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Segment

The PCI segment number for Address.

Address

The address within the PCI configuration space. See Table 12-1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Count

The number of PCI operations to perform. The number of bytes moved is *Width* size * *Count*, starting at *Address*.

Buffer

The source buffer from which to write the data. The buffer size is *Width* size * *Count*.

Description

This function adds a PCI configuration space write record into a specified boot script table. When the script is executed, this operation writes the presaved value into the specified location in PCI configuration space.

Status Codes Returned

See "Status Codes Returned" in **`Write()`**.

EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE

Summary

Adds a record for a PCI configuration space modify operation into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT16                                    Segment,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Segment

The PCI segment number for *Address*.

Address

The address within the PCI configuration space. See Table 12 1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Data

A pointer to the data to be **OR**-ed. The size depends on *Width*.

DataMask

A pointer to the data mask to be **AND**-ed.

Description

This function adds a PCI configuration read and write record into a specified boot script table. When the script is executed, the PCI configuration space location at *Address* is read, **AND**-ed with *DataMask*, and **OR**-ed with, and finally the result is written back.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE

Summary

Adds a record for PCI configuration space reads and continues when the exit criteria is satisfied or after a defined duration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN  UINT16                                    OpCode,
  IN  EFI_BOOT_SCRIPT_WIDTH                     Width,
  IN  UINT16                                    Segment,
  IN  UINT64                                    Address,
  IN  VOID                                      *Data,
  IN  VOID                                      *DataMask,
  IN  UINT64                                    Delay
  )
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE**. Type **EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

The width of the PCI operations. Enumerated in **EFI_BOOT_SCRIPT_WIDTH**. Type **EFI_BOOT_SCRIPT_WIDTH** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Segment

The PCI segment number for *Address*.

Address

The address within the PCI configuration space. See Table 12 1 in the *Extensible Firmware Interface Specification*, version 1.10, for the address format.

Data

The comparison value used for the polling exit criteria.

DataMask

Mask used for the polling criteria. The bits in the bytes below Width which are zero in *Data* are ignored when polling the memory address.

Delay

The number of 100ns units to poll. Note that timer available may be of poorer granularity so the delay may be longer.

Description

This function adds a delay to the boot script table. The PCI configuration read operation is repeated until either a timeout of *Delay* 100 ns units has expired, or (*Data* & *DataMask)* is equal to *Data*. At least one PCI configuration access is always performed regardless of the value of *Delay*.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE

Summary

Adds a record for an SMBus command execution into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN CONST EFI_S3_SAVE_STATE_PROTOCOL          *This,
  IN UINT16                                    OpCode,
  IN EFI_SMBUS_DEVICE_ADDRESS                  SlaveAddress,
  IN EFI_SMBUS_DEVICE_COMMAND                  Command,
  IN EFI_SMBUS_OPERATION                       Operation,
  IN BOOLEAN                                   PecCheck,
  IN UINTN                                     *Length,
  IN VOID                                      *Buffer
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to *EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE*. Type **EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

SlaveAddress

The SMBus address for the slave device that the operation is targeting. Type **EFI_SMBUS_DEVICE_ADDRESS** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *PI Specification*.

Command

The command that is transmitted by the SMBus host controller to the SMBus slave device. The interpretation is SMBus slave device specific. It can mean the offset to a list of functions inside an SMBus slave device. Type **EFI_SMBUS_DEVICE_COMMAND** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *PI Specification*.

Operation

Indicates which particular SMBus protocol it will use to execute the SMBus transactions. Type **EFI_SMBUS_OPERATION** is defined in **EFI_PEI_SMBUS_PPI.Execute()** in the *PI Specification*.

PecCheck

Defines if Packet Error Code (PEC) checking is required for this operation.

Length

A pointer to signify the number of bytes that this operation will do.

Buffer

Contains the value of data to execute to the SMBUS slave device.

Description

This function adds an SMBus command execution record into a specified boot script table. When the script is executed, this operation executes a specified SMBus command.

Status Codes Returned

See "Status Codes Returned" in **`Write()`**.

EFI_BOOT_SCRIPT_STALL_OPCODE

Summary

Adds a record for an execution stall on the processor into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  UINTN                                   Duration
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_STALL_OPCODE**. Type **EFI_BOOT_SCRIPT_STALL_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Duration

Duration in microseconds of the stall.

Description

This function adds a stall record into a specified boot script table. When the script is executed, this operation will stall the system for Duration number of microseconds.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_DISPATCH_OPCODE

Summary

Adds a record for dispatching specified arbitrary code into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  EFI_PHYSICAL_ADDRESS                    EntryPoint
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_DISPATCH_OPCODE**. Type **EFI_BOOT_SCRIPT_DISPATCH_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

EntryPoint

Entry point of the code to be dispatched. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI 2.0 Specification*.

Description

This function adds a dispatch record into a specified boot script table, with which it can run the arbitrary code that is specified. This script can be used to initialize the processor. When the script is executed, the script incurs jumping to the entry point to execute the arbitrary code. After the execution is returned, it goes on executing the next opcode in the table.

The *EntryPoint* must point to memory of type of *EfiRuntimeServicesCode*, *EfiRuntimeServicesData*, or *EfiACPIMemoryNVS*. The EntryPoint must have the same calling convention as the PI DXE Phase.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE

Summary

Adds a record for dispatching specified arbitrary code into a specified boot script table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  EFI_PHYSICAL_ADDRESS                    EntryPoint,
  IN  EFI_PHYSICAL_ADDRESS                    Context
  )
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_DISPATCH_OPCODE**. Type **EFI_BOOT_SCRIPT_DISPATCH_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

EntryPoint

Entry point of the code to be dispatched. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification*.

Context

Argument to be passed into the *EntryPoint* of the code to be dispatched. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification*.

Description

This function adds a dispatch record into a specified boot script table, with which it can run the arbitrary code that is specified. This script can be used to initialize the processor. When the script is executed, the script incurs jumping to the entry point to execute the arbitrary code. After the execution is returned, it goes on executing the next opcode in the table.

The *EntryPoint* and *Context* must point to memory of type of *EfiRuntimeServicesCode*, *EfiRuntimeServicesData*, or *EfiACPIMemoryNVS*. The *EntryPoint* must have the same calling convention as the PI DXE Phase.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_BOOT_SCRIPT_INFORMATION_OPCODE

Summary

Store arbitrary information in the boot script table. This opcode is a no-op on dispatch and is only used for debugging script issues.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN  UINT16                                  OpCode,
  IN  UINT32                                  InformationLength,
  IN  EFI_PHYSICAL_ADDRESS                    Information
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_DISPATCH_OPCODE**. Type **EFI_BOOT_SCRIPT_DISPATCH_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

InformationLenght

Length of the data in bytes.

Information

Information to be logged in the boot scrpit.

Description

This function adds a record that has no impact on the S3 replay. This function is used to store debug information in the S3 data stream.

The *Information* must point to memory of type of *EfiRuntimeServicesCode*, *EfiRuntimeServicesData*, or *EfiACPIMemoryNVS*.

Status Codes Returned

See "Status Codes Returned" in **Write()**.

EFI_S3_SAVE_STATE_PROTOCOL.Insert()

Summary

Record operations that need to be replayed during an S3 resume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_S3_SAVE_STATE_INSERT) (
  IN     CONST EFI_S3_SAVE_STATE_PROTOCOL     *This,
  IN     BOOLEAN                              BeforeOrAfter,
  IN OUT EFI_S3_BOOT_SCRIPT_POSITION          *Position OPTIONAL,
  IN     UINT16                               OpCode,
  ...
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCO**L instance.

BeforeOrAfter

Specifies whether the opcode is stored before (**TRUE**) or after (**FALSE**) the position in the boot script table specified by *Position*. If *Position* is **NULL** or points to **NULL** then the new opcode is inserted at the beginning of the table (if **TRUE**) or end of the table (if **FALSE**).

Position

On entry, specifies the position in the boot script table where the opcode will be inserted, either before or after, depending on *BeforeOrAfter*. On exit, specifies the position of the inserted opcode in the boot script table.

OpCode

The operation code (opcode) number. See "Related Definitions" in **Write()** for the defined opcode types.

...

Argument list that is specific to each opcode. See the following subsections for the definition of each opcode.

Description

This function is used to store an *OpCode* to be replayed as part of the S3 resume boot path. It is assumed this protocol has platform specific mechanism to store the *OpCode* set and replay them during the S3 resume.

The opcode is stored before (**TRUE**) or after (**FALSE**) the position in the boot script table specified by *Position*. If *Position* is **NULL** or points to **NULL** then the new opcode is inserted at the beginning of the table (if **TRUE**) or end of the table (if **FALSE**).

The position which is pointed to by Position upon return can be used for subsequent insertions.

This function has a variable parameter list. The exact parameter list depends on the *OpCode* that is passed into the function. If an unsupported *OpCode* or illegal parameter list is passed in, this function returns **EFI_INVALID_PARAMETER**.

If there are not enough resources available for storing more scripts, this function returns **EFI_OUT_OF_RESOURCES**.

OpCode values of 0x80 - 0xFE are reserved for implementation specific functions.

Related Definitions

```
typedef VOID *EFI_S3_BOOT_SCRIPT_POSITION;
```

Status Codes Returned

EFI_SUCCESS	The operation succeeded. An opcode was added into the script table.
EFI_INVALID_PARAMETER	The *Opcode* is an invalid opcode value.
EFI_INVALID_PARAMETER	The *Position* is not a valid position in the boot script table.
EFI_OUT_OF_RESOURCES	There is insufficient memory to store the boot script.

EFI_S3_SAVE_STATE_PROTOCOL.Label()

Summary

Find a label within the boot script table and, if not present, optionally create it.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_S3_SAVE_STATE_LABEL) (
  IN     CONST EFI_S3_SAVE_STATE_PROTOCOL    *This,
  IN     BOOLEAN                             BeforeOrAfter,
  IN     BOOLEAN                             CreateIfNotFound,
  IN OUT EFI_S3_BOOT_SCRIPT_POSITION         *Position OPTIONAL,
  IN     CONST CHAR8                         *Label
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

BeforeOrAfter

Specifies whether the label is stored before (**TRUE**) or after (**FALSE**) the position in the boot script table specified by *Position*. If *Position* is **NULL** or points to **NULL** then the new label is inserted at the beginning of the table (if **TRUE**) or end of the table (if **FALSE)** .

CreateIfNotFound

Specifies whether the label will be created if the label does not exists (**TRUE**) or not (**FALSE**).

Position

On entry, specifies the position in the boot script table where the label will be inserted, either before or after, depending on *BeforeOrAfter*. On exit, specifies the position of the inserted label in the boot script table.

Label

Points to the label which will be inserted in the boot script table.

Description

If the label *Label* is already exists in the boot script table, then no new label is created, the position of the *Label* is returned in ***Position** and **EFI_SUCCESS** is returned.

If the label *Label* does not already exist and *CreateIfNotFound* is **TRUE**, then it will be created before or after the specified position and **EFI_SUCCESS** is returned.

If the label *Label* does not already exist and *CreateIfNotFound* is **FALSE**, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	The label already exists or was inserted.
EFI_NOT_FOUND	The label did not already exist and *CreateifNotFound* was **FALSE**.
EFI_INVALID_PARAMETER	The *Opcode* is an invalid opcode value.
EFI_INVALID_PARAMETER	The *Position* is not a valid position in the boot script table.
EFI_OUT_OF_RESOURCES	There is insufficient memory to store the boot script.

EFI_S3_SAVE_STATE_PROTOCOL.Compare()

Summary

Compare two positions in the boot script table and return their relative position.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_S3_SAVE_STATE_COMPARE) (
  IN  CONST EFI_S3_SAVE_STATE_PROTOCOL      *This,
  IN  EFI_S3_BOOT_SCRIPT_POSITION           Position1,
  IN  EFI_S3_BOOT_SCRIPT_POSITION           Position2,
  OUT UINTN                                 *RelativePosition
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

Position1, Position2

The positions in the boot script table to compare.

RelativePosition

On return, points to the result of the comparison.

Description

This function compares two positions in the boot script table and returns their relative positions. If *Position1* is before *Position2*, then -1 is returned. If *Position1* is equal to *Position2*, then 0 is returned. If *Position1* is after *Position2*, then 1 is returned.

Status Codes Returned

EFI_SUCCESS	The label already exists or was inserted.
EFI_INVALID_PARAMETER	The *Position1* or *Position2* is not a valid position in the boot script table.

7.8 S3 SMM Save State Protocol

This chapter defines how a SMM PI module can record IO operations to be performed as part of the S3 resume. This is done via the **EFI_S3_SMM_SAVE_STATE_PROTOCOL** and this allows the implementation of the S3 resume boot path to be abstracted from SMM drivers.

The S3 SMM Save State Protocol shares the function **Save()** function definition with the S3 SMM Save State Protocol but implements it on a separate protocol. Having separate protocols for SMM and DXE makes it easier to accommodate the differences in the operating environment between SMM and DXE.

EFI_S3_SMM_SAVE_STATE_PROTOCOL

Summary

Used to store or record various IO operations to be replayed during an S3 resume.

GUID

```
#define EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID \
  { 0x320afe62, 0xe593, 0x49cb, \
  { 0xa9, 0xf1, 0xd4, 0xc2, 0xf4, 0xaf, 0x1, 0x4c } }
```

Protocol Interface Structure

```
typedef struct _EFI_S3_SMM_SAVE_STATE_PROTOCOL {
  EFI_S3_SAVE_STATE_WRITE           Write;
  EFI_S3_SAVE_STATE_INSERT          Insert;
  EFI_S3_SAVE_STATE_LABEL           Label;
  EFI_S3_SAVE_STATE_COMPARE         Compare;
} EFI_S3_SMM_SAVE_STATE_PROTOCOL;
```

Parameters

Write

Write an opcode at the end of the boot script table. See the **Write()** function description under the **EFI_S3_SAVE_STATE_PROTOCOL** definition.

Insert

Write an opcode at the specified position in the boot script table. See the **Insert()** function description under the **EFI_S3_SAVE_STATE_PROTOCOL** definition.

Label

Find an existing label in the boot script table or, if not present, create it. See the Label() function description under the **EFI_S3_SAVE_STATE_PROTOCOL** definition.

Compare

Compare two positions in the boot script table to determine their relative location. See the **Compare()** function description under the **EFI_S3_SAVE_STATE_PROTOCOL** definition.

Description

The **EFI_S3_SMM_SAVE_STATE_PROTOCOL** publishes the PI SMMboot script abstractions

On an S3 resume boot path the data stored via this protocol is replayed in the order it was stored.

The order of replay is the order either of the S3 Save State Protocol or S3 SMM Save State Protocol **Write()** functions were called during the boot process. **Insert()**, **Label()**, and **Compare()** operations are ordered relative other S3 SMM Save State Protocol write() operations and the order relative to S3 State Save **Write()** operations is not defined. Due to these ordering restrictions it is recommended that the S3 State Save Protocol be used during the DXE phase when every possible.

The **EFI_S3_SMM_SAVE_STATE_PROTOCOL** can be called at runtime and **EFI_OUT_OF_RESOURCES** may be returned from a runtime call. It is the responsibility of the platform to ensure enough memory resource exists to save the system state. It is recommended that runtime calls be minimized by the caller.

8
PCI Host Bridge

8.1 PCI Host Bridge Overview

This specification defines the core code and services that are required for an implementation of the PCI Host Bridge Resource Allocation Protocol. This protocol is used by a PCI bus driver to program the PCI host bridge and configure the root PCI buses. The registers inside the PCI host bridge that control root PCI bus configuration are not governed by the PCI specification and vary from chipset to chipset. The PCI Host Bridge Resource Allocation Protocol is therefore specific to a particular chipset.

This specification does the following:

- Describes the basic components of the PCI Host Bridge Resource Allocation Protocol
- Describes several sample PCI architectures and a sample implementation of the PCI Host Bridge Resource Allocation Protocol
- Provides code definitions for the PCI Host Bridge Resource Allocation Protocol and the PCI-host-bridge-related type definitions that are architecturally required by this specification.

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** does not describe platform policies.The platform policies are described by the **EFI_PCI_PLATFORM_PROTOCOL**, which is desribed in section 9.6.1. Silicon-related policies are described by the **EFI_PCI_OVERRIDE_PROTOCOL**, which is described in section 9.6.2

8.2 PCI Host Bridge Design Discussion

This section provides background and design information for the PCI Host Bridge Resource Allocation Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to program PCI host bridge hardware. This protocol abstracts a PCI host bridge. In particular, functions for programming a PCI host bridge are defined here although other bus types may be supported in a similar fashion as extensions to this specification.

This chapter discusses the following:

- PCI terms that are used in this document
- An overview of the PCI Host Bridge Resource Allocation Protocol
- Sample PCI architectures
- ISA aliasing considerations
- Programming of standard PCI configuration registers
- Sample implementation

8.3 PCI Host Bridge Resource Allocation Protocol

8.3.1 PCI Host Bridge Resource Allocation Protocol Overview

The PCI Host Bridge Resource Allocation Protocol is used by a PCI bus driver to program a PCI host bridge. The registers inside a PCI host bridge that control configuration of PCI root buses are not governed by the PCI specification and vary from chipset to chipset. The PCI Host Bridge Resource Allocation Protocol implementation is therefore specific to a particular chipset.

Each PCI host bridge is comprised of one or more PCI root bridges, and there are hardware registers associated with each PCI root bridge. These registers control the bus, I/O, and memory resources that are decoded by the PCI root bus that the PCI root bridge produces and all the PCI buses that are children of that PCI root bus.

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** allows for future innovation of the chipsets. It abstracts the PCI bus driver from the chipset details. This design allows system designers to make changes to the host bridge hardware without impacting a platform-independent PCI bus driver.

See PCI Host Bridge Resource Allocation Protocol in Code Definitions for the definition of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

8.3.2 Host Bus Controllers

A platform can be viewed as the following:

- A set of processors
- A set of core chipset components that may produce one or more host buses

The figure below shows a platform with ***n*** processors (CPUs) and a set of core chipset components that produce ***m*** host bridges (HBs).

Most systems with one PCI host bus controller will contain a single instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. More complex systems may contain multiple instances of this protocol.

Note: *There is no relationship between the number of chipset components in a platform and the number of* **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** *instances. This protocol is an abstraction from a software point of view. This protocol is attached to the device handle of a PCI host bus controller, which itself is composed of one or more PCI root bridges. A PCI root bridge is a chipset component(s) that produces a physical PCI bus whose parent is not another physical PCI bus.*

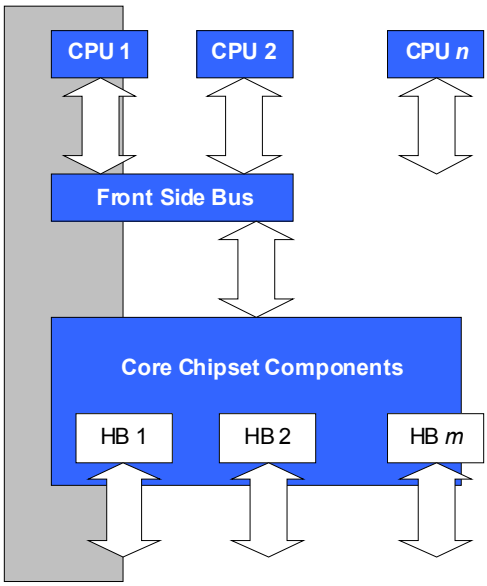

Figure 4. Host Bus Controllers

8.3.3 Producing the PCI Host Bridge Resource Allocation Protocol

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL instances are produced by DXE drivers—most often by early DXE drivers.

The figure below shows how the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is used to identify the associated PCI root bridges. After the steps in the figure are completed, the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** can then be queried to identify the device handles of the associated PCI root bridges. See the *UEFI 2.1 Specification* for details of the PCI Root Bridge I/O Protocol.

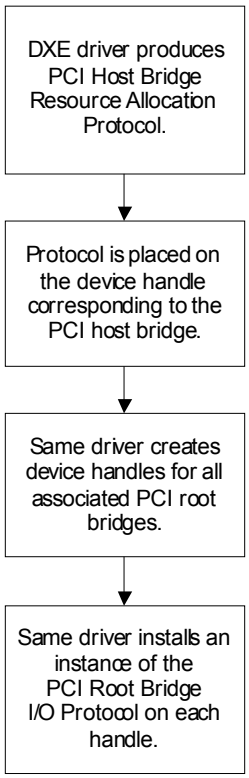

Figure 5. Producing the PCI Host Bridge Resource Allocation Protocol

8.3.4 Required PCI Protocols

The following protocols are mandatory if the system supports PCI devices or slots:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

See the *UEFI 2.1 Specification* for more information on the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

8.3.5 Relationship with EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL

It is expected, although not necessary, that a chipset-aware driver will produce the following protocol instances:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

Care has been taken to avoid overlap between the member functions of the two protocols. For example, **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** does not describe the *SegmentNumber* or the final resource assignment for a root bridge, because these attributes are available using the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. Both protocols contain links to the associated instances of the other protocols, as follows:

- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**: Includes the handle of the PCI host bridge that is associated with the root bridge.
- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**: Provides a member function to retrieve the handles of the associated root bridges.

The definition of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** attempts to maintain compatibility with the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** definition.

See the *UEFI 2.1 Specification* for more information on the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

8.4 Sample PCI Architectures

8.4.1 Sample PCI Architectures Overview

The PCI Host Bridge Resource Allocation Protocol is a protocol that is designed to provide a software abstraction for a wide variety of PCI architectures. This section provides examples of the following PCI architectures:

- Desktop system with 1 PCI root bridge
- Server system with 4 PCI root bridges
- Server system with 2 PCI segments
- Server system with 2 PCI host buses

This section is not intended to be an exhaustive list of the PCI architectures that the PCI Host Bridge Resource Allocation Protocol can support. Instead, it is intended to show the flexibility of this protocol to adapt to current and future platform designs.

8.4.2 Desktop System with 1 PCI Root Bridge

The figure below shows an example of a PCI host bus with one PCI root bridge. This PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard and/or PCI slots. This setup would be typical of a desktop system. In this system, the PCI root bridge needs minimal setup. Typically, the PCI root bridge will decode the following:

- The entire bus range on Segment 0
- The entire I/O space of the processor
- All the memory above the top of system memory

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- One instance of PCI Root Bridge I/O Protocol

See the *UEFI 2.1 Specification*, Chapter 13, for details of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

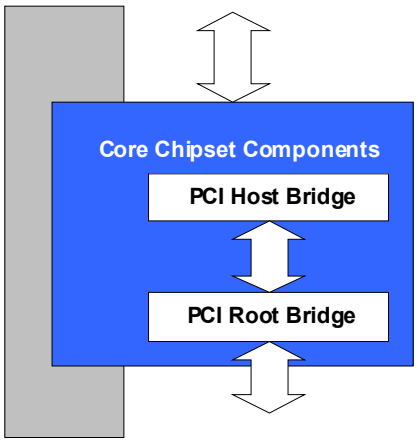

Figure 6. Desktop System with 1 PCI Root Bridge

8.4.3 Server System with 4 PCI Root Bridges

The figure below shows an example of a larger server with one PCI host Bus with four PCI root bridges (RBs). The PCI devices that are attached to the PCI root bridges are all part of the same coherency domain, which means they share the following:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

As a result, each PCI root bridge must get resources out of a common pool. Each PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard or PCI slots. The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Four instances of the PCI Root Bridge I/O Protocol

See the *UEFI 2.1 Specification*, Chapter 13, for details of the **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`**.

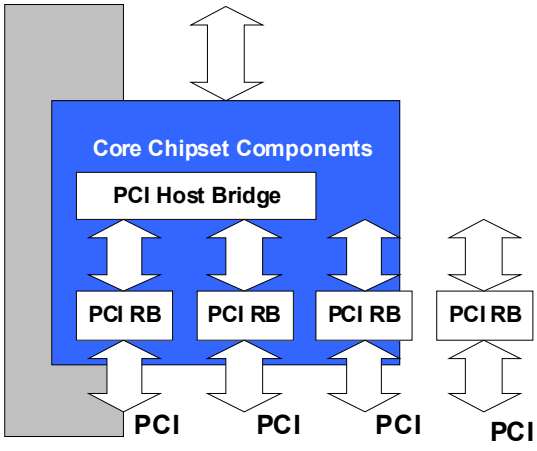

Figure 7. Server System with 4 PCI Root Bridges

8.4.4 Server System with 2 PCI Segments

The figure below shows an example of a server with one PCI host bus and two PCI root bridges (RBs). Each of these PCI root bridges is on a different PCI segment, which allows the system to have up to 512 PCI buses. A single PCI segment is limited to 256 PCI buses. These two segments do not share the same PCI configuration space, but they do share the following, which is why they can be described with a single PCI host bus:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

See the *UEFI 2.1 Specification*, Chapter 13, for details of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

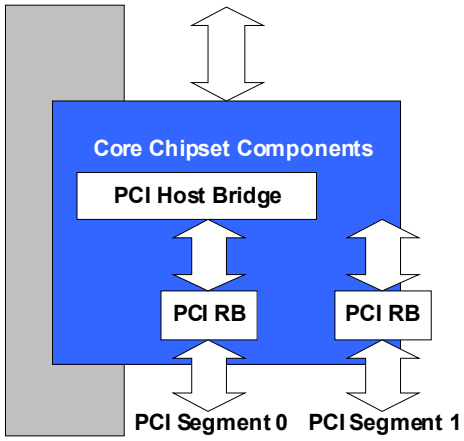

Figure 8. Server System with 2 PCI Segments

8.4.5 Server System with 2 PCI Host Buses

The figure below shows a server system with two PCI host buses and one PCI root bridge (RB) per PCI host bus. As in Figure 8, this system supports up to 512 PCI buses, but the following resources are not shared between the two PCI root bridges:

- PCI I/O space
- PCI memory space
- PCI prefetchable memory space

The firmware for this platform would produce the following:

- Two instances of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

See the *UEFI 2.1 Specification*, Chapter 13, for details of t the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

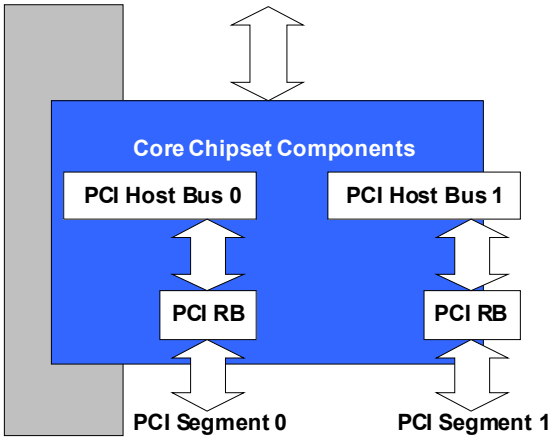

Figure 9. Server System with 2 PCI Host Buses

8.5 ISA Aliasing Considerations

The PCI host bridge driver will handle the ISA alias addresses based on the platform policy. The platform communicates the policy to the PCI host bridge driver using the **EFI_PCI_PLATFORM_PROTOCOL**. If the PCI host bridge driver cannot locate an instance of **EFI_PCI_PLATFORM_PROTOCOL**, it will not reserve the ISA alias addresses. The PCI bus driver is not aware of this policy and probes devices to gather resource requirements regardless of this policy. The **EFI_PCI_PLATFORM_PROTOCOL** is defined in section 9.6.1.

Note: *When it is started, a PCI device may request that the ISA alias ranges be forwarded to it through the* **EFI_PCI_IO_PROTOCOL.Attributes()** *member function by setting the input parameter* `Attributes` *to* **EFI_PCI_IO_ATTRIBUTE_ISA_IO**. *If the ISA alias I/O addresses are not reserved during enumeration, such a request may fail because one or more PCI devices may be occupying aliased addresses.*

If the ISA alias I/O addresses are to be reserved during enumeration, the PCI host bridge driver is responsible for allocating four times the amount of the requested I/O. The PCI bus driver obtains the resources by calling one of the following member functions:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetProposedResources()**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()**

The PCI host bridge driver sets the _RNG bit to communicate the availability of the ISA alias range to the PCI bus driver. If the _RNG flag is set, the PCI bus enumerator is not allowed to allocate the ISA alias addresses to any PCI device. See Table 5 in the "Description" section of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** for the definition of the _RNG flag. In this case, a PCI device's request to turn on aliasing will succeed because one or more PCI devices may be occupying aliased addresses. The _RNG flag is the only aspect of the protocol interface structure that is affected by ISA aliasing.

8.6 Programming of Standard PCI Configuration Registers

This topic defines design guidelines for programming PCI configuration registers in the standard PCI header. It defines roles and responsibilities of various drivers.

Table 1. Standard PCI Devices – Header Type 0

PCI Configuration Register Bits	Programmed By
PCI command register – I/O, Memory, and Bus Master enable	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
PCI command register – SERR, PERR, MWI, Special Cycle Enable, Fast Back to Back Enable	Chipset/platform-specific code
PCI command register – VGA palette snoop	PCI device driver.
Cache line size	Chipset/platform code to match the processor's cache line size or some other value.
Latency timer	PCI bus driver. This driver programs this register to default values before it sends the **EfiPciBeforeResourceCollection** notification. For PCI devices, this value is 0x20. PCI-X* devices come out of reset with this register set to 0x40. The PCI bus driver does not change the setting. The PCI bus driver will also make sure that the default value for PCI devices is consistent with the **MIN_LAT** and **MAX_LAT** register values in the device's PCI configuration space. Chipset/platform code can overwrite this register during the **EfiPciBeforeResourceCollection** notification call. The new value may come from the end user using configuration options. The device driver may overwrite this value during its own **Start()** function.
BIST	PCI bus driver.
Base address registers	PCI bus driver.
Interrupt line	Not touched.
Subsystem vendor ID and Device ID	Chipset/platform code. Per the *PCI Specification,* these registers must get programmed before system software accesses the device. Some noncompliant or chipset devices may require that these registers be programmed during the preboot phase.

Table 2. PCI-to-PCI Bridge – Header Type 1

PCI Configuration Register Bits	Programmed By
PCI command register – I/O, Memory, Bus Master enable, VGA palette snoop	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
PCI command register – SERR, PERR, MWI, Fast Back to Back Enable, Special Cycle Enable	Chipset/platform-specific code.
Cache line size	Chipset/platform code to match the processor's cache line size or some other value.
Latency timer	PCI bus driver. This driver programs to default values before it sends the **EfiPciBeforeResourceCollection** notification. For PCI devices, this value is 0x20. PCI-X devices come out of reset with this register set to 0x40.The PCI bus driver does not change the setting. The PCI bus driver will also make sure that the default value for PCI devices is consistent with the **MIN_LAT** and **MAX_LAT** register values in the device's PCI configuration space. Chipset/platform code can overwrite this register during the **EfiPciBeforeResourceCollection** notification call. The new value may come from the end user using configuration options.
Base addresses registers, bus, I/O, and memory aperture registers	PCI bus driver.
Interrupt line	Not touched.
Bridge control register – ISA Enable, VGA Enable	PCI bus driver. This driver sets these values as requested by the device driver through the **EFI_PCI_IO_PROTOCOL** member functions.
Bridge control register – PERR Enable, SERR Enable, Fast Back to Back, Discard Timers	Chipset/platform-specific code.
Bridge control register – Secondary Bus Reset	PCI bus driver is permitted to reset the secondary bus during enumeration. The chipset/platform code may also reset the secondary bus during the **EfiPciBeforeChildBusEnumeration** notification.

8.7 Sample Implementation

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. Actual implementations may vary. Calls to **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. **PreprocessController()** are not included for the sake of clarity.

Unless noted otherwise, all functions that are listed below are member functions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

1. If the hardware supports dynamically changing the number of PCI root buses or changing the segment number that is associated with a PCI root bus, such changes must be completed before the next steps.
2. The chipset/platform driver(s) creates a device handle for the PCI host bridges in the system(s) and installs an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** on that handle.
3. The chipset/platform driver(s) creates a device handle for every PCI root bridge and installs the following on that handle:
   - An instance of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
   - An instance of **EFI_DEVICE_PATH_PROTOCOL**

   It is expected that a single driver will handle a PCI host bridge, as well as all the associated PCI root bridges. The *ParentHandle* field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** must be initialized with the handle for the PCI host bridge that contains an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

...Other initialization activities take place.

4. The **EFI_DRIVER_BINDING_PROTOCOL.Start()** function of the PCI bus driver is called and is passed the device handle of a PCI root bridge. The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI root bridge can be found by using the *ParentHandle* field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** must be present in PI Architecture systems.
5. Begin the PCI enumeration process. The order in which the various member functions are called cannot be changed. Between any two steps, there can be any amount of implementation-specific code as long as it does not call any member functions of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. This requirement is necessary to keep the state machines in the PCI host bridge allocation driver and the PCI bus enumerator in sync.
6. Notify the host bridge driver that PCI enumeration is about to begin by calling **NotifyPhase(EfiPciHostBridgeBeginEnumeration)**. This member function must be the first one that gets called. PCI enumeration has two steps: bus enumeration and resource enumeration.
7. Notify the host bridge driver that bus enumeration is about to begin by calling **NotifyPhase (EfiPciHostBridgeBeginBusAllocation)**.
8. Do the following for every PCI root bridge handle:
   - Call **StartBusEnumeration(***This*,*RootBridgeHandle*)**.**
   - Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.
   - Allocate memory to hold resource requirements. These resources can be two resource trees: one to hold bus requirements and another to hold the I/O and memory requirements.
   - Call **GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.
   - Scan all the devices in the specified bus range and on the specified segment. If it is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If it is an ordinary device, collect the resource request and add up all of

these requests in multiple pools (e.g., I/O, 32-bit prefetchable memory). Combine different types of memory requests at an appropriate level based on the PCI root bridge attributes. Update the resource requirement information accordingly. On every PCI root bridge, reserve space to cover the largest expansion ROMs on that bus, which will allow the PCI bus driver to retrieve expansion ROMs from the PCI card or device without having to reprogram the PCI host bridge. Because the memory and I/O resource collection step does not call any member function of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**, it can be performed at a later time.

- Once the number of PCI buses under this PCI root bridge is known, call **SetBusNumbers()** with this information.

9. Notify the host bridge driver that the bus allocation phase is over by calling **NotifyPhase (EfiPciHostBridgeEndBusAllocation)**.
10. Notify the host bridge driver that resource allocation is about to begin by calling **NotifyPhase(EfiPciHostBridgeBeginResourceAllocation)**.
11. For every PCI root bridge handle, call **SubmitResources()**. The *Configuration* information is derived from the resource requirements that were computed in step 8 above.
12. Call **NotifyPhase(EfiPciHostBridgeAllocateResources)** to allocate the necessary resources. This call should not be made unless resource requirements for all the PCI root bridges have been submitted. If the call succeeds, go to next step. Otherwise, there are two options:
    - Make do with the smaller ranges.
    - Call **GetProposedResources()** to retrieve the proposed settings and examine the differences. Prioritize various requests and drop lower-priority requests. Call **NotifyPhase(EfiPciHostBridgeFreeResources)** to undo the previous allocation. Go back to step 11 with reduced requirements, which includes resubmitting requests for all the root bridges.
13. Call **NotifyPhase(EfiPciHostBridgeSetResources)** to program the hardware. At this point, the decode logic in this host bridge is fully set up.
14. Do the following for every root bridge handle:
    - Obtain the resource range that is assigned to a PCI root bridge by calling the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function on that handle.
    - From the resource range that is assigned to the PCI root bridge, assign resources to all the devices. Program the Base Address Registers (BARs) in all the PCI functions and decode registers in PCI-to-PCI bridges. If a PCI device has a PCI option ROM, copy the contents to a buffer in memory. It is possible to defer the BAR programming for a PCI controller until a connect request for the device is received.
    - Create a device handle for each PCI device as required.
    - Install an instance of **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on each of these handles.
15. Notify the host bridge driver that resource allocation is complete by calling **NotifyPhase(EfiPciHostBridgeEndResourceAllocation)**.
16. Deallocate any temporary buffers.

Looping on PCI root bridges is accomplished with the following algorithm:

```
RootBridgeHandle = NULL;
while (GetNextRootBridge(RootBridgeHandle) == EFI_SUCCESS) {
    . . .
```

8.7.1 PCI enumeration process

1. If the hardware supports dynamically changing the number of PCI root buses or changing the segment number that is associated with a PCI root bus, such changes must be completed before the next steps.
2. The PCI host bridge driver (s) creates a device handle for the PCI host bridges in the system(s) and installs an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** on that handle.
3. The PCI root bridge driver(s) creates a device handle for every PCI root bridge and installs the following on that handle:
   - An instance of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
   - An instance of **EFI_DEVICE_PATH_PROTOCOL**

   It is expected that a single driver will handle a PCI host bridge, as well as all the associated PCI root bridges. The *ParentHandle* field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** must be initialized with the handle for the PCI host bridge that contains an instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**.

8.7.1.1 Other initialization activities take place.

4. The **EFI_DRIVER_BINDING_PROTOCOL.Start()** function of the PCI bus driver is called and is passed the device handle of a PCI root bridge. The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI root bridge can be found by using the ParentHandle field of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** must be present.
5. Begin the PCI enumeration process. The order in which the various member functions are called cannot be changed. Between any two steps, there can be any amount of implementation-specific code as long as it does not call any member functions of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. This requirement is necessary to keep the state machines in the PCI host bridge allocation driver and the PCI bus enumerator in sync.
6. Notify drivers that PCI enumeration is about to begin using **EfiPciHostBridgeBeginenumeration**.

8.7.1.2 PCI enumeration has two steps: bus enumeration and resource enumeration.

7. Notify drivers that PCI bus enumeration is about to begin using EfiPciHostBridgeBeginBusAllocation.
8. Do the following for every PCI root bridge handle:
   - Call **StartBusEnumeration** (*This*, *RootBridgeHandle*).
   - Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.
   - Allocate memory to hold resource requirements.

- Call **GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.
- Scan all the devices in the specified bus range and on the specified segment.

If it is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. Call the drivers for preprocess notifications using EfiPciBeforeChildBusEnumeration.

If it is an ordinary device, collect the resource request and add up all of these requests in multiple pools (e.g., I/O, 32-bit prefetchable memory). Combine different types of memory requests at an appropriate level based on the PCI root bridge attributes. Update the resource requirement information accordingly.

On every PCI root bridge, reserve space to cover the largest expansion ROMs on that bus, which will allow the PCI bus driver to retrieve expansion ROMs from the PCI card or device without having to reprogram the PCI host bridge. Because the memory and I/O resource collection step does not call any member function of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**, it can be performed at a later time.

- Once the number of PCI buses under this PCI root bridge is known, call **SetBusNumbers()** with this information.

9. Notify drivers that the bus allocation phase is over using **EfiPciHostBridgeEndBusAllocation**.
10. Notify drivers that resource allocation is about to begin using **EfiPciHostBridgeBeginResourceAllocation**.
11. For every PCI root bridge handle, call **SubmitResources()**. The *Configuration* information is derived from the resource requirements that were computed in step 8 above.
12. Notify the drivers to allocate the necessary resources using **EfiPciHostBridgeAllocateResources**. This call should not be made unless resource requirements for all the PCI root bridges have been submitted. If the call succeeds, go to next step. Otherwise, there are two options:
    - Make do with the smaller ranges.
    - Call **GetProposedResources()** to retrieve the proposed settings and examine the differences. Prioritize various requests and drop lower-priority requests. Notify the drivers using **EfiPciHostBridgeFreeResources** to undo the previous allocation. Go back to step 11 with reduced requirements, which includes resubmitting requests for all the root bridges.
13. Notify the drivers using **EfiPciHostBridgeSetResources** to program the hardware. At this point, the decode logic in this host bridge is fully set up.
14. Do the following for every root bridge handle:
    - Obtain the resource range that is assigned to a PCI root bridge by calling the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function on that handle.
    - From the resource range that is assigned to the PCI root bridge, assign resources to all the devices. Program the Base Address Registers (BARs) in all the PCI functions and decode registers in PCI-to-PCI bridges. If a PCI device has a PCI option ROM, copy the contents to a buffer in memory. It is possible to defer the BAR programming for a PCI controller until a connect request for the device is received.

- Create a device handle for each PCI device as required.
- Install an instance of **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on each of these handles.

15. Notify the drivers that resource allocation is complete by using **EfiPciHostBridgeEndResourceAllocation**.
16. Notify the drivers that bus enumeration is complete by calling **EfiPciHostBridgeEndEnumeration**.
17. Deallocate any temporary buffers.
18. Install the **EFI_PCI_ENUMERATION_COMPLETE_GUID** protocol.

Related Definitions

```
#define EFI_PCI_ENUMERATION_COMPLETE_GUID \
  { 0x30cfe3e7, 0x3de1, 0x4586, \
  0xbe, 0x20, 0xde, 0xab, 0xa1, 0xb3, 0xb7, 0x93 }
```

Note: *This protocol is always installed with a NULL pointer.*

8.7.1.3 Sample PCI Device Set Up Implementation

This section describes further the outlines of the process in step 14, second bullet (above).

1. Call the PCI enumeration preprocess functions using **EfiPciBeforeResourceCollection**.
2. Gather PCI device resource requirements.
3. If present, call **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** to see if there is an alternate set of resources for this device.
4. Call the **EFI_PCI_PLATFORM_PROTOCOL** function **GetPciRom()**. If it returns EFI_SUCCESS, go to step 7.
5. Call the **EFI_PCI_OVERRIDE_PROTOCOL** function **GetPciRom()**. If it returns EFI_SUCCESS, go to step 7.
6. Find the PCI device's option ROM and copy its contents into memory. If there is no option ROM, go to step 8.
7. Find and decompress the UEFI image within the option ROM image.
8. Exit

8.7.2 Sample Enumeration Implementation

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL**. Actual implementations may vary.

8.7.2.1 PCI Enumeration Phases

There are several phases of the PCI enumeration process. For each phase, the PCI platform drivers and the PCI host bridge drivers are notified as follows:

1. The **PlatformNotify()** function of the **EFI_PCI_PLATFORM_PROTOCOL** is called with the enumeration phase and the execution phase BeforePciHostBridge.
2. The **PlatformNotify()** function of the **EFI_PCI_OVERRIDE_PROTOCOL** is called with the enumeration phase and the execution phase BeforePciHostBridge.
3. The NotifyPhase function of each instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is called with the enumeration phase.
4. The **PlatformNotify()** function of the **EFI_PCI_PLATFORM_PROTOCOL** is called with the enumeration phase and the execution phase AfterPciHostBridge.
5. The **PlatformNotify()** function of the **EFI_PCI_OVERRIDE_PROTOCOL** is called with the execution phase AfterPciHostBridge.

8.7.2.2 Additional locations to preprocess PCI devices

There are a few additional places during the PCI enumeration process where the platform or PCI host bridge drivers are given the opportunity to preprocess individual PCI devices.

1. The **PlatformPrepController** function of the **EFI_PCI_PLATFORM_PROTOCOL** is called with the preprocess phase and the execution phase of BeforePciHostBridge.
2. The **PlatformPrepController** function of each instance of the **EFI_PCI_OVERRIDE_PROTOCOL** is called with the preprocess phase and the execution phase of BeforePciHostBridge.
3. The **PreprocessController** function of each instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is called with the preprocess phase.
4. The **PlatformPrepController** function of each instance of the **EFI_PCI_PLATFORM_PROTOCOL** is called with the preprocess phase and the execution phase of AfterPciHostBridge.
5. The **PlatformPrepController** function of the **EFI_PCI_OVERRIDE_PROTOCOL** is called with the preprocess phase and the execution phase of **AfterPciHostBridge**.

8.8 PCI HostBridge Code Definitions

8.8.1 Introduction

This section contains the basic definitions of the PCI Host Bridge Resource Allocation Protocol. This section defines the protocol
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE**
- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES**
- **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE**

8.8.2 PCI Host Bridge Resource Allocation Protocol

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL

Summary

Provides the basic interfaces to abstract a PCI host bridge resource allocation.

Note: *This protocol is mandatory if the system includes PCI devices.*

GUID

```
#define EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GUID \
  {
0xCF8034BE,0x6768,0x4d8b,0xB7,0x39,0x7C,0xCE,0x68,0x3A,0x9F,0xBE
}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL
{
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE
                                              NotifyPhase;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_B
RIDGE
                                              GetNextRootBridge;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_ATTRIBUTES
                                              GetAllocAttributes;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUME
RATION
                                              StartBusEnumeration;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS
                                              SetBusNumbers;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCE
S
                                              SubmitResources;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RE
SOURCES
                                              GetProposedResources;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONT
ROLLER
                                              PreprocessController;
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL;
```

Parameters

NotifyPhase

The notification from the PCI bus enumerator that it is about to enter a certain phase during the enumeration process. See the **NotifyPhase()** function description.

GetNextRootBridge

Retrieves the device handle for the next PCI root bridge that is produced by the host bridge to which this instance of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is attached. See the **GetNextRootBridge()** function description. See section 1.2 for a definition of a PCI root bridge.

GetAllocAttributes

Retrieves the allocation-related attributes of a PCI root bridge. See the **GetAllocAttributes()** function description.

StartBusEnumeration

Sets up a PCI root bridge for bus enumeration. See the **StartBusEnumeration()** function description.

SetBusNumbers

Sets up the PCI root bridge so that it decodes a specific range of bus numbers. See the **SetBusNumbers()** function description.

SubmitResources

Submits the resource requirements for the specified PCI root bridge. See the **SubmitResources()** function description.

GetProposedResources

Returns the proposed resource assignment for the specified PCI root bridges. See the **GetProposedResources()** function description.

PreprocessController

Provides hooks from the PCI bus driver to every PCI controller (device/function) at various stages of the PCI enumeration process that allow the host bridge driver to preinitialize individual PCI controllers before enumeration. See the **PreprocessController()** function description.

Description

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** provides the basic resource allocation services to the PCI bus driver. There is one **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance for each PCI host bridge in a system. The following will typically have only one PCI host bridge:

- Embedded systems
- Desktops
- Workstations
- Most servers

High-end servers may have multiple PCI host bridges. A PCI bus driver that wishes to manage a PCI bus in a system will have to retrieve the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI bus to be managed. A device handle for a PCI host bridge will not contain an

EFI_DEVICE_PATH_PROTOCOL instance because the PCI host bridge is a software abstraction and has no equivalent in the ACPI name space.

All applicable member functions use ACPI 2.0 or ACPI 3.0 resource descriptors to describe resources. Using ACPI resource descriptors does the following:

- Allows other types of resources to be described in the future because they are very generic in nature.
- Avoids multiple structure definitions for describing resources.
- Maintains compatibility with the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** definition.

Only the following two resource descriptor types from the *ACPI Specification* may be used to describe the current resources that are allocated to a PCI root bridge:

- QWORD Address Space Descriptor (*ACPI 3.0*)
- End Tag (*ACPI 3.0)*

The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors, followed by an End Tag. Table 3 and Table 4 below contain these two descriptor types. Table 5 and Table 6 define how resource-specific flags are used. See the *ACPI Specification* for details on the field values.

Table 3. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes, not including the first two fields.
0x03	0x01		Resource type: 0: Memory range 1: I/O range 2: Bus number range
0x04	0x01		General flags. Flags that are common to all resource types: **Bits[7:4]:** Reserved (must be 0) **Bit[3] _MAF:** Always returned as 1 while returning allocated requests to indicate that the specified max address is fixed. **Bit[2] _MIF:** Always returned as 1 while returning allocated requests to indicate that the specified min address is fixed. **Bit[1] _DEC:** Ignored. **Bit[0]:** Ignored.
0x05	0x01		Type-specific flags. Ignored except as defined in Table 3-3 and Table 3-4 below.
0x06	0x08		Address Space Granularity. Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. Ignored for I/O and bus resource requests. Ignored during **`GetProposedResources()`**.
0x0E	0x08		Address Range Minimum. Set to the base of the allocated address range (bus, I/O, memory) during **`GetProposedResources()`**. Ignored during **`SubmitResources()`**.
0x16	0x08		Address Range Maximum. Used to indicate alignment requirement during **`SubmitResources()`** and ignored during **`GetProposedResources()`**. This value must be 2^n-1. The address base must be a multiple of the granularity field. That is, if this field is 4k-1, the allocated address must be a multiple of 4 KB. **Note:** The interpretation of this field is different from the *ACPI Specification* and PCI Root Bridge I/O Protocol.
0x1E	0x08		Address Translation Offset. Used to indicate the allocation status during **`GetProposedResources()`** and ignored during **`SubmitResources()`**. Allocation status is defined in "Related Definitions" in **`GetProposedResources()`**. **Note:** The interpretation of this field is different from the *ACPI Specification* and PCI Root Bridge I/O Protocol.
0x26	0x08		Address Range Length. This field specifies the amount of resources that are requested or allocated in number of bytes.

Table 4. ACPI 2.0 & 3.0 End Tag Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag.
0x01	0x01	0x00	Checksum. Set to 0 to indicate that checksum is to be ignored.

Table 5. I/O Resource Flag (Resource Type = 1) Usage

Bits	Meaning
Bits[7:1]	Ignored.
Bit[0]	_RNG. Ignored during an allocation request. Setting this bit while returning allocated resources means that the I/O allocation must be limited to the ISA I/O ranges. In that case, the PCI bus driver must allocate I/O addresses out of the ISA I/O ranges. The following are the ISA I/O ranges: n100–n3FF n500–n7FF n900–nBFF nD00–nFFF See ISA Aliasing Considerations for more details.

Table 6. Memory Resource Flag (Resource Type = 0) Usage

Bits	Meaning
Bits[7:3]	Ignored.
Bit[2:1]	_MEM. Memory attributes. **Value and Meaning:** 0 The memory is nonprefetchable. 1 Invalid. 2 Invalid. 3 The memory is prefetchable. **Note:** The interpretation of these bits is somewhat different from the *ACPI Specification.* According to the *ACPI Specification,* a value of 0 implies noncacheable memory and the value of 3 indicates prefetchable and cacheable memory.
Bit[0]	Ignored.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase()

Summary

These are the notifications from the PCI bus driver that it is about to enter a certain phase of the PCI enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE)
(
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL    *This,
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE        Phase
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

Phase

The phase during enumeration. Type **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This member function can be used to notify the host bridge driver to perform specific actions, including any chipset-specific initialization, so that the chipset is ready to enter the next phase. Nine notification points are defined at this time. See "Related Definitions" below for definitions of various notification points and section 8.7 for usage.

More synchronization points may be added as required in the future.

Related Definitions

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
  EfiPciHostBridgeBeginEnumeration,
  EfiPciHostBridgeBeginBusAllocation,
  EfiPciHostBridgeEndBusAllocation,
  EfiPciHostBridgeBeginResourceAllocation,
  EfiPciHostBridgeAllocateResources,
  EfiPciHostBridgeSetResources,
  EfiPciHostBridgeFreeResources,
  EfiPciHostBridgeEndResourceAllocation,
  EfiPciHostBridgeEndEnumeration,
  EfiMaxPciHostBridgeEnumeratonPhase
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE;
```

Table 7 provides a description of the fields in the above enumeration:

Table 7. Enumeration Descriptions

Enumeration	Description
EfiPciHostBridgeBeginEnumeration	Resets the host bridge PCI apertures and internal data structures. The PCI enumerator should issue this notification before starting a fresh enumeration process. Enumeration cannot be restarted after sending any other notification such as **EfiPciHostBridgeBeginBusAllocation**.
EfiPciHostBridgeBeginBusAllocation	The bus allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeEndBusAllocation	The bus allocation and bus programming phase is complete. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeBeginResourceAllocation	The resource allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeAllocateResources	Allocates resources per previously submitted requests for all the PCI root bridges. These resource settings are returned on the next call to **GetProposedResources()**. Before calling **NotifyPhase()** with a *Phase* of **EfiPciHostBridgeAllocateResource**, the PCI bus enumerator is responsible for gathering I/O and memory requests for all the PCI root bridges and submitting these requests using **SubmitResources()**. This function pads the resource amount to suit the root bridge hardware, takes care of dependencies between the PCI root bridges, and calls the Global Coherency Domain (GCD) with the allocation request. In the case of padding, the allocated range could be bigger than what was requested. Note that the size of the allocated range could be smaller than what was requested. This scenario could happen due to an allocation failure, a host bridge hardware limitation, or any other reason. In that case, the call will return an **EFI_OUT_OF_RESOURCES** error. If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. The PCI bus driver can call **GetProposedResouces()** to find out which of the resource types were partially allocated and the difference between the amount that was requested and the amount that was allocated. The PCI bus enumerator should readjust the requested sizes (by dropping certain PCI devices or PCI buses) to obtain a best fit. The PCI bus driver can call **NotifyPhase (EfiPciHostBridgeFreeResources)** to free up the original assignments and resubmit the adjusted resource requests with **SubmitResources()**.

EfiPciHostBridgeSetResources	Programs the host bridge hardware to decode previously allocated resources (proposed resources) for all the PCI root bridges. After the hardware is programmed, reassigning resources will not be supported. The bus settings are not affected.
EfiPciHostBridgeFreeResources	Deallocates resources that were previously allocated for all the PCI root bridges and resets the I/O and memory apertures to their initial state. The bus settings are not affected. If the request to allocate resources fails, the PCI enumerator can use this notification to deallocate previous resources, adjust the requests, and retry allocation.
EfiPciHostBridgeEndResourceAllocation	The resource allocation phase is completed. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeEndBusEnumeration	The bus enumeration phase is completed. No specific action is required here. This notification can be used to perform any chipset-specific programming.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_INVALID_PARAMETER	The *Phase* is invalid.
EFI_NOT_READY	This phase cannot be entered at this time. For example, this error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if **SubmitResources()** has not been called for one or more PCI root bridges before this call.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. This error is valid for a *Phase* of **EfiPciHostBridgeSetResources**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources. This error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if the previously submitted resource requests cannot be fulfilled or were only partially fulfilled.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetNextRootBridge()

Summary

Returns the device handle of the next PCI root bridge that is associated with this host bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_
BRIDGE) (
  IN     EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN OUT EFI_HANDLE                                        *RootBridgeHandle
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

Returns the device handle of the next PCI root bridge. On input, it holds the *RootBridgeHandle* that was returned by the most recent call to **GetNextRootBridge()**. If *RootBridgeHandle* is **NULL** on input, the handle for the first PCI root bridge is returned. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This function is called multiple times to retrieve the device handles of all the PCI root bridges that are associated with this PCI host bridge. Each PCI host bridge is associated with one or more PCI root bridges. On each call, the handle that was returned by the previous call is passed into the interface, and on output the interface returns the device handle of the next PCI root bridge. The caller can use the handle to obtain the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** for that root bridge. When there are no more PCI root bridges to report, the interface returns **EFI_NOT_FOUND**. A PCI enumerator must enumerate the PCI root bridges in the order that they are returned by this function.

The search is initiated by passing in a **NULL** device handle as input. Some of the member functions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** operate on a PCI root bridge and expect the *RootBridgeHandle* as an input.

There is no requirement that this function return the root bridges in any specific relation with the EFI device paths of the root bridges.

This function can also be used to determine the number of PCI root bridges that were produced by this PCI host bridge. The host bridge hardware may provide mechanisms to change the number of

root bridges that it produces, but such changes must be completed before the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not an **EFI_HANDLE** that was returned on a previous call to **GetNextRootBridge()**.
EFI_NOT_FOUND	There are no more PCI root bridge device handles.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()

Summary

Returns the allocation attributes of a PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_GET_ATTRIBUTES) (
 IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
 IN  EFI_HANDLE                                       RootBridgeHandle,
 OUT UINT64                                           *Attributes
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The device handle of the PCI root bridge in which the caller is interested. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Attributes

The pointer to attributes of the PCI root bridge. The permitted attribute values are defined in "Related Definitions" below.

Description

The function returns the allocation attributes of a specific PCI root bridge. The attributes can vary from one PCI root bridge to another. These attributes are different from the decode-related attributes that are returned by the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()** member function. The *RootBridgeHandle* parameter is used to specify the instance of the PCI root bridge. The device handles of all the root bridges that are associated with this host bridge must be obtained by calling **GetNextRootBridge()**. The attributes are static in the sense that they do not change during or after the enumeration process. The hardware may provide mechanisms to change the attributes on the fly, but such changes must be completed before **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed. The permitted values of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES** are defined in "Related Definitions" below. The caller uses these attributes to combine multiple resource requests. For example, if the flag **EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM** is set, the PCI bus enumerator needs to include requests for the prefetchable memory in the nonprefetchable memory pool and not request any prefetchable memory.

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES
//*******************************************************

#define EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM       1
#define EFI_PCI_HOST_BRIDGE_MEM64_DECODE           2
```

Following is a description of the fields in the above definition:

Table 8. EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES field descriptions

EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM	If this bit is set, then the PCI root bridge does not support separate windows for nonprefetchable and prefetchable memory. A PCI bus driver needs to include requests for prefetchable memory in the nonprefetchable memory pool.
EFI_PCI_HOST_BRIDGE_MEM64_DECODE	If this bit is set, then the PCI root bridge supports 64-bit memory windows. If this bit is not set, the PCI bus driver needs to include requests for a 64-bit memory address in the corresponding 32-bit memory pool.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Attributes* is **NULL**.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration()

Summary

Sets up the specified PCI root bridge for the bus enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUMERATION) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                        RootBridgeHandle,
  OUT VOID                                             **Configuration
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge to be set up. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

Pointer to the pointer to the PCI bus resource descriptor.

Description

This member function sets up the root bridge for bus enumeration and returns the PCI bus range over which the search should be performed in ACPI (2.0 & 3.0) resource descriptor format. The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **StartBusEnumeration()**.

Table 9. ACPI 2.0 & 3.0 Resource Descriptor Field Values for StartBusEnumeration()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be scanned.
Address Range Length	Set to the number of PCI buses that may be scanned. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

This function cannot return resource descriptors for anything other than bus resources. This function can be used to prevent a PCI bus driver from scanning certain PCI buses to work around a chipset limitation. Because the size of ACPI resource descriptors is not fixed, **`StartBusEnumeration()`** is responsible for allocating memory for the buffer *`Configuration`*.

The PCI segment is implicit and is identified by the *`SegmentNumber`* field in the instance of the **`EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL`** that is installed on the PCI root bridge handle *`RootBridgeHandle`*.

Status Codes Returned

EFI_SUCCESS	The PCI root bridge was set up and the bus range was returned in *`Configuration`*.
EFI_INVALID_PARAMETER	*`RootBridgeHandle`* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()

Summary

Programs the PCI root bridge hardware so that it decodes the specified PCI bus range.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN EFI_HANDLE                                       RootBridgeHandle,
  IN VOID                                             *Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose bus range is to be programmed. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the PCI bus resource descriptor.

Description

This member function programs the specified PCI root bridge to decode the bus range that is specified by the input parameter *Configuration*.

The bus range information is specified in terms of the ACPI (2.0 & 3.0) resource descriptor format. The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **SetBusNumbers()**.

Table 10. ACPI 2.0 & 3.0 Resource Descriptor Field Values for SetBusNumbers()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be decoded.
Address Range Length	Set to the number of PCI buses that should be decoded. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

This call will return **EFI_INVALID_PARAMETER** without programming the hardware if either of the following are specified:

- Any descriptors other than bus type descriptors
- Any invalid descriptors

The bus range is typically a subset of what was returned during **StartBusEnumeration()**. If **SetBusNumbers()** is called with incorrect (but valid) parameters, it may cause system failure.

The PCI segment is implicit and is identified by the *SegmentNumber* field in the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the PCI root bridge handle *RootBridgeHandle*. This call cannot alter the following:

- The *SegmentNumber* field in the corresponding instances of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
- The segment number settings in the hardware

The caller is responsible for allocating and deallocating a buffer to hold *Configuration*. If the call returns **EFI_DEVICE_ERROR**, the PCI bus enumerator can optionally attempt another bus setting.

Status Codes Returned

EFI_SUCCESS	The bus range for the PCI root bridge was programmed.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Configuration* is **NULL**.
EFI_INVALID_PARAMETER	*Configuration* does not point to a valid ACPI (2.0 & 3.0) resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* does not include a valid ACPI 2.0 bus resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* includes valid ACPI (2.0 & 3.0) resource descriptors other than bus descriptors.
EFI_INVALID_PARAMETER	*Configuration* contains one or more invalid ACPI resource descriptors.
EFI_INVALID_PARAMETER	"Address Range Minimum" is invalid for this root bridge.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this root bridge.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()

Summary

Submits the I/O and memory resource requirements for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCES) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                        RootBridgeHandle,
  IN  VOID                                             *Configuration
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose I/O and memory resource requirements are being submitted. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the PCI I/O and PCI memory resource descriptor.

Description

This function is used to submit all the I/O and memory resources that are required by the specified PCI root bridge. The input parameter *Configuration* is used to specify the following:

- The various types of resources that are required
- The associated lengths in terms of ACPI (2.0 & 3.0) resource descriptor format

The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **SubmitResources()**.

Table 11. ACPI 2.0& 3.0 Resource Descriptor Field Values for SubmitResources()

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. All other values result in this function returning the error code of **EFI_INVALID_PARAMETER**.
Address Range Maximum	Used to specify the alignment requirement. If "Address Range Maximum" is of the form 2^n-1, this member function returns the error code **EFI_INVALID_PARAMETER**. The address base must be a multiple of the granularity field. That is, if this field is 4 KB-1, the allocated address must be a multiple of 4 KB.
Address Range Minimum	Ignored.
Address Translation Offset	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

The caller must ask for appropriate alignment using the "Address Range Maximum" field. The caller is responsible for allocating and deallocating a buffer to hold *Configuration*.

It is considered an error if no resource requests are submitted for a PCI root bridge. If a PCI root bridge does not require any resources, a zero-length resource request must explicitly be submitted.

If the *Configuration* includes one or more invalid resource descriptors, all the resource descriptors are ignored and the function returns **EFI_INVALID_PARAMETER**.

Status Codes Returned

EFI_SUCCESS	The I/O and memory resource requests for a PCI root bridge were accepted.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Configuration* is **NULL**.
EFI_INVALID_PARAMETER	*Configuration* does not point to a valid ACPI (2.0 & 3.0) resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* includes requests for one or more resource types that are not supported by this PCI root bridge. This error will happen if the caller did not combine resources according to *Attributes* that were returned by **GetAllocAttributes()**.
EFI_INVALID_PARAMETER	"Address Range Maximum" is invalid.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	"Address Space Granularity" is invalid for this PCI root bridge.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetProposedResources()

Summary

Returns the proposed resource settings for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RESOURCES) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT VOID                                              **Configuration
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the pointer to the PCI I/O and memory resource descriptor.

Description

This member function returns the proposed resource settings for the specified PCI root bridge. The proposed resource settings are prepared when **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeAllocateResources.** The output parameter *Configuration* specifies the following:

- The various types of resources, excluding bus resources, that are allocated
- The associated lengths in terms of ACPI (2.0 & 3.0) resource descriptor format

The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **GetProposedResources()**.

Table 12. ACPI 2.0 & 3.0 Resource Descriptor Field Values for GetProposedResources()

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Ignored.
Address Range Minimum	Indicates the starting address of the allocated ranges.
Address Translation Offset	Indicates the allocation status. Allocation status is defined in "Related Definitions" below.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

The callee is responsible for allocating a buffer to hold *Configuration* because the caller does not know the number of descriptors that are required. The caller is also responsible for deallocating the buffer.

If **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeAllocateResources** and returns **EFI_OUT_OF_RESOURCES**, the PCI bus enumerator may use **GetProposedResources()** to retrieve the proposed settings. The **EFI_OUT_OF_RESOURCES** error status indicates that one or more requests could not be fulfilled or were partially fulfilled. Additional details of the allocation status for each type of resource can be retrieved from the "Address Translation Offset" field in the resource descriptor that was returned by this function; also see "Related Definitions" below for defined allocation status values. This error could happen for the following reasons:

- Allocation failure
- A limitation in the host bridge hardware
- Any other reason

If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. In that case, the PCI bus enumerator may choose to readjust the requested sizes (by dropping certain devices or PCI buses) to obtain a best fit. The PCI bus driver calls **NotifyPhase()** with a *Phase* of **EfiPciHostBridgeFreeResources** to free the original assignments.

If this member function is able to only partially fulfill the requests for one or more resource types, the root bridges that are first in the list will get resources first. The ordering of the root bridges is determined by the output of **GetNextRootBridge()**. The handle to the first root bridge is obtained by calling **GetNextRootBridge()** with an input handle of **NULL**.

In the case of I/O resources, the PCI bus enumerator must check the _RNG flag. If this flag is set, the I/O ranges that are allocated to the devices must come from the non-ISA I/O subset.

For example, if this flag is set, the "Address Range Minimum" is 0x1000, and the "Address Range Length" is 0x1000, then the following I/O ranges can be allocated to PCI devices:

- 0x1000–0x10FF
- 0x1400–0x14FF

- 0x1800–0x18FF
- 0x1C00–0x1CFF

This call is made before **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeSetResources**. After that time, the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function should be used to obtain the resources that were consumed by a particular PCI root bridge.

Related Definitions

```
//
*******************************************************************
// EFI_RESOURCE_ALLOCATION_STATUS
//
*******************************************************************
typedef  UINT64    EFI_RESOURCE_ALLOCATION_STATUS;

#define EFI_RESOURCE_SATISFIED                    0
#define EFI_RESOURCE_NOT_SATISFIED                (UINT64) -1
```

Following is a description of the fields in the above definition. All other values indicate that the request of this resource type could be partially fulfilled. The exact value indicates how much more space is still required to fulfill the requirement.

Table 13. EFI_RESOURCE_ALLOCATION_STATUS field descriptions

EFI_RESOURCE_SATISFIED	The request of this resource type could be fulfilled.
EFI_RESOURCE_NOT_SATISFIED	The request of this resource type could not be fulfilled for its absence in the host bridge resource pool.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController()

Summary

Provides the hooks from the PCI bus driver to every PCI controller (device/function) at various stages of the PCI enumeration process that allow the host bridge driver to preinitialize individual PCI controllers before enumeration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONTROLLER)(
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN  EFI_HANDLE                                       RootBridgeHandle,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS      PciAddress,
  IN  EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE     Phase
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The associated PCI root bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** member functions to access the PCI configuration space of the device. See *UEFI 2.1 Specification* for the definition of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**.

Phase

The phase of the PCI device enumeration. Type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This function is called during the PCI enumeration process. No specific action is expected from this member function. It allows the host bridge driver to preinitialize individual PCI controllers before enumeration.

The parameter *RootBridgeHandle* can be used to locate the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the root bridge that is the parent of

the specific PCI function. The parameter *PciAddress* can be passed to the **Pci.Read()** and **Pci.Write()** functions of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance to access the PCI configuration space of the specific PCI function.

This member function is invoked during PCI enumeration and before the PCI enumerator has created a handle for the PCI function. As a result, the **EFI_PCI_IO_PROTOCOL** cannot be used at this point.

Two notification points are defined at this time. See type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** in "Related Definitions" below for definitions of these notification points and ISA Aliasing Considerations for usage. More synchronization points may be added as required in the future.

Related Definitions

```
//*******************************************************
// EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
 EfiPciBeforeChildBusEnumeration,
 EfiPciBeforeResourceCollection
} EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE;
```

Following is a description of the fields in the above enumeration:

Table 14. EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE field descriptions

EfiPciBeforeChildBusEnumeration	This notification is applicable only to PCI-to-PCI bridges and indicates that the PCI enumerator is about to begin enumerating the bus behind the PCI-to-PCI bridge. This notification is sent after the primary bus number, the secondary bus number, and the subordinate bus number registers in the PCI-to-PCI bridge are programmed to valid (but not necessary final) values. Programming of the bus number register allows the chipset code to scan devices on the bus that are immediately behind the PCI-to-PCI bridge. This notification can be used to reset the secondary PCI bus. Some PCI-to-PCI bridges can drive their secondary bus at various clock speeds (33 MHz or 66 MHz, for example) and support PCI-X* or conventional PCI mode. These bridges must be set up to operate at the correct speed and correct mode before the downstream devices and buses are enumerated. This notification can be used to perform that activity. The host bridge code cannot reprogram the bus numbers in the PCI-to-PCI bridge or reprogram any upstream devices during this notification. It can touch the downstream devices because the PCI enumerator has not found these devices. If there are multiple PCI-to-PCI bridges on the same PCI bus, the order in which the notification is sent to these bridges is implementation specific. On the other hand, it is guaranteed that a PCI-to-PCI bridge will see this notification before the downstream bridge receives this notification or its child devices receive the **EfiPciBeforeResourceCollection** notification.
EfiPciBeforeResourceCollection	This notification is sent before the PCI enumerator probes the Base Address Register (BAR) registers for every valid PCI function. This notification can be used to program the backside registers that determine the BAR size or any other programming such as the master latency timer, cache line size, and PERR and SERR control. This notification is sent regardless of whether the function implements BAR or not. In the case of a multifunction device, this notification is sent for every function of the device. The order within the functions is not specified. The order in which this notification is sent to various devices/functions on the same bus is implementation specific.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Phase* is not a valid phase that is defined in **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE**.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. The PCI enumerator should not enumerate this device, including its child devices if it is a PCI-to-PCI bridge.

9
PCI Platform

9.1 Introduction

This section contains the basic definitions of protocols that provide PCI platform support. The following protocols are defined in this section:

EFI_PCI_PLATFORM_PROTOCOL
EFI_PCI_OVERRIDE_PROTOCOL
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

EFI_PCI_EXECUTION_PHASE
EFI_PCI_PLATFORM_POLICY

9.2 PCI Platform Overview

This section defines the core code and services that are required for an implementation of the following protocols in this specification:

- PCI Platform Protocol
- PCI Override Protocol
- Incompatible PCI Device Support Protocol

The PCI Platform Protocol allows a PCI bus driver to obtain the platform policy and call a platform driver at various points in the enumeration phase. The Incompatible PCI Device Support Protocol allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification.*

This specification does the following:

- Describes the basic components of the PCI Platform Protocol
- Describes the basic components of the Incompatible PCI Device Support Protocol and how firmware configures incompatible PCI devices
- Provides code definitions for the PCI Platform Protocol, the Incompatible PCI Device Support Protocol, and their related type definitions that are architecturally required by this specification.

This document is intended for the following readers:

- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in IntelÆ architecture-based products.

Readers of this specification are assumed to have solid knowledge of the *UEFI 2.1 Specification*

9.3 PCI Platform Support Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

9.3.1 Industry Specifications

- *Advanced Configuration and Power Interface Specification* (hereafter referred to as the *ACPI Specification*), version 3.0.

9.3.2 PCI Specifications

- *Conventional PCI Specification,* version 3.0: http://www.pcisig.com*
- *PCI-to-PCI Bridge Architecture Specification,* revision 1.2: http://www.pcisig.com*
- *PCI-to-PCI Bridges and CardBus Controllers on Windows 2000, Windows XP, and Windows Server 2003:* http://www.microsoft.com/whdc/system/bus/PCI/pcibridge-cardbus.mspx*

9.4 PCI Platform Protocol

9.4.1 PCI Platform Protocol Overview

"PCI Host Bridge Resource Allocation Protocol", Section 8.8.2 defines and describes the PCI Host Bridge Resource Allocation Protocol. The PCI Host Bridge Resource Allocation Protocol driver provides chipset-specific functionality that works across processor architectures and unique platform features. It does not address issues where an implementation varies across platforms.

In contrast, the PCI Override Protocol and PCI Platform Protocol provide interfaces allow a platform driver or codebase driver to perform platform-specific actions. For example:

- Allow a PCI bus driver to obtain platform policy. The platform can use this protocol to control whether the PCI bus driver reserves I/O ranges for ISA aliases and VGA aliases. The default policy for the PCI bus driver is to reserve I/O ranges for both ISA aliases and VGA aliases, which may result in a large amount of I/O space being unavailable for PCI devices. This protocol allows the platform driver to change this policy.
- Call a platform driver at various points in the enumeration phase. The platform driver can use these hooks to perform various platform-specific activities. Examples of such activities include but are not limited to the following:
- **`PlatformPrepController()`** can be used to program the PCI subsystem vendor ID and device ID into onboard and chipset devices.
- **`PlatformPrepController()`** and **`PlatformNotify()`** can be used for implementing hardware workarounds.
- **`PlatformPrepController()`** can be used for preprogramming any backside registers that control the Base Address Register (BAR) window sizes.
- **`PlatformPrepController()`** can be used to set PCI or PCI-X* bus speeds for PCI bridges that support multiple bus speeds.

- Allow PCI option ROMs to be stored in local storage. The platform can store PCI option ROMs in local storage (e.g., a firmware volume) and report their existence to the PCI bus driver using the **GetPciRom()** member function. Option ROMs for embedded PCI controllers are often stored in a platform-specific location. The same member function can be used to override the default PCI ROM on an add-in card with one from platform-specific storage.

A platform should implement this protocol if any of the functionality that is listed above is required.

See Code Definitions for the definition of **EFI_PCI_PLATFORM_PROTOCOL** and the member functions listed above. See Section 8.8.2 for additional PCI-related design discussion.

9.5 Incompatible PCI Device Support Protocol

9.5.1 Incompatible PCI Device Support Protocol Overview

Some PCI devices do not fully comply with the PCI Specification. For example, a PCI device may request that its I/O Base Address Register (BAR) be placed on a 0x200 boundary even though it is requesting an I/O with a length of 0x100. The Incompatible PCI Device Support Protocol allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification.*

In the PI Architecture, the platform-specific PCI host bridge driver works with the generic, standard PCI bus driver to configure the entire PCI subsystem. Even though the exact configuration is up to individual incompatible devices, it is a platform choice to support those incompatible PCI devices. For example, one platform may not want to support those incompatible devices while another platform appears more tolerant of those devices.

See Code Definitions for the definition of the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**.

9.5.2 Usage Model for the Incompatible PCI Device Support Protocol

The following describes the usage model for the Incompatible PCI Device Support Protocol:

1. The PCI bus driver locates **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**. If the PCI bus driver cannot find this protocol, simply follow the regular PCI enumeration path. Otherwise, go to step 2.
2. For each PCI device that was detected, the PCI bus driver begins collecting the required PCI resources by probing the Base Address Register (BAR) for each device.
3. For each device, call **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()** to check whether this PCI device is an incompatible device. If this device in not an incompatible device, go to step 5.
4. Use the *Configuration* that is returned by **CheckDevice()** to override or modify the original PCI resource requirements.
5. Follow the normal PCI enumeration process.

9.6 PCI Code Definitions

This section contains the basic definitions of protocols that provide PCI platform support. The following protocols are defined in this section:

- **EFI_PCI_PLATFORM_PROTOCOL**
- **EFI_PCI_OVERRIDE_PROTOCOL**
- **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

EFI_PCI_CHIPSET_EXECUTION_PHASE

EFI_PCI_PLATFORM_POLICY

9.6.1 PCI Platform Protocol

EFI_PCI_PLATFORM_PROTOCOL

Summary

This protocol provides the interface between the PCI bus driver/PCI Host Bridge Resource Allocation driver and a platform-specific driver to describe the unique features of a platform. This protocol is optional.

GUID

```
#define EFI_PCI_PLATFORM_PROTOCOL_GUID \
  { 0x7d75280, 0x27d4, 0x4d69, 0x90, 0xd0, 0x56, 0x43, \
  0xe2, 0x38, 0xb3, 0x41 }
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_PLATFORM_PROTOCOL {
  EFI_PCI_PLATFORM_PHASE_NOTIFY           PlatformNotify;
  EFI_PCI_PLATFORM_PREPROCESS_CONTROLLER PlatformPrepController;
  EFI_PCI_PLATFORM_GET_PLATFORM_POLICY    GetPlatformPolicy;
  EFI_PCI_PLATFORM_GET_PCI_ROM            GetPciRom;
} EFI_PCI_PLATFORM_PROTOCOL;
```

Parameters

PlatformNotify

The notification from the PCI bus enumerator to the platform that it is about to enter a certain phase during the enumeration process. See the **PlatformNotify()** function description.

PlatformPrepController

The notification from the PCI bus enumerator to the platform for each PCI controller at several predefined points during PCI controller initialization. See the **PlatformPrepController()** function description.

GetPlatformPolicy

Retrieves the platform policy regarding enumeration. See the **GetPlatformPolicy()** function description.

GetPciRom

Gets the PCI device's option ROM from a platform-specific location. See the **GetPciRom()** function description.

Description

The **EFI_PCI_PLATFORM_PROTOCOL** is published by a platform-aware driver. This protocol is optional; see PCI Platform Protocol Overview in Design Discussion for scenarios in which this protocol is required. There cannot be more than one instance of this protocol in the system.

If the PCI bus driver detects the presence of this protocol before enumeration, it will use the PCI Platform Protocol to obtain information about the platform policy. The PCI bus driver will use this protocol to get the PCI device's option ROM from a platform-specific location in storage. It will also call the various member functions of this protocol at predefined points during PCI bus enumeration. The member functions can be used for performing any platform-specific initialization that is appropriate during the particular phase.

EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PHASE_NOTIFY) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL                  *This,
  IN  EFI_HANDLE                                       HostBridge,
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE    Phase,
  IN  EFI_PCI EXECUTION_PHASE                          ExecPhase
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

HostBridge

The handle of the host bridge controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

Phase

The phase of the PCI bus enumeration. Type **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE** is defined in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase()**.

ExecPhase

Defines the execution phase of the PCI chipset driver. Type **EFI_PCI_EXECUTION_PHASE** is defined in "Related Definitions" below.

Description

The **PlatformNotify()** function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Several notification points are defined at this time. More notification points may be added as required in the future. The function should return **EFI_UNSUPPORTED** for any value of Phase that that the function does not support.

The PCI bus driver calls this function twice for every Phase-once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.

This member function may not perform any error checking on the input parameters. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Related Definitions

```
//***************************************************************
// EFI_PCI_EXECUTION_PHASE
//***************************************************************
typedef enum {
  BeforePciHostBridge = 0,
  ChipsetEntry        = 0,
  AfterPciHostBridge  = 1,
  ChipsetExit         = 1,
  MaximumExecutionPhase
} EFI_PCI_EXECUTION_PHASE;

typedef EFI_PCI_EXECUTION_PHASE EFI_PCI_CHIPSET_EXECUTION_PHASE;
```

Note: **EFI_PCI_EXECUTION_PHASE** *is used to call a platform protocol and execute platform-specific code. Following is a description of the fields in the above enumeration.*

BeforePciHostBridge

The phase that indicates the entry point to the PCI Bus Notify phase. This platform hook is called before the PCI bus driver calls the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** driver.

AfterPciHostBridge

The phase that indicates the exit point to the PCI Bus Notify phase before returning to the PCI Bus Driver Notify phase. This platform hook is called after the PCI bus driver calls the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** driver.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_UNSUPPORTED	The function does not support the phase specified by *Phase*.

EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController()

Summary

The platform driver receives notifications from the PCI bus enumerator at various phases during PCI controller initialization, just like the PCI host bridge driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PREPROCESS_CONTROLLER) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL                *This,
  IN  EFI_HANDLE                                     HostBridge,
  IN  EFI_HANDLE                                  RootBridge,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS    PciAddress,
  IN  EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE   Phase,
  IN  EFI_PCI_EXECUTION_PHASE                        ExecPhase
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

HostBridge

The associated PCI host bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

RootBridge

The associated PCI root bridge handle.

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** functions to access the PCI configuration space of the device. See the *UEFI 2.1 Specification* for the definition of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**.

Phase

The phase of the PCI controller enumeration. Type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** is defined in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PrepareProcessController()**.

ExecPhase

Defines the execution phase of the PCI chipset driver. Type **EFI_PCI_CHIPSET_EXECUTION_PHASE** is defined in **EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()**.

Description

The **`PlatformPrepController()`** function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Several notification points are defined at this time. More synchronization points may be added as required in the future. The function should return EFI_UNSUPPORTED for any value of Phase that that the function does not support.

The PCI bus driver calls the platform driver twice for every PCI controller—once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.

This member function may not perform any error checking on the input parameters. It also does not return any error codes. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

EFI_PCI_PLATFORM_PROTOCOL.GetPlatformPolicy()

Summary

The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve platform policies regarding PCI enumeration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_GET_PLATFORM_POLICY) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL            *This,
  OUT EFI_PCI_PLATFORM_POLICY                    *PciPolicy,
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

PciPolicy

The platform policy with respect to VGA and ISA aliasing. Type **EFI_PCI_PLATFORM_POLICY** is defined in "Related Definitions" below.

Description

The **GetPlatformPolicy()** function retrieves the platform policy regarding PCI enumeration. The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve the policy.

The **EFI_PCI_IO_PROTOCOL.Attributes()** function allows a PCI device driver to ask for various legacy ranges. Because PCI device drivers run after PCI enumeration, a request for legacy allocation comes in after PCI enumeration. The only practical way to guarantee that such a request from a PCI device driver will be fulfilled is to preallocate these ranges during enumeration. The PCI bus enumerator does not know which legacy ranges may be requested and therefore must rely on **GetPlatformPolicy()**. The data that is returned by **GetPlatformPolicy()** determines the supported attributes that are returned by the **EFI_PCI_IO_PROTOCOL.Attributes()** function.

See "Related Definitions" below for a description of the output parameter *PciPolicy*. For example, the platform can decide if it wishes to support devices that require ISA aliases using this parameter. Note that the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()** function returns the attributes that the root bridge hardware supports and does not depend upon preallocations.

Related Definitions

```
typedef UINT32 EFI_PCI_PLATFORM_POLICY;
```

EFI_PCI_PLATYFORM_POLICY is a bitmask with the following legal combinations.

```
#define  EFI_RESERVE_NONE_IO_ALIAS        0x0000
#define  EFI_RESERVE_ISA_IO_ALIAS         0x0001
#define  EFI_RESERVE_ISA_IO_NO_ALIAS      0x0002
#define  EFI_RESERVE_VGA_IO_ALIAS         0x0004
#define  EFI_RESERVE_VGA_IO_NO_ALIAS      0x0008
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_UNSUPPORTED	The function is not supported.
EFI_INVALID_PARAMETER	*PciPolicy* is **NULL**.

EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()

Summary

Gets the PCI device's option ROM from a platform-specific location.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_GET_PCI_ROM) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL        *This,
  IN  EFI_HANDLE                             PciHandle,
  OUT VOID                                   **RomImage,
  OUT UINTN                                  *RomSize
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

PciHandle

The handle of the PCI device. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

RomImage

If the call succeeds, the pointer to the pointer to the option ROM image. Otherwise, this field is undefined. The memory for RomImage is allocated by **EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()** using the UEFI Boot Service **AllocatePool()**. It is the caller's responsibility to free the memory using the UEFI Boot Service **FreePool()**, when the caller is done with the option ROM.

RomSize

If the call succeeds, a pointer to the size of the option ROM size. Otherwise, this field is undefined.

Description

The **GetPciRom()** function gets the PCI device's option ROM from a platform-specific location. The option ROM will be loaded into memory. This member function is used to return an image that is packaged as a PCI 2.2 option ROM. The image may contain both legacy and UEFI option ROMs. See the *UEFI 2.1 Specification* for details. This member function can be used to return option ROM images for embedded controllers. Option ROMs for embedded controllers are typically stored in platform-specific storage, and this member function can retrieve it from that storage and return it to the PCI bus driver. The PCI bus driver will call this member function before scanning the ROM that is attached to any controller, which allows a platform to specify a ROM image that is different from the ROM image on a PCI card.

Status Codes Returned

EFI_SUCCESS	The option ROM was available for this device and loaded into memory.
EFI_NOT_FOUND	No option ROM was available for this device.
EFI_OUT_OF_RESOURCES	No memory was available to load the option ROM.
EFI_DEVICE_ERROR	An error occurred in getting the option ROM.

9.6.2 PCI Override Protocol

EFI_PCI_OVERRIDE_PROTOCOL

Summary

This protocol provides the interface between the PCI bus driver/PCI Host Bridge Resource Allocation driver and an implementation's driver to describe the unique features of a platform. This protocol is optional.

GUID

```
#define EFI_PCI_OVERRIDE_GUID \
  { 0xb5b35764, 0x460c, 0x4a06, { 0x99, 0xfc, 0x77, 0xa1,
  0x7c, 0x1b, 0x5c, 0xeb } }
```

Protocol Interface Structure

```
typedef EFI_PCI_PLATFORM_PROTOCOL EFI_PCI_OVERRIDE_PROTOCOL;
```

Description

The **EFI_PCI_OVERRIDE_PROTOCOL** is published by an implementation aware driver. This protocol is optional. But it must be called, if present, during PCI enumeration. There cannot be more than one instance of this protocol in the system.

If the PCI bus driver detects the presence of this protocol before bus enumeration, it will use the PCI Override Protocol to obtain information about the platform policy. If the PCI Platform Protocol does not exist or returns an error, then this protocol is called.

The PCI bus driver will use this protocol to get the PCI device's option ROM from an implementation-specific location in storage. If the PCI Platform Protocol does not exist or returns an error, then this function is called.

It will also call the various member functions of this protocol at predefined points during PCI bus enumeration. The member functions can be used for performing any implementation-specific initialization that is appropriate during the particular phase.

9.6.3 Incompatible PCI Device Support Protocol

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL

Summary

Allows the PCI bus driver to support resource allocation for some PCI devices that do not comply with the PCI Specification.

Note: *This protocol is optional. Only those platforms that implement this protocol will have the capability to support incompatible PCI devices. The absence of this protocol can cause the PCI bus driver to configure these incompatible PCI devices incorrectly. As a result, these devices may not work properly.*

GUID

```
#define EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL_GUID \
  {0xeb23f55a, 0x7863, 0x4ac2, 0x8d, 0x3d, 0x95, 0x65, \
  0x35, 0xde, 0x3, 0x75}
```

Protocol Interface Structure

```
typedef struct _EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL {
  EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_CHECK_DEVICE  CheckDevice;
} EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL;
```

Parameters

CheckDevice

Returns a list of ACPI resource descriptors that detail any special resource configuration requirements if the specified device is a recognized incompatible PCI device. See the **CheckDevice()** function description.

Description

The **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** is used by the PCI bus driver to support resource allocation for some PCI devices that do not comply with the *PCI Specification*. This protocol can find some incompatible PCI devices and report their special resource requirements to the PCI bus driver. The generic PCI bus driver does not have prior knowledge of any incompatible PCI devices. It interfaces with the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** to find out if a device is incompatible and to obtain the special configuration requirements for a specific incompatible PCI device.

This protocol is optional, and only one instance of this protocol can be present in the system. If a platform supports this protocol, this protocol is produced by a Driver Execution Environment (DXE) driver and must be made available before the Boot Device Selection (BDS) phase. The PCI bus driver will look for the presence of this protocol before it begins PCI enumeration.

If this protocol exists in a platform, it indicates that the platform has the capability to support those incompatible PCI devices. However, final support for incompatible PCI devices still depends on the implementation of the PCI bus driver. The PCI bus driver may fully, partially, or not even support these incompatible devices.

During PCI bus enumeration, the PCI bus driver will probe the PCI Base Address Registers (BARs) for each PCI device—regardless of whether the PCI device is incompatible or not—to determine the resource requirements so that the PCI bus driver can invoke the proper PCI resources for them. Generally, this resource information includes the following:

- Resource type
- Resource length
- Alignment

However, some incompatible PCI devices may have special requirements. As a result, the length or the alignment that is derived through BAR probing may not be exactly the same as the actual resource requirement of the device. For example, there are some devices that request I/O resources at a length of 0x100 from their I/O BAR, but these incompatible devices will never work correctly if an odd I/O base address, such as 0x100, 0x300, or 0x500, is assigned to the BAR. Instead, these devices request an even base address, such as 0x200 or 0x400. The Incompatible PCI Device Support Protocol can then be used to obtain these special resource requirements for these incompatible PCI devices. In this way, the PCI bus driver will take special consideration for these devices during PCI resource allocation to ensure that they can work correctly.

This protocol may support the following incompatible PCI BAR types:

- I/O or memory length that is different from what the BAR reports
- I/O or memory alignment that is different from what the BAR reports
- Fixed I/O or memory base address

See the *Conventional PCI Specification 3.0* for the details of how a PCI BAR reports the resource length and the alignment that it requires.

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()

Summary

Returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_CHECK_DEVICE) (
  IN  EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL *This,
  IN  UINTN                                        VendorId,
  IN  UINTN                                        DeviceId,
  IN  UINTN                                        RevisionId,
  IN  UINTN                                        SubsystemVendorId,
  IN  UINTN                                        SubsystemDeviceId,
  OUT VOID                                         **Configuration
  );
```

Parameters

This

Pointer to the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** instance.

VendorID

A unique ID to identify the manufacturer of the PCI device. See the *Conventional PCI Specification 3.0* for details.

DeviceID

A unique ID to identify the particular PCI device. See the *Conventional PCI Specification 3.0* for details.

RevisionID

A PCI device-specific revision identifier. See the *Conventional PCI Specification 3.0* for details.

SubsystemVendorId

Specifies the subsystem vendor ID. See the *Conventional PCI Specification 3.0* for details.

SubsystemDeviceId

Specifies the subsystem device ID. See the *Conventional PCI Specification* 3.0 for details.

Configuration

A list of ACPI resource descriptors that detail the configuration requirement. See Table 15 in the "Description" subsection below for the definition.

Description

The **CheckDevice()** function returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prior to bus enumeration, the PCI bus driver will look for the presence of the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**. Only one instance of this protocol can be present in the system. For each PCI device that the PCI bus driver discovers, the PCI bus driver calls this function with the device's vendor ID, device ID, revision ID, subsystem vendor ID, and subsystem device ID. If the *VendorId*, *DeviceId*, *RevisionId*, *SubsystemVendorId*, or *SubsystemDeviceId* value is set to **(UINTN)-1**, that field will be ignored. The ID values that are not **(UINTN)-1** will be used to identify the current device.

This function will only return **EFI_SUCCESS**. However, if the device is an incompatible PCI device, a list of ACPI resource descriptors will be returned in *Configuration*. Otherwise, **NULL** will be returned in *Configuration* instead. The PCI bus driver does not need to allocate memory for *Configuration*. However, it is the PCI bus driver's responsibility to free it. The PCI bus driver then can configure this device with the information that is derived from this list of resource nodes, rather than the result of BAR probing.

Only the following two resource descriptor types from the *ACPI Specification* may be used to describe the incompatible PCI device resource requirements:

- QWORD Address Space Descriptor (ACPI 2.0, section 6.4.3.5.1; also ACPI 3.0)
- End Tag (ACPI 2.0, section 6.4.2.8; also ACPI 3.0)

The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors, followed by an End Tag. Table 15 and Table 16 below contain these two descriptor types. See the *ACPI Specification* for details on the field values.

Table 15. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes, not including the first two fields.
0x03	0x01		Resource type: 0: Memory range 1: I/O range Other values will be ignored.
0x04	0x01		General flags. Ignored.
0x05	0x01		Type-specific flags. Ignored.
0x06	0x08		Address Space Granularity. Ignored.
0x16	0x08		Address Range Maximum. Used to convey the alignment information. This value must be 2^n-1. If no special alignment is required for the BAR, it must be 0. Then the alignment will set to `(length-1)`, where the `length` is derived through the BAR probing.
0x1E	0x08		Address Translation Offset. Used to indicate the BAR Index from 0 to 5. Specially, `(UINT64)-1` in this field means all the PCI BARs on the device.
0x26	0x08		Address Range Length. Length of the requested resource. If the device has no special length request, it must be 0. Then the length that was obtained from BAR probing will be applied.

Table 16. ACPI 2.0 & 3.0 End Tag Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag.
0x01	0x01	0x00	Checksum. Set to 0 to indicate that checksum is to be ignored.

Status Codes Returned

EFI_SUCCESS	The function always returns `EFI_SUCCESS`.

10
Hot Plug PCI

10.1 HotPlug PCI Overview

This specification defines the core code and services that are required for an implementation of the Hot-Plug PCI Initialization Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. The same protocol may be used by other buses such as CardBus that support hot plugging. This specification does the following:

- Describes the basic components of the hot-plug PCI subsystem and the Hot-Plug PCI Initialization Protocol
- Provides code definitions for the Hot-Plug PCI Initialization Protocol and the hot-plug-PCI–related type definitions that are architecturally required.

10.2 Hot-Plug PCI Initialization Protocol Introduction

This chapter describes the Hot-Plug PCI Initialization Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. This protocol is generic enough to include PCI-to-CardBus bridges and PCI Express* systems. This protocol abstracts the hot-plug controller initialization and resource padding. This protocol is required on platforms that support PCI Hot Plug* or PCI Express slots. For the purposes of initialization, a CardBus PC Card bus is treated in the same way. This protocol is not required on all other platforms.

This protocol is not intended to support hot plugging of PCI cards during the preboot stage. Separate components can be developed if such support is desired.

See Hot-Plug PCI Initialization Protocol in Code Definitions for the definition of **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL`**.

10.3 Hot-Plug PCI Initialization Protocol Related Information

The following resources are referenced throughout this specification or may be useful to you:

- *Conventional PCI Specification,* revision 3.0: http://www.pcisig.com/*
- *PC Card Standard,* volumes 1, 7, and 8: http://www.pcmcia.org/*
- *PCI Express Base Specification,* revision 1.0a: http://www.pcisig.com/*
- *PCI Hot-Plug Specification,* revision 1.1: http://www.pcisig.com/*
- *PCI Standard Hot-Plug Controller and Subsystem Specification,* revision 1.0: http://www.pcisig.com/*

10.4 Requirements

PI Architecture firmware must support platforms with PCI Hot Plug* slots and PCI Express* Hot Plug slots, as well as CardBus PC Card sockets. In both cases, the user is allowed to plug in new devices or remove existing devices during runtime. PCI Hot Plug slots are controlled by a PCI Hot Plug controller whereas CardBus sockets are controlled by a PCI-to-CardBus bridge. PCI Express Hot Plug slots are controlled by a PCI Express root port or a downstream port in a switch. The term "Hot Plug Controller" (HPC) in this document refers to all of these types of controllers. From the standpoint of initialization, all three are identical and have the same general requirements, as follows:

- The root HPCs may come up uninitialized after system reset. These HPCs must be initialized by the system firmware.
- Every HPC may require resource padding. The padding must be taken into account during PCI enumeration. This scenario is true for conventional PCI, PCI Express, and CardBus PC Cards because they all consume shared system resources (I/O, memory, and bus). These resources are produced by the root PCI bridge.

These general requirements place the following specific requirements on an implementation of the PI Architechture PCI hot plug support:

- PI Architecture firmware must handle root HPCs differently than other regular PCI devices. When a root HPC is initialized, the hot-plug slots or CardBus sockets are enabled and this process may uncover more PCI buses and devices. In that respect, root HPCs are somewhat like PCI bridges. The root HPC initialization process may involve detecting bus type and optimum bus speed. The initialization process may also detect faults and voltage mismatches. The initialization process may be specific to the controller and the platform. At the time of the root HPC initialization, the PCI bus may not be fully initialized and the standard PCI bus-specific protocols are not available. PI Architecture firmware must provide an alternate infrastructure for the initialization code. In other words, the HPC initialization code should not be required to understand the bus numbering scheme and other chipset details.
- PI Architecture firmware must support an unlimited number of HPCs in the system. PI Architecture firmware must support various types of HPCs as long as they follow industry standards or conventions. A mix of various types of HPCs is allowed.
- PI Architecture firmware must support legacy PCI Hot Plug Controllers (PHPCs; class code 0x6, subclass code 0x4) as well as Standard (PCI) Hot Plug Controllers (SHPCs). Other conventional PCI Hot Plug controllers are not supported.
- PI Architecture firmware must be capable of supporting a PHPC that is a child of another PHPC. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that the child PHPC must be initialized without firmware assistance because it is not a root PHPC.
- PI Architecture firmware must be capable of supporting SHPCs on an add-in card. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that such an SHPC must be initialized without firmware assistance because it is not a root PHPC. PI Architecture firmware must also support plug-in CardBus bridges that follow the *CardBus Specification,* which is part of the *PC Card Standard.*

- As stated above, root HPCs may require firmware initialization. PI Architecture firmware must be capable of supporting root HPCs that are initialized by hardware and do not require any firmware initialization.
- A PI Architecture PCI bus enumerator must overallocate resources for PCI Hot Plug buses and CardBus sockets. The amount of overallocation may be platform specific.
- The root HPC initialization process may be time consuming. An SHPC can take as long as 15 seconds to enable power to a hot-plug bus without violating the PCI Special Interest Group (PCI-SIG*) requirements. PI Architecture firmware should be able to initialize multiple HPCs in parallel to reduce boot time. In contrast, CardBus initialization is quick.
- PI Architecture firmware should be able to handle when an HPC fails. PI Architecture firmware should be able to handle an HPC that has been disabled.
- The PCI bus driver in PI Architecture firmware is not required to assume anything that is not in one of the PCI-SIG specifications.
- It must be possible to produce legacy Hot Plug Resource Tables (HPRTs) if necessary. HPRTs are described in the *PCI Standard Hot-Plug Controller and Subsystem Specification.*

10.5 Sample Implementation for a Platform Containing PCI Hot Plug* Slots

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI bus enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. Actual implementations may vary although the relative ordering of events is critical. The activities related to PCI Hot Plug* are underlined. Please note that multiple passes of bus enumeration are required in a system containing PCI Hot Plug slots.

See section 8.3 for definitions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** and its member functions.

If the platform supports PCI Hot Plug, an instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is installed.

The PCI enumeration process begins.

Look for instances of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. If it is not found, all the hot-plug subsystem initialization steps can be skipped. If one exists, create a list of root Hot Plug Controllers (HPCs) by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()**.

Notify the host bridge driver that bus enumeration is about to begin by calling **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase** (EfiPciHostBridgeBeginBusAllocation).

For every PCI root bridge handle, do the following:

1. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration** (This,RootBridgeHandle).

2. Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. See the *UEFI 2.1 Specification* for the definition of the PCI Root Bridge I/O Protocol.
3. Allocate memory to hold resource requirements. These can be two resource descriptors, one to hold bus requirements and another to hold the I/O and memory requirements.
4. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.

   Scan all the devices in the specified bus range and the specified segment, one bus at a time. If the device is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If the device path of a device matches that of a root HPC and it is not a PCI-to-CardBus bridge, it must be initialized by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()** before the bus it controls can be fully enumerated. The PCI bus enumerator determines the PCI address of the PCI Hot Plug Controller (PHPC) and passes it as an input to **InitializeRootHpc()**.
5. Continue to scan devices on that root bridge and start the initialization of all root HPCs.
6. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()** so that the HPCs under initialization are still accessible. SetBusNumbers() cannot affect the PCI addresses of the HPCs.

Wait until all the HPCs that were found on various root bridges in step 5 to complete initialization.

Go back to step 5 for another pass and rescan the PCI buses. For all the root HPCs and the nonroot HPCs, call **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()** to obtain the amount of overallocation and add that amount to the requests from the physical devices. Reprogram the bus numbers by taking into account the bus resource padding information. This action will require calling **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()**. The rescan is not required if there is only one root bridge in the system.

Once the memory resources are allocated and a PCI-to-CardBus bridge is part of the HpcList, it will be initialized.

10.6 Code Definitions

This section contains the basic definitions that are related to PCI Hot Plug*. **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is defined in this section.

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- **EFI_HPC_LOCATION**
- **EFI_HPC_STATE**
- **EFI_HPC_PADDING_ATTRIBUTES**

10.7 Hot-Plug PCI Initialization Protocol

EFI_PCI_HOT_PLUG_INIT_PROTOCOL

Summary

This protocol provides the necessary functionality to initialize the Hot Plug Controllers (HPCs) and the buses that they control. This protocol also provides information regarding resource padding.

Note: *This protocol is required only on platforms that support one or more PCI Hot Plug* slots or CardBus sockets.*

GUID

```
#define EFI_PCI_HOT_PLUG_INIT_PROTOCOL_GUID \
  { 0xaa0e8bc1, 0xdabc, 0x46b0, 0xa8, 0x44, 0x37, 0xb8, \
  0x16, 0x9b, 0x2b, 0xea }
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOT_PLUG_INIT_PROTOCOL {
  EFI_GET_ROOT_HPC_LIST                    GetRootHpcList;
  EFI_INITIALIZE_ROOT_HPC                  InitializeRootHpc;
  EFI_GET_HOT_PLUG_PADDING                 GetResourcePadding;
} EFI_PCI_HOT_PLUG_INIT_PROTOCOL;
```

Parameters

GetRootHpcList

Returns a list of root HPCs and the buses that they control. See the **GetRootHpcList()** function description.

InitializeRootHpc

Initializes the specified root HPC. See the **InitializeRootHpc()** function description.

GetResourcePadding

Returns the resource padding that is required by the HPC. See the **GetResourcePadding()** function description.

Description

The **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** provides a mechanism for the PCI bus enumerator to properly initialize the HPCs and CardBus sockets that require initialization. The HPC initialization takes place before the PCI enumeration process is complete. There cannot be more than one instance of this protocol in a system. This protocol is installed on its own separate handle.

Because the system may include multiple HPCs, one instance of this protocol should represent all of them. The protocol functions use the device path of the HPC to identify the HPC. When the PCI bus enumerator finds a root HPC, it will call **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**. If InitializeRootHpc() is unable to initialize a root HPC, the PCI enumerator will ignore that root HPC and continue the

enumeration process. If the HPC is not initialized, the devices that it controls may not be initialized, and no resource padding will be provided.

From the standpoint of the PCI bus enumerator, HPCs are divided into the following two classes:

Root HPC

These HPCs must be initialized by calling InitializeRootHpc() during the enumeration process. These HPCs will also require resource padding. The platform code must have *a priori* knowledge of these devices and must know how to initialize them. There may not be any way to access their PCI configuration space before the PCI enumerator programs all the upstream bridges and thus enables the path to these devices. The PCI bus enumerator is responsible for determining the PCI bus address of the HPC before it calls InitializeRootHpc().

Nonroot HPC

These HPCs will not need explicit initialization during enumeration process. These HPCs will require resource padding. The platform code does not have to have *a priori* knowledge of these devices.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()

Summary

Returns a list of root Hot Plug Controllers (HPCs) that require initialization during the boot process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_ROOT_HPC_LIST) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL   *This,
  OUT UINTN                            *HpcCount,
  OUT EFI_HPC_LOCATION                 **HpcList
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcCount

The number of root HPCs that were returned.

HpcList

The list of root HPCs. HpcCount defines the number of elements in this list. Type **EFI_HPC_LOCATION** is defined in "Related Definitions" below.

Description

This procedure returns a list of root HPCs. The PCI bus driver must initialize these controllers during the boot process. The PCI bus driver may or may not be able to detect these HPCs. If the platform includes a PCI-to-CardBus bridge, it can be included in this list if it requires initialization. The HpcList must be self consistent. An HPC cannot control any of its parent buses. Only one HPC can control a PCI bus. Because this list includes only root HPCs, no HPC in the list can be a child of another HPC. This policy must be enforced by the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. The PCI bus driver may not check for such invalid conditions.

The callee allocates the buffer HpcList.

Related Definitions

```
//*******************************************************
// EFI_HPC_LOCATION
//*******************************************************
typedef struct {
  EFI_DEVICE_PATH_PROTOCOL  *HpcDevicePath;
  EFI_DEVICE_PATH_PROTOCOL  *HpbDevicePath;
} EFI_HPC_LOCATION;
```

HpcDevicePath

The device path to the root HPC. An HPC cannot control its parent buses. The PCI bus driver requires this information so that it can pass the correct HpcPciAddress to the **InitializeRootHpc()** and **GetResourcePadding()** functions. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *UEFI 2.1 Specification.*

HpbDevicePath

The device path to the Hot Plug Bus (HPB) that is controlled by the root HPC. The PCI bus driver uses this information to check if a particular PCI bus has hot-plug slots. The device path of a PCI bus is the same as the device path of its parent. For Standard (PCI) Hot Plug Controllers (SHPCs) and PCI Express*, HpbDevicePath is the same as HpcDevicePath.

Status Codes Returned

EFI_SUCCESS	HpcList was returned.
EFI_OUT_OF_RESOURCES	HpcList was not returned due to insufficient resources.
EFI_INVALID_PARAMETER	HpcCount is NULL.
EFI_INVALID_PARAMETER	HpcList is NULL.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()

Summary

Initializes one root Hot Plug Controller (HPC). This process may causes initialization of its subordinate buses.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INITIALIZE_ROOT_HPC) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL    *This,
  IN  EFI_DEVICE_PATH_PROTOCOL          *HpcDevicePath,
  IN  UINT64                            HpcPciAddress,
  IN  EFI_EVENT                         Event, OPTIONAL
  OUT EFI_HPC_STATE                     *HpcState
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC that is being initialized. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *UEFI 2.1 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

Event

The event that should be signaled when the HPC initialization is complete. Set to **NULL** if the caller wants to wait until the entire initialization process is complete. The event must be of type **EFI_EVENT_NOTIFY_SIGNAL**. Type **EFI_EVENT** is defined in **CreateEvent()** in the *UEFI2.1 Specification.*

HpcState

The state of the HPC hardware. The type **EFI_HPC_STATE** is defined in "Related Definitions" below.

Description

This function initializes the specified HPC. At the end of initialization, the hot-plug slots or sockets (controlled by this HPC) are powered and are connected to the bus. All the necessary registers in the HPC are set up. For a Standard (PCI) Hot Plug Controller (SHPC), the registers that must be set up are defined in the *PCI Standard Hot Plug Controller and Subsystem Specification.* For others HPCs, they are specific to the HPC hardware. The initialization process may choose not to enable certain PCI Hot Plug* slots or sockets for any reason. The PCI Hot Plug slots or CardBus sockets that are left disabled at this stage are not available to the system. A PCI slot may be disabled due to a power fault, PCI bus type mismatch, or power budget constraints. The HPC initialization process can be

time consuming. Powering up the slots that are controlled by SHPCs can take up to 15 seconds. In a system with multiple HPCs, it is desirable to perform these activities in parallel. Therefore, this procedure supports nonblocking execution mode.

If **InitializeRootHpc()** is called with a non-NULL event, HPC initialization is considered complete after the event is signaled. If **InitializeRootHpc()** is called with a non-NULL event, a return from **InitializeRootHpc()** with EFI_SUCCESS marks the completion of the HPC initialization.

The PCI bus enumerator will call this function for every root HPC that is returned by **GetRootHpcList()**.

The PCI bus enumerator must make sure that the registers that are required during HPC initialization are accessible before calling **InitializeRootHpc()**. The determination of whether the registers are accessible is based on the following rules:

- For HPCs (legacy HPCs, SHPCs inside a PCI-to-PCI bridge, and PCI Express* HPCs), the PCI configuration space of the HPC device must be accessible. In other words, all the upstream bridges including root bridges and special-purpose PCI-to-PCI bridges are programmed to forward PCI configuration cycles to the HPC.
- SHPCs inside a root bridge are accessible without any initialization of the PCI bus.
- PCI-to-CardBus bridges have their registers mapped into the memory space using a memory Base Address Register (BAR).

This function takes the device path of the HPC as an input. At the time of HPC initialization, the PCI bus enumeration is not complete. The PCI bus enumerator may not have created a handle for the HPC and the hot-plug initialization code cannot use the **EFI_PCI_IO_PROTOCOL** or **EFI_DEVICE_PATH_PROTOCOL** like other PCI device drivers. The device path uniquely identifies the HPC and also the PCI bus that it controls.

If the HPC is a PCI device, the hot-plug initialization code may need its address on the PCI bus (**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**; see the *UEFI 2.1 Specification* for its definition) to access its registers. The PCI address of a regular PCI device is dynamic but is known to the PCI bus driver. Therefore, the PCI bus driver provides it through the input parameter HpcPciAddress to this function. Passing this address eliminates the need for **InitializeRootHpc()** to convert the device path into the PCI address. If the HPC is a function in a multifunction device, this address is the PCI address of that function. The HPC's configuration space must be accessible at the specified HpcPciAddress until the HPC initialization is complete. In other words, the PCI bus driver cannot renumber PCI buses that are upstream to the HPC while it is being initialized.

This member function can use the **LocateDevicePath()** function to locate the appropriate instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

If the *Event* is not NULL, this function will return control to the caller without completing the entire initialization. This function must perform some basic checks to make sure that it knows how to initialize the specified HPC before returning control. The *Event* is signaled when the initialization process completes, regardless of whether it results in a failure. The caller must check HpcState to get the initialization status after the event is signaled.

If *Event* is not NULL, it is possible that the *Event* may be signaled before this function returns. There are at least two cases where that may happen:

- A simple implementation of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** may force the caller to wait until the initialization is complete. In that case, the InitializeRootHpc() function may signal the event before it returns control back to the caller.
- The HPC may already have been initialized by the time **InitializeRootHpc()** is called. In that case, **InitializeRootHpc()** will signal *Event* and return control back to the caller.

HpcState returns the state of the HPC at the time when control returns. If Event is NULL, HpcState must indicate that the HPC has completed initialization. If Event is not NULL, HpcState can indicate that the HPC has not completed initialization when this function returns, but HpcState must be updated before Event is signaled.

The firmware may not wait until InitializeRootHpc() to start HPC initialization. The firmware may start the initialization earlier in the boot process and the initialization may be completely done by the time the PCI bus enumerator calls InitializeRootHpc(). An HPC can be initialized by hardware alone, and no firmware initialization may be needed. For such HPCs, this member function does not have to do any real work. In such cases, InitializeRootHpc() merely acts as a synchronization point.

Related Definitions

```
//*******************************************************
// EFI_HPC_STATE
//*******************************************************
// Describes current state of an HPC

typedef UINT16 EFI_HPC_STATE;

#define  EFI_HPC_STATE_INITIALIZED    0x01
#define  EFI_HPC_STATE_ENABLED        0x02
```

Following is a description of the possible states for **EFI_HPC_STATE**.

Table 18. Description of possible states for EFI_HPC_STATE

0	Not initialized
EFI_HPC_STATE_INITIALIZED	The HPC initialization function was called and the HPC completed initialization, but it was not enabled for some reason. The HPC may be disabled in hardware, or it may be disabled due to user preferences, hardware failure, or other reasons. No resource padding is required.
EFI_HPC_STATE_INITIALIZED | EFI_HPC_ENABLED	The HPC initialization function was called, the HPC completed initialization, and it was enabled. Resource padding is required.

Status Codes Returned

EFI_SUCCESS	If *Event* is NULL, the specific HPC was successfully initialized. If *Event* is not NULL, Event will be signaled at a later time when initialization is complete.

EFI_UNSUPPORTED	This instance of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC. If *Event* is not NULL, it will not be signaled.
EFI_OUT_OF_RESOURCES	Initialization failed due to insufficient resources. If *Event* is not NULL, it will not be signaled.
EFI_INVALID_PARAMETER	HpcState is NULL.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()

Summary

Returns the resource padding that is required by the PCI bus that is controlled by the specified Hot Plug Controller (HPC).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_HOT_PLUG_PADDING) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL    *This,
  IN  EFI_DEVICE_PATH_PROTOCOL          *HpcDevicePath,
  IN  UINT64                            HpcPciAddress,
  OUT EFI_HPC_STATE                     *HpcState,
  OUT VOID                              **Padding,
  OUT EFI_HPC_PADDING_ATTRIBUTES        *Attributes
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC. Type **EFI_DEVICE_PATH** is defined in **LocateDevicePath()** in the *UEFI 2.1 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

HpcState

The state of the HPC hardware. Type **EFI_HPC_STATE** is defined in **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**.

Padding

The amount of resource padding that is required by the PCI bus under the control of the specified HPC. Because the caller does not know the size of this buffer, this buffer is allocated by the callee and freed by the caller.

Attributes

Describes how padding is accounted for. The padding is returned in the form of ACPI (2.0 & 3.0) resource descriptors. The exact definition of each of the fields is the same as in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()** in section 8.8.2**.** Type **EFI_HPC_PADDING_ATTRIBUTES** is defined in "Related Definitions" below.

Description

This function returns the resource padding that is required by the PCI bus that is controlled by the specified HPC. This member function is called for all the root HPCs and nonroot HPCs that are detected by the PCI bus enumerator. This function will be called before PCI resource allocation is completed. This function must be called after all the root HPCs, with the possible exception of a PCI-to-CardBus bridge, have completed initialization. Waiting until initialization is completed allows the HPC driver to optimize the padding requirement. The calculation may take into account the number of empty and/or populated PCI Hot Plug* slots, the number of PCI-to-PCI bridges among the populated slots, and other factors. This information is available only after initialization is complete. PCI-to-CardBus bridges require memory resources before the initialization is started and therefore are considered an exception. The padding requirements are relatively constant for PCI-to-CardBus bridges and an estimated value must be returned.

If **InitializeRootHpc()** is called with a non-NULL event, HPC initialization is considered complete after the event is signaled. If **InitializeRootHpc()** is called with a non-NULL event, a return from **InitializeRootHpc()** with **EFI_SUCCESS** marks the completion of HPC initialization.

The input parameters *HpcDevicePath*, *HpcPciAddress*, and *HpcState* are described in **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**. The value of HpcPciAddress for the same root HPC may be different from what was passed to I**nitializeRootHpc()**. The HPC's configuration space must be accessible at the specified HpcPciAddress until this function returns control.

The padding is returned in the form of ACPI (2.0 & 3.0) resource descriptors. The exact definition of each of the fields is the same as in the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()** function. See the section 8.8 for the definition of this function.

The PCI bus driver is responsible for adding this resource request to the resource requests by the physical PCI devices. If *Attributes* is EfiPaddingPciBus, the padding takes effect at the PCI bus level. If *Attributes* is EfiPaddingPciRootBridge, the required padding takes effect at the root bridge level. For details, see the definition of **EFI_HPC_PADDING_ATTRIBUTES** in "Related Definitions" below.

Note that the padding request cannot ask for specific legacy resources such as COM port addresses. Legacy PC Card devices may require such resources. Supporting these resource requirements is outside the scope of this specification.

Related Definitions

```
//*****************************************************
// EFI_HPC_PADDING_ATTRIBUTES
//*****************************************************
// Describes how resource padding should be applied

typedef enum {
  EfiPaddingPciBus,
  EfiPaddingPciRootBridge
} EFI_HPC_PADDING_ATTRIBUTES;
```

Following is a description of the fields in the above definition.

Table 19. EFI_HPC_PADDING_ATTRIBUTES field descriptions

EfiPaddingPciBus	Apply the padding at a PCI bus level. In other words, the resources that are allocated to the bus containing hot-plug slots are padded by the specified amount. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures will indicate the padding.
EfiPaddingPciRootBridge	Apply the padding at a PCI root bridge level. If a PCI root bridge includes more than one hot-plug bus, the resource padding requests for these buses are added together and the resources that are allocated to the root bridge are padded by the specified amount. This strategy may reduce the total amount of padding, but requires reprogramming of PCI-to-PCI bridges in a hot-add event. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures do not indicate the padding for that bus.

Status Codes Returned

EFI_SUCCESS	The resource padding was successfully returned.
EFI_UNSUPPORTED	This instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC.
EFI_NOT_READY	This function was called before HPC initialization is complete.
EFI_INVALID_PARAMETER	HpcState is NULL.
EFI_INVALID_PARAMETER	Padding is NULL.
EFI_INVALID_PARAMETER	Attributes is NULL.
EFI_OUT_OF_RESOURCES	ACPI (2.0 & 3.0) resource descriptors for Padding cannot be allocated due to insufficient resources.

10.7.1 PCI Hot Plug Request Protocol

A hot-plug capable PCI bus driver should produce the EFI PCI Hot Plug Request protocol. When a PCI device or a PCI-like device (for example, 32-bit PC Card) is installed after PCI bus does the enumeration, the PCI bus driver can be notified through this protocol. For example, when a 32-bit PC Card is inserted into the PC Card socket, the PC Card bus driver can call interface of this protocol to notify PCI bus driver to allocate resource and create handles for this PC Card.

Summary

Provides services to notify PCI bus driver that some events have happened in a hot-plug controller (for example, PC Card socket, or PHPC), and ask PCI bus driver to create or destroy handles for the PCI-like devices.

GUID

```
#define EFI_PCI_HOTPLUG_REQUEST_PROTOCOL_GUID \
  {0x19cb87ab,0x2cb9,0x4665,0x83,0x60,0xdd,0xcf,\
  0x60,0x54,0xf7,0x9d}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOTPLUG_REQUEST_PROTOCOL {
  EFI_PCI_HOTPLUG_REQUEST_NOTIFY     Notify;
} EFI_PCI_HOTPLUG_REQUEST_PROTOCOL;
```

Parameters

Notify

Notify the PCI bus driver that some events have happened in a hot-plug controller (for example, PC Card socket, or PHPC), and ask PCI bus driver to create or destroy handles for the PCI-like devices. See Section 0 for a detailed description.

Description

The **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** is installed by the PCI bus driver on a separate handle when PCI bus driver starts up. There is only one instance in the system. Any driver that wants to use this protocol must locate it globally.

The **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** allows the driver of hot-plug controller, for example, PC Card Bus driver, to notify PCI bus driver that an event has happened in the hot-plug controller, and the PCI bus driver is requested to create (add) or destroy (remove) handles for the specified PCI-like devices. For example, when a 32-bit PC Card is inserted, this protocol interface will be called with an add operation, and the PCI bus driver will enumerate and start the devices inserted; when a 32-bit PC Card is removed, this protocol interface will be called with a remove operation, and the PCI bus driver will stop the devices and destroy their handles.

The existence of this protocol represents the capability of the PCI bus driver. If this protocol exists in system, it means PCI bus driver is hot-plug capable, thus together with the effort of PC Card bus driver, hot-plug of PC Card can be supported. Otherwise, the hot-plug capability is not provided.

EFI_PCI_HOTPLUG_REQUEST_PROTOCOL.Notify()

Summary

This function is used to notify PCI bus driver that some events happened in a hot-plug controller, and the PCI bus driver is requested to start or stop specified PCI-like devices.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_HOTPLUG_REQUEST_NOTIFY) (
  IN EFI_PCI_HOTPLUG_REQUEST_PROTOCOL *This,
  IN EFI_PCI_HOTPLUG_OPERATION        Operation,
  IN EFI_HANDLE                       Controller,
  IN EFI_DEVICE_PATH_PROTOCOL         *RemainingDevicePath OPTIONAL,
  IN OUT UINT8                        *NumberOfChildren,
  IN OUT EFI_HANDLE                   *ChildHandleBuffer
  );
```

Parameters

This

A pointer to the **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** instance. Type **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** is defined in Section 0.

Operation

The operation the PCI bus driver is requested to make. See "Related Definitions" for the list of legal values.

Controller

The handle of the hot-plug controller.

RemainingDevicePath

The remaining device path for the PCI-like hot-plug device. It only contains device path nodes behind the hot-plug controller. It is an optional parameter and only valid when the Operation is a add operation. If it is NULL, all devices behind the PC Card socket are started.

NumberOfChildren

The number of child handles. For a add operation, it is an output parameter. For a remove operation, it's an input parameter. When it contains a non-zero value, children handles specified in *ChildHandleBuffer* are destroyed. Otherwise, PCI bus driver is notified to stop managing the controller handle.

ChildHandleBuffer

The buffer which contains the child handles. For a add operation, it is an output parameter and contains all newly created child handles. For a remove operation, it contains child handles to be destroyed when *NumberOfChildren* contains a non-zero value. It can be NULL when *NumberOfChildren* is 0. It's the caller's responsibility to allocate and free memory for this buffer.

Description

This function allows the PCI bus driver to be notified to act as requested when a hot-plug event has happened on the hot-plug controller. Currently, the operations include add operation and remove operation.

If it is a add operation, the PCI bus driver will enumerate, allocate resources for devices behind the hot-plug controller, and create handle for the device specified by *RemainingDevicePath*. The *RemainingDevicePath* is an optional parameter. If it is not NULL, only the specified device is started; if it is NULL, all devices behind the hot-plug controller are started. The newly created handles of PC Card functions are returned in the *ChildHandleBuffer*, together with the number of child handle in *NumberOfChildren*.

If it is a remove operation, when *NumberOfChildren* contains a non-zero value, child handles specified in *ChildHandleBuffer* are stopped and destroyed; otherwise, PCI bus driver is notified to stop managing the controller handle.

Related Definitions

```
//*******************************************************
// EFI PCI HOTPLUG NOTIFY OPERATION
//*******************************************************
typedef enum {
  EfiPciHotPlugRequestAdd,
  EfiPciHotplugRequestRemove
} EFI_PCI_HOTPLUG_OPERATION;
```

EfiPciHotplugRequestAdd

The PCI bus driver is requested to create handles for the specified devices. An array of **EFI_HANDLE** is returned, a NULL element marks the end of the array.

EfiPciHotplugRequestRemove

The PCI bus driver is requested to destroy handles for the specified devices.

Status Codes Returned

EFI_SUCCESS	The handles for the specified device have been created or destroyed as requested, and for an add operation, the new handles are returned in *ChildHandleBuffer*.
EFI_INVALID_PARAMETER	*Operation* is not a legal value.
EFI_INVALID_PARAMETER	*Controller* is NULL or not a valid handle.
EFI_INVALID_PARAMETER	*NumberOfChildren* is NULL.
EFI_INVALID_PARAMETER	*ChildHandleBuffer* is NULL while *Operation* is ìremoveî and *NumberOfChildren* contains a non-zero value.
EFI_INVALID_PARAMETER	*ChildHandleBuffer* is NULL while *Operation* is ìaddî.
EFI_OUT_OF_RESOURCES	There are no enough resources to start the devices.

10.8 Sample Implementation for a Platform Containing PCI Hot Plug* Slots

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI bus enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. Actual implementations may vary although the relative ordering of events is critical. The activities related to PCI Hot Plug* are underlined. Please note that hot plug PCI devices may require that multiple passes of bus enumeration are required.

There are several phases during the PCI bus enumeration process when PCI hot plug slots are present. At each phase, the PlatformNotify function of the **EFI_PCI_PLATFORM_PROTOCOL** and **EFI_PCI_OVERRIDE_PROTOCOL** will be called with the execution phase BeforePciHostBridge. Then the PCI host bridge driver function NotifyPhase is called. Finally, the PlatformNotify functions are called again, but with the execution phase **AfterPciHostBridge**.

1. If the platform supports PCI Hot Plug, an instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is installed.
2. The PCI enumeration process begins.
3. Look for instances of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. If it is not found, all the hot-plug subsystem initialization steps can be skipped. If one exists, create a list of root Hot Plug Controllers (HPCs) by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()**.
4. Notify the drivers using **EfiPciHostBridgeBeginBusAllocation**.
5. For every PCI root bridge handle, do the following:
   - Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration**(*This, RootBridgeHandle*).
   - Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. See the *UEFI 2.1 `Specification* for the definition of the PCI Root Bridge I/O Protocol.
   - Allocate memory to hold resource requirements.

- Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.

Scan all the devices in the specified bus range and the specified segment, one bus at a time. If the device is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If the device path of a device matches that of a root HPC and it is not a PCI-to-CardBus bridge, it must be initialized by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()** before the bus it controls can be fully enumerated. The PCI bus enumerator determines the PCI address of the PCI Hot Plug Controller (PHPC) and passes it as an input to I**nitializeRootHpc()**.

- Continue to scan devices on that root bridge and start the initialization of all root HPCs.
- Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()** so that the HPCs under initialization are still accessible. SetBusNumbers() cannot affect the PCI addresses of the HPCs.

6. Wait until all the HPCs that were found on various root bridges in step 5 to complete initialization.
7. Go back to step 5 for another pass and rescan the PCI buses. For all the root HPCs and the nonroot HPCs, call **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()** to obtain the amount of overallocation and add that amount to the requests from the physical devices. Reprogram the bus numbers by taking into account the bus resource padding information. This action requires calling **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()**. The rescan is not required if there is only one root bridge in the system.

Once the memory resources are allocated and a PCI-to-CardBus bridge is part of the HpcList, it will be initialized.

Appendix A
Error Codes

A.1 Error Code Definitions

For 32-bit architecture:

```
#define EFI_INTERRUPT_PENDING                   0xa0000000
#define EFI_WARN_INTERRUPT_SOURCE_PENDING       0x20000000
#define EFI_WARN_INTERRUPT_SOURCE_QUIESCED      0x20000001
```

For 64-bit architecture:

```
#define EFI_INTERRUPT_PENDING                   0xa000000000000000
#define EFI_WARN_INTERRUPT_SOURCE_PENDING       0x2000000000000000
#define EFI_WARN_INTERRUPT_SOURCE_QUIESCED      0x2000000000000001
```