VOLUME 5: Platform Initialization Specification

Standards

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

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

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	Revises typographical errors and minor omissions--see Errata for details	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
1.2	• 271 Support For Large Firmware Files And Firmware File Sections • 284 CPU I/O protocol update • 286 Legacy Region protocol • 289 Recovery API • 292 PCD Specification Update • 354 ACPI Manipulation Protocol • 355 EFI_SIO_PROTOCOL Errata • 365 UEFI Capsule HOB • 382 IDE Controller Specification • 385 Report Status Code Router Specification • 386 Status Code Specification	01/19/09

1.2	• 401 SMM Volume 4 issue	02/23/09
	• 402 SMM PI spec issue w.r.t. CRC	
	• 407 Add LMA Pseudo-Register to SMM Save State Protocol	
	• 409 PCD_PROTOCOL Errata	
	• 411 Draft Errata, Volume 5, Section 8	
	• 412 Comment: PEI_S3_RESUME_PPI should be EFI_PEI_S3_RESUME_PPI	
	• 414 Draft Chapter 7 Comments	
	• 415 Comment: Report Status Code Routers	
	• 416 EFI_CPU_IO_PROTOCOL2 Name should be EFI_CPU_IO2_PROTOCOL	
	• 417 Volume 5, Chapter 4 & 5 order is reversed	
	• 423 Comment: Section 15.2.1 Formatting Issues vol5	
	• 424 Comments: Volume 5, Appendix A.1 formatting issues	
	• 425 Comment: Formatting in Section 6.1 of Volume 3	
	• 426 Comments: Volume 2	
	• 427 Comment: Volume 3, Section 6	
	• 433 Editorial issues in PI 1.2 draft	
1.2	• 407 Comment: additional change to LMA Pseudo-Register	05/13/09
	• 441 Comment: PI Volume 3, Incorrect Struct Declaration (esp PCD_PPI)	
	• 455 Comment: Errata - Clarification of InstallPeiMemory()	
	• 465 Comment: Errata on PMI interface	
	• 466 Comment: Vol 4 EXTENDED_SAL_PROC definition	
	• 467 Comments: PI1.1 errata	
	• 480 Comment: FIX to PCD_PROTOCOL and PCD_PPI	

1.2 errata	• 345 PI1.0 errata	12/16/09
	• 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 PI1.1 errata from SMM development	
	• 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	
	• 521Add 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	

1.2 errata A	• 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 • 551 Name conflicts w/ Legacy region • 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 • 565 CANCEL_CALL_BACK should be CANCEL_CALLBACK • 569 Recovery: EFI_PEI_GET_NUMBER_BLOCK_DEVICES decl has EFI_STATUS w/o return code & errror on stage 3 recovery description • 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 • 594 ATA/ATAPI clarification • 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
1.2 errata B	• 628 ACPI SDT protocol errata • 629 Typos in PCD GetSize() • 630EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL service clarification • 631 System Management System Table (SMST) MP-related field clarification	5/27/10

1.2 Errata C	• 550 Naming conflicts w/ PI SMM	10/27/11
	• 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4)	
	• 654 UEFI PI specific handle for SMBIOS is now available	
	• 688 Status Code errata	
	• 690 Clarify agent in IDE Controller chapter	
	• 691 SMM a priori file and SOR support	
	• 692 Clarify the SMM SW Register API	
	• 694 PEI Temp RAM PPI ambiguity	
	• 703 End of PEI phase PPI publication for the S3 boot mode case	
	• 706 GetPeiServicesTablePointer () changes for the ARM architecture	
	• 714 PI Service Table Versions	
	• 717 PI Extended File Size Errata	
	• 718 PI Extended Header cleanup / Errata	
	• 730 typo in EFI_SMM_CPU_PROTOCOL.ReadSaveState() return code	
	• ERROR: listed by mistake:737	
	• 738 Errata to Volume 2 of the PI1.2 specification	
	• 739 Errata for PI SMM Volume 4 Control protocol	
	• 742 Errata for SMBUS chapter in Volume 5	
	• 743 Errata - PCD_PPI declaration	
	• 745 Errata – PI Firmware Section declarations	
	• 746 Errata - PI status code	
	• 747 Errata - Text for deprecated HOB	
	• 752 Binary Prefix change	
	• ERROR: listed by mistake: 753	
	• 764 PI Volume 4 SMM naming errata	
	• 775 errata/typo in EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT, Volume 3	
	• 781 S3 Save State Protocol Errata	
	• 782 Format Insert(), Compare() and Label() as for Write()	
	• 783 TemporaryRamMigration Errata	
	• 784 Typos in status code definitions	
	• 787 S3 Save State Protocol Errata 2	
	• 810 Set Memory Attributes return code clarification	
	• 811 SMBIOS API Clarification	
	• 814 PI SMBIOS Errata	
	• 821Location conflict for EFI_RESOURCE_ATTRIBUTE_xxx_PROTECTABLE #defines	
	• 823 Clarify max length of SMBIOS Strings in SMBIOS Protocol	
	• 824 EFI_SMM_SW_DISPATCH2_PROTOCOL.Register() Errata	
	• 837 ARM Vector table can not support arbitrary 32-bit address	
	• 838 Vol 3 EFI_FVB2_ALIGNMNET_512K should be EFI_FVB2_ALIGNMENT_512K	
	• 840 Vol 3 Table 5 Supported FFS Alignments contains values not supported by FFS	
	• 844 correct references to Platform Initialization Hand-Off Block Specification	

1.2.1	• 527 PI Volume 2 DXE Security Architecture Protocol (SAP) clarification	05/02/12
	• 562 Add SetMemoryCapabilities to GCD interface	
	• 719 End of DXE event	
	• 731 Volume 4 SMM - clarify the meaning of NumberOfCpus	
	• 737 Remove SMM Communication ACPI Table definition .	
	• 753 SIO PEI and UEFI-Driver Model Architecture	
	• 769 Signed PI sections	
	• 813 Add a new EFI_GET_PCD_INFO_PROTOCOL and EFI_GET_PCD_INFO_PPI instance.	
	• 818 New SAP2 return code	
	• 822 Method to disable Temporary RAM when Temp RAM Migration is not required	
	• 833 Method to Reserve Interrupt and Exception Vectors	
	• 839 Add support for weakly aligned FVs	
	• 892 EFI_PCI_ENUMERATION_COMPLETE_GUID Protocol	
	• 894 SAP2 Update	
	• 895 Status Code Data Structures Errata	
	• 902 Errata on signed firmware volume/file	
	• 903 SmiManage Update	
	• 906 Volume 3 errata - Freeform type	
	• 916 Service table revisions	
1.2.1 Errata A	• 922 Add a "Boot with Manufacturing" boot mode setting	10/26/12
	• 925 Errata on signed FV/Files	
	• 931 DXE Volume 2 - Clarify memory map construction from the GCD	
	• 936 Clarify memory usage in PEI on S3	
	• 937 SMM report protocol notify issue errata	
	• 951 Root Handler Processing by SmiManage	
	• 958 Omissions in PI1.2.1 integration for M816 and M894	
	• 969 Vol 1 errata: TE Header parameters	
1.3	• 945 Integrated Circuit (I2C) Bus Protocol	3/29/13
	• 998 PI Status Code additions	
	• 999 PCI enumeration complete GUID	
	• 1005 NVMe Disk Info guid	
	• 1006 Security Ppi Fixes	
	• 1025 PI table revisions	

1.3 Errata	• 1041 typo in HOB Overview	2/19/15
	• 1067 PI1.3 Errata for SetBootMode	
	• 1068 Updates to PEI Service table/M1006	
	• 1069 SIO Errata - pnp end node definition	
	• 1070 Typo in SIO chapter	
	• 1072 Errata – SMM register protocol notify clarification/errata	
	• 1093 Extended File Size Errata	
	• 1095 typos/errata	
	• 1097 PI SMM GPI Errata	
	• 1098 Errata on I2C IO status code	
	• 1099 I2C Protocol stop behavior errata	
	• 1104 ACPI System Description Table Protocol Errata	
	• 1105 ACPI errata - supported table revision	
	• 1177 PI errata - make CPU IO optional	
	• 1178 errata - allow PEI to report an additional memory type	
	• 1283 Errata - clarify sequencing of events	
1.4	• 1210 Adding persistence attribute to GCD	2/20/15
	• 1235 PI.Next Feature - no execute support	
	• 1236 PI.Next feature - Graphics PPI	
	• 1237 PI.Next feature - add reset2 PPI	
	• 1239 PI.Next feature - Disk Info Guid UFS	
	• 1240 PI.Next feature - Recovery Block IO PPI - UFS	
	• 1259 PI.Next feature - MP PPI	
	• 1273 PI.Next feature - capsule PPI	
	• 1274 Recovery Block I/O PPI Update	
	• 1275 GetMemoryMap Update	
	• 1277 PI1.next feature - multiple CPU health info	
	• 1278 PI1.next - Memory relative reliability definition	
	• 1305 PI1.next - specification number encoding	
	• 1331 Remove left-over Boot Firmware Volume references in the SEC Platform Information PPI	
	• 1366 PI 1.4 draft - M1277 issue BIST / CPU. So health record needs to be indexed / CPU.	

1.4 Errata A	• 1574 Fix artificial limitation in the PCD.SetSku support	3/15/16
	• 1565 Update status code to include AArch64 exception error codes	
	• 1564 SMM Software Dispatch Protocol Errata	
	• 1562 Errata to remove statement from DXE vol about PEI dispatch behavior	
	• 1561 Errata to provide Equivalent of DXE-CIS Mantis 247 for the PEI-CIS	
	• 1532 Allow S3 Resume without having installed permanent memory (via InstallPeiMemory)	
	• 1530 errata on dxe report status code	
	• 1529 address space granularity errata	
	• 1525 PEI Services Table Retrieval for AArch64	
	• 1515 EFI_PEIM_NOTIFY_ENTRY_POINT return values are undefined	
	• 1497 Fixing language in SMMStartupThisAP	
	• 1489 GCD Conflict errata	
	• 1485 Minor Errata in SMM Vo2 description of SMMStartupThisAP	
	• 1397 PEI 1.4 specification revision errata	
	• 1394 Errata to Relax requirements on CPU rendez in SEC	
	• 1351 EndOfDxe and SmmReadyToLock	
	• 1322 Minor Updates to handle Asynchronous CPU Entry Into SMM	
1.5	• 1315 SMM Environment to Support Newer Architecture/Platform Designs	4/26/16
	• 1317 additional I2C PPI's (vol5)	
	• 1321 ARM Extensions to Volume 4	
	• 1330 Add PPI to allow SEC pass HOBs into PEI	
	• 1336 Provide For Pre-DXE Initialization Of The SM Foundation	
	• 1369 Handling PEI PPI descriptor notifications from SEC	
	• 1387 Variable services errors not consistent	
	• 1390 SM stand-alone infrastructure	
	• 1396 Update SEC HOB Capabilities of 1330 with additional guidance	
	• 1413 Communicate protocol enhancements	
	• 1506 New MP protocol	
	• 1513 Need a way to propagate PEI-phase FV verification status to DXE	
	• 1563 Update MM PPIs to match existing implementations	
	• 1566 PI.next - update the specification revisions	
	• 1568 Add SD/MMC GUID to DiskInfo protocol	
	• 1592 Add EFI_FV_FILETYPE_SMM_CORE_STANDALONE file type	
	• 1594 Pei GetVaiable M1387 issue	
	• 1595 M1568 Disk Info issue	
	• 1596 M1489 GCD issue	
	• 1603 Minor erratas in Vol4 PI 1.5 draft related to ECR 0001506	

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 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 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 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_SMBUS_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_SMBUS_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_SMBUS_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_STRINGUpdateString;
  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, the handle of the SMBIOS record to add. If FFFEh, 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_ADDITIONAL_INFORMATION              40
#define EFI_SMBIOS_TYPE_ONBOARD_DEVICES_EXTENDED_INFORMATION 41
#define EFI_SMBIOS_TYPE_MANAGEMENT_CONTROLLER_HOST_INTERFACE 42

#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;
```

Note: *These types are consistent with the DMTF SMBIOS 2.7 specification.*

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 because it is longer than the SMBIOS Table supports.
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_HANDLE         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 FFFEh on entry, then the first SMBIOS record handle will be returned. If it returns FFFEh 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
IDE Controller

7.1 IDE Controller Overview

This specification defines the core code and services that are required for an implementation of the IDE Controller Initialization Protocol of the UEFI Platform Initialization Specification. This protocol isa driver entity such as a driver entity to program an IDE controller and to obtain IDE device timing information. This protocol abstracts the nonstandard parts of an IDE controller. This protocol is not tied to any specific bus.

This specification does the following:

- Describes the basic components of the IDE Controller Initialization Protocol
- Provides code definitions for the IDE Controller Initialization Protocol and other IDE-controller-related type definitions that are architecturally required

7.2 Design Discussion

7.2.1 IDE Controller Initialization Protocol Overview

This section discusses the IDE Controller Initialization Protocol. This protocol is used by a driver entity to program an IDE controller and to obtain IDE device timing information. This protocol abstracts the nonstandard parts of IDE controller. This protocol is mandatory on platforms with IDE controllers that are managed by a driver entity.

See IDE Controller Initialization Protocol in Code Definitions for the definition of **EFI_IDE_CONTROLLER_INIT_PROTOCOL**.

7.2.1.1 IDE Controller Terms

The following terms are used throughout this document.

AHCI

> Advanced Host Controller Interface.

ATAPI

> AT Attachment Packet Interface

enumeration group

> The set of IDE devices that must be enumerated as a group. In other words, if device A and device B belong to an enumeration group and device A needs to be configured, device B must be configured at the same time and vice versa. There are two possible enumeration groupings for an IDE controller:

- "All the devices on a channel. In this case, the number of enumeration groups is equal to the number of channels.

- "All the devices on all the channels behind an IDE controller. This enumeration grouping may arise because multiple channels share some hardware registers or have some other dependencies. In this case, the number of enumeration groups is 1.

The IDE controller indicates the type of enumeration group that is applicable. In case 2, the driver entity must enumerate all the devices on all the channels if there is a request to configure a single device. In case 1, the driver entity must enumerate all the devices on the same channel if there is a request to configure a single device. Case 1 will lead to faster boot.

IDE controller

The hardware device that produces one or more IDE buses (channels). Each channel can host one or more IDE devices.

PATA

Parallel ATA.

PATA controller

An IDE controller that supports PATA devices. Traditionally, a PATA controller supports up to two channels: primary and secondary. Each channel traditionally supports up to two devices: master and slave.

SATA

Serial ATA.

SATA controller

An IDE controller that supports the SATA driver. SATA controllers can emulate PATA behavior. The behavior of command and control block registers, PIO and DMA data transfers, resets, and interrupts are all emulated. In addition, SATA controllers can implement a more modern register interface, namely AHCI. AHCI allows the host software to overcome the limitations that are imposed by PATA emulation and to use advanced SATA features.

Some chipsets contain both PATA and SATA controllers and support a combined mode. In combined mode, the two controllers are logically merged into one controller. The PATA drives can appear behind the SATA controller to the host software. In such a mode, all the PATA rules in terms of IDE timing configuration apply to SATA controllers.

7.2.2 IDE Controller Initialization Protocol References

The following sources of information are referenced in this specification or may be useful to you.

- "*ATA Host Adapter Standards*, Working Draft Version of: http://www.t13.org/*
- "*Information Technology - AT Attachment with Packet Interface - 6* (ATA/ATAPI-6): http://www.t13.org/*
- *Serial ATA Advanced Host Controller Interface (AHCI) Specification*, version 1.0: http://developer.intel.com/technology/serialata/ahci.htm
- *Serial ATA: High Speed Serialized AT Attachment*, revision 1.0a (may also be referred to as Serial ATA Specification 1.0a): http://www.serialata.org/*
- "*Serial ATA II: Port Multiplier Specification*, revision 1.1: http://www.serialata.org/*

7.2.3 Background

7.2.3.1 IDE Requirements

The IDE Controller Initialization Protocol is designed to work for both Parallel ATA (PATA) and Serial ATA (SATA) IDE controllers.

This protocol is designed with the following requirements in mind:

1. The timing registers in a PATA IDE controller are vendor specific. (See *ATA Host Adapter Standards*, Working Draft Version 0f, for more information.) The programming of these registers needs to be abstracted from the driver entity.

2. The IDE Controller Initialization Protocol should also support a case where a specific channel is disabled and/or it should not be scanned. This protocol also needs a mechanism to address individual devices in various SATA and PATA configurations. This protocol needs to support the following:

   • "A variable number of channels per controller
   • "A variable number of devices per channel

7.2.3.1.1 PATA Controllers

PATA controllers support up to two channels and each channel can have a maximum of two devices.

7.2.3.1.2 SATA Contollers

SATA controllers can support standard ATA emulation. As described in the *Serial ATA Specification 1.0a*, ATA emulation can either be master-only emulation or master-slave emulation. In either case, the SATA controller appears to have one or two channels. In master-only emulation, a maximum of one drive appears on a channel. In master-slave emulation, one or two drives can show up behind a channel.

When an SATA controller is operating in Advanced Host Controller Interface (AHCI) mode, it can support up to 32 ports. The SATA port that is generated by an SATA controller can host an SATA port multiplier. There can be up to 16 SATA devices on the other side of the SATA port multiplier.

In this geometry, each SATA port that is generated by the SATA controller is treated as a channel, and this channel can have up to 16 devices. This is done so that PATA drives as well as SATA drives can be represented using a (*Channel*, *Device*) address pair. Note that the SATA channels work very differently from PATA channels in the sense that the SATA channels do not have the concept of master/slave or daisy chaining.

See Figure 2 1 and Figure 2 2 below for explanations how the devices are addressed.

7.2.3.1.3 Bus Neutral

It should be possible to use the same abstractions to support an IDE controller on the PCI bus or some other bus. The IDE controller driver will know which controller devices it can support. Because the majority of IDE controllers that exist today are located on the PCI bus, all the examples will refer to PCI IDE controllers, but the protocol is not tied to the PCI bus.

7.2.3.2 PCI IDE controller

PCI IDE controllers can operate in native PCI mode or compatibility mode. The IDE Controller Initialization Protocol should permit both modes.

The design should use the EFI Driver Model to support the quick boot feature. The smallest unit of initialization is one channel. By default, the driver entity initializes only the channel on which the user-requested drive resides. The IDE Controller Initialization Protocol should support the case where various channels share the same hardware bits and cannot be independently enumerated. The controller driver can specify that all the channels should be enumerated as one unit.

The IDE Controller Initialization Protocol must support SATA controllers that may or may not implement AHCI register interface.

7.2.4 Simplifying the Design of IDE Drivers

The IDE bus is not a general-purpose bus. The standard ATA and ATAPI command sets support only a storage class of devices. The following design decisions can be made to simplify the IDE Controller Initialization Protocol and the design of IDE drivers:

- "The driver entity is the only driver that will send commands to the ATA devices. No device-specific drivers are needed for IDE devices because all the devices belong to the same class (i.e., storage) and the driver entity can have inherent knowledge of these commands. IDE bus equivalents of **EFI_PCI_IO_PROTOCOL** and **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** for accessing IDE devices are not required. It is possible to further simplify the design of the driver entity if it does not have to deal with the ATAPI devices. It can enumerate the ATA and ATAPI devices and install the **EFI_SCSI_PASSTHRU_PROTOCOL** on ATAPI device handles. Either way, IDE-bus-specific I/O protocols are not needed. See the *UEFI Specification* for the definitions of the EFI PCI I/O Protocol, PCI Root Bridge I/O Protocol, and the SCSI Pass Thru Protocol.

- "IDE devices are accessed and configured through a set of standard registers in the IDE controller. The ATA committee is standardizing the layout of these registers. (See *ATA Host Adapter Standards*, Working Draft Version 0f, for more information.) For Serial ATA (SATA) controllers, the *Serial ATA Advanced Host Controller Interface (AHCI) Specification* defines a standard register interface. Although the layout is dependent on the bus on which the controller is located, the layout for a particular bus is fixed. As a result, the driver entity can be required to know about the register layout for buses that it chooses to support. For example, for a PCI IDE controller, the IDE driver can access the base of the command block register for channel 0 using the following steps:

1. Check bit 0 of register 0x9 (Programming Interface Code) in the PCI configuration space of the controller to determine whether it is operating in compatibility mode or native PCI mode. For this example, we will assume that the controller is operating in native mode.

2. Read register 0x10 (Base Address Register [BAR] 0) of the controller. Clear bit 0 of the value that was read to get the command block base

7.2.5 Configuring Devices on the IDE Bus

The table below lists the various drivers that may participate in configuring the devices on the IDE bus.

Table 1. Drivers Involved in Configuring IDE Devices

Driver	Follows the EFI Driver Model?	Description
IDE controller driver	Yes	Produces the **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. Consumes the bus-specific I/O protocol. **EFI_IDE_CONTROLLER_INIT_PROTOCOL** abstracts the chipset-specific IDE controller registers and is responsible for early initialization of the IDE controller. Note that **EFI_IDE_CONTROLLER_INIT_PROTOCOL** is not tied to a specific bus although most IDE controllers today are on the PCI or ISA bus.
Driver entity	Yes	Consumes the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** and the bus-specific I/O protocol. It enumerates the IDE buses. This driver will check for the presence of the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** on the controller handle before enumerating the child devices. This driver uses the presence of the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** to determine whether a controller is an IDE controller or not. This driver will use bus-specific methods to access the standard ATA registers (such as the control block, command block, and bus master DMA registers) for a particular device. The driver not only knows the address of a specific register block, but it also knows the layout of that register block. This driver may produce the **EFI_SCSI_PASSTHRU_PROTOCOL** for ATAPI devices or it may directly manage the ATAPI devices by producing the **EFI_BLOCK_IO_PROTOCOL**. This driver produces the **EFI_BLOCK_IO_PROTOCOL** for ATA devices.
Generic SCSI or ATAPI storage driver	Yes	This optional driver manages the ATAPI device using the **EFI_SCSI_PASSTHRU_PROTOCOL** and produces the **EFI_BLOCK_IO_PROTOCOL** if requested.
Driver entity and IDE controller driver combined as one driver	Yes	It is also possible to combine the driver entity and the IDE controller driver into one driver. In this case, **EFI_IDE_CONTROLLER_INIT_PROTOCOL** is not installed on the IDE controller handle. The monolithic driver is responsible for initializing the IDE controller as well as the IDE devices behind that controller. **EFI_IDE_CONTROLLER_INIT_PROTOCOL** is mandatory if the IDE devices behind the controller are to be enumerated by the generic driver entity.

See the UEFI Specification for the definitions of the Block I/O Protocol and the SCSI Pass Thru Protocol. The IDE Controller Initialization Protocol is defined in Code Definitions of this specification.

7.2.6 Sample Implementation for a Simple PCI IDE Controller

This topic provides a sample implementation only. The sequencing of various notifications cannot be changed. The steps below apply if **EFI_IDE_CONTROLLER_INIT_PROTOCOL.EnumAll** = **FALSE.**

See the *UEFI Specification* for definitions of the Driver Binding Protocol, EFI PCI I/O Protocol, Device Path Protocol, and Block I/O Protocol. See Code Definitions in this specification for the definition of the IDE Controller Initialization Protocol.

1. The IDE controller driver as well as the driver entity follow the EFI Driver Model. They are loaded and both install (at least) one instance of the **EFI_DRIVER_BINDING_PROTOCOL** on their image handle. An ATA hard drive behind a PCI IDE controller is one of the boot devices.

2. The PCI bus driver enumerates the PCI bus, finds the PCI IDE controller, creates a handle for it, and installs an instance of **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on that handle.

3. The Boot Device Selection (BDS) phase searches for an appropriate driver to own the IDE controller device and finds the IDE controller driver. It then connects the IDE controller device and the IDE controller driver. The IDE controller driver opens the **EFI_PCI_IO_PROTOCOL BY_DRIVER**. It may perform some other preprogramming at this point.

4. BDS searches for a driver to own the IDE device and finds the driver entity. The driver entity's Supported() function checks for the presence of **EFI_IDE_CONTROLLER_INIT_PROTOCOL** on the parent of the IDE device (i.e., the IDE controller).

5. The EFI Boot Services function **ConnectController()** calls the **Start()** function of the driver entity, which starts the IDE bus enumeration. The following steps are performed by the **Start()** function.

   - The driver entity locates the **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. It opens the **EFI_IDE_CONTROLLER_INIT_PROTOCOL BY_DRIVER**. If it needs to open **EFI_PCI_IO_PROTOCOL**, it may open it by **GET_PROTOCOL**. The driver entity reads the *EnumAll* and *ChannelCount* fields in **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. In this case, *EnumAll* is **FALSE**. The driver entity also obtains the channel number from **Start().RemainingDevicePath**.

   - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeChannelEnumeration*, *Channel*).

   - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo** (*This*, *Channel*, **Enabled*, **MaxDevices*) to find out the number of devices on this channel. If **Enabled* **= FALSE,** it exits with an error code. If the device number of the device to be connected is too large, it exits with an error code.

   - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeChannelReset*, *Channel*).

   - The driver entity resets the channel.

   - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterChannelReset*, *Channel*).

   - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeDevicePresenceDetection*, *Channel*). The IDE controller driver may insert a predelay here or may ensure that various IDE bus signals are at desired levels.

- The driver entity attempts to detect devices on the channel. Note than there can be no more than MaxDevices on the channel.
- The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterDevicePresenceDetection*, *Channel*).
- The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeResetMode*, *Channel*). The IDE controller sets up the controller with the default timings.

6. For all the devices on this channel:
   - The driver entity gathers **EFI_IDENTIFY_DATA** for the device and submits it to the IDE controller driver using **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()**. Submit **NULL** data for devices that do not exist.
   - The driver entity may call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** to disqualify modes that it does not support.

7. For all the detected devices on this channel:
   - Call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** to get the optimum mode settings. The IDE controller driver uses controller-specific algorithms and platform information to calculate the best modes.
   - The driver entity enables the appropriate modes by sending an ATA **SET_FEATURES** command to the device. It the device returns an error, it disqualifies that mode for that device and goes back to step 7. This time step 7 (first bullet) will not consider the failed mode. The implementation then returns here to step 7 (second bullet) with new (less optimum) modes.

8. For all the detected devices on this channel, call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()** to program the timings. Note that we reset the mode settings in step 5(last bullet), so the settings for nonexistent devices will remain at their default levels.

9. The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterChannelEnumeration*, *Channel*).

10. Install **EFI_BLOCK_IO_PROTOCOL** on that device handle.

7.3 Code Definitions

This section contains the basic definitions of the IDE Controller Initialization Protocol. The IDE Controller Initialization Protocol

following protocol is defined in this section:

EFI_IDE_CONTROLLER_INIT_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_IDE_CONTROLLER_ENUM_PHASE**
> **EFI_IDENTIFY_DATA**
> **EFI_ATA_IDENTIFY_DATA**
> **EFI_ATAPI_IDENTIFY_DATA**
> **EFI_ATA_COLLECTIVE_MODE**
> **EFI_ATA_MODE**
> **EFI_ATA_EXTENDED_MODE**
> **EFI_ATA_EXT_TRANSFER_PROTOCOL**

EFI_IDE_CONTROLLER_INIT_PROTOCOL

Summary

Provides the basic interfaces to abstract an IDE controller.

GUID

```
#define EFI_IDE_CONTROLLER_INIT_PROTOCOL_GUID \
  { 0xa1e37052, 0x80d9, 0x4e65, 0xa3, 0x17, 0x3e, 0x9a, \
  0x55, 0xc4, 0x3e, 0xc9 }
```

Protocol Interface Structure

```
typedef struct _EFI_IDE_CONTROLLER_INIT_PROTOCOL {
  EFI_IDE_CONTROLLER_GET_CHANNEL_INFO    GetChannelInfo;
  EFI_IDE_CONTROLLER_NOTIFY_PHASE        NotifyPhase;
  EFI_IDE_CONTROLLER_SUBMIT_DATA         SubmitData;
  EFI_IDE_CONTROLLER_DISQUALIFY_MODE     DisqualifyMode;
  EFI_IDE_CONTROLLER_CALCULATE_MODE      CalculateMode;
  EFI_IDE_CONTROLLER_SET_TIMING          SetTiming;
  BOOLEAN                                EnumAll;
  UINT8                                  ChannelCount;
} EFI_IDE_CONTROLLER_INIT_PROTOCOL;
```

Parameters

GetChannelInfo

> Returns the information about a specific channel. See the GetChannelInfo() function description.

NotifyPhase

> The notification that the driver entity is about to enter the specified phase during the enumeration process. See the **NotifyPhase()** function description.

SubmitData

> Submits the Drive Identify data that was returned by the device. See the **SubmitData()** function description.

DisqualifyMode

>   Submits information about modes that should be disqualified. The specified IDE device does not support these modes and these modes should not be returned by CalculateMode . See the **DisqualifyMode()** function description.

CalculateMode

>   Calculates and returns the optimum mode for a particular IDE device. See the **CalculateMode()** function description.

SetTiming

>   Programs the IDE controller hardware to the default timing or per the modes that were returned by the last call to **CalculateMode()**. See the **SetTiming()** function description.

EnumAll

>   Set to **TRUE** if the enumeration group includes all the channels that are produced by this controller. **FALSE** if an enumeration group consists of only one channel.

ChannelCount

>   The number of channels that are produced by this controller. Parallel ATA (PATA) controllers can support up to two channels. Advanced Host Controller Interface (AHCI) Serial ATA (SATA) controllers can support up to 32 channels, each of which can have up to one device.   In the presence of a multiplier, each channel can have 15 devices.

Description

The **EFI_IDE_CONTROLLER_INIT_PROTOCOL** provides the chipset-specific information to the driver entity. This protocol is mandatory for IDE controllers if the IDE devices behind the controller are to be enumerated by a driver entity.

There can only be one instance of **EFI_IDE_CONTROLLER_INIT_PROTOCOL** for each IDE controller in a system. It is installed on the handle that corresponds to the IDE controller. A driver entity that wishes to manage an IDE bus and possibly IDE devices in a system will have to retrieve the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance that is associated with the controller to be managed.

A device handle for an IDE controller must contain an **EFI_DEVICE_PATH_PROTOCOL**.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo()

Summary

Returns the information about the specified IDE channel.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_GET_CHANNEL_INFO) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN UINT8                              Channel,
  OUT BOOLEAN                           *Enabled,
  OUT UINT8                             *MaxDevices
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

> Zero-based channel number.

Enabled

> **TRUE** if this channel is enabled. Disabled channels are not scanned to see if any devices are present.

MaxDevices

> The maximum number of IDE devices that the bus driver can expect on this channel. For the ATA/ATAPI specification, version 6, this number will either be 1 or 2. For Serial ATA (SATA) configurations with a port multiplier, this number can be as large as 15.

Description

This function can be used to obtain information about a particular IDE channel. The driver entity uses this information during the enumeration process.

If *Enabled* is set to **FALSE**, the driver entity will not scan the channel. Note that it will not prevent an operating system driver from scanning the channel.

For most of today's controllers, *MaxDevices* will either be 1 or 2. For SATA controllers, this value will always be 1. SATA configurations can contain SATA port multipliers. SATA port multipliers behave like SATA bridges and can support up to 16 devices on the other side. If an SATA port out of the IDE controller is connected to a port multiplier, *MaxDevices* will be set to the number of SATA devices that the port multiplier supports. Because today's port multipliers support up to 15 SATA devices, this number can be as large as 15. The driver entity is required to scan for the presence of port multipliers behind an SATA controller and enumerate up to *MaxDevices* number of devices behind the port multiplier.

In this context, the devices behind a port multiplier constitute a channel.

Status Codes Returned

EFI_SUCCESS	Information was returned without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).

EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()

Summary

The notifications from the driver entity that it is about to enter a certain phase of the IDE channel enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_NOTIFY_PHASE) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL   *This,
  IN EFI_IDE_CONTROLLER_ENUM_PHASE       Phase,
  IN UINT8                               Channel
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Phase

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

Channel

> Zero-based channel number.

Description

This function can be used to notify the IDE controller driver to perform specific actions, including any chipset-specific initialization, so that the chipset is ready to enter the next phase. Seven notification points are defined at this time. See "Related Definitions" below for the definition of various notification points and Sample Implementation for a Simple PCI IDE Controller in the Design Discussion chapter for usage.

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

Related Definitions

```
//*************************************************
// EFI_IDE_CONTROLLER_ENUM_PHASE
//*************************************************
typedef enum {
  EfiIdeBeforeChannelEnumeration,
  EfiIdeAfterChannelEnumeration,
  EfiIdeBeforeChannelReset,
  EfiIdeAfterChannelReset,
  EfiIdeBusBeforeDevicePresenceDetection,
  EfiIdeBusAfterDevicePresenceDetection,
  EfiIdeResetMode,
  EfiIdeBusPhaseMaximum
} EFI_IDE_CONTROLLER_ENUM_PHASE;
```

Table 2. Field descriptiond for EFI_IDE_CONTROLLER_ENUM_PHASE

EfiIdeBeforeChannelEnumeration	The driver entity is about to begin enumerating the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeAfterChannelEnumeration	The driver entity has completed enumerating the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeBeforeChannelReset	The driver entity is about to reset the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeAfterChannelReset	The driver entity has completed resetting the devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeBusBeforeDevicePresenceDetection	The driver entity is about to detect the presence of devices behind the specified channel. This notification can be used to set up the bus signals to default levels or for implementing predelays.
EfiIdeBusAfterDevicePresenceDetection	The driver entity is done with detecting the presence of devices behind the specified channel. This notification can be used to perform any chipset-specific programming.
EfiIdeResetMode	The IDE bus is requesting the IDE controller driver to reprogram the IDE controller hardware and thereby reset all the mode and timing settings to default settings.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_UNSUPPORTED	*Phase* is not supported.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).

EFI_NOT_READY	This phase cannot be entered at this time; for example, an attempt was made to enter a *Phase* without having entered one or more previous *Phase*.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()

Summary

Submits the device information to the IDE controller driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_SUBMIT_DATA) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL  *This,
  IN UINT8                             Channel,
  IN UINT8                             Device,
  IN EFI_IDENTIFY_DATA                 *IdentifyData
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

> Zero-based channel number.

Device

> Zero-based device number on the *Channel*.

IdentifyData

> The device's response to the ATA **IDENTIFY_DEVICE** command. Type **EFI_IDENTIFY_DATA** is defined in "Related Definitions" below.

Related Definitions

```
//*****************************************************
// EFI_IDENTIFY_DATA
//*****************************************************
typedef union {
  EFI_ATA_IDENTIFY_DATA      AtaData;
  EFI_ATAPI_IDENTIFY_DATA    AtapiData;
} EFI_IDENTIFY_DATA;

#define   EFI_ATAPI_DEVICE_IDENTIFY_DATA  0x8000
```

AtaData

> The data that is returned by an ATA device upon successful completion of the ATA **IDENTIFY_DEVICE** command. The **IDENTIFY_DEVICE** command is defined in the ATA/ATAPI specification. Type **EFI_ATA_IDENTIFY_DATA** is defined below.

AtapiData

The data that is returned by an ATAPI device upon successful completion of the ATA
IDENTIFY_PACKET_DEVICE command. The **IDENTIFY_PACKET_DEVICE**
command is defined in the ATA/ATAPI specification. Type
EFI_ATAPI_IDENTIFY_DATA is defined below.

AtapiData

Table 3. `EFI_ATAPI_IDENTIFY_DATA` Definition .

EFI_ATAPI_DEVICE_IDENTIFY_DATA	This flag indicates whether the IDENTIFY data is a response from an ATA device (`EFI_ATA_IDENTIFY_DATA`) or response from an ATAPI device (`EFI_ATAPI_IDENTIFY_DATA`). According to the ATA/ATAPI specification, `EFI_IDENTIFY_DATA` is for an ATA device if bit 15 of the Config field is zero. The Config field is common to both `EFI_ATA_IDENTIFY_DATA` and `EFI_ATAPI_IDENTIFY_DATA`.

```
//***************************************************
// EFI_ATA_IDENTIFY_DATA
//***************************************************
//
// This structure definition is not part of the protocol
// definition because the ATA/ATAPI Specification controls
// the definition of all the fields. The ATA/ATAPI
// Specification can obsolete old fields or redefine existing
// fields. This definition is provided here for reference only.
//

#pragma pack(1)
///
/// EFI_ATA_IDENTIFY_DATA is strictly complied with ATA/ATAPI-8
Spec
///
typedef struct {
  UINT16  config;                           ///< General
Configuration
  UINT16  obsolete_1;
  UINT16  specific_config;                  ///< Specific
Configuration
  UINT16  obsolete_3;
  UINT16  retired_4_5[2];
  UINT16  obsolete_6;
  UINT16  cfa_reserved_7_8[2];
  UINT16  retired_9;
  CHAR8   SerialNo[20];                     ///< word 10~19
  UINT16  retired_20_21[2];
  UINT16  obsolete_22;
  CHAR8   FirmwareVer[8];                   ///< word 23~26
  CHAR8   ModelName[40];                    ///< word 27~46
  UINT16  multi_sector_cmd_max_sct_cnt;
  UINT16  trusted_computing_support;
  UINT16  capabilities_49;
```

```
       UINT16  capabilities_50;
       UINT16  obsolete_51_52[2];
       UINT16  field_validity;
       UINT16  obsolete_54_58[5];
       UINT16  multi_sector_setting;
       UINT16  user_addressable_sectors_lo;
       UINT16  user_addressable_sectors_hi;
       UINT16  obsolete_62;
       UINT16  multi_word_dma_mode;
       UINT16  advanced_pio_modes;
       UINT16  min_multi_word_dma_cycle_time;
       UINT16  rec_multi_word_dma_cycle_time;
       UINT16  min_pio_cycle_time_without_flow_control;
       UINT16  min_pio_cycle_time_with_flow_control;
       UINT16  reserved_69_74[6];
       UINT16  queue_depth;
       UINT16  reserved_76_79[4];                 ///< reserved for
     Serial ATA
       UINT16  major_version_no;
       UINT16  minor_version_no;
       UINT16  command_set_supported_82;          ///< word 82
       UINT16  command_set_supported_83;          ///< word 83
       UINT16  command_set_feature_extn;          ///< word 84
       UINT16  command_set_feature_enb_85;        ///< word 85
       UINT16  command_set_feature_enb_86;        ///< word 86
       UINT16  command_set_feature_default;       ///< word 87
       UINT16  ultra_dma_mode;                    ///< word 88
       UINT16  time_for_security_erase_unit;
       UINT16  time_for_enhanced_security_erase_unit;
       UINT16  advanced_power_management_level;
       UINT16  master_password_identifier;
       UINT16  hardware_configuration_test_result;
       UINT16  acoustic_management_value;
       UINT16  stream_minimum_request_size;
       UINT16  streaming_transfer_time_for_dma;
       UINT16  streaming_access_latency_for_dma_and_pio;
       UINT16  streaming_performance_granularity[2];///< word 98~99
       UINT16  maximum_lba_for_48bit_addressing[4]; ///< word 100~103
       UINT16  streaming_transfer_time_for_pio;
       UINT16  reserved_105;
       UINT16  phy_logic_sector_support;          ///< word 106
       UINT16  interseek_delay_for_iso7779;
       UINT16  world_wide_name[4];                 ///< word 108~111
       UINT16  reserved_for_128bit_wwn_112_115[4];
       UINT16  reserved_for_technical_report;
       UINT16  logic_sector_size_lo;              ///< word 117
       UINT16  logic_sector_size_hi;              ///< word 118
```

```
  UINT16   features_and_command_sets_supported_ext; ///< word 119
  UINT16   features_and_command_sets_enabled_ext;    ///< word 120
  UINT16   reserved_121_126[8];
  UINT16   obsolete_127;
  UINT16   security_status;                          ///< word 128
  UINT16   vendor_specific_129_159[31];
  UINT16   cfa_power_mode;                           ///< word 160
  UINT16   reserved_for_compactflash_161_175[15];
  CHAR8    media_serial_number[60];                  ///< word
176~205
  UINT16   sct_command_transport;                    ///< word 206
  UINT16   reserved_207_208[2];
  UINT16   alignment_logic_in_phy_blocks;            ///< word 209
  UINT16   write_read_verify_sector_count_mode3[2];  ///< word
210~211
  UINT16   verify_sector_count_mode2[2];
  UINT16   nv_cache_capabilities;
  UINT16   nv_cache_size_in_logical_block_lsw;       ///< word 215
  UINT16   nv_cache_size_in_logical_block_msw;       ///< word 216
  UINT16   nv_cache_read_speed;
  UINT16   nv_cache_write_speed;
  UINT16   nv_cache_options;                         ///< word 219
  UINT16   write_read_verify_mode;                   ///< word 220
  UINT16   reserved_221;
  UINT16   transport_major_revision_number;
  UINT16   transport_minor_revision_number;
  UINT16   reserved_224_233[10];
  UINT16   min_number_per_download_microcode_mode3; ///< word 234
  UINT16   max_number_per_download_microcode_mode3; ///< word 235
  UINT16   reserved_236_254[19];
  UINT16   integrity_word;
} EFI_ATA_IDENTIFY_DATA;
#pragma pack()

//*****************************************************
// EFI_ATAPI_IDENTIFY_DATA
//*****************************************************
#pragma pack(1)
///
/// EFI_ATAPI_IDENTIFY_DATA is strictly complied with ATA/ATAPI-
8 Spec
///
typedef struct {
  UINT16  config;                        ///< General Configuration
  UINT16  reserved_1;
  UINT16  specific_config;               ///< Specific Configuration
  UINT16  reserved_3_9[7];
```

```
CHAR8   SerialNo[20];                           ///< word 10~19
UINT16  reserved_20_22[3];
CHAR8   FirmwareVer[8];                          ///< word 23~26
CHAR8   ModelName[40];                           ///< word 27~46
UINT16  reserved_47_48[2];
UINT16  capabilities_49;
UINT16  capabilities_50;
UINT16  obsolete_51;
UINT16  reserved_52;
UINT16  field_validity;                          ///< word 53
UINT16  reserved_54_61[8];
UINT16  dma_dir;
UINT16  multi_word_dma_mode;                     ///< word 63
UINT16  advanced_pio_modes;                      ///< word 64
UINT16  min_multi_word_dma_cycle_time;
UINT16  rec_multi_word_dma_cycle_time;
UINT16  min_pio_cycle_time_without_flow_control;
UINT16  min_pio_cycle_time_with_flow_control;
UINT16  reserved_69_70[2];
UINT16  obsolete_71_72[2];
UINT16  reserved_73_74[2];
UINT16  queue_depth;
UINT16  reserved_76_79[4];
UINT16  major_version_no;                        ///< word 80
UINT16  minor_version_no;                        ///< word 81
UINT16  cmd_set_support_82;
UINT16  cmd_set_support_83;
UINT16  cmd_feature_support;
UINT16  cmd_feature_enable_85;
UINT16  cmd_feature_enable_86;
UINT16  cmd_feature_default;
UINT16  ultra_dma_select;
UINT16  time_required_for_sec_erase;        ///< word 89
UINT16  time_required_for_enhanced_sec_erase; ///< word 90
UINT16  reserved_91;
UINT16  master_pwd_revison_code;
UINT16  hardware_reset_result;                   ///< word 93
UINT16  current_auto_acoustic_mgmt_value;
UINT16  reserved_95_107[13];
UINT16  world_wide_name[4];                      ///< word 108~111
UINT16  reserved_for_128bit_wwn_112_115[4];
UINT16  reserved_116_124[9];
UINT16  atapi_byte_count_0_behavior;        ///< word 125
UINT16  obsolete_126;
UINT16  removable_media_status_notification_support;
UINT16  security_status;
UINT16  reserved_129_160[32];
```

```
    UINT16  cfa_reserved_161_175[15];
    UINT16  reserved_176_254[79];
    UINT16  integrity_word;
} EFI_ATAPI_IDENTIFY_DATA;
#pragma pack()
```

Description

This function is used by the driver entity to pass detailed information about a particular device to the IDE controller driver. The driver entity obtains this information by issuing an ATA or ATAPI **IDENTIFY_DEVICE** command. *IdentifyData* is the pointer to the response data buffer. The *IdentifyData* buffer is owned by the driver entity, and the IDE controller driver must make a local copy of the entire buffer or parts of the buffer as needed. The original *IdentifyData* buffer pointer may not be valid when **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** or **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** is called at a later point.

The IDE controller driver may consult various fields of **EFI_IDENTIFY_DATA** to compute the optimum mode for the device. These fields are not limited to the timing information. For example, an implementation of the IDE controller driver may examine the vendor and type/mode field to match known bad drives.

The driver entity may submit drive information in any order, as long as it submits information for all the devices belonging to the enumeration group before **CalculateMode()** is called for any device in that enumeration group. If a device is absent, **SubmitData()** should be called with *IdentifyData* set to **NULL**. The IDE controller driver may not have any other mechanism to know whether a device is present or not. Therefore, setting *IdentifyData* to **NULL** does not constitute an error condition. **SubmitData()** can be called only once for a given (*Channel*, *Device*) pair.

Status Codes Returned

EFI_SUCCESS	The information was accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()

Summary

Disqualifies specific modes for an IDE device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_DISQUALIFY_MODE) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL  *This,
  IN UINT8                             Channel,
  IN UINT8                             Device,
  IN EFI_ATA_COLLECTIVE_MODE           *BadModes
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

> Zero-based channel number.

Device

> Zero-based device number on the Channel.

BadModes

> The modes that the device does not support and that should be disqualified. Type **EFI_ATA_COLLECTIVE_MODE** is defined in "Related Definitions" below.

Description

This function allows the driver entity or other drivers (such as platform drivers) to reject certain timing modes and request the IDE controller driver to recalculate modes. This function allows the driver entity and the IDE controller driver to negotiate the timings on a per-device basis. This function is useful in the case of drives that lie about their capabilities. An example is when the IDE device fails to accept the timing modes that are calculated by the IDE controller driver based on the response to the Identify Drive command.

If the driver entity does not want to limit the ATA timing modes and leave that decision to the IDE controller driver, it can either not call this function for the given device or call this function and set the *Valid* flag to **FALSE** for all modes that are listed in **EFI_ATA_COLLECTIVE_MODE**.

The driver entity may disqualify modes for a device in any order and any number of times.

This function can be called multiple times to invalidate multiple modes of the same type (e.g., Programmed Input/Output [PIO] modes 3 and 4). See the ATA/ATAPI specification for more information on PIO modes.

For Serial ATA (SATA) controllers, this member function can be used to disqualify a higher transfer rate mode on a given channel. For example, a platform driver may inform the IDE controller driver to not use second-generation (Gen2) speeds for a certain SATA drive.

Related Definitions

```
//****************************************************
// EFI_ATA_COLLECTIVE_MODE
//****************************************************
typedef struct {
  EFI_ATA_MODE            PioMode;
  EFI_ATA_MODE            SingleWordDmaMode;
  EFI_ATA_MODE            MultiWordDmaMode;
  EFI_ATA_MODE            UdmaMode;
  UINT32                  ExtModeCount;
  EFI_ATA_EXTENDED_MODE   ExtMode[1];
} EFI_ATA_COLLECTIVE_MODE;
```

PioMode

> This field specifies the PIO mode. PIO modes are defined in the ATA/ATAPI specification. The ATA/ATAPI specification defines the enumeration. In other words, a value of 1 in this field means PIO mode 1. The actual meaning of PIO mode 1 is governed by the ATA/ATAPI specification. Type EFI_ATA_MODE is defined below.

SingleWordDmaMode

> This field specifies the single word DMA mode. Single word DMA modes are defined in the ATA/ATAPI specification, versions 1 and 2. Single word DMA support was obsoleted in the ATA/ATAPI specification, version 3; therefore, most devices and controllers will not support this transfer mode. The ATA/ATAPI specification defines the enumeration. In other words, a value of 1 in this field means single word DMA mode 1. The actual meaning of single word DMA mode 1 is governed by the ATA/ATAPI specification.

MultiWordDmaMode

> This field specifies the multiword DMA mode. Various multiword DMA modes are defined in the ATA/ATAPI specification. A value of 1 in this field means multiword DMA mode 1. The actual meaning of multiword DMA mode 1 is governed by the ATA/ATAPI specification.

UdmaMode

> This field specifies the ultra DMA (UDMA) mode. UDMA modes are defined in the ATA/ATAPI specification. A value of 1 in this field means UDMA mode 1. The actual meaning of UDMA mode 1 is governed by the ATA/ATAPI specification.

ExtModeCount

>The number of extended-mode bitmap entries. Extended modes describe transfer protocols beyond PIO, single word DMA, multiword DMA, and UDMA. This field can be zero and provides extensibility.

ExtMode

>*ExtModeCount* number of entries. Each entry represents a transfer protocol other than the ones defined above (i.e., PIO, single word DMA, multiword DMA, and UDMA). This field is defined for extensibility. At this time, only one extended transfer protocol is defined to cover SATA transfers. Type **EFI_ATA_EXTENDED_MODE** is defined below.

```
//*****************************************************
// EFI_ATA_MODE
//*****************************************************
typedef struct {
  BOOLEAN   Valid;
  UINT32    Mode;
} EFI_ATA_MODE;
```

Valid

>**TRUE** if *Mode* is valid.

Mode

>The actual ATA mode. This field is not a bit map.

```
//*****************************************************
// EFI_ATA_EXTENDED_MODE
//*****************************************************
typedef struct {
  EFI_ATA_EXT_TRANSFER_PROTOCOL   TransferProtocol;
  UINT32                          Mode;
} EFI_ATA_EXTENDED_MODE;
```

TransferProtocol

>An enumeration defining various transfer protocols other than the protocols that exist at the time this specification was developed (i.e., PIO, single word DMA, multiword DMA, and UDMA). Each transfer protocol is associated with a mode. The various transfer protocols are defined by the ATA/ATAPI specification. This enumeration makes the interface extensible because we can support new transport protocols beyond UDMA. Type **EFI_ATA_EXT_TRANSFER_PROTOCOL** is defined below.

Mode

The mode for operating the transfer protocol that is identified by
TransferProtocol.

```
//****************************************************
// EFI_ATA_EXT_TRANSFER_PROTOCOL
//****************************************************
//
// This extended mode describes the SATA physical protocol.
// SATA physical layers can operate at different speeds.
// These speeds are defined below. Various PATA protocols
// and associated modes are not applicable to SATA devices.
//
typedef enum {
  EfiAtaSataTransferProtocol
} EFI_ATA_EXT_TRANSFER_PROTOCOL;

#define  EFI_SATA_AUTO_SPEED  0
#define  EFI_SATA_GEN1_SPEED  1
#define  EFI_SATA_GEN2_SPEED  2
```

Table 4. EFI_ATA_EXT_TRANSFER_PROTOCOL field descriptions

EFI_SATA_AUTO_SPEED	Automatically detects the optimum SATA speed.
EFI_SATA_GEN1_SPEED	Indicates a first-generation (Gen1) SATA speed.
EFI_SATA_GEN2_SPEED	Indicates a second-generation (Gen2) SATA speed.

Status Codes Returned

EFI_SUCCESS	The modes were accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_INVALID_PARAMETER	*IdentifyData* is **NULL**.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()

Summary

Returns the information about the optimum modes for the specified IDE device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_CALCULATE_MODES) (
  IN  EFI_IDE_CONTROLLER_INIT_PROTOCOL  *This,
  IN  UINT8                              Channel,
  IN  UINT8                              Device,
  OUT EFI_ATA_COLLECTIVE_MODE           **SupportedModes
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

> Zero-based channel number.

Device

> Zero-based device number on the *Channel*.

SupportedModes

> The optimum modes for the device. Type **EFI_ATA_COLLECTIVE_MODE** is
> defined in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()**.

Description

This function is used by the driver entity to obtain the optimum ATA modes for a specific device.
The IDE controller driver takes into account the following while calculating the mode:

- "The *IdentifyData* inputs to
  EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()

- "The *BadModes* inputs to
  EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()

The driver entity is required to call **SubmitData()** for all the devices that belong to an
enumeration group before calling **CalculateMode()** for any device in the same group.

The IDE controller driver will use controller- and possibly platform-specific algorithms to arrive at
SupportedModes. The IDE controller may base its decision on user preferences and other
considerations as well. This function may be called multiple times because the driver entity may
renegotiate the mode with the IDE controller driver using **DisqualifyMode()**.

The driver entity may collect timing information for various devices in any order. The driver entity
is responsible for making sure that all the dependencies are satisfied; for example, the

SupportedModes information for device A that was previously returned may become stale after a call to **DisqualifyMode()** for device B.

The buffer *SupportedModes* is allocated by the callee because the caller does not necessarily know the size of the buffer. The type **EFI_ATA_COLLECTIVE_MODE** is defined in a way that allows for future extensibility and can be of variable length. This memory pool should be deallocated by the caller when it is no longer necessary.

The IDE controller driver for a Serial ATA (SATA) controller can use this member function to force a lower speed (first-generation [Gen1] speeds on a second-generation [Gen2]-capable hardware). The IDE controller driver can also allow the driver entity to stay with the speed that has been negotiated by the physical layer.

Status Codes Returned

EFI_SUCCESS	*SupportedModes* was returned.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_INVALID_PARAMETER	*SupportedModes* is **NULL**.
EFI_NOT_READY	Modes cannot be calculated due to a lack of data. This error may happen if **SubmitData()** and **DisqualifyData()** were not called for at least one drive in the same enumeration group.

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()

Summary

Commands the IDE controller driver to program the IDE controller hardware so that the specified device can operate at the specified mode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_IDE_CONTROLLER_SET_TIMING) (
  IN EFI_IDE_CONTROLLER_INIT_PROTOCOL  *This,
  IN UINT8                             Channel,
  IN UINT8                             Device,
  IN EFI_ATA_COLLECTIVE_MODE           *Modes
  );
```

Parameters

This

> Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

> Zero-based channel number.

Device

> Zero-based device number on the *Channel*.

Modes

> The modes to set. Type **EFI_ATA_COLLECTIVE_MODE** is defined in **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()**.

Description

This function is used by the driver entity to instruct the IDE controller driver to program the IDE controller hardware to the specified modes. This function can be called only once for a particular device. For a Serial ATA (SATA) Advanced Host Controller Interface (AHCI) controller, no controller-specific programming may be required.

Status Codes Returned

EFI_SUCCESS	The command was accepted without any errors.
EFI_INVALID_PARAMETER	*Channel* is invalid (*Channel* **>=** *ChannelCount*).
EFI_INVALID_PARAMETER	*Device* is invalid.
EFI_NOT_READY	*Modes* cannot be set at this time due to lack of data.
EFI_DEVICE_ERROR	*Modes* cannot be set due to hardware failure. The driver entity should not use this device.

7.3.1 IDE Disk Information Protocol

This section contains the basic definitions of the IDE Disk Information Protocol.

EFI_DISK_INFO_PROTOCOL

Summary

Provides the basic interfaces to abstract platform information regarding an IDE controller.

GUID

```
#define EFI_DISK_INFO_PROTOCOL_GUID \
  { 0xd432a67f, 0x14dc, 0x484b, 0xb3, 0xbb, 0x3f, 0x02, 0x91,\
  0x84, 0x93, 0x27 }
```

Protocol Interface Structure

```
typedef struct _EFI_DISK_INFO_PROTOCOL {
  EFI_GUID                    Interface;
  EFI_DISK_INFO_INQUIRY       Inquiry;
  EFI_DISK_INFO_IDENTIFY      Identify;
  EFI_DISK_INFO_SENSE_DATA    SenseData;
  EFI_DISK_INFO_WHICH_IDE     WhichIde;
} EFI_DISK_INFO_PROTOCOL;
```

Parameters

Interface

A GUID that defines the format of buffers for the other member functions of this protocol.

Inquiry

Return the results of the Inquiry command to a drive in *InquiryData*.   Data format of *Inquiry* data is defined by the Interface GUID.

Identify

Return the results of the *Identify* command to a drive in *IdentifyData*. Data format of *Identify* data is defined by the Interface GUID.

SenseData

>   Return the results of the Request Sense command to a drive in *SenseData*.   Data format of Sense data is defined by the Interface GUID.

WhichIde

>   Specific controller.

Description

The **EFI_DISK_INFO_PROTOCOL** provides controller specific information.

There can only various instances of **EFI_DISK_INFO_PROTOCOL** for different interface types.

EFI_DISK_INFO_PROTOCOL.Interface

Summary

GUID of the type of interfaces

Related Definitions

```
#define EFI_DISK_INFO_IDE_INTERFACE_GUID \
  { \
    0x5e948fe3, 0x26d3, 0x42b5, 0xaf, 0x17, 0x61, 0x2, \
    0x87, 0x18, 0x8d, 0xec \
  }

#define EFI_DISK_INFO_SCSI_INTERFACE_GUID \
  { \
    0x8f74baa, 0xea36, 0x41d9, 0x95, 0x21, 0x21, 0xa7, \
    0xf, 0x87, 0x80, 0xbc \
  }

#define EFI_DISK_INFO_USB_INTERFACE_GUID \
  { \
    0xcb871572, 0xc11a, 0x47b5, 0xb4, 0x92, 0x67, 0x5e, \
    0xaf, 0xa7, 0x77, 0x27 \
  }

#define EFI_DISK_INFO_AHCI_INTERFACE_GUID \
  { \
    0x9e498932, 0x4abc, 0x45af, 0xa3, 0x4d, 0x2, 0x47, \
    0x78, 0x7b, 0xe7, 0xc6 \
  }

#define EFI_DISK_INFO_NVME_INTERFACE_GUID \
{ \
  0x3ab14680, 0x5d3f, 0x4a4d, 0xbc, 0xdc, 0xcc, 0x38, \
  0x0, 0x18, 0xc7, 0xf7 \
}

#define EFI_DISK_INFO_UFS_INTERFACE_GUID \
{ \
  0x4b3029cc, 0x6b98, 0x47fb, 0xbc, 0x96, 0x76, 0xdc, \
  0xb8, 0x4, 0x41, 0xf0 \
}
#define EFI_DISK_INFO_SD_MMC_INTERFACE_GUID \
  { \
    {0x8deec992, 0xd39c, 0x4a5c, { 0xab, 0x6b, 0x98, 0x6e, \
    0x14, 0x24, 0x2b, 0x9d } \
  }
```

The data format of *InquiryData* of **EFI_DISK_INFO_PROTOCOL.Inquiry()** is the card CID register content defined at SD physical layer specification or MMC/eMMC electrical standard.

Description

The type of interface being described.

EFI_DISK_INFO_PROTOCOL.Inquiry()

Summary

Provides inquiry information for the controller type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DISK_INFO_INQUIRY) (
  IN EFI_DISK_INFO_PROTOCOL  *This,
  IN OUT VOID                *InquiryData,
  IN OUT UINT32              *InquiryDataSize
  );
```

Parameters

This

Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

InquiryData

Pointer to a buffer for the inquiry data.

InquiryDataSize

Pointer to the value for the inquiry data size.

Description

This function is used by the driver entity to get inquiry data. Data format of *Identify* data is defined by the Interface GUID.

Status Codes Returned

EFI_SUCCESS	The command was accepted without any errors.
EFI_NOT_FOUND	Device does not support this data class
EFI_DEVICE_ERROR	Error reading *InquiryData* from device
EFI_BUFFER_TOO_SMALL	*InquiryDataSize* not big enough

EFI_DISK_INFO_PROTOCOL.Identify()

Summary

Provides identify information for the controller type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DISK_INFO_IDENTIFY) (
  IN EFI_DISK_INFO_PROTOCOL  *This,
  IN OUT VOID                *IdentifyData,
  IN OUT UINT32              *IdentifyDataSize
  );
```

Parameters

This

> Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

IdentifyData

> Pointer to a buffer for the identify data.

IdentifyDataSize

> Pointer to the value for the identify data size.

Description

This function is used by the driver entity to get identify data. Data format of Identify data is defined by the Interface GUID.

Status Codes Returned

EFI_SUCCESS	The command was accepted without any errors.
EFI_NOT_FOUND	Device does not support this data class
EFI_DEVICE_ERROR	Error reading *IdentifyData* from device
EFI_BUFFER_TOO_SMALL	*IdentifyDataSize* not big enough

EFI_DISK_INFO_PROTOCOL.SenseData()

Summary

Provides sense data information for the controller type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DISK_INFO_SENSE_DATA) (
  IN     EFI_DISK_INFO_PROTOCOL  *This,
  IN OUT VOID                    *SenseData,
  IN OUT UINT32                  *SenseDataSize
  OUT    UINT8                   *SenseDataNumber
  );
```

Parameters

This

> Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

SenseData

> Pointer to the *SenseData*.

SenseDataSize

> Size of *SenseData* in bytes.

SenseDataNumber

> Pointer to the value for the sense data size.

Description

This function is used by the driver entity to get sense data.   Data format of *Identify* data is defined by the Interface GUID.

Status Codes Returned

EFI_SUCCESS	The command was accepted without any errors.
EFI_NOT_FOUND	Device does not support this data class
EFI_DEVICE_ERROR	Error reading *SenseData* from device
EFI_BUFFER_TOO_SMALL	*SenseDataSize* not big enough

EFI_DISK_INFO_PROTOCOL.WhichIde()

Summary

Provides IDE channel and device information for the interface

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DISK_INFO_WHICH_IDE) (
  IN  EFI_DISK_INFO_PROTOCOL     *This,
  OUT UINT32                     *IdeChannel,
  OUT UINT32                     *IdeDevice
  );
```

Parameters

This

>Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

IdeChannel

>Pointer to the Ide Channel number. Primary or secondary.This should also return the port.

IdeDevice

>Pointer to the Ide Device number.   Master or slave.This should also return the port-multiplier port for AHCI.   The format will be the same as for port above.

Description

This function is used by the driver entity to get controller information.

Status Codes Returned

EFI_SUCCESS	IdeChannel and IdeDevice are valid
EFI_UNSUPPORTED	This is not an IDE Device

8
S3 Resume

8.1 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.*

8.2 Goals

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

Extensibility:

The PI S3 resume boot path should easily adapt to different platforms, such as Itanium®-based platforms those based on 32-bit Intel® architecture (IA-32), and x64 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.

8.3 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").

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

8.4 Assumptions

8.4.1 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.

8.4.2 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.

8.5 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 Architechture 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 Architechture 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.
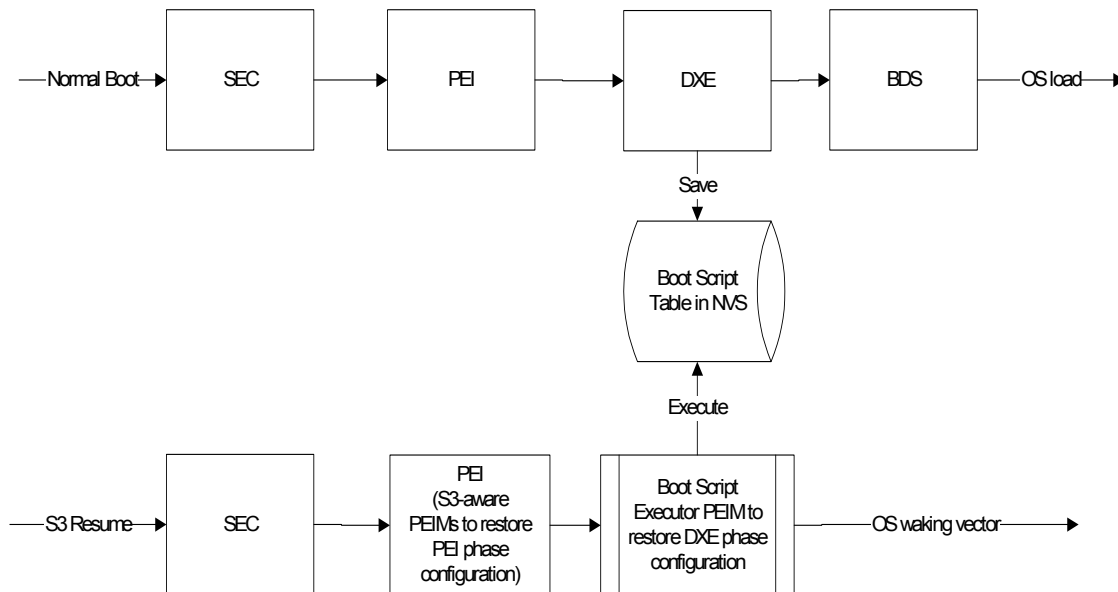
Figure 1. PI Architechture S3 Resume Boot Path

8.5.1 Phases in the S3 Resume Boot Path

8.5.1.1 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.

8.5.1.2 PEI

8.5.1.2.1 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 PI Architechture 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.

```
Normal Boot Path                                                S3 Resume Boot Path

┌──────────────┐                                              ┌──────────────┐
│ PEIM initializes │                 ┌─────────┐              │ PEIM initializes │
│ the platform     │ Save Configuration │ Nonvolatile │ Retrieve Configuration │ the platform │
│ without known    │ ───────────────▶ │ storage   │ ──────────────▶ │ with known   │
│ configuration    │                 └─────────┘              │ configuration │
└──────────────┘                                              └──────────────┘
```
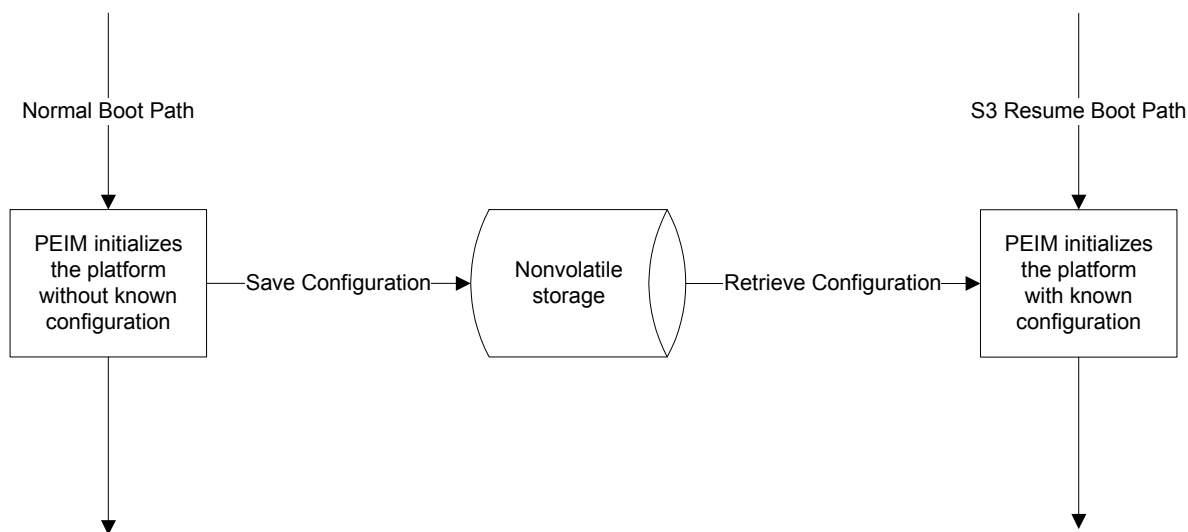
Figure 2. PEI Phase in S3 Resume Boot Path

8.5.1.2.2 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.
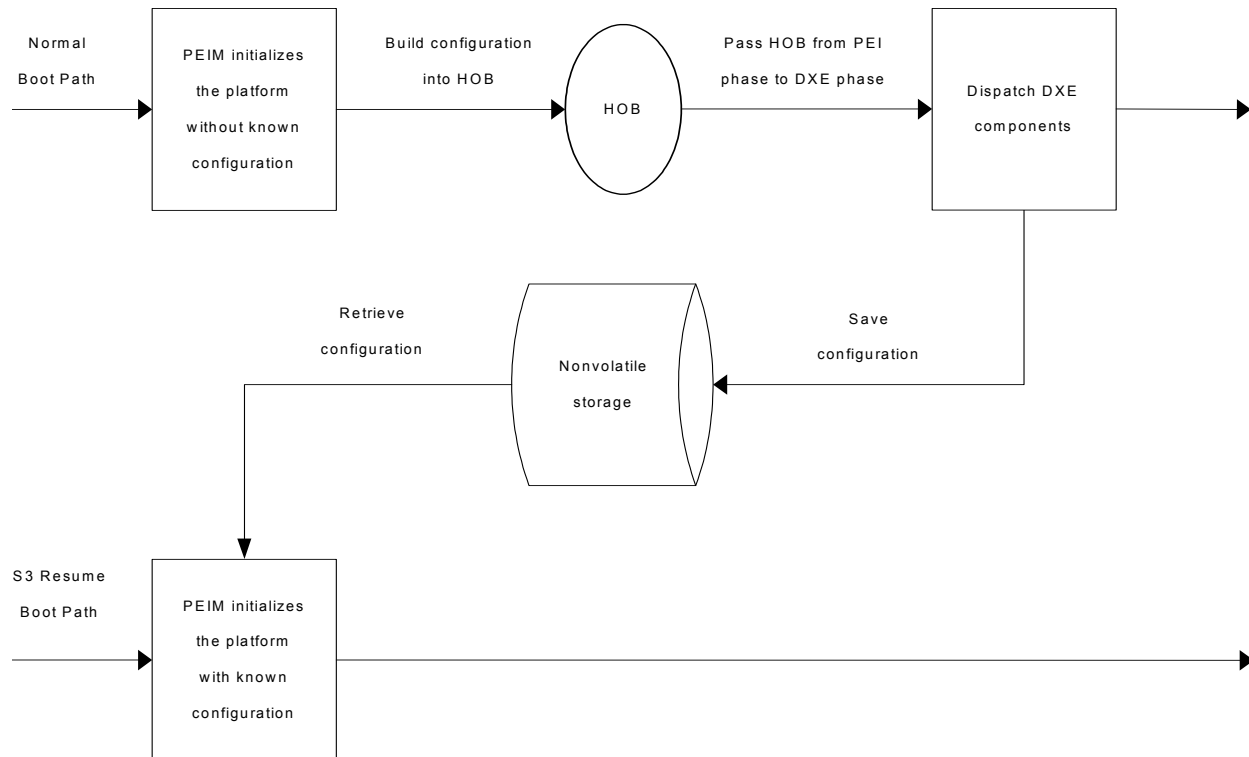
Figure 3. Configuration Save for PEI Phase

8.5.1.3 DXE

8.5.1.3.1 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 PI Architechture 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 PI Architechture 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.

8.5.1.3.2 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 PI Architechture 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.

8.5.1.4 SMM

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

In the S3 boot mode 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

8.6 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.

EFI_PEI_S3_RESUME_PPI. S3RestoreConfig()

Summary

Restores the platform to its pre-boot configuration for an S3 resume and jumps to the OS waking vector.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_S3_RESUME_PPI_RESTORE_CONFIG) (
  IN EFI_PEI_S3_RESUME_PPI                *This
  );
```

Parameters

This

> A pointer to this instance of the **PEI_S3_RESUME_PPI.**

Description

This function will restore the platform to its pre-boot configuration that was pre-stored in the boot script table and transfer control to OS waking vector.

Upon invocation, this function is responsible for locating the following information before jumping to OS waking vector:

- ACPI tables
- boot script table
- any other information that it needs

The **S3RestoreConfig()** function then executes the pre-stored boot script table and transitions the platform to the pre-boot state. The boot script is recorded during regular boot using the **EFI_S3_SAVE_STATE_PROTOCOL.Write()** and **EFI_S3_SMM_SAVE_STATE_PROTOCOL.Write()** functions. Finally, this function transfers control to the OS waking vector. If the OS supports only a real-mode waking vector, this function will switch from flat mode to real mode before jumping to the waking vector.

If all platform pre-boot configurations are successfully restored and all other necessary information is ready, this function will never return and instead will directly jump to the OS waking vector. If this function returns, it indicates that the attempt to resume from the ACPI S3 sleep state failed.

Status Codes Returned

EFI_ABORTED	Execution of the S3 resume boot script table failed.
EFI_NOT_FOUND	Some necessary information that is used for the S3 resume boot path could not be located.

8.7 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_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.

8.7.1 Save State Write

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.

8.7.1.1 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 *Delay* of at least 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 the pointer to the 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_INFORMATION_OPCODE**. Type **EFI_BOOT_SCRIPT_INFORMATION_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

InformationLenght

> Length of the data in bytes.

Information

> Pointer to the information to be logged in the boot script.

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 *Informatio*n must point to memory of type of *EfiRuntimeServicesCode*, *EfiRuntimeServicesData*, or *EfiACPIMemoryNVS*.

Status Codes Returned

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

8.7.2 Save State Insert

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, if not NULL, 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.

8.7.3 Save State Label

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, if not NULL, specifies the position of the inserted label in the boot script table.

Label

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

Description

If the label *Label* already exists in the boot script table, then no new label is created, the position of the *Label* is returned in **Position* (if *Position* is not NULL) and **EFI_SUCCESS** is returned. If the label already exists, the input value of the *Position* is ignored.

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 *Label* is *NULL* or points to an empty string.
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.

8.7.4 Save State Compare

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 operation succeeded.
EFI_INVALID_PARAMETER	The *Position1* or *Position2* is not a valid position in the boot script table.
EFI_INVALID_PARAMETER	The *RelativePosition* is *NULL*.

8.8 S3 SMM Save State Protocol

This section 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 interface definition with the S3 Save State Protocol but it has a different GUID. It is an SMM 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** provides the PI SMMboot script abstraction.

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.

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.

9
ACPI System Description Table Protocol

9.1 EFI_ACPI_SDT_PROTOCOL

Summary

Provides services for creating ACPI system description tables.

GUID

```
#define EFI_ACPI_SDT_PROTOCOL_GUID \
  { 0xeb97088e, 0xcfdf, 0x49c6, \
  { 0xbe, 0x4b, 0xd9, 0x6, 0xa5, 0xb2, 0xe, 0x86 } }
```

Protocol Interface Structure

```
typedef struct _EFI_ACPI_SDT_PROTOCOL {
 EFI_ACPI_TABLE_VERSION    AcpiVersion;
 EFI_ACPI_GET_TABLE2       GetAcpiTable;
 EFI_ACPI_REGISTER_NOTIFY  RegisterNotify;
 EFI_ACPI_OPEN             Open;
 EFI_ACPI_OPEN_SDT         OpenSdt;
 EFI_ACPI_CLOSE            Close;
 EFI_ACPI_GET_CHILD        GetChild;
 EFI_ACPI_GET_OPTION       GetOption;
 EFI_ACPI_SET_OPTION       SetOption;
 EFI_ACPI_FIND_PATH        FindPath;
} EFI_ACPI_SDT_PROTOCOL;
```

Related Definitions

```
#define UINT32 EFI_ACPI_TABLE_VERSION;

#define EFI_ACPI_TABLE_VERSION_NONE (1 << 0)
#define EFI_ACPI_TABLE_VERSION_1_0B (1 << 1)
#define EFI_ACPI_TABLE_VERSION_2_0  (1 << 2)
#define EFI_ACPI_TABLE_VERSION_3_0  (1 << 3)
#define EFI_ACPI_TABLE_VERSION_4_0  (1 << 4)
#define EFI_ACPI_TABLE_VERSION_5_0  (1 << 5)
```

Members

AcpiVersion

A bit map containing all the ACPI versions supported by this protocol.

GetTable

        Enumerate the ACPI tables.

RegisterNotify

        Register a notification when a table is installed.

Open

        Create a handle from an ACPI opcode.

OpenSdt

        Create a handle from an ACPI table.

Close

        Close an ACPI handle.

GetChild

        Cycle through the child objects of an ACPI handle.

GetOption

        Return one of the optional pieces of the opcode.

SetOption

        Change one of the optional pieces of the opcode.

FindPath

        Given an ACPI path, return an ACPI handle.

EFI_ACPI_SDT_PROTOCOL.GetAcpiTable()

Summary

Returns a requested ACPI table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_GET_ACPI_TABLE) (
  IN  UINTN                     Index,
  OUT EFI_ACPI_SDT_HEADER       **Table,
  OUT EFI_ACPI_TABLE_VERSION    *Version,
  OUT UINTN                     *TableKey
  );
```

Parameters

Index

> The zero-based index of the table to retrieve.

Table

> Pointer for returning the table buffer. Type **EFI_ACPI_SDT_HEADER** is defined in "Related Definitions" below.

Version

> On return, updated with the ACPI versions to which this table belongs. Type **EFI_ACPI_TABLE_VERSION** is defined in "Related Definitions" in the **EFI_ACPI_SDT_PROTOCOL**.

TableKey

> On return, points to the table key for the specified ACPI system definition table. This is identical to the table key used in the **EFI_ACPI_TABLE_PROTOCOL**. The *TableKey* can be passed to **EFI_ACPI_TABLE_PROTOCOL.UninstallAcpiTable()** to uninstall the table.

Description

The **GetAcpiTable(**) function returns a pointer to a buffer containing the ACPI table associated with the Index that was input. The following structures are not considered elements in the list of ACPI tables:

- Root System Description Pointer (RSD_PTR)
- Root System Description Table (RSDT)
- Extended System Description Table (XSDT)

Version is updated with a bit map containing all the versions of ACPI of which the table is a member.

For tables installed via the **EFI_ACPI_TABLE_PROTOCOL.InstallAcpiTable()** interface, the function returns the value of **EFI_ACPI_STD_PROTOCOL.AcpiVersion**.

Related Definitions

```
typedef struct {
  UINT32 Signature;
  UINT32 Length;
  UINT8  Revision;
  UINT8  Checksum;
  CHAR8  OemId[6];
  CHAR8  OemTableId[8];
  UINT32 OemRevision;
  UINT32 CreatorId;
  UINT32 CreatorRevision;
} EFI_ACPI_SDT_HEADER;
```

This structure is based on the **DESCRIPTION_HEADER** structure, defined in section 5.2.6 of the *ACPI 3.0 specification*.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_FOUND	The requested index is too large and a table was not found.

EFI_ACPI_SDT_PROTOCOL.RegisterNotify()

Summary

Register or unregister a callback when an ACPI table is installed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_REGISTER_NOTIFY) (
  IN BOOLEAN                  Register,
  IN EFI_ACPI_NOTIFICATION_FN Notification
  );
```

Parameters

Register

> If **TRUE**, then the specified function will be registered. If **FALSE**, then the specified function will be unregistered.

Notification

> Points to the callback function to be registered or unregistered. Type **EFI_ACPI_NOTIFICATION_FN** is defined in "Related Definitions" below.

Description

This function registers or unregisters a function which will be called whenever a new ACPI table is installed.

Status Codes Returned

EFI_SUCCESS	Callback successfully registered or unregistered.
EFI_INVALID_PARAMETER	Notification is NULL
EFI_INVALID_PARAMETER	Register is FALSE and Notification does not match a known registration function.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_NOTIFICATION_FN)(
  IN EFI_ACPI_SDT_HEADER        *Table,
  IN EFI_ACPI_TABLE_VERSION     Version,
  IN UINTN                      TableKey
  );
```

Table

> A pointer to the ACPI table header.

Version

> The ACPI table's version. Type **EFI_ACPI_TABLE_VERSION** is defined in "Related Definitions" in the **EFI_ACPI_SDT_PROTOCOL**.

TableKey

> The table key for this ACPI table. This is identical to the table key used in the **EFI_ACPI_TABLE_PROTOCOL**.

This function is called each time a new ACPI table is added using **EFI_ACPI_TABLE_PROTOCOl.InstallAcpiTable()**.

EFI_ACPI_SDT_PROTOCOL.Open()

Summary

Create a handle from an ACPI opcode

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_OPEN) (
  IN  VOID           *Buffer,
  OUT EFI_ACPI_HANDLE *Handle
  );
```

Parameters

Buffer

      Points to the ACPI opcode.

Handle

      Upon return, holds the handle.

Related Definitions

```
typedef VOID *EFI_ACPI_HANDLE;
```

Description

Creates a handle from a single ACPI opcode.

Status Code Values

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*Buffer* is NULL or *Handle* is NULL or *Buffer* points to an invalid opcode.

EFI_ACPI_SDT_PROTOCOL.OpenSdt()

Summary

Create a handle for the first ACPI opcode in an ACPI system description table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_OPEN_SDT) (
  IN  UINTN            TableKey,
  OUT EFI_ACPI_HANDLE *Handle
  );
```

Parameters

TableKey

> The table key for the ACPI table, as returned by **GetTable()**.

Handle

> On return, points to the newly created ACPI handle. Type **EFI_ACPI_HANDLE** is defined in "Related Definitions" below.

Description

Creates an ACPI handle for the top-level opcodes in the ACPI system description table specified by *TableKey*.

Related Definitions

```
typedef VOID *EFI_ACPI_HANDLE;
```

Status Codes Returned

EFI_SUCCESS	*Handle* created successfully.
EFI_NOT_FOUND	*TableKey* does not refer to a valid ACPI table.

EFI_ACPI_SDT_PROTOCOL.Close()

Summary

Close an ACPI handle.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_CLOSE) (
  IN  EFI_ACPI_HANDLE Handle
  );
```

Parameters

Handle

> Returns the handle. Type **EFI_ACPI_HANDLE** is defined in **Open()**.

Description

Closes the ACPI handle and, if any changes were made, updates the table checksum.

Status Code Values

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*Handle* is NULL or does not refer to a valid ACPI object.

EFI_ACPI_SDT_PROTOCOL.GetChild()

Summary

Return the child ACPI objects.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_ENUM) (
  IN     EFI_ACPI_HANDLE ParentHandle,
  IN OUT EFI_ACPI_HANDLE *Handle
  );
```

Parameters

ParentHandle

> Parent handle.

Handle

> On entry, points to the previously returned handle or NULL to start with the first handle. On return, points to the next returned ACPI handle or NULL if there are no child objects.

Description

Iterates through all children ACPI objects of the ACPI object specified by the handle *ParentHandle*.

Status Code Values

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*ParentHandle* is NULL or does not refer to a valid ACPI object.

EFI_ACPI_SDT_PROTOCOL.GetOption()

Summary

Retrieve information about an ACPI object.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_GET_OPTION) (
  IN  EFI_ACPI_HANDLE    Handle,
  IN  UINTN              Index,
  OUT EFI_ACPI_DATA_TYPE *DataType,
  OUT CONST VOID         **Data,
  OUT UINTN              *DataSize
  );
```

Parameters

Handle

ACPI object handle.

Index

Index of the data to retrieve from the object. In general, indexes read from left-to-right
in the ACPI encoding, with index 0 always being the ACPI opcode.

DataType

Points to the returned data type or **EFI_ACPI_DATA_TYPE_NONE** if no data exists
for the specified index. See **EFI_ACPI_DATA_TYPE** in Related Definitions.

Data

Upon return, points to the pointer to the data.

DataSize

Upon return, points to the size of *Data*.

Related Definitions

```
typedef UINT32 EFI_ACPI_DATA_TYPE;

#define EFI_ACPI_DATA_TYPE_NONE        0
#define EFI_ACPI_DATA_TYPE_OPCODE      1
#define EFI_ACPI_DATA_TYPE_NAME_STRING 2
#define EFI_ACPI_DATA_TYPE_OP          3
#define EFI_ACPI_DATA_TYPE_UINT        4
#define EFI_ACPI_DATA_TYPE_STRING      5
#define EFI_ACPI_DATA_TYPE_CHILD       6
```

Description

Retrieves various fields encoded within the ACPI object. All ACPI objects support at least index 0.

The **EFI_ACPI_DATA_TYPE_NONE** indicates that the specified ACPI object does not support the specified option. The **EFI_ACPI_DATA_TYPE_OPCODE** indicates that the option is an ACPI opcode. The **EFI_ACPI_DATA_TYPE_NAME_STRING** indicates that the option is an ACPI name string. The **EFI_ACPI_DATA_TYPE_OP** indicates that the option is an ACPI opcode. The **Open()** function can be used to manipulate the contents of this ACPI opcode. The **EFI_ACPI_DATA_TYPE_UINT** indicates that the option is an unsigned integer. The size of the integer is indicated by *DataSize*. The **EFI_ACPI_DATA_TYPE_STRING** indicates that the option is a string whose length is indicated by *DataSize*. The **EFI_ACPI_DATA_TYPE_CHILD** indicates that the opcode has child data, pointed to by *Data*, with the size *DataSize*.

Table 5. AML terms and supported options

Term	0	1	2	3	4	5	6
ACPI_OP_ZERO	0000						
ACPI_OP_ONE	0001						
ACPI_OP_ALIAS	0006	N	N				
ACPI_OP_NAME	0008	N	O				
ACPI_OP_BYTE	000A	U8					
ACPI_OP_WORD	000B	U16					
ACPI_OP_DWORD	000C	U32					
ACPI_OP_STRING	000D	S					
ACPI_OP_QWORD	000E	U64					
ACPI_OP_SCOPE	0010	N					
ACPI_OP_BUFFER	0011	O					
ACPI_OP_PACKAGE	0012	U8					
ACPI_OP_PACKAGE1	0013	O					
ACPI_OP_METHOD	0014	N	U8				
ACPI_OP_LOCAL0	0060						
ACPI_OP_LOCAL1	0061						
ACPI_OP_LOCAL2	0062						
ACPI_OP_LOCAL3	0063						
ACPI_OP_LOCAL4	0064						
ACPI_OP_LOCAL5	0065						
ACPI_OP_LOCAL6	0066						
ACPI_OP_LOCAL7	0067						
ACPI_OP_ARG0	0068						
ACPI_OP_ARG1	0069						
ACPI_OP_ARG2	006A						
ACPI_OP_ARG3	006B						
ACPI_OP_ARG4	006C						
ACPI_OP_ARG5	006D						
ACPI_OP_ARG6	006E						
ACPI_OP_STORE	0070	O	O				
ACPI_OP_REFOF	0071	O					
ACPI_OP_ADD	0072	O	O	O			
ACPI_OP_CONCAT	0073	O	O	O			
ACPI_OP_SUBTRACT	0074	O	O	O			
ACPI_OP_INCREMENT	0075	O					
ACPI_OP_DECREMENT	0076	O					
ACPI_OP_MULTIPLY	0077	O	O	O			
ACPI_OP_DIVIDE	0078	O	O	O	O		

Term	0	1	2	3	4	5	6
ACPI_OP_SHIFTLEFT	0079	O	O	O			
ACPI_OP_SHIFTRIGHT	007A	O	O	O			
ACPI_OP_AND	007B	O	O	O			
ACPI_OP_NAND	007C	O	O	O			
ACPI_OP_OR	007D	O	O	O			
ACPI_OP_NOR	007E	O	O	O			
ACPI_OP_XOR	007F	O	O	O			
ACPI_OP_NOT	0080	O	O				
ACPI_OP_FINDSETLEFTBIT	0081	O	O				
ACPI_OP_FINDSETRIGHTBIT	0082	O	O				
ACPI_OP_DEREFOF	0083	O					
ACPI_OP_CONCATENATE	0084	O	O	O			
ACPI_OP_MODULO	0085	O	O	O			
ACPI_OP_NOTIFY	0086	O	O				
ACPI_OP_SIZEOF	0087	O					
ACPI_OP_INDEX	0088	O	O	O			
ACPI_OP_MATCH	0089	O	U8	O	U8	O	O
ACPI_OP_OBJECTTYPE	008E	O					
ACPI_OP_LAND	0090	O	O				
ACPI_OP_LOR	0091	O	O				
ACPI_OP_LNOT	0092	O					
ACPI_OP_LEQUAL	0093	O	O				
ACPI_OP_LGREATER	0094	O	O				
ACPI_OP_LLESS	0095	O	O				
ACPI_OP_TOBUFFER	0096	O	O				
ACPI_OP_TODECIMALSTRING	0097	O	O				
ACPI_OP_TOHEXSTRING	0098	O	O				
ACPI_OP_TOINTEGER	0099	O	O				
ACPI_OP_TOSTRING	009C	O	O	O			
ACPI_OP_COPYOBJECT	009D	O	O				
ACPI_OP_MID	009E	O	O	O			
ACPI_OP_CONTINUE	009F						
ACPI_OP_IF	00A0	O					
ACPI_OP_ELSE	00A1						
ACPI_OP_WHILE	00A2	O					
ACPI_OP_NOP	00A3						
ACPI_OP_RETURN	00A4	O					
ACPI_OP_BREAK	00A5						
ACPI_OP_BREAKPOINT	00CC						

Term	0	1	2	3	4	5	6
ACPI_OP_ONES	00FF						
ACPI_OP_MUTEX	5B01	N	U8				
ACPI_OP_EVENT	5B02	N					
ACPI_OP_CONDREFOF	5B12	O	O				
ACPI_OP_CREATEFIELD	5B13	O	O	O	N		
ACPI_OP_LOADTABLE	5B1F	O	O	O	O	O	O
ACPI_OP_LOAD	5B20	N	O				
ACPI_OP_STALL	5B21	O					
ACPI_OP_SLEEP	5B22	O					
ACPI_OP_ACQUIRE	5B23	O	U16				
ACPI_OP_SIGNAL	5B24	O					
ACPI_OP_WAIT	5B25	O	O				
ACPI_OP_RESET	5B26	O					
ACPI_OP_RELEASE	5B27	O					
ACPI_OP_FROMBCD	5B28	O	O				
ACPI_OP_TOBCD	5B29	O	O				
ACPI_OP_UNLOAD	5B2A	O					
ACPI_OP_REVISION	5B30						
ACPI_OP_DEBUG	5B31						
ACPI_OP_FATAL	5B32	U8	U32	O			
ACPI_OP_TIMER	5B33						
ACPI_OP_OPERATIONREGION	5B80	N	U8	O	O		
ACPI_OP_FIELD	5B81	N	U8				
ACPI_OP_DEVICE	5B82	N					
ACPI_OP_PROCESSOR	5B83	N	U8	U32	U8		
ACPI_OP_POWERRESOURCE	5B84	N	U8	U16			
ACPI_OP_THERMALZONE	5B85	N					
ACPI_OP_INDEXFIELD	5B86	N	N	U8			
ACPI_OP_BANKFIELD	5B87	N	N	O	U8		
ACPI_OP_DATAREGION	5B88	N	O	O	O		
ACPI_OP_CREATEDWORDFIELD	5B8A	O	O	N			
ACPI_OP_CREATEWORDFIELD	5B8B	O	O	N			
ACPI_OP_CREATEBYTEFIELD	5B8C	O	O	N			
ACPI_OP_CREATEBITFIELD	5B8D	O	O	N			
ACPI_OP_CREATEQWORDFIELD	5B8F	O	O	N			

Status Code Returns

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*Handle* is NULL or does not refer to a valid ACPI object.

EFI_ACPI_SDT_PROTOCOL.SetOption()

Summary

Change information about an ACPI object.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_SET_OPTION) (
  IN  EFI_ACPI_HANDLE    Handle,
  IN  UINTN              Index,
  IN  CONST VOID         *Data,
  IN  UINTN              DataSize
  );
```

Parameters

Handle

> ACPI object handle.

Index

> Index of the data to retrieve from the object. In general, indexes read from left-to-right in the ACPI encoding, with index 0 always being the ACPI opcode.

Data

> Points to the data.

DataSize

> The size of the *Data*.

Description

> Changes fields within the ACPI object. If the new size will not fit in the space occupied by the previous option, then his function will return **EFI_BAD_BUFFER_SIZE**. The list of opcodes and their related options can be found in **GetOption()**.

Status Code Returns

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*Handle* is NULL or does not refer to a valid ACPI object.
EFI_BAD_BUFFER_SIZE	Data cannot be accommodated in the space occupied by the option.

EFI_ACPI_SDT_PROTOCOL.FindPath()

Summary

Returns the handle of the ACPI object representing the specified ACPI path.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ACPI_FIND_PATH) (
  IN  EFI_ACPI_HANDLE HandleIn,
  IN  VOID            *AcpiPath,
  OUT EFI_ACPI_HANDLE *HandleOut
  );
```

Parameters

HandleIn

>   Points to the handle of the object representing the starting point for the path search.

AcpiPath

>   Points to the ACPI path, which conforms to the ACPI encoded path format.

HandleOut

>   On return, points to the ACPI object which represents *AcpiPath*, relative to
>   *HandleIn*.

Description

Starting with the ACPI object represented by *HandleIn*, walk the specified ACPI path *AcpiPath*
and return the handle of the ACPI object it refers to. This function supports absolute paths, relative
paths and the special rules applied to single name segments.

Status Code Returns

EFI_SUCCESS	Success
EFI_INVALID_PARAMETER	*HandleIn* is NULL or does not refer to a valid ACPI object.

10
PCI Host Bridge

10.1 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 11.6.1*. Silicon-related policies are described by the **EFI_PCI_OVERRIDE_PROTOCOL**, which is described in section 11.6.2.

10.2 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

10.3 PCI Host Bridge Resource Allocation Protocol

10.3.1 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**.

10.3.2 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.*
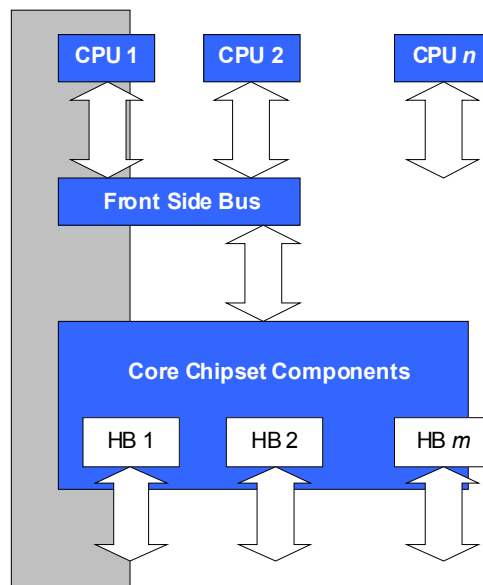
Figure 4. Host Bus Controllers

10.3.3 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.
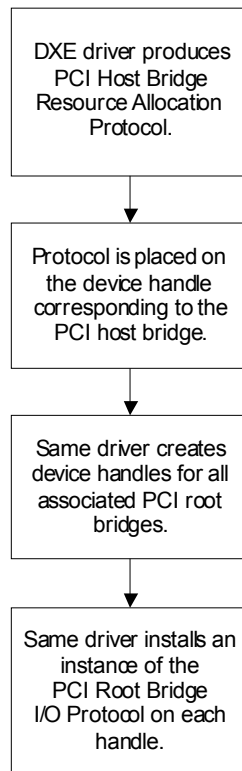
Figure 5. Producing the PCI Host Bridge Resource Allocation Protocol

10.3.4 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.

10.3.5 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`**.

10.4 Sample PCI Architectures

10.4.1 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.

10.4.2 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`**.
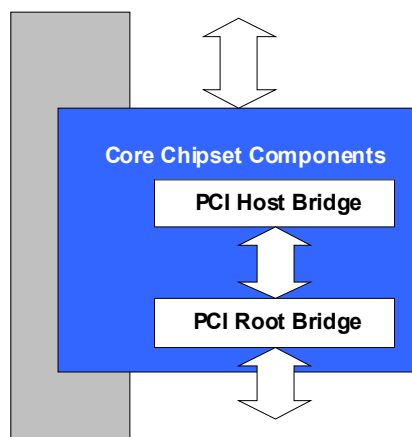
Figure 6. Desktop System with 1 PCI Root Bridge

10.4.3 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.
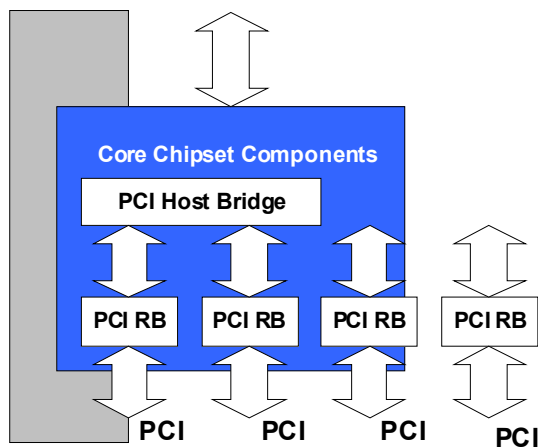
Figure 7. Server System with 4 PCI Root Bridges

10.4.4 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`.
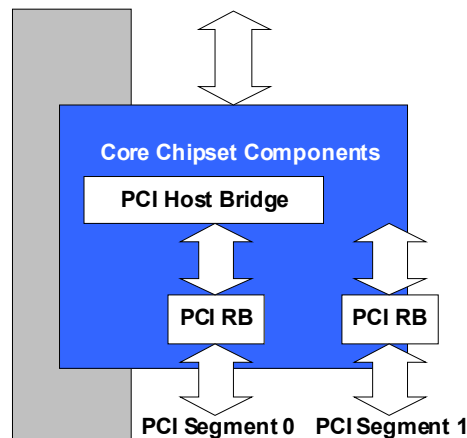
Figure 8. Server System with 2 PCI Segments

10.4.5 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.
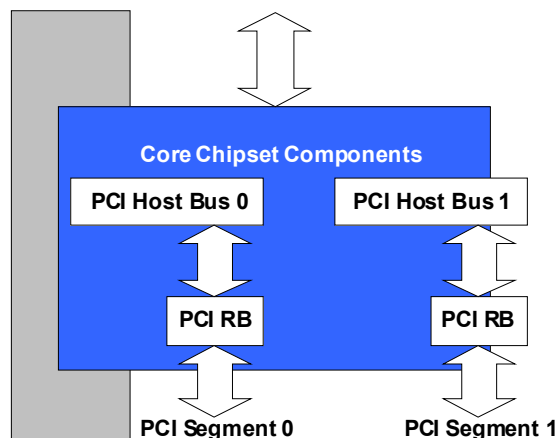
Figure 9. Server System with 2 PCI Host Buses

10.5 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 11.6.1.

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 10 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.

10.6 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 6. 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 7. 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.

10.7 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) {
    . . .
```

10.7.1 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**.

10.7.1.1 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**.

10.7.1.2 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.

10.7.1.3 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

10.7.2 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.

10.7.2.1 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.

10.7.2.2 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**.

10.8 PCI HostBridge Code Definitions

10.8.1 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**

10.8.2 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 8 and Table 9 below contain these two descriptor types. Table 10 and Table 11 define how resource-specific flags are used. See the *ACPI Specification* for details on the field values.

Table 8. 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 4KiB-1, the allocated address must be a multiple of 4 KiB. **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 9. 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 10. 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 SA I/O ranges: n100–n3FF n500–n7FF n900–nBFF nD00–nFFF See ISA Aliasing Considerations for more details.

Table 11. 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.Noti fyPhase()

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 10.7 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 12 provides a description of the fields in the above enumeration:

Table 12. 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()`.

Enumeration	Description
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.Get NextRootBridge()

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.Get AllocAttributes()

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 13. 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.Star tBusEnumeration()

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_ENUMERAT
ION) (
  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 14. 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.Set BusNumbers()

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 15. 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.Sub mitResources()

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 16. 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 KiB-1, the allocated address must be a multiple of 4 KiB.
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.Get ProposedResources()

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_RESOUR
CES) (
  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 17. ACPI 2.0 & 3.0 GetProposedResources() Resource Descriptor Field Values

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 18. 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.Preprocessor Controller()

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_CONT
ROLLER)(
  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 19. 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.

10.9 End of PCI Enumeration Overview

This specification defines the indicia to inform the platform when the PCI enumeration process has completed.   This allows for some post enumeration finalization actions to occur, if necessary.

10.9.1 End of PCI Enumeration Protocol

The indicia for this finalization action is a protocol. The obligation of the platform that supports this capability is as follows. Once PCI enumeration is complete, the **EFI_PCI_ENUMERATION_PROTOCOL** shall be installed on the same handle as the host bridge protocol.

This protocol is always installed with a NULL pointer.

GUID

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

The protocol can be used as an indicia by other DXE agents that the process of PCI device enumeration has been completed.

<div align="right">

11
PCI Platform

</div>

11.1 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

11.2 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.

• Operating system developers who will be adapting their shrink-wrapped operating system products to run on Intel architecture-based platforms.

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

11.3 PCI Platform Support Related Information

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

11.3.1 Industry Specifications

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

11.3.2 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*

11.4 PCI Platform Protocol

11.4.1 PCI Platform Protocol Overview

"PCI Host Bridge Resource Allocation Protocol", Section 10.8.2 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 10.8.2 for additional PCI-related design discussion.

11.5 Incompatible PCI Device Support Protocol

11.5.1 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**.

11.5.2 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.

11.6 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**

11.6.1 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.PreprocessController()**.

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.

11.6.2 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 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.

11.6.3 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 20 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 20 and Table 21 below contain these two descriptor types. See the *ACPI Specification* for details on the field values.

Table 20. 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 if the value is 0. Ignored if the PCI BAR is I/O. Ignored if the PCI BAR is 32-bt memory. If PCI BAR is 64-bit memory and this field is 32, then the PCI BAR resource is allocated below 4GB. If the PCI BAR is 64-bit memory and this field is 64, then the PCI BAR resource is allocated above 4GB.
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 21. 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**.

12
Hot Plug PCI

12.1 HOT PLUG 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.

12.2 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`.

12.3 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/*

12.4 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.*

12.5 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 10.3 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.

12.6 PCI 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_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of 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_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of 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 *UEFI 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 22. 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 i*s 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_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of 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 10.8.2. 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 10.8 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 23. 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.

12.7 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.

12.8 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.GetAllocAttr ibutes() 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()**.

- Continue to scan devices on that root bridge and start the initialization of all root HPCs.
- Call
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumber s() 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.

13
Super I/O Protocol

13.1 Super I/O Protocol

EFI_SIO_PROTOCOL

Summary

The Super I/O driver installs an instance of this protocol on the handle of every device within the Super I/O chip.

GUID

```
#define EFI_SIO_PROTOCOL_GUID  \
  { 0x215fdd18, 0xbd50, 0x4feb, { 0x89, 0xb, 0x58, 0xca, \
  0xb, 0x47, 0x39, 0xe9 } }
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_PROTOCOL {
    EFI_SIO_REGISTER_ACCESS    RegisterAccess;
    EFI_SIO_GET_RESOURCES      GetResources;
    EFI_SIO_SET_RESOURCES      SetResources;
    EFI_SIO_POSSIBLE_RESOURCES PossibleResources;
    EFI_SIO_MODIFY             Modify;
} EFI_SIO_PROTOCOL;
```

Parameters

RegisterAccess

Provides a low level access to the registers for the Super I/O.

GetResources

Provides a list of current resources consumed by the device in ACPI Resource Descriptor Format.

SetResources

Sets resources for a device.

PossibleResources

Provides a collection of possible resource descriptors for the device. Each resource descriptor in the collection defines a combination of resources that can potentially be used by the device.

Modify

Provides an interface for table based programming of the Super I/O registers.

Description

The Super I/O Protocol is installed by the Super I/O driver. The Super I/O driver is a UEFI driver model compliant driver. In the **Start()** routine of the Super I/O driver, a handle with an instance of **EFI_SIO_PROTOCOL** is created for each device within the Super I/O. The device within the Super I/O is powered up, enabled, and assigned with the default set of resources. In the **Stop()** routine of the Super I/O driver, the device is disabled and Super I/O protocol is uninstalled.

EFI_SIO_PROTOCOL.RegisterAccess()

Summary

Provides a low level access to the registers for the Super I/O.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_REGISTER_ACCESS) (
  IN CONST EFI_SIO_PROTOCOL *This,
  IN BOOLEAN                Write,
  IN BOOLEAN                ExitCfgMode,
  IN UINT8                  Register,
  IN OUT UINT8              *Value
  );
```

Parameters

This

Indicates a pointer to the calling context.

Write

Specifies the type of the register operation. If this parameter is **TRUE**, *Value* is interpreted as an input parameter and the operation is a register write. If this parameter is **FALSE**, *Value* is interpreted as an output parameter and the operation is a register read.

ExitCfgMode

Exit Configuration Mode Indicator. If this parameter is set to **TRUE**, the Super I/O driver will turn off configuration mode of the Super I/O prior to returning from this function. If this parameter is set to **FALSE**, the Super I/O driver will leave Super I/O in the configuration mode.

The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to register access.

Register

Register number.

Value

If *Write* is **TRUE**, *Value* is a pointer to the buffer containing the byte of data to be written to the Super I/O register. If *Write* is **FALSE**, *Value* is a pointer to the destination buffer for the byte of data to be read from the Super I/O register.

Description

The **RegisterAccess()** function provides low level interface to the registers in the Super I/O.

Note: *This function only provides access to the internal registers of the Super I/O chip. For example, on a typical desktop system, these are the registers accessed via the 0x2E/0x2F indexed port I/O.*

This function cannot be used to access I/O or memory locations assigned to individual logical devices.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	The *Value* is NULL
EFI_INVALID_PARAMETER	Invalid *Register* number

EFI_SIO_PROTOCOL.GetResources()

Summary

Provides an interface to get a list of the current resources consumed by the device in the ACPI Resource Descriptor format.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_GET_RESOURCES)(
  IN  CONST EFI_SIO_PROTOCOL    *This,
  OUT ACPI_RESOURCE_HEADER_PTR  *ResourceList
  );
```

Parameters

This

> Indicates a pointer to the calling context.

ResourceList

> A pointer to an ACPI resource descriptor list that defines the current resources used by the device. Type **ACPI_RESOURCE_HEADER_PTR** is defined in the "Related Definitions" below.

Description

GetResources() returns a list of resources currently consumed by the device. The *ResourceList* is a pointer to the buffer containing resource descriptors for the device. The descriptors are in the format of Small or Large ACPI resource descriptor as defined by ACPI specification (2.0 & 3.0). The buffer of resource descriptors is terminated with the 'End tag' resource descriptor.

Related Definitions

```
typedef union {
  UINT8 Byte;
  struct{
    UINT8 Length : 3;
    UINT8 Name   : 4;
    UINT8 Type   : 1;
  }Bits;
} ACPI_SMALL_RESOURCE_HEADER;

typedef struct {
  union {
    UINT8 Byte;
    struct{
      UINT8 Name   : 7;
      UINT8 Type   : 1;
    }Bits;
  } Header;
  UINT16 Length;
} ACPI_LARGE_RESOURCE_HEADER;

typedef union {
  ACPI_SMALL_RESOURCE_HEADER *SmallHeader;
  ACPI_LARGE_RESOURCE_HEADER *LargeHeader;
} ACPI_RESOURCE_HEADER_PTR;
```

Length

Length of the resource descriptor in bytes.

Name

Resource descriptor name. Possible values for this field are defined in the ACPI specification.

Type

Descriptor type.

0 – ACPI Small Resource Descriptor

1 – ACPI Large Resource Descriptor

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	ResourceList is NULL

EFI_SIO_PROTOCOL.SetResources()

Summary

Sets the resources for the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_SET_RESOURCES)(
  IN CONST EFI_SIO_PROTOCOL   *This,
  IN ACPI_RESOURCE_HEADER_PTR ResourceList
  );
```

Parameters

This

> Indicates a pointer to the calling context.

ResourceList

> Pointer to the ACPI resource descriptor list. Type **ACPI_RESOURCE_HEADER_PTR**
> is defined in the "Related Definitions" section of
> **EFI_SIO_PROTOCOL.GetResources()**.

Description

SetResources() sets the resources for the device. *ResourceList* is a pointer to the ACPI
resource descriptor list containing requested resources for the device.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*ResourceList* is invalid
EFI_ACCESS_DENIED	Some of the resources in *ResourceList* are in use

EFI_SIO_PROTOCOL.PossibleResources()

Summary

Provides a collection of resource descriptor lists. Each resource descriptor list in the collection defines a combination of resources that can potentially be used by the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_POSSIBLE_RESOURCES)(
  IN  CONST EFI_SIO_PROTOCOL     *This,
  OUT ACPI_RESOURCE_HEADER_PTR   *ResourceCollection
  );
```

Parameters

This

       Indicates a pointer to the calling context.

ResourceCollection

       Collection of the resource descriptor lists. Type **ACPI_RESOURCE_HEADER_PTR** is defined in the "Related Definitions" section of **EFI_SIO_PROTOCOL.GetResources()**.

Description

PossibleResources() returns a collection of resource descriptor lists. Each resource descriptor list in the collection defines a combination of resources that can potentially be used by the device. The descriptors are in the format of Small or Large ACPI Resource Descriptor as defined by the *ACPI Specification* (2.0 & 3.0). The collection is terminated with the 'End tag' resource descriptor.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*ResourceCollection* is NULL

EFI_SIO_PROTOCOL.Modify()

Summary

Provides an interface for a table based programming of the Super I/O registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_MODIFY)(
  IN CONST EFI_SIO_PROTOCOL        *This,
  IN CONST EFI_SIO_REGISTER_MODIFY *Command,
  IN UINTN                          NumberOfCommands
  );
```

Parameters

This

> Indicates a pointer to the calling context.

Command

> A pointer to an array of *NumberOfCommands* **EFI_SIO_REGISTER_MODIFY** structures. Each structure specifies a single Super I/O register modify operation. Type **EFI_SIO_REGISTER_MODIFY** is defined in the "Related Definitions" below.

NumberOfCommands

> Number of elements in the *Command* array.

Description

The **Modify()** function provides an interface for table based programming of the Super I/O registers. This function can be used to perform programming of multiple Super I/O registers with a single function call. For each table entry, the *Register* is read, its content is bitwise ANDed with *AndMask,* and then ORed with *OrMask* before being written back to the *Register*. The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to table processing. Once the table is processed, the Super I/O device has to be returned to the original state.

Note: *This function only provides access to the internal registers of the Super I/O chip. For example, on a typical desktop system, these are the registers accessed via the 0x2E/0x2F indexed port I/O.*

This function cannot be used to access I/O or memory locations assigned to individual logical devices.

Related Definitions

```
typedef struct {
  UINT8      Register;
  UINT8      AndMask;
  UINT8      OrMask;
} EFI_SIO_REGISTER_MODIFY;
```

Register

> Register number.

AndMask

> Bitwise AND mask.

OrMask

> Bitwise OR mask.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*Command* is NULL

14
Super I/O and ISA Host Controller Interactions

14.1 Design Descriptions

The PI architecture provides a means to interact in a standard fashion with Super I/O devices. For the purposes of this specification, the Super I/O is a device residing on an ISA or LPC or similar bus that consumes I/O and/or memory resources and provides multiple standard logical devices, such as PC/ AT compatible floppy, serial port, parallel port, keyboard or mouse. There may be more than one of these devices behind each of the ISA/LPC buses.
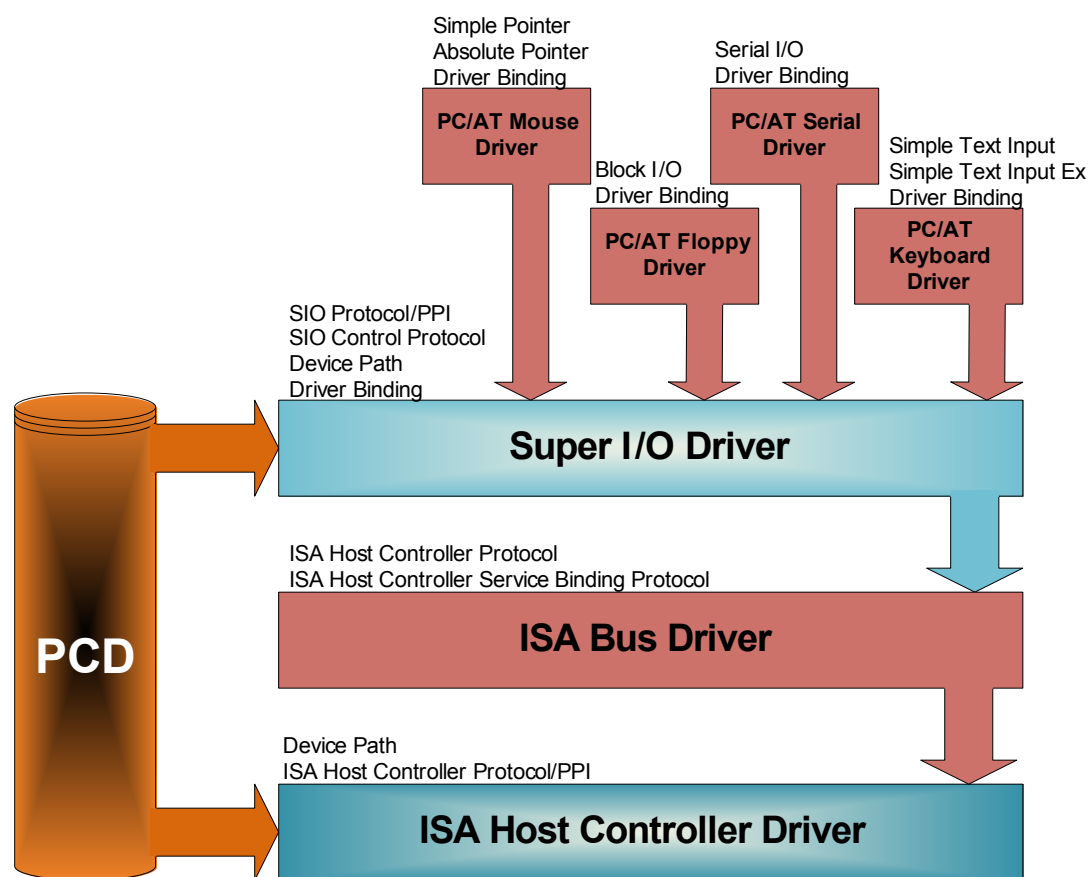
Figure 10. Super I/O and ISA Host Controller Interactions

Mouse, Floppy, Serial and keyboard Drivers

The Mouse, Floppy, Serial and Keyboard drivers are UEFI driver-model drivers that support devices produced by the Super I/O component. When started, they use the optional SIO Control protocol to enable the logical device, to produce the standard UEFI protocols used for console or booting, such as Serial I/O or Block I/O. They typically examine the device paths on the child handles created by

the Super I/O drivers for the ACPI device path nodes that refer to their devices (e.g. PNP0501, PNP0303, etc.).

Super I/O Driver

The Super I/O driver consists of a UEFI driver-model driver (in DXE) and PEIM (in PEI) that supports a Super I/O component. The Super I/O components support multiple logical devices, such as the PS/2 keyboard controller, a floppy controller or serial/IrDa controller. When started, the Super I/O driver verifies it is present on the board and produces child handles for each of the logical devices that are enabled. On each child handle it installs an instance of the Device Path protocol, the SIO protocol and the SIO Control protocol.

ISA Bus Driver

The ISA Bus driver consists of a UEFI driver-model driver (DXE only) that produces the ISA Host Controller Service Binding protocol, which manages the many-to-one relationship between Super I/O drivers in the system and an ISA Host Controller.

ISA Host Controller Driver

The ISA Host Controller driver is a DXE driver that supports a PCI-ISA or PCI-LPC bridge component. It creates a child handle that represents the ISA Bus and installs the ISA Host Controller protocol and the Device Path protocol with an ACPI device path node (PNP0A05/PNP0A06).

PCD

The Platform Configuration Database (PCD) provides configuration information about the device configuration. Information concerning configured I/O addresses can be placed into the PCD by platform drivers and then used by the various silicon drivers, including SIO to find base addresses and logical device configuration.

14.1.1 Super I/O

The Super I/O DXE driver and PEIM encapsulate the functionality of the Super I/O component. They are both responsible to:

- Detect the presence of the component, using information from the PCD and the apertures opened by the ISA host controller.
- Configure the component and its logical devices using information from the PCD.
- Publish information about the component and the logical devices it supports using the SIO protocol/PPI.

14.1.1.1 DXE

The Super I/O DXE Driver is responsible for:

- Producing the Driver Binding protocol's **Supported()**, **Start()** and **Stop()** member functions on the driver image handle.
- Installing the same GUID as used for the SioGuid member of the Super I/O PPI on the image handle. This allows other drivers to detect which Super I/O is present in the system.

- Checking Super I/O controller presence. The **Supported()** function must check whether the Super I/O controller is present in the system and whether the handle has an instance of the ISA Host Controller Service Binding protocol installed on it. For more information, see "Working With The ISA Bus", below.

- Creatingchild handles for each logical device. The **Start()** function creates a child handle for each logical device using the ISA Host Controller Service Binding protocol and installs the SIO and SIO Control protocols on each one. For more information, see "Working With Logical Devices", below.

14.1.1.1.1 Working with the ISA Bus

The system may contain an ISA bus bridge and zero or more Extended I/O bus bridges. The Super I/O driver checks each of these to see whether it is present.

Supported()

The Super I/O DXE driver's Driver Binding protocol **Supported()** function typically performs the following steps:

1. Verifies that the controller handle has an installed instance of the ISA Host Controller Service Binding protocol.
2. Opens the apertures necessary to access the component's configuration I/O address (i.e. 0x2e/ 0x2f) using the ISA Host Controller protocol.
3. Verifies the device's signature to determine whether the component is actually present using these configuration I/O addresses. For example, it might read a device-specific register and check for a signature.
4. Closes the aperture and any opened protocols.

Start()

The Super I/O DXE driver's Driver Binding protocol **Start()** function typically performs the following steps:

1. Detects whether Super I/O DXE driver is already managing the device indicated by the configuration I/O address. One method of doing this is to create a Device Path with the configuration I/O address embedded in one device node, then use *LocateDevicePath* to determine whether a child handle with the ISA Host Controller protocol installed, exists.
2. Creates a child handle for the SIO using the ISA Host Controller Service Binding protocol.
   - Opens the I/O apertures used for the configuration I/O address.
   - Installs an instance of the Device Path and (optionally) the SIO and SIO Control protocol
3. Creates child handles for each logical device. Install an instance of the Device Path and SIO protocol and (optionally) the SIO Control protocol on each child handle.
4. Installs an instance of the Device Path, SIO and SIO Control protocols on each of the child handles.

Stop()

The Super I/O DXE driver's Driver Binding protocol **Stop()** function typically performs the following steps:

1. Uninstalls the instances of the Device Path, SIO and SIO control protocols from each of the child handles.
2. Destroys the Super I/O's own child handle using the ISA Host Controller Service Binding protocol.

SetResources()

The Super I/O DXE driver's SIO protocol **SetResources()** function typically calls the **OpenIoAperture()** and **CloseIoAperture()** member functions of the ISA Host Controller protocol for the I/O addresses related to the individual logical devices.

14.1.1.1.2 Working with Logical Devices

The Super I/O controller supports many different logical devices. Some of these devices, such as the floppy controller, keyboard controller, MIDI controller and serial port are standard PC/AT devices. These drivers produce interfaces based on these industry-standard interfaces. Also, the Super I/O component itself may act as a logical device.

For each logical device, the following steps are taken during **Start()**:

1. Create a child handle.
2. Install the **EFI_SIO_PROTOCOL** (with correct current resource settings) on the child handle.
3. Install the **EFI_SIO_CONTROL_PROTOCOL** on the same child handle. This protocol allows a standard drivers to correctly enable and disable their resources when the **Start()** and **Stop()** members of the Driver Binding protocol that they produce is called.
4. If the device implements one of the standard PC/AT devices, install the **EFI_DEVICE_PATH_PROTOCOL** by appending a device node containing the ACPI HID of the PC/AT device to the device path of the ISA bus on which it is installed..

For more information, see "Logical Devices"

14.1.1.2 PEI

The Super I/O PEIM is responsible to:

- Read its configuration information from the PCD.
- Detect if the Super I/O device is present in the system. If necessary, it should open the aperture required to access the configuration registers using the ISA Host Controller PPI. If the Super I/O device is not detected, the driver should close the aperture and exit immediately.
- Install the **EFI_SIO_PPI** for the Super I/O. The Identifier field allows consumers of the PPI to know which device's register set can be accessed by using the PPI's functions, in cases where multiple Super I/Os are supported on a platform.
- Allocate I/O and memory resources. All I/O and memory resources are allocated using the **EFI_ISA_HC_PPI**, which handles opening and closing bridge apertures.

The Super I/O PEIM should have the **EFI_ISA_HC_PPI** in its dependency expression.

14.1.2 ISA Bus

The ISA Bus is the logical device that manages the child devices attached to the ISA Host Controller.

It consumes the ISA Host Controller protocol produced by the ISA Host Controller and installs the ISA Host Controller Service Binding protocol on the ISA Host Controller's handle.

14.1.3 ISA Host Controller

The Host Controller is the device that translates the memory and I/O cycles from a parent device (such as a PCI bus) into memory and I/O cycles for the target devices.

14.1.3.1 DXE

The ISA Host Controller driver creates a child handle for the ISA Host Controller and installs an instance of the ISA Host Controller protocol and Device Path protocol on it. The Device Path instance for the child handle will have an extra ACPI device path node for either PNP0A05 (standard subtractive-decode ISA bus) or PNP0A06 (positive-decode extended I/O bus). If a bridge device can support more than one of these simultaneously, the _UID field of the device path node must contain a different value.

For PCI-ISA/LPC bridges, there are two classes of the ISA Host Controller Driver: generic and chipset-specific. The generic ISA Bus driver connects to any standard subtractive-decode PCI-ISA bridge device (class code:6, sub-class: 1, programming I/F 0).

Chipset-specific ISA Bus Drivers are used for PCI-ISA (or PCI-LPC) bridges that support positive decode. These bridges have device-specific mechanisms for opening and closing the I/O and memory apertures. These apertures determine which address ranges will be passed through to devices attached to the ISA/LPC side of the bridge. In this case, the registration process includes opening of apertures and guaranteeing that I/O access falls within the addresses that go the specified bus.

The ISA Host Controller is responsible for reporting the actual address and size of the apertures using the DXE GCD services.

14.1.3.2 PEI

The ISA Bus PEIM comes in two versions: generic and chipset-specific.

The generic version is used for subtractive-decode ISA (or LPC) buses. It implements the **EFI_ISA_HC_PPI** with a device identifier of all zeroes. All of the aperture functions report **EFI_UNSUPPORTED**.

The chipset-specific version implements the **EFI_ISA_HC_PPI**, which opens and close apertures for ISA/LPC buses that are positive decode. The device identifier is filled in with the PCI PFA of the PCI-ISA bridge device.

14.1.4 Logical Devices

Logical Device drivers are UEFI driver model drivers that support many of the standard PC/AT peripherals. They are designed to connect to the device paths produced by the Super I/O DXE driver. Each of these drivers produces the Driver Binding and related protocols used in implementing UEFI driver model drivers.

Each of these drivers supports more than one instance of a specific device can be in a system. Calls to **Stop()** and **Start()** will disable or enable the device and stop consumption of all system resources. This allows Super I/O drivers to be loaded and unloaded. These drivers can use the SIO

Control protocol to enable consumption of system I/O and memory resources when they are started or stopped.

14.2 Code Definitions

14.2.1 EFI_SIO_PPI

Summary

Super I/O register access.

GUID

```
#define EFI_SIO_PPI_GUID \
  {0x23a464ad, 0xcb83, 0x48b8, \
  {0x94, 0xab, 0x1a, 0x6f, 0xef, 0xcf, 0xe5, 0x22}}
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_PPI {
  EFI_PEI_SIO_REGISTER_READ     Read;
  EFI_PEI_SIO_REGISTER_WRITE    Write;
  EFI_PEI_SIO_REGISTER_MODIFY   Modify;
  EFI_GUID                      SioGuid;
  PEFI_SIO_INFO                 Info;
} EFI_SIO_PPI, *PEFI_SIO_PPI;
```

Members

Read

> This function reads a register's value from the Super I/O controller.

Write

> This function writes a value to a register in the Super I/O controller.

Modify

> This function modifies zero or more registers in the Super I/O controller using a table.

SioGuid

> This GUID uniquely identifies the Super I/O controller.

Info

> This pointer is to an array which maps EISA identifiers to logical devices numbers.

Description

This PPI provides low-level access to Super I/O registers using **Read()** and **Write()**. It also uniquely identifies this Super I/O controller using a GUID and provides mappings between ACPI-style PNP IDs and the logical device numbers. There is one instance of this PPI per Super I/O device.

This PPI is produced by the Super I/O PEIM after the driver has determined that it is present in the system.

Related Definitions

```
typedef struct _EFI_SIO_INFO {
  EFI_ACPI_HID   Hid;
  EFI_ACPI_UID   Uid;
  UINT8          Ldn;
} EFI_SIO_INFO, *PEFI_SIO_INFO;
Hid
```

This is the EISA-style Plug-and-Play identifier for one of the devices on the super I/O controller. The standard values are:

EFI_ACPI_PNP_HID_KBC - 101/102-key Keyboard

EFI_ACPI_PNP_HID_LPT - Standard parallel port

EFI_ACPI_PNP_HID_COM - Standard serial port

EFI_ACPI_PNP_HID_FDC - Standard floppy controller

EFI_ACPI_PNP_HID_MIDI - Standard MIDI controller

EFI_ACPI_PNP_HID_GAME - Standard joystick controller

EFI_ACPI_PNP_HID_END - Specifies the end of the information list.

Uid

> This is the unique zero-based instance number for a device on the super I/O. For example, if there are two serial ports, one of them would have a Uid of 0 and the other would have a Uid of 1.

Ldn

> This is the Logical Device Number for this logical device in the Super I/O. This value can be used in the **Read()** and **Write()** functions. The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.


```
typedef UINT32 EFI_ACPI_HID;
typedef UINT32 EFI_ACPI_UID;

#define EFI_ACPI_PNP_HID_KBC  EFI_PNP_ID(0x0303)
#define EFI_ACPI_PNP_HID_LPT  EFI_PNP_ID(0x0400)
#define EFI_ACPI_PNP_HID_COM  EFI_PNP_ID(0x0500)
#define EFI_ACPI_PNP_HID_FDC  EFI_PNP_ID(0x0700)
#define EFI_ACPI_PNP_HID_MIDI EFI_PNP_ID(0xB006)
#define EFI_ACPI_PNP_HID_END  EFI_PNP_ID(0x0000)
#define EFI_ACPI_PNP_HID_GAME EFI_PNP_ID(0xB02F)
```

```
#pragma pack(1)
typedef struct _EFI_SIO_INFO {
  EFI_ACPI_HID      Hid;
  EFI_ACPI_UID      Uid;
  UINT8             Ldn;
} EFI_SIO_INFO, *PEFI_SIO_INFO;
#pragma pack()
```

14.2.1.1 EFI_SIO_PPI.Read()

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SIO_REGISTER_READ) (
  IN EFI_PEI_SERVICES   **PeiServices,
  IN CONST EFI_SIO_PPI  *This,
  IN BOOLEAN            ExitCfgMode,
  IN EFI_SIO_REGISTER   Register,
  OUT UINT8             *IoData
  );
```

Parameters

PeiServices

> A pointer to a pointer to the PEI Services.

This

> A pointer to this instance of the **EFI_SIO_PPI**.

ExitCfgMode

> A boolean specifying whether the driver should turn on configuration mode (FALSE) or turn off configuration mode (TRUE) after completing the read operation. The driver must track the current state of the configuration mode (if any) and turn on configuration mode (if necessary) prior to register access.

Register

> A value specifying the logical device number (bits 15:8) and the register to read (bits 7:0). The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.

IoData

> A pointer to the returned register value.

Description

This function provides low-level read access to a device register. The register is specified as an 8-bit logical device number and an 8-bit register value. The logical device numbers for specific SIO devices can be determined using the Info member of the PPI structure.

If this function completes successfully, it will return **EFI_SUCCESS** and *IoData* will point to the returned Super I/O register value. If the register value was invalid for this device or *IoData* was NULL, then it will return **EFI_INVALID_PARAMETERS**. If the register could not be read within the correct amount of time, it will return **EFI_TIMEOUT**. If the device had some sort of fault or the device was not present, it will return **EFI_DEVICE_ERROR**.

Return Values

This function returns standard EFI status codes.

Status Code Value	Description
EFI_SUCCESS	Success.
EFI_TIMEOUT	The register could not be read in the a reasonable amount of time. The exact time is device-specific.
EFI_INVALID_PARAMETERS	Register was out of range for this device. *IoData* was NULL
EFI_DEVICE_ERROR	There was a device fault or the device was not present.

Related Definitions

```
typedef UINT16 EFI_SIO_REGISTER;

#define EFI_SIO_REG(ldn,reg) (EFI_SIO_REGISTER)(((ldn)<<8)|reg)

#define EFI_SIO_LDN_GLOBAL 0xFF
```

14.2.1.2 EFI_SIO_PPI.Write()

Write a Super I/O register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SIO_REGISTER_WRITE) (
  IN EFI_PEI_SERVICES     **PeiServices,
  IN CONST EFI_SIO_PPI    *This,
  IN BOOLEAN              ExitCfgMode,
  IN EFI_SIO_REGISTER     Register,
  IN UINT8                IoData
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services.

This

A pointer to this instance of the **EFI_SIO_PPI**.

ExitCfgMode

>   A boolean specifying whether the device should turn on configuration mode (FALSE) or turn off configuration mode (TRUE) after completing the write operation. The driver must track the current state of the configuration mode (if any) and turn on configuration mode (if necessary) prior to register access.

Register

>   A value specifying the logical device number and the register to read. The logical device number can be determined by using the Super I/O chip specification or by looking up the value in the Info field of the **EFI_SIO_PPI**. The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.

IoData

>   An 8-bit register value.

Status Code Return

Status Code Value	Description
EFI_SUCCESS	Success.
EFI_TIMEOUT	The register could not be read in the a reasonable amount of time. The exact time is device-specific.
EFI_INVALID_PARAMETERS	Register was out of range for this device. *IoData* was NULL
EFI_DEVICE_ERROR	There was a device fault or the device was not present.

Description

This function provides low-level write access to a Super I/O register.

The register is specified as an 8-bit logical device number and an 8-bit register value. The logical device numbers for specific SIO devices can be determined using the Info member of the PPI structure.

If this function completes successfully, it will return **EFI_SUCCESS** and *IoData* will point to the returned Super I/O register value. If the register value was invalid for this device or *IoData* was NULL, then it will return **EFI_INVALID_PARAMETERS**. If the register could not be read within the correct amount of time, it will return **EFI_TIMEOUT**. If the device had some sort of fault or the device was not present, it will return **EFI_DEVICE_ERROR**.

14.2.1.3 EFI_SIO_PPI.Modify()

Summary

Provides an interface for a table based programming of the Super I/O registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_MODIFY)(
  IN EFI_PEI_SERVICES            **PeiServices,
  IN CONST EFI_SIO_PPI            *This,
  IN CONST EFI_SIO_REGISTER_MODIFY *Command,
  IN UINTN                         NumberOfCommands
  );
```

Parameters

PeiServices

> A pointer to a pointer to the PEI Services.

This

> A pointer to this instance of the **EFI_SIO_PPI**.

Command

> A pointer to an array of *NumberOfCommands* **EFI_SIO_REGISTER_MODIFY** structures. Each structure specifies a single Super I/O register modify operation. Type **EFI_SIO_REGISTER_MODIFY** is defined in **EFI_SIO_PROTOCOL.Modify()**.

NumberOfCommands

> The number of elements in the Command array.

Description

The **Modify()** function provides an interface for table based programming of the Super I/O registers. This function can be used to perform programming of multiple Super I/O registers with a single function call. For each table entry, the *Register* is read, its content is bitwise ANDed with *AndMask*, and then ORed with *OrMask* before being written back to the *Register*. The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to table processing. Once the table is processed, the Super I/O device must be returned to the original state.

Status Code Return

Status Code Value	Description
EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETERS	*Command* is NULL

14.2.2 EFI_ISA_HC_PPI

GUID

```
#define EFI_ISA_HC_PPI_GUID \
  {0x8d48bd70, 0xc8a3, 0x4c06, \
  {0x90, 0x1b, 0x74, 0x79, 0x46, 0xaa, 0xc3, 0x58}}
```

PPI Structure

```
typedef struct _EFI_ISA_HC_PPI {
  UINT32 Version;

  UINT32 Address;
  EFI_PEI_ISA_HC_OPEN_IO      OpenIoAperture;
  EFI_PEI_ISA_HC_CLOSE_IO     CloseIoAperture;
} EFI_ISA_HC_PPI, *PEFI_ISA_HC_PPI;
```

Members

Version

An unsigned integer that specifies the version of the PPI structure. Initialized to zero.

PciAddress

The address of the ISA/LPC Bridge device. For PCI, this is the segment, bus, device and function of the a ISA/LPC Bridge device.

If bits 24-31 are 0, then the definition is:

Bits 0:2 – Function

Bits 3-7 – Device

Bits 8:15 – Bus

Bits 16-23 – Segment

Bits 24-31 – Bus Type

If bits 24-31 are 0xff, then the definition is platform-specific.

OpenIoAperture

Opens an aperture on a positive-decode ISA Host Controller.

CloseIoAperture

Closes an aperture on a positive-decode ISA Host Controller.

14.2.2.1 EFI_ISA_HC_PPI.OpenIoAperture()

Open I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ISA_HC_OPEN_IO) (
  IN CONST EFI_ISA_HC_PPI  *This,
  IN UINT16                IoAddress,
  IN UINT16                IoLength,
  OUT UINT64               *IoApertureHandle
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services Table.

This

A pointer to this instance of the **EFI_ISA_HC_PPI**.

IoAddress

An unsigned integer that specifies the first byte of the I/O space required.

IoLength

An unsigned integer that specifies the number of bytes of the I/O space required.

IoApertureHandle

A pointer to the returned I/O aperture handle. This value can be used on subsequent calls to **CloseIoAperture()**.

Description

This function opens an I/O aperture in a ISA Host Controller for the I/O addresses specified by *IoAddress* to *IoAddress* + *IoLength* - 1. It is possible that more than one caller may be assigned to the same aperture.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and doesa not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

If this function completes successfully, then it returns **EFI_SUCCESS**. If there is no available I/O aperture, then this function returns **EFI_OUT_OF_RESOURCES**.

14.2.2.2 EFI_ISA_HC_PPI.CloseIoAperture()

Close I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ISA_HC_CLOSE_IO) (
  IN CONST EFI_ISA_HC_PPI    *This,
  IN UINT64                  IoApertureHandle
  );
```

Parameters

PeiServices

> A pointer to a pointer to the PEI Services Table.

This

> A pointer to this instance of the **EFI_ISA_HC_PPI**.

IoApertureHandle

> The I/O aperture handle previously returned from a call to **OpenIoAperture()**.

Description

This function closes a previously opened I/O aperture handle. If there are no more I/O aperture handles that refer to the hardware I/O aperture resource, then the hardware I/O aperture is closed.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

14.2.3 EFI_ISA_HC_PROTOCOL

Summary

Provides registration and enumeration of ISA devices.

GUID

```
#define EFI_ISA_HC_PROTOCOL_GUID \
  {0xbcdaf080, 0x1bde, 0x4e22, \
  {0xae, 0x6a, 0x43, 0x54, 0x1e, 0x12, 0x8e, 0xc4}}
```

Protocol Interface Structure

```
typedef struct _EFI_ISA_HC_PROTOCOL {
  UINT32 Version;

  EFI_ISA_HC_OPEN_IO       OpenIoAperture;
  EFI_ISA_HC_CLOSE_IO      CloseIoAperture;
} EFI_ISA_HC_PROTOCOL, *PEFI_ISA_HC_PROTOCOL;
```

Members

Version

> The version of this protocol. Higher version numbers are backward compatible with lower version numbers. The current version is 0.

OpenIoAperture

> Open an I/O aperture.

CloseIoAperture

> Close an I/O aperture.

Description

This protocol provides registration for ISA devices on a positive- or subtractive-decode ISA bus. It allows devices to be registered and also handles opening and closing the apertures which are positively-decoded.

14.2.3.1 EFI_ISA_HC_PROTOCOL.OpenIoAperture()

Open I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ISA_HC_OPEN_IO) (
  IN CONST EFI_ISA_HC_PROTOCOL  *This,
  IN UINT16                     IoAddress,
  IN UINT16                     IoLength,
  OUT UINT64                    *IoApertureHandle
  );
```

Parameters

This

> A pointer to this instance of the **EFI_ISA_HC_PROTOCOL**.

IoAddress

> An unsigned integer that specifies the first byte of the I/O space required.

IoLength

> An unsigned integer that specifies the number of bytes of the I/O space required.

IoApertureHandle

> A pointer to the returned I/O aperture handle. This value can be used on subsequent calls to **CloseIoAperture()**.

Description

This function opens an I/O aperture in a ISA Host Controller for the I/O addresses specified by *IoAddress* to *IoAddress* + *IoLength* - 1.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

If this function completes successfully, then it returns **EFI_SUCCESS**. If there is no available I/O aperture, then this function returns **EFI_OUT_OF_RESOURCES**.

14.2.3.2 EFI_ISA_HC_PROTOCOL.CloseIoAperture()

Close I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ISA_HC_CLOSE_IO) (
  IN CONST EFI_ISA_HC_PROTOCOL *This,
  IN UINT64                     IoApertureHandle
  );
```

Parameters

PeiServices

> A pointer to a pointer to the PEI Services Table.

This

> A pointer to this instance of the **EFI_ISA_HC_PROTOCOL**.

IoApertureHandle

> The I/O aperture handle previously returned from a call to **OpenIoAperture()**.

Description

This function closes a previously opened I/O aperture handle. If there are no more I/O aperture handles that refer to the hardware I/O aperture resource, then the hardware I/O aperture is closed.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

14.2.4 EFI_ISA_HC_SERVICE_BINDING_PROTOCOL

Summary

Manages child devices for an ISA Host Controller.

GUID

```
#define EFI_ISA_HC_SERVICE_BINDING_PROTOCOL_GUID \
  {0xfad7933a, 0x6c21, 0x4234, \
  {0xa4, 0x34, 0x0a, 0x8a, 0x0d, 0x2b, 0x07, 0x81}}
```

Protocol Interface Structure

The protocol interface structure is the same for all service binding protocols and can be found in Section 10.6 ("EFI Service Binding Protocol").

Description

The ISA Host Controller Service Binding protocol permits multiple Super I/O devices to use the services provide by an ISA Host Controller. The function **CreateChild()** installs an instance of the ISA Host Controller protocol on each child handle created.

14.2.5 EFI_SIO_CONTROL_PROTOCOL

Summary

Provide low-level services for SIO devices that enable them to be used in the UEFI driver model.

GUID

```
#define EFI_SIO_CONTROL_PROTOCOL_GUID \
  {0xb91978df, 0x9fc1, 0x427d, \
  {0xbb, 0x5, 0x4c, 0x82, 0x84, 0x55, 0xca, 0x27}}
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_CONTROL_PROTOCOL {
  UINT32 Version;

  EFI_SIO_CONTROL_ENABLE        EnableDevice;
  EFI_SIO_CONTROL_DISABLE       DisableDevice;
} EFI_SIO_CONTROL_PROTOCOL, PEFI_SIO_CONTROL_PROTOCOL;
```

Members

Version

The version of this protocol. Higher version numbers are backward compatible with lower version numbers. The current version is 0.

EnableDevice

Enable a device.

DisableDevice

Disable a device.

Description

The **EFI_SIO_CONTROL_PROTOCOL** provides control over the decoding of Super I/O and memory resources by a logical device within a Super I/O. While the logical devices often implement industry standard interfaces (such as PS/2 keyboard or serial port), these standard interfaces do not describe how to enable or disable the memory and I/O resources for those devices. Instead, this control is usually implemented within the Super I/O device itself through proprietary means. The industry standard drivers may utilize these functions in their implementations of the Driver Binding protocol's **Start()** and **Stop()** functions.

The Super I/O driver installs this protocol on the same child handle as the **EFI_SIO_PROTOCOL**.

14.2.5.1 EFI_SIO_CONTROL_PROTOCOL.Enable()

Summary

Enable an ISA-style device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_CONTROL_ENABLE)(
  IN CONST EFI_SIO_CONTROL_PROTOCOL *This
  );
```

Parameters

This

A pointer to this instance of the **EFI_SIO_CONTROL_PROTOCOL**.

Description

This function enables a logical ISA device and, if necessary, configures it to default settings, including memory, I/O, DMA and IRQ resources.

If the function completed successfully, then this function returns **EFI_SUCCESS**.

If the device could not be enabled because there were insufficient resources either for the device itself or for the records needed to track the device, then this function returns **EFI_OUT_OF_RESOURCES**.

If this device is already enabled, then this function returns **EFI_ALREADY_STARTED**. If this device cannot be enabled, then this function returns **EFI_UNSUPPORTED**.

14.2.5.2 EFI_SIO_CONTROL_PROTOCOL.Disable()

Summary

Disable a logical ISA device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_CONTROL_DISABLE)(
  IN CONST EFI_SIO_CONTROL_PROTOCOL *This
  );
```

Parameters

This

A pointer to this instance of the **EFI_SIO_CONTROL_PROTOCOL**.

Description

This function disables a logical ISA device so that it no longer consumes system resources, such as memory, I/O, DMA and IRQ resources. Enough information must be available so that subsequent **Enable()** calls would properly reconfigure the device.

If this function completed successfully, then it returns **EFI_SUCCESS**.

If the device could not be disabled because there were insufficient resources either for the device itself or for the records needed to track the device, then this function returns **EFI_OUT_OF_RESOURCES**.

If this device is already disabled, then this function returns **EFI_ALREADY_STARTED**. If this device cannot be disabled, then this function returns **EFI_UNSUPPORTED**.

15
CPU I/O Protocol

 This document describes the CPU I/O Protocol. This protocol provides an I/O abstraction for a system processor. This protocol is used by a PCI root bridge I/O driver to perform memory-mapped I/O and I/O transactions. The I/O or memory primitives can be used by the consumer of the protocol to materialize bus-specific configuration cycles, such as the transitional configuration address and data ports for PCI. Only drivers that require direct access to the entire system should use this protocol. This is a boot-services only protocol.

15.1 CPU I/O Protocol Terms

The following are the terms that are used throughout this document to describe the CPU I/O Protocol.

coherency domain

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

CPU I/O Protocol

> A software abstraction that provides access to the I/O and memory regions in a single coherency domain.

SMP

> Symmetric multiprocessing. A collection of processors that share a common view of I/O and memory-mapped I/O.

15.2 CPU I/O Protocol2 Description

This section describes the CPU I/O Protocol. This protocol is used by code—typically PCI root bridge I/O drivers and drivers that need I/O prior to the loading of the PCI root bridge I/O driver—that is running in the EFI Boot Services environment to access memory and I/O.This protocol can be also used by non-PC-AT* systems to abstract the I/O mechanism published by the processor and/or integrated CPU-I/O complex.

See Code Definitions for the definition of **EFI_CPU_IO_PROTOCOL2**.

15.2.1 EFI CPU I/O Overview

The interfaces that are provided in the **EFI_CPU_IO2_PROTOCOL** are for performing basic operations to memory and I/O. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **EFI_CPU_IO2_PROTOCOL** allows for future innovation of the platform. It abstracts processor-device-specific code from the system memory map. This abstraction allows system

designers to make changes to the system memory map without impacting platform-independent code that is consuming basic system resources.

Systems with one to many processors in a symmetric multiprocessing (SMP) configuration will contain a single instance of the *E***FI_CPU_IO2_PROTOCOL**. This protocol is an abstraction from a software point of view. This protocol is attached to the device handle of a processor driver. The CPU I/O Protocol is the parent to a set of PCI Root Bridge I/O Protocol instances that may contain many PCI segments. A CPU I/O Protocol instance might also be the parent of a series of protocols that abstract host-bus attached devices.

CPU I/O Protocol instances are either produced by the system firmware or an EFI driver. When a CPU I/O Protocol is produced, it is placed on a device handle without an EFI Device Path Protocol instance. The figure below shows a device handle that has the **EFI_CPU_IO2_PROTOCOL** installed on it.

FG10010
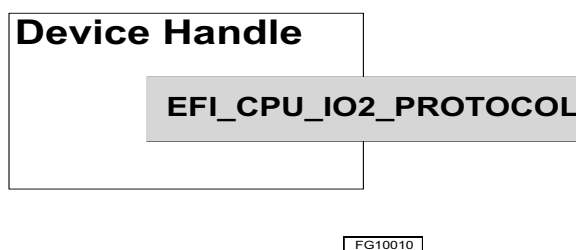
Figure 11. EFI CPU I/O2 Protocol

Other characteristics of the CPU I/O Protocol include the following:

- The protocol uses re-entrancy to enable possible use by a debugger agent that is outside of the generic EFI Task Priority Level (TPL) priority mechanism.

See Code Definitions for the definition of **EFI_CPU_IO2_PROTOCOL**.

15.3 Code Definitions

This section contains the basic definitions of the CPU I/O Protocol (**EFI_CPU_IO2_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 protocol or function definition:

- **EFI_CPU_IO_PROTOCOL_ACCESS**
- **EFI_CPU_IO_PROTOCOL_WIDTH**

15.3.1 CPU I/O Protocol

EFI_CPU_IO2_PROTOCOL

Summary

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices in a system.

GUID

```
#define EFI_CPU_IO2_PROTOCOL_GUID  \
  {0xad61f191, 0xae5f, 0x4c0e, 0xb9, 0xfa, 0xe8, 0x69, 0xd2, \
  0x88, 0xc6, 0x4f}
```

Protocol Interface Structure

```
typedef struct _EFI_CPU_IO2_PROTOCOL {
  EFI_CPU_IO_PROTOCOL_ACCESS        Mem;
  EFI_CPU_IO_PROTOCOL_ACCESS        Io;
} EFI_CPU_IO2_PROTOCOL;
```

Parameters

Mem.Read

> Allows reads from memory-mapped I/O space. See the **Mem.Read()** function description. Type **EFI_CPU_IO_PROTOCOL_ACCESS** is defined in "Related Definitions" below.

Mem.Write

> Allows writes to memory-mapped I/O space. See the **Mem.Write()** function description.

Io.Read

> Allows reads from I/O space. See the **Io.Read()** function description. Type **EFI_CPU_IO_PROTOCOL_ACCESS** is defined in "Related Definitions" below.

Io.Write

> Allows writes to I/O space. See the **Io.Write()** function description.

Description

The **EFI_CPU_IO2_PROTOCOL** provides the basic memory and I/O interfaces that are used to abstract accesses to platform hardware. This hardware can include PCI- or host-bus-attached peripherals and buses. There is one **EFI_CPU_IO2_PROTOCOL** instance for each PI System. Embedded systems, desktops, and workstations will typically have only one PI System. Non–symmetric multiprocessing (non-SMP), high-end servers may have multiple PI Systems. A device driver that wishes to make I/O transactions in a system will have to retrieve the **EFI_CPU_IO2_PROTOCOL** instance. A device handle for an PI System will minimally contain an **EFI_CPU_IO2_PROTOCOL** instance.

Related Definitions

```
//***************************************************
// EFI_CPU_IO2_PROTOCOL_ACCESS
//***************************************************
typedef struct {
  EFI_CPU_IO_PROTOCOL_IO_MEM  Read;
  EFI_CPU_IO_PROTOCOL_IO_MEM  Write;
} EFI_CPU_IO_PROTOCOL_ACCESS;
```

Read

> This service provides the various modalities of memory and I/O read.

Write

> This service provides the various modalities of memory and I/O write.

EFI_CPU_IO2_PROTOCOL.Mem.Read() and Mem.Write()

Summary

Enables a driver to access memory-mapped registers in the PI System memory space.

Prototype

```
typedef
EFI_STATUS

(EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN     EFI_CPU_IO2_PROTOCOL          *This,
  IN     EFI_CPU_IO_PROTOCOL_WIDTH     Width,
  IN     UINT64                        Address,
  IN     UINTN                         Count,
  IN OUT VOID                          *Buffer
  );
```

Parameters

This

> A pointer to the **EFI_CPU_IO2_PROTOCOL** instance.

Width

> Signifies the width of the memory operation. Type
> **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in "Related Definitions" below.

Address

> The base address of the memory operation.

Count

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

Buffer

> For read operations, the destination buffer to store the results. For write operations, the
> source buffer from which to write data.

Description

The **Mem.Read()** and **Mem.Write()** functions enable a driver to access memory-mapped
registers in the PI System memory space.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying
any alignment and memory width restrictions that a PI System on a platform might require. For
example, on some platforms, width requests of **EfiCpuIoWidthUint64** do not work.
Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**,
or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the
Count operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16, EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Related Definitions

```
//*****************************************************
// EFI_CPU_IO_PROTOCOL_WIDTH
//*****************************************************
typedef enum {
  EfiCpuIoWidthUint8,
  EfiCpuIoWidthUint16,
  EfiCpuIoWidthUint32,
  EfiCpuIoWidthUint64,
  EfiCpuIoWidthFifoUint8,
  EfiCpuIoWidthFifoUint16,
  EfiCpuIoWidthFifoUint32,
  EfiCpuIoWidthFifoUint64,
  EfiCpuIoWidthFillUint8,
  EfiCpuIoWidthFillUint16,
  EfiCpuIoWidthFillUint32,
  EfiCpuIoWidthFillUint64,
  EfiCpuIoWidthMaximum
} EFI_CPU_IO_PROTOCOL_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PI System.
EFI_INVALID_PARAMETER	*Width* is invalid for this PI System.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this PI System.

EFI_CPU_IO2_PROTOCOL.Io.Read() and Io.Write()

Summary

Enables a driver to access registers in the PI CPU I/O space.

Prototype

```
typedef
EFI_STATUS
 (EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN      EFI_CPU_IO2_PROTOCOL           *This,
  IN      EFI_CPU_IO_PROTOCOL_WIDTH      Width,
  IN      UINT64                         Address,
  IN      UINTN                          Count,
  IN OUT  VOID                           *Buffer
  );
```

Parameters

This

> A pointer to the **EFI_CPU_IO2_PROTOCOL** instance.

Width

> Signifies the width of the I/O operation. Type **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in **EFI_CPU_IO2_PROTOCOL.Mem()**.

Address

> The base address of the I/O operation. The caller is responsible for aligning the *Address* if required.

Count

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

Buffer

> For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The *Io.Read()* and **Io.Write()** functions enable a driver to access PCI controller registers in the PI CPU I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and I/O width restrictions that a PI System on a platform might require. For example on some platforms, width requests of **EfiCpuIoWidthUint64** do not work. Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**, or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16**, **EfiCpuIoWidthFillUint32**, or **EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PI System.
EFI_INVALID_PARAMETER	*Width* is invalid for this PI System.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this PI System.

16
Legacy Region Protocol

This section describes the legacy region protocol that abstracts the platform capability for the BIOS memory region from 0xC0000 to 0xFFFFF. The Legacy Region Protocol is used to abstract the hardware control of the Option ROM and Compatibility 16-bit code region shadowing.

16.1 Legacy Region Protocol

The Legacy Region Protocol controls the read, write and boot-lock attributes for the region 0xC0000 to 0xFFFFF. The table below lists the functions that are included in the Legacy Region Protocol. See **EFI_LEGACY_REGION2_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 24. Functions in Legacy Region Protocol

Function	Description
Decode()	Programs the chipset to decode or not decode regions in the 0xC0000 to 0xFFFFF range. Governs the read attribute.
Lock()	Programs the chipset to lock (write protect) regions in the 0xC0000 to 0xFFFFF range. Disables the write attribute.
BootLock()	Programs the chipset to boot-lock regions in the 0xC0000 to 0xFFFFF range. Enables the boot-lock attribute.
Unlock()	Programs the chipset to unlock regions in the 0xC0000 to 0xFFFFF range. Enables the write attribute.
GetInfo()	Get information about the granularity of the regions for each attribute.

16.2 Code Definitions

16.2.1 Legacy Region Protocol

EFI_LEGACY_REGION2_PROTOCOL

Summary

Abstracts the hardware control of the physical address region 0xC0000–0xFFFFF.

GUID

```
#define EFI_LEGACY_REGION2_PROTOCOL_GUID \
  { 0x70101eaf, 0x85, 0x440c, 0xb3, 0x56, 0x8e, 0xe3, 0x6f,\
  0xef, 0x24, 0xf0 }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_REGION2_PROTOCOL {
  EFI_LEGACY_REGION2_DECODE                  Decode;
  EFI_LEGACY_REGION2_LOCK                    Lock;
  EFI_LEGACY_REGION2_BOOT_LOCK               BootLock;
  EFI_LEGACY_REGION2_UNLOCK                  UnLock;
  EFI_LEGACY_REGION_GET_INFO                 GetInfo;
} EFI_LEGACY_REGION2_PROTOCOL;
```

Parameters

Decode

>   Modify the read attribute of a memory region. See the **Decode()** function
>   description.

Lock

>   Modify the write attribute of a memory region to prevent writes. See the **Lock()**
>   function description.

BootLock

>   Modify the boot-lock attribute of a memory region to prevent future changes to the
>   memory attributes for this region. See the **BootLock()** function description.

Unlock

>   Modify the write attribute of a memory region to allow writes. See the **Unlock()**
>   function description.

GetInfo

>   Modify the write attribute of a memory region to allow writes. See the **GetInfo()**
>   function description.

Description

The **EFI_LEGACY_REGION2_PROTOCOL** is used to abstract the hardware control of the memory
attributes of the Option ROM shadowing region, 0xC0000 to 0xFFFFF.

There are three memory attributes that can be modified through this protocol: read, write and boot-
lock. These protocols may be set in any combination.

EFI_LEGACY_REGION2_PROTOCOL.Decode()

Summary

Modify the hardware to allow (decode) or disallow (not decode) memory reads in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_DECODE) (
  IN  EFI_LEGACY_REGION2_PROTOCOL  *This,
  IN  UINT32                        Start,
  IN  UINT32                        Length,
  OUT UINT32                       *Granularity ,
  IN  BOOLEAN                      *On
  );
```

Parameters

This

> Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

> The beginning of the physical address of the region whose attributes should be modified.

Length

> The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

> The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

On

> Decode / Non-Decode flag.

Description

If the On parameter evaluates to **TRUE**, this function enables memory reads in the address range Start to (Start + Length - 1).

If the *On* parameter evaluates to **FALSE**, this function disables memory reads in the address range Start to (Start + Length - 1).

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.Lock()

Summary

Modify the hardware to disallow memory writes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_LOCK) (
  IN  EFI_LEGACY_REGION2_PROTOCOL *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                      *Granularity
  );
```

Parameters

This

>Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

>The beginning of the physical address of the region whose attributes should be modified.

Length

>The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

>The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function changes the attributes of a memory range to not allow writes.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.BootLock()

Summary

Modify the hardware to disallow memory attribute changes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_BOOT_LOCK) (
  IN  EFI_LEGACY_REGION2_PROTOCOL  *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                       *Granularity
  );
```

Parameters

This

> Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

> The beginning of the physical address of the region whose attributes should be modified.

Length

> The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

> The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function makes the attributes of a region read only. Once a region is boot-locked with this function, the read and write attributes of that region cannot be changed until a power cycle has reset the boot-lock attribute. Calls to **Decode()**, **Lock()** and **Unlock()** will have no effect.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully locked.
EFI_INVALID_PARAMETER	If Start or Length describe an address not in the Legacy Region.
EFI_UNSUPPORTED	The chipset does not support locking the configuration registers in a way that will not affect memory regions outside the legacy memory region.

EFI_LEGACY_REGION2_PROTOCOL.UnLock()

Summary

Modify the hardware to allow memory writes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_UNLOCK) (
  IN  EFI_LEGACY_REGION2_PROTOCOL  *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                       *Granularity
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

The beginning of the physical address of the region whose attributes should be modified.

Length

The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function changes the attributes of a memory range to allow writes.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.GetInfo()

Summary

Get region information for the attributes of the Legacy Region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_GET_INFO) (
  IN  EFI_LEGACY_REGION2_PROTOCOL   *This,
  OUT UINT32                        *DescriptorCount,
  OUT EFI_LEGACY_REGION_DESCRIPTOR  **Descriptor
  );
```

Parameters

This

> Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

DescriptorCount

> The number of region descriptor entries returned in the *Descriptor* buffer. Type **EFI_LEGACY_REGION_DESCRIPTOR** is defined in the "Related Definitions" section.

Descriptor

> A pointer to a pointer used to return a buffer where the legacy region information is deposited. This buffer will contain a list of *DescriptorCount* number of region descriptors. This function will provide the memory for the buffer.

Description

This function is used to discover the granularity of the attributes for the memory in the legacy region. Each attribute may have a different granularity and the granularity may not be the same for all memory ranges in the legacy region.

Status Codes Returned

EFI_SUCCESS	The information structure was returned.
EFI_UNSUPPORTED	This function is not supported.

Related Definitions

```
typedef enum {
  LegacyRegionDecoded,
  LegacyRegionNotDecoded,
  LegacyRegionWriteEnabled,
  LegacyRegionWriteDisabled,
  LegacyRegionBootLocked,
  LegacyRegionNotLocked
} EFI_LEGACY_REGION_ATTRIBUTE;
```

LegacyRegionDecoded

This region is currently set to allow reads.

LegacyRegionNotDecoded

This region is currently set to not allow reads.

LegacyRegionWriteEnabled

This region is currently set to allow writes.

LegacyRegionWriteDisabled

This region is currently set to write protected.

LegacyRegionBootLocked

This region's attributes are locked, cannot be modified until after a power cycle.

LegacyRegionNotLocked

This region's attributes are not locked.

```
typedef struct {
  UINT32                                Start;
  UINT32                                Length;
  EFI_LEGACY_REGION_ATTRIBUTE           Attribute;
  UINT32                                Granularity;
} EFI_LEGACY_REGION_DESCRIPTOR;
```

Start

The beginning of the physical address of this region.

Length

The number of bytes in this region.

Attribute

Attribute of the Legacy Region Descriptor that describes the capabilities for that memory region.

Granularity

Describes the byte length programmability associated with the *Start* address and the specified *Attribute* setting.

17
I²C Protocol Stack

17.1 Design Discussion

The Inter-Integrated Circuit (I²C) protocol stack enables third party silicon vendors to write UEFI drivers for their products by decoupling the I²C chip details from the I²C controller and I²C bus configuration details.

17.1.1 I²C Bus Overview

The Inter-Integrated Circuit (I²C) bus enables simple low speed communications between chips. The following sections describe the attributes of the I²C bus configurations supported by the I²C protocol stack and the I²C-bus specification and user manual.

17.1.1.1 Single Master
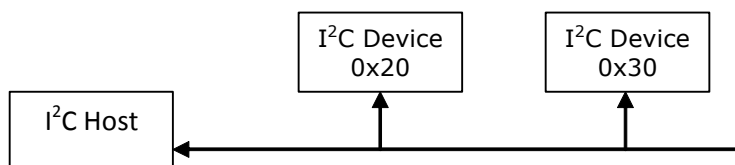
Figure 12. Simple 1²C Bus

Figure 12 shows a simple I²C bus configuration consisting of one host controller and two I²C devices which use the same I²C clock frequency. In this configuration the I²C host controller gets initialized with a single clock frequency and performs transactions to the I²C devices using their slave addresses.

17.1.1.2 Multiple I²C Bus Frequencies
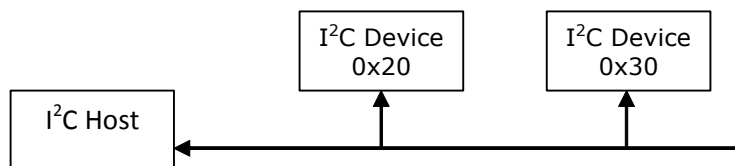
Figure 13. Multiple I²C Bus Frequencies

Two I^{2}C bus configurations are shown in Figure 13 separated by a switch. This allows the I^{2}C bus to operate at two different frequencies depending on the state of the switch. Device requiring higher bus frequencies are placed closer to the I^{2}C host controller and are accessed when the switch is turned off. Devices using lower bus frequencies are placed after the switch and may only be accessed when the switch is on. Note that the I^{2}C bus frequency needs to be set to a frequency supported by all devices currently accessible by the I^{2}C host controller.

17.1.1.3 Limited Address Space
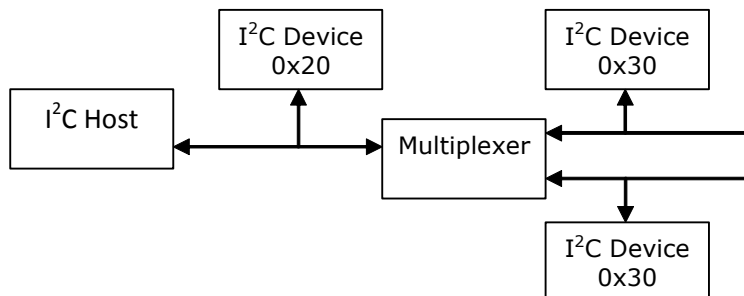
Figure 14. Limited address Space

I^{2}C devices have a limited number of address settings, sometimes only one. When the hardware design requires more I^{2}C devices than the address space supports a multiplexer may be introduced to create additional bus configurations (address spaces). Note that the host must first select the appropriate bus configuration before communicating with the I^{2}C device.

17.1.1.4 I^{2}C Bus Configurations

A bus configuration is a concept introduced by the I^{2}C protocol stack to configure the state of the switches and multiplexers in the I^{2}C bus. The I^{2}C protocol stack calls into the platform code with a value from zero (0) to N-1 to request the platform code enable a specific configuration of the switches and multiplexers. The platform code then sets the requested state for the switches and multiplexers and sets the I^{2}C clock frequency for this I^{2}C bus configuration. Upon return the I^{2}C protocol stack is able to access the I^{2}C devices in this configuration.

17.1.2 I^{2}C Protocol Stack Overview

The following is a representation of the I^{2}C protocol stack and an I^{2}C bus layout.

```
+-----------------+
|   Application   |
+-----------------+
         |
         | Third Party or UEFI
         |
         V
+-----------------+
```

```
        |   Third Party  |
        |   I2C Device   |
        |     Driver     |
        +----------------+
                |
                |
                V
        +----------------+
        | I2C IO Protocol |
        +----------------+
                |
                |
                V
        +----------------+   +-----------------------+
        | I2C Bus Driver |-->| I2C Enumerate Protocol |
        +----------------+   +-----------------------+
                |
                |
                V
       +------------------+
       | I2C Host Protocol |---------.
       +------------------+          |
                |                     V
                |          +----------------------+
                |          | I2C Bus Configuration |
                |<---------| Management Protocol   |
                V          +----------------------+
+--------------------+             |      |
| I2C Master Protocol |            |      |
+--------------------+             |      |
                |
Software   |                       |      |
-------------------------------------------------
Hardware   |                       |      |
                |
                V                   |      |
        +----------------+
        | I2C Controller |          |      |
        +----------------+
                |                   |      |
-----------------------
I2C Bus    |                        |      |
                |    +------------+
           +----| High speed |      |      |
                |    | I2C device |
                |    |    0x01    |   |      |
                |    +------------+
                |                    |      |
        +---------+  0
```

```
            | Switch  |<- - - - - - - - `       |
           +---------+  1
                |                                |
                |     +------------+             |
           +----| Fast speed |              |
                |     | I2C device |
                |     |    0x02    |              |
                |     +------------+
                |                                |
        +-------------+
        | Multiplexer |< - - - - - - - - - - - `
        +------------+
         0 |         | 1
           |         |
           |         |
           |         |   +-------------+
           |     +----| Third Party |
           |         |   | I2C Device  |
           |         |   |  0x03, 0x04 |
           |         |   +-------------+
           |         |
           |
           |             +-------------+
        +------------| Third Party |
           |             | I2C Device  |
           |             |  0x03, 0x04 |
           |             +-------------+
           |
```

Figure 15. I^{2}C Protocol Stack

The platform hardware designer chooses the bus layout based upon the platform, I^{2}C chip and software requirements. The design uses switches to truncate the bus to enable higher bus frequencies for a subset of devices which are placed closer to the controller. When the switch is on, the extended bus must operate at a lower bus frequency. The design uses multiplexer to create separate address spaces enabling the use of multiple devices which would otherwise have conflicting addresses. See the I^{2}C-bus specification and user manual for more details.

N.B. Some operating systems may prohibit the changing of switches and multiplexers in the I^{2}C bus. In this case the platform hardware and software designers must select a single I^{2}C bus configuration consisting of constant input values for the switches and multiplexers. The I^{2}C subsystem must be placed in the OS compatible I^{2}C bus configuration upon successful completion of **ExitBootServices()**.

The platform hardware designer needs to provide the platform software designer the following data for each I^{2}C bus:

1. Which controller controls this bus

2. A list of logic blocks contained in one or more I^{2}C devices:

- I^{2}C device which contains this logic block
- Logic block I^{2}C slave address
- Logic block description

3. For each configuration of the switches and multiplexers in the I^{2}C bus

   - What is the maximum frequency of operation for the I^{2}C bus
   - What I^{2}C slave addresses are accessible

4. The settings for the switches and multiplexers when control is given to the operating system.

17.1.2.1 Handles

The I^{2}C protocol stack uses two groups of handles:

- I^{2}C controller handles
- I^{2}C device handles

Some bus driver (PCI, USB, etc.) or the platform specific code may expose a handle for each of the I^{2}C controllers. The platform specific code installs the I^{2}C bus configuration management and I^{2}C enumeration protocols on the controller handle. As the I^{2}C stack is initialized, additional protocols are placed on the I^{2}C controller handle. When the I^{2}C stack initialization is complete, the controller handle contains:

I2C Controller Handle
EFI_I2C_HOST_PROTOCOL
EFI_I2C_ENUMERATE_PROTOCOL
EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL
EFI_I2C_MASTER_PROTOCOL
EFI_DEVICE_PATH_PROTOCOL
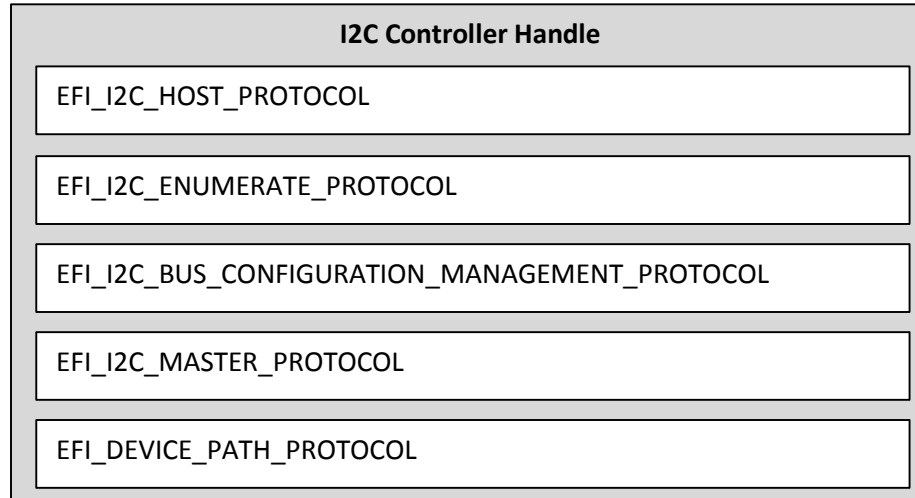
The I^{2}C Bus Driver uses the **EFI_I2C_ENUMERATE_PROTOCOL** to enumerate the set of I^{2}C devices connected to an I^{2}C controller, and creates an I^{2}C device handle for each I^{2}C device installing the following protocols on each:

```
┌──────────────────────────────────────────────────────────┐
│                    I2C Device Handle                       │
│  ┌──────────────────────────────────────────────────────┐ │
│  │  EFI_I2C_IO_PROTOCOL                                   │ │
│  └──────────────────────────────────────────────────────┘ │
│  ┌──────────────────────────────────────────────────────┐ │
│  │  EFI_DEVICE_PATH_PROTOCOL                              │ │
│  └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
```
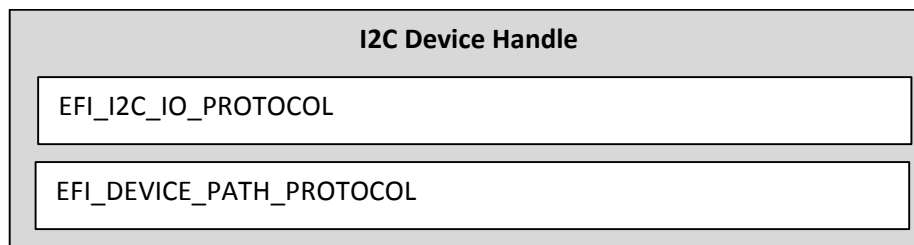
It is possible for the SMBus Host Controller Protocol to be implemented using the services on an I^{2}C Controller Handle. The SMBus Host Controller Protocol does not support the concept of multiple bus configurations, so the state of the I^{2}C controller handle required for the SMBus Host Controller Protocol to be produced on an I^{2}C Controller Handle is as follows:

```
┌──────────────────────────────────────────────────────────┐
│                  I2C Controller Handle                     │
│  ┌──────────────────────────────────────────────────────┐ │
│  │  EFI_I2C_MASTER_PROTOCOL                               │ │
│  └──────────────────────────────────────────────────────┘ │
│  ┌──────────────────────────────────────────────────────┐ │
│  │  EFI_DEVICE_PATH_PROTOCOL                              │ │
│  └──────────────────────────────────────────────────────┘ │
└──────────────────────────────────────────────────────────┘
```
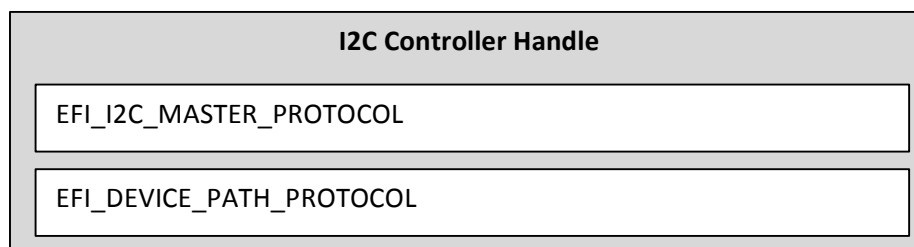
17.1.2.2 Driver Loading Order

A race condition potentially exists between the platform specific code and a layered SMBus driver when a driver for a PCI or USB I^{2}C controller installs the **EFI_I2C_MASTER_PROTOCOL** on its handle. The layered SMBus driver may start on this controller as soon as the **EFI_I2C_MASTER_PROTOCOL** is installed as long as the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** is not installed on the controller handle. However if the platform specific code wants to use this controller with the **EFI_I2C_HOST_PROTOCOL** then the platform specific code needs to prevent the SMBus driver from starting by installing the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL**. Note that the I^{2}C host protocol opens the **EFI_I2C_MASTER_PROTOCOL** only if the handle contains the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL**.

Chapter 10 of the *Universal Extensible Firmware Interface Specification* describes several ways for the platform specific code to adjust the driver load order. One possible way to eliminate this race condition is to use the version number for the driver binding protocol. The platform specific code implements the driver binding protocol's *Supported()* and *Start()* routines and sets the *version* field to a value in the range of **0xfffffff0 – 0xffffffff**. The SMBus driver should set the version field of the driver binding protocol to a value in the range of **0x00000010 – 0xffffffef**. This selection delays the SMBus driver to execute its *Supported()* and *Start()* routines after the platform specific code, enabling the platform specific code to install the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** and the **EFI_I2C_ENUMERATE_PROTOCOL** on the controller's handle.

17.1.2.3 Third Party I^{2}C Drivers

Third party I^{2}C drivers are I^{2}C chip specific but platform and host controller independent.

Third party I^{2}C driver writers, typically silicon vendors, need to provide:

- The vendor specific GUID that is used to select their driver.

- I^{2}C slave address array guidance (described below) when the I^{2}C device uses more than one I^{2}C slave address consisting of the order for the blocks of logic that get referenced by the entries in the slave address array.

The hardware version of the I^{2}C device, this value is passed to the third party I^{2}C driver to enable it to perform workarounds for the specific hardware version. It is recommended that value match the value in the ACPI _HRV tag. See the *Advanced Configuration and Power Interface Specification, Revision 5.0* for the field format and the *Plug and play support for I2C* web-page for restriction on values.

The third party I^{2}C driver uses relative addressing to abstract the platform specific details of the I^{2}C device. Using an example I^{2}C device containing an accelerometer and a magnetometer which consumes two I^{2}C slave addresses, one for each logic block. The third party I^{2}C driver writer may choose to write two drivers, one for each block of logic, in which case each driver refers to the single I^{2}C slave address using the relative value of zero (0). However if the third party I^{2}C driver writer chooses to write a single driver which consumes multiple I^{2}C slave addresses then the third party I^{2}C driver writer needs to convey the order of the I^{2}C slave address entries in the I^{2}C slave address array to the platform software designer. For the example:

> 0: Accelerometer
> 1: Magnetometer

The platform hardware designer picks the actual slave addresses from the I^{2}C device's data sheet and provides this information to the platform software designer. The platform software designer then places the I^{2}C slave addresses into the I^{2}C slave address array in the **EFI_I2C_ENUMERATE_PROTOCL** in the order specified by the third party I^{2}C driver writer. The third party I^{2}C driver writer uses the index into the I^{2}C slave address array as the relative I^{2}C slave address. The I^{2}C IO protocol uses the I^{2}C slave address array to translate the relative I^{2}C slave address into the platform specific I^{2}C slave address. The relative value always starts at zero (0) and its maximum value is the number of entries in I^{2}C slave address array minus one.

Each I^{2}C slave address entry is specified as a 32-bit integer to allow room for future I^{2}C slave address expansion. Only the I^{2}C master protocol knows the maximum I^{2}C slave address value. All other drivers and applications must look for the **EFI_NOT_FOUND** status for the indication that a reserve bit was set in the I^{2}C slave address.

31	30	8 7	0
0	Reserved (Must Be Zero)		7-bit Slave Address

31	30	10 9	0
1	Reserved (Must Be Zero)		10-bit Slave Address

17.1.2.3.1 Driver Binding Protocol Supported() API

The driver binding protocol's *Supported()* routine looks for controllers which declare the **EFI_I2C_IO_PROTOCOL** and match the device path supplied by the silicon vendor or third party I^{2}C driver writer to the platform integrator.

The third party I^{2}C device driver creates a GUID for a Vendor-Defined Hardware Device Path Node when describing the I^{2}C device. The third party I^{2}C device driver writer provides this GUID to the person writing the platform specific code to identify the type of I^{2}C device.

The third party I^{2}C driver which consumes the **EFI_I2C_IO_PROTOCOL** compares the known GUID with the GUID pointed to by the *DeviceGuid* field.

An example algorithm for the driver binding protocol *Supported()* routine:

1. Open the **EFI_I2C_IO_PROTOCOL** using **EFI_OPEN_PROTOCOL_BY_DRIVER**
2. If OpenProtocol() fails return the error status
3. Get the vendor GUID from the **EFI_I2C_IO_PROTOCOL**
4. Close the **EFI_I2C_IO_PROTOCOL**
5. Compare the expected vendor GUID to the GUID from the **EFI_I2C_IO_PROTOCOL** structure.
6. If the GUIDS don't match then return **EFI_NOT_SUPPORTED**
7. Return **EFI_SUCCESS**

17.1.2.3.2 Supporting Multiple Hardware Versions

Note that package markings are important to allow the platform integrator to verify the hardware revision after the part is integrated! The platform integrator includes the hardware revision information into the **EFI_I2C_ENUMERATE_PROTOCOL**. The I^{2}C bus driver gets this data during the I^{2}C device enumeration and makes it available to the third party I^{2}C device driver via the **EFI_I2C_IO_PROTOCOL**. There are a couple of ways in which the silicon vendor or third party I^{2}C driver writer may support multiple hardware versions of the I^{2}C device:

*   Provide a different GUID value to the platform integrator for each hardware revision
*   Provide a different hardware version value to the platform integrator with the devices

Each of the above methods describes an interface to the I^{2}C device. The interface specifies the number of slave addresses as well as the features and software workarounds for the I^{2}C device.

17.1.2.4 I^{2}C IO Protocol

The I2C IO protocol is platform, host controller, and I^{2}C chip independent.

The I^{2}C bus driver creates a handle for each of the I^{2}C devices returned by the I^{2}C enumerate protocol. The I^{2}C controller's device path is extended with the vendor GUID and unique ID value returned by the I^{2}C enumerate protocol and attached to the handle. The vendor GUID is used to extend the device path with a Vendor-define Hardware Device Path Node and the unique ID is used to further extend the device path with a Controller Device Path Node. If the unique ID is 0, then the Controller Device Path Node is optional. The third party I^{2}C device driver uses the device GUID to determine if it may connect.

When a third party I^{2}C device driver or application calls *QueueRequest()*, the I^{2}C IO protocol validates the *SlaveAddressIndex* (relative I^{2}C address) for the I^{2}C device and then converts the *SlaveAddressIndex* to a I^{2}C slave address. The request is then passed to the I^{2}C host protocol along with the tuple BusConfiguration:I^{2}C slave address.

17.1.2.5 I^{2}C Host Protocol

The I^{2}C host protocol is platform, host controller, and I^{2}C chip independent.

Note: For proper operation of the I^{2}C bus, only the I^{2}C IO protocol and I^{2}C test applications connect to the **EFI_I2C_HOST_PROTOCOL**.

The I^{2}C host protocol may access any device on the I^{2}C bus. The I^{2}C host protocol has the following responsibilities:

*   Limits the number of requests to the I^{2}C master protocol to one. The I^{2}C host protocol holds on to additional requests until the I^{2}C master protocol is available to process the request. The I^{2}C requests are issued in FIFO order to the I^{2}C master protocol.
*   Enable the proper I
* ^{2}C bus configuration before starting the I^{2}C request using the I^{2}C master protocol

I^{2}C devices are addressed as the tuple: BusConfiguration:SlaveAddress. I^{2}C bus configuration zero (0) is the portion of the I^{2}C bus that connects to the host controller. The bus configuration specifies the control values for the switches and multiplexers in the I^{2}C bus. After the switches and multiplexers are properly configured, the I^{2}C controller uses the slave address to access the requested I^{2}C device.

Since the I^{2}C protocol stack supports asynchronous transactions the I^{2}C host protocol maintains a queue of I^{2}C requests until the I^{2}C controller is available them. When a request reaches the head of

the queue the necessary bus configuration is enabled and then the request is sent to the I²C master protocol.

17.1.2.6 I²C Master Protocol

The I2C master protocol is I2C controller specific but platform independent.

This protocol is designed to allow the implementation to be built as a driver which may be delivered in binary form as an EFI image.

The master protocol manipulates the I2C controller to perform a transaction on the I2C bus. The I2C master protocol does not configure the I2C bus so it is up to the caller to ensure that the I2C bus is in the proper configuration before issuing the I²C request.

The I²C master protocol typically needs the following information:

• Host controller address

• Controller's input clock frequency

Depending upon the I2C controller, more data may be necessary. This protocol may use any method to get these values: hard coded values, PCD values, or may choose to communicate with the platform specific code using an undefined mechanism to get these values.

If the I2C master protocol requires data from the platform specific code then the I2C master protocol writer needs to provide the platform interface details to the platform software designer.

17.1.2.7 Platform Specific Code

The platform specific code installs the **EFI_I2C_ENUMERATE_PROTOCOL** to provide the I2C device descriptions to the I2C bus driver using the **EFI_I2C_DEVICE** structure. These descriptions include the bus configuration number required for the I2C device, the slave address array, the vendor GUID and a unique ID value.

The **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** enables the I2C host protocol to call into the platform specific code to enable a specific I2C bus configuration and set the I2C bus frequency. This protocol is required to get the I2C host protocol to start for the I2C controller's handle.

The platform software designer collects the data requirements from third party I2C driver writers, the vendor specific I2C master protocol writer, the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** and **EFI_I2C_ENUMERATE_PROTOCOL**. The platform software designer gets the necessary data from the platform hardware designer. The platform software designer then builds the data structures and implements the necessary routines to construct the platform specific code for I²C.

17.1.2.8 Switches and Multiplexers

There are some I2C switches and I2C multiplexers where the control is done via I2C commands. When the control inputs come via the same I²C bus that is being configured then the platform specific code must use the **EFI_I2C_MASTER_PROTOCOL**. While the I²C host protocol makes the call to *EnableI2cBusConfiguration* to configure the I²C bus, the I²C host protocol

keeps the I2C master protocol idle, enabling the platform specific code to perform the necessary I2C configuration transactions.

If however the configuration control is done via an I2C device connected to a different I2C bus (host controller), then the platform software designer may choose between the following:

- Call into a third party I^{2}C driver to manipulate the I^{2}C bus control device.

- Call into the **EFI_I2C_IO_PROTOCOL** if no third party I^{2}C driver exists for the I^{2}C bus control device

- Call into the **EFI_I2C_HOST_PROTOCOL** if the platform does not expose the I^{2}C bus control device.

17.1.3 PCI Comparison

PCI provides several features to describe the device to the operating system as well decoupling the driver from the specific platform.

17.1.3.1 Device Description

PCI uses the *Vendor ID* and *Device ID* fields in configuration space to identify the piece of hardware. Where the Vendor ID is assigned by the PCI committee and the Device ID is assigned by the hardware manufacture.

PCI also uses the *Base Class*, *Sub Class* and *Programming Interface* fields to help identify the operating system driver.

The I^{2}C protocol stack uses the vendor GUID associated with the I^{2}C device to identify the UEFI driver. This GUID is supplied by the silicon vendor or third party I^{2}C driver writer to the platform integrator and gets included in the I^{2}C platform driver. The **EFI_I2C_ENUMERATE_PROTOCOL** provides this GUID to the I^{2}C bus driver during the I^{2}C bus enumeration.

The driver binding protocol's *Supported()* routine of the third party I^{2}C device driver looks for controllers which have the **EFI_I2C_IO_PROTOCOL** and have a match for the vendor GUID.

17.1.3.2 Hardware Features and Workarounds

PCI provides a *Revision ID* field to allow the driver to determine which version of hardware is present and which features and software workarounds are necessary to support this device.

The I^{2}C protocol stack uses the *HardwareRevision* field in the **EFI_I2C_IO_PROTOCOL** for this same purpose. It is recommended that this value match the _HRV value in the DSDT for this I^{2}C device. See the *Advanced Configuration and Power Interface Specification, Revision 5.0* for the field format and the *Plug and play support for I2C* web-page for restriction on values.

17.1.3.3 Device Relative Addressing

PCI provides **Base Address Registers** (BARs) to decouple the device driver software from the details of the platform's PCI bus configuration. Typically, all device register references are fixed offsets from one of the BAR addresses.

The I^{2}C protocol stack provides a similar mechanism using an index into an array of slave addresses. The silicon vendor or third party driver writer provides the structure of the array listing the major functions to the platform integrator. An example is:

   0: Accelerometer

   1: Compass

The platform integrator works with the platform's hardware designer to get the I^{2}C slave addresses of the I^{2}C device and builds the array which is included in the platform specific code. During I^{2}C device enumeration, this array is passed to the I^{2}C bus driver for use by the I^{2}C IO protocol.

The third party I^{2}C driver references the major components within the I^{2}C device using the index values, thus remaining platform independent. The I^{2}C IO protocol performs the array lookup, translating the index into an actual slave address on the I^{2}C bus.

Most I^{2}C devices only have a single I^{2}C slave address and thus the third party I^{2}C device driver will only use index zero (0). Also depending upon the I^{2}C device architecture, the silicon vendor or third party I^{2}C device writer may choose to write multiple drivers, each supporting a single I^{2}C slave address.

17.1.4 Hot Plug Support

I^{2}C protocol stack enables the platform specific code to support hot-plug with the following algorithm:

1. Describe all possible devices on all possible busses, including the hot-plug devices.
2. The platform specific code detects hot-plug events: Add and Remove
3. For a removal event:
    - The platform specific code opens the **EFI_I2C_IO_PROTOCOL** on the hot-plug device's handle exclusively. This operation tears down any upper layer protocols on this handle. Note that the open request may fail if I/O is pending in the lower protocols.
    - When the step above fails, delay below **TPL_NOTIFY** to allow the current I
    - ^{2}C transaction complete and then retry until the open is successful
    - After the open is successful, the platform specific code may use the I
    - ^{2}C IO protocol to perform I^{2}C transactions for device probing.
4. For an add event:
    - The platform specific code waits for completion any outstanding I/O that the platform specific code initiated on the hot-plug I^{2}C device.
    - The platform specific code closes the **EFI_I2C_IO_PROTOCOL**
    - The platform specific code issues a *ConnectController()* on the hot-plug device's handle. This causes the protocol stack which uses the hot-plug device to be reloaded.

17.2 DXE Code definitions

The I^{2}C protocol stack consists of the following protocols:

- **EFI_I2C_IO_PROTOCOL** – Third party silicon vendors use this protocol to access their I^{2}C device. This protocol enables a driver or application to perform I/O transactions to a single I^{2}C device independent of the I^{2}C bus configuration.

- **EFI_I2C_HOST_PROTOCOL** – The I^{2}C bus driver uses this protocol to produce the **EFI_I2C_IO_PROTOCOL** that provides access a device on the I^{2}C bus.

- **EFI_I2C_MASTER_PROTOCOL** – The I^{2}C host protocol uses this protocol to manipulate the I^{2}C host controller and perform transactions as a master on the I^{2}C bus.

- **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** – The I^{2}C host protocol uses this protocol to request the proper state for the switches and multiplexers in the I^{2}C bus and set the I^{2}C clock frequency.

- **EFI_I2C_ENUMERATE_PROTOCOL** – The I^{2}C bus driver uses this protocol to enumerate the devices on the I^{2}C bus, getting the bus configuration and an array of slave addresses for each of the I^{2}C devices.

The following sections describe these protocols in detail.

17.2.1 I^{2}C Master Protocol

EFI_I2C_MASTER_PROTOCOL

Summary

This protocol manipulates the I2C host controller to perform transactions as a master on the I2C bus using the current state of any switches or multiplexers in the I2C bus.

GUID

```
#define EFI_I2C_MASTER_PROTOCOL_GUID \
{ 0xcd72881f, 0x45b5, 0x4feb, { 0x98, 0xc8, 0x31, 0x3d, \
0xa8, 0x11, 0x74, 0x62 }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_MASTER_PROTOCOL {
  EFI_I2C_MASTER_PROTOCOL_SET_BUS_FREQUENCY  SetBusFrequency;
  EFI_I2C_MASTER_PROTOCOL_RESET              Reset;
  EFI_I2C_MASTER_PROTOCOL_START_REQUEST      StartRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES
*I2cControllerCapabilities;
} EFI_I2C_MASTER_PROTOCOL;
```

Parameters

SetBusFrequency

> Set the clock frequency for the I^2C bus.

Reset

> Reset the I^2C host controller.

StartRequest

> Start an I^2C transaction in master mode on the host controller.

I2cControllerCapabilities

> Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^2C host controller.

Description

The **EFI_I2C_MASTER_PROTOCOL** is typically used by the I^2C host protocol to perform transactions on the I^2C bus. This protocol may also be used to configure the I^2C clock frequency and use I^2C transactions to set the state of switches and multiplexers in the I^2C bus.

Related Definitions

31	30	8	7	0
0	Reserved (Must Be Zero)		7-bit Slave Address	

31	30	10	9	0
1	Reserved (Must Be Zero)		10-bit Slave Address	

A 10-bit slave address is or'ed with the following value enabling the I^2C protocol stack to address the duplicated address space between 0 and 127 in 10-bit mode.

```
#define I2C_ADDRESSING_10_BIT     0x80000000
```

The I^{2}C protocol stack uses the **EFI_I2C_REQUEST_PACKET** structure to describe I^{2}C transactions on the I^{2}C bus. The **EFI_I2C_OPERATION** describes a portion of the I^{2}C transaction. The transaction starts with a start bit followed by the first operation in the operation array. Subsequent operations are separated with repeated start bits and the last operation is followed by a stop bit which concludes the transaction.

```
typedef struct {
  UINTN                OperationCount;
  EFI_I2C_OPERATION   Operation[];
} EFI_I2C_REQUEST_PACKET;
```

Parameters

OperationCount

Number of elements in the operation array.

Operation

Description of the I^{2}C operation

Description

The **EFI_I2C_REQUEST_PACKET** describes a single I^{2}C transaction. The transaction starts with a start bit followed by the first operation in the operation array. Subsequent operations are separated with repeated start bits and the last operation is followed by a stop bit which concludes the transaction. Each operation is described by one of the elements in the *Operation* array.

```
typedef struct {
  UINT32  Flags;
  UINT32  LengthInBytes;
  UINT8   *Buffer;
} EFI_I2C_OPERATION;
```

Parameters

Flags

Flag bits qualify the I^{2}C operation.

Flag Bits:

```
///
/// Define the I2C flags
///
/// I2C read operation when set
#define I2C_FLAG_READ              0x00000001


///
/// Define the flags for SMBus operation
///
/// The following flags are also present in only the first I2C operation
/// and are ignored when present in other operations.  These flags
/// describe a particular SMB transaction as shown in the following table.
///


/// SMBus operation
#define I2C_FLAG_SMBUS_OPERATION    0x00010000

/// SMBus block operation
///   The flag I2C_FLAG_SMBUS_BLOCK causes the I2C master protocol to update
///   the LengthInBytes field of the operation in the request packet with
///   the actual number of bytes read or written.  These values are only
///   valid when the entire I2C transaction is successful.
///   This flag also changes the LengthInBytes meaning to be: A maximum
///   of LengthInBytes is to be read from the device.  The first byte
///   read contains the number of bytes remaining to be read, plus an
///   optional PEC value.
#define I2C_FLAG_SMBUS_BLOCK        0x00020000

/// SMBus process call operation
#define I2C_FLAG_SMBUS_PROCESS_CALL 0x00040000

/// SMBus use packet error code (PEC)
///   Note that the I2C master protocol may clear the I2C_FLAG_SMBUS_PEC bit
///   to indicate that the PEC value was checked by the hardware and is
///   not appended to the returned read data.
///
#define I2C_FLAG_SMBUS_PEC          0x00080000

//-------------------------------------------------------------------
///
/// QuickRead:        OperationCount=1,
///                   LengthInBytes=0,   Flags=I2C_FLAG_READ
/// QuickWrite:       OperationCount=1,
///                   LengthInBytes=0,   Flags=0
///
///
/// ReceiveByte:      OperationCount=1,
///                   LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                      | I2C_FLAG_READ
/// ReceiveByte+PEC:  OperationCount=1,
///                   LengthInBytes=2,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                      | I2C_FLAG_READ
///                                      | I2C_FLAG_SMBUS_PEC
///
///
```

```
/// SendByte:          OperationCount=1,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
/// SendByte+PEC:      OperationCount=1,
///                    LengthInBytes=2,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_PEC
///
///
/// ReadDataByte:      OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                    LengthInBytes=1,   Flags=I2C_FLAG_READ
/// ReadDataByte+PEC:  OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_PEC
///                    LengthInBytes=2,   Flags=I2C_FLAG_READ
///
///
/// WriteDataByte:     OperationCount=1,
///                    LengthInBytes=2,   Flags=I2C_FLAG_SMBUS_OPERATION
/// WriteDataByte+PEC: OperationCount=1,
///                    LengthInBytes=3,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_PEC
///
///
/// ReadDataWord:      OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                    LengthInBytes=2,   Flags=I2C_FLAG_READ
/// ReadDataWord+PEC:  OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_PEC
///                    LengthInBytes=3,   Flags=I2C_FLAG_READ
///
///
/// WriteDataWord:     OperationCount=1,
///                    LengthInBytes=3,   Flags=I2C_FLAG_SMBUS_OPERATION
/// WriteDataWord+PEC: OperationCount=1,
///                    LengthInBytes=4,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_PEC
///
///
/// ReadBlock:         OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_BLOCK
///                    LengthInBytes=33,  Flags=I2C_FLAG_READ
/// ReadBlock+PEC:     OperationCount=2,
///                    LengthInBytes=1,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_BLOCK
///                                             | I2C_FLAG_SMBUS_PEC
///                    LengthInBytes=34,  Flags=I2C_FLAG_READ
///
///
/// WriteBlock:        OperationCount=1,
///                    LengthInBytes=N+2, Flags=I2C_FLAG_SMBUS_OPERATION
///                                             | I2C_FLAG_SMBUS_BLOCK
/// WriteBlock+PEC:    OperationCount=1,
///                    LengthInBytes=N+3, Flags=I2C_FLAG_SMBUS_OPERATION
```

```
///                                                    | I2C_FLAG_SMBUS_BLOCK
///                                                    | I2C_FLAG_SMBUS_PEC
///
///
/// ProcessCall:        OperationCount=2,
///                     LengthInBytes=3,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                          | I2C_FLAG_SMBUS_PROCESS_CALL
///                     LengthInBytes=2,   Flags=I2C_FLAG_READ
/// ProcessCall+PEC:    OperationCount=2,
///                     LengthInBytes=3,   Flags=I2C_FLAG_SMBUS_OPERATION
///                                            | I2C_FLAG_SMBUS_PROCESS_CALL
///                                            | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=3,   Flags=I2C_FLAG_READ
///
///
/// BlkProcessCall:     OperationCount=2,
///                     LengthInBytes=N+2, Flags=I2C_FLAG_SMBUS_OPERATION
///                                          | I2C_FLAG_SMBUS_PROCESS_CALL
///                                          | I2C_FLAG_SMBUS_BLOCK
///                     LengthInBytes=33,  Flags=I2C_FLAG_READ
/// BlkProcessCall+PEC: OperationCount=2,
///                     LengthInBytes=N+2, Flags=I2C_FLAG_SMBUS_OPERATION
///                                          | I2C_FLAG_SMBUS_PROCESS_CALL
///                                          | I2C_FLAG_SMBUS_BLOCK
///                                          | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=34,  Flags=I2C_FLAG_READ
///
//-------------------------------------------------------------------
```

LengthInBytes

> Number of bytes to send to or receive from the I^{2}C device. A ping (address only byte/bytes) is indicated by setting the *LengthInBytes* to zero.

Buffer

> Pointer to a buffer containing the data to send or to receive from the I^{2}C device. The *Buffer* must be at least *LengthInBytes* in size.

Description

The **EFI_I2C_OPERATION** describes a subset of an I^{2}C transaction in which the I^{2}C controller is either sending or receiving bytes from the bus. Some transactions will consist of a single operation while others will be two or more.

Note: *Some I^{2}C controllers do not support read or write ping (address only) operation and will return* **EFI_UNSUPPORTED** *status when these operations are requested.*

Note: *I^{2}C controllers which do not support complex transactions requiring multiple repeated start bits return* **EFI_UNSUPPORTED** *without processing any of the transaction.*

```
typedef struct {
  UINT32  StructureSizeInBytes;
  UINT32  MaximumReceiveBytes;
  UINT32  MaximumTransmitBytes;
  UINT32  MaximumTotalBytes;
} EFI_I2C_CONTROLLER_CAPABILITIES;
```

Parameters

StructureSizeInBytes

Length of this data structure in bytes

MaximumReceiveBytes;

The maximum number of bytes the I^{2}C host controller is able to receive from the I^{2}C bus.

MaximumTransmitBytes

The maximum number of bytes the I^{2}C host controller is able to send on the I^{2}C bus.

MaximumTotalBytes

The maximum number of bytes in the I^{2}C bus transaction.

Description

The **EFI_I2C_CONTROLLER_CAPABILITIES** specifies the capabilities of the I^{2}C host controller. The *StructureSizeInBytes* enables variations of this structure to be identified if there is need to extend this structure in the future.

EFI_I2C_MASTER_PROTOCOL.SetBusFrequency()

Summary

Set the frequency for the I^{2}C clock line.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_SET_BUS_FREQUENCY) (
  IN CONST EFI_I2C_MASTER_PROTOCOL   *This,
  IN OUT UINTN                       *BusClockHertz
  );
```

Parameters

This

> Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

BusClockHertz

> Pointer to the requested I^{2}C bus clock frequency in Hertz. Upon return this value contains the actual frequency in use by the I^{2}C controller.

Description

This routine must be called at or below **TPL_NOTIFY**.

The software and controller do a best case effort of using the specified frequency for the I^{2}C bus. If the frequency does not match exactly then the I^{2}C master protocol selects the next lower frequency to avoid exceeding the operating conditions for any of the I^{2}C devices on the bus. For example if 400 KHz was specified and the controller's divide network only supports 402 KHz or 398 KHz then the I^{2}C master protocol selects 398 KHz. If there are not lower frequencies available, then return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The bus frequency was set successfully.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_UNSUPPORTED	The controller does not support this frequency.

EFI_I2C_MASTER_PROTOCOL.Reset()

Summary

Reset the I^{2}C controller and configure it for use.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_RESET) (
  IN CONST EFI_I2C_MASTER_PROTOCOL  *This
  );
```

Parameters

This

Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

Description

This routine must be called at or below **TPL_NOTIFY**.

The I^{2}C controller is reset. The caller must call *SetBusFrequency()* after calling *Reset()*.

Status Codes Returned

EFI_SUCCESS	The reset completed successfully.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_DEVICE_ERROR	The reset operation failed.

EFI_I2C_MASTER_PROTOCOL.StartRequest()

Summary

Start an I^{2}C transaction on the host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_START_REQUEST) (
  IN CONST EFI_I2C_MASTER_PROTOCOL  *This,
  IN UINTN                          SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET         *RequestPacket,
  IN EFI_EVENT                      Event              OPTIONAL,
  OUT EFI_STATUS                    *I2cStatus         OPTIONAL
  );
```

Parameters

This

>> Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

SlaveAddress

>> Address of the device on the I^{2}C BUS. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^{2}C slave addresses and bits 0-9 are used for 10-bit I^{2}C slave addresses.

RequestPacket

>> Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction.

Event

>> Event to signal for asynchronous transactions, NULL for synchronous transactions

I2cStatus

>> Optional buffer to receive the I^{2}C transaction completion status

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

This function initiates an I^{2}C transaction on the controller. To enable proper error handling by the I^{2}C protocol stack, the I^{2}C master protocol does not support queuing but instead only manages one I^{2}C transaction at a time. This API requires that the I^{2}C bus is in the correct configuration for the I^{2}C transaction.

The transaction is performed by sending a start-bit and selecting the I^{2}C device with the specified I^{2}C slave address and then performing the specified I^{2}C operations. When multiple operations are

requested they are separated with a repeated start bit and the slave address. The transaction is terminated with a stop bit.

When Event is NULL, *StartRequest* operates synchronously and returns the I^{2}C completion status as its return value.

When Event is not NULL, *StartRequest* synchronously returns **EFI_SUCCESS** indicating that the I^{2}C transaction was started asynchronously. The transaction status value is returned in the buffer pointed to by *I2cStatus* upon the completion of the I^{2}C transaction when I2cStatus is not NULL. After the transaction status is returned the *Event* is signaled.

Note: *The typical consumer of this API is the I^{2}C host protocol. Extreme care must be taken by other consumers of this API to prevent confusing the third party I^{2}C drivers due to a state change at the I^{2}C device which the third party I^{2}C drivers did not initiate. I^{2}C platform specific code may use this API within these guidelines.*

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully started when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.2 I^{2}C Host Protocol

EFI_I2C_HOST_PROTOCOL

Summary

This protocol provides callers with the ability to do I/O transactions to all of the devices on the I^{2}C bus.

GUID

```
#define EFI_I2C_HOST_PROTOCOL_GUID \
{ 0xa5aab9e3, 0xc727, 0x48cd, { 0x8b, 0xbf, 0x42, 0x72, \
0x33, 0x85, 0x49, 0x48 }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_HOST_PROTOCOL {
  EFI_I2C_HOST_PROTOCOL_QUEUE_REQUEST  QueueRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES
*I2cControllerCapabilities;
} EFI_I2C_HOST_PROTOCOL;
```

Parameters

QueueRequest

Queue an transaction for execution on the I^{2}C bus

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

The I^{2}C bus driver uses the services of the **EFI_I2C_HOST_PROTOCOL** to produce an instance of the **EFI_I2C_IO_PROTOCOL** for each I^{2}C device on an I^{2}C bus.

The **EFI_I2C_HOST_PROTOCOL** exposes an asynchronous interface to callers to perform transactions to any device on the I^{2}C bus. Internally, the I^{2}C host protocol manages the flow of the I^{2}C transactions to the host controller, keeping them in FIFO order. Prior to each transaction, the I^{2}C host protocol ensures that the switches and multiplexers are properly configured. The I^{2}C host protocol then starts the transaction on the host controller using the **EFI_I2C_MASTER_PROTOCOL**.

EFI_I2C_HOST_PROTOCOL.QueueRequest()

Summary

Queue an I^2C transaction for execution on the I^2C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_HOST_PROTOCOL_QUEUE_REQUEST) (
  IN CONST EFI_I2C_HOST_PROTOCOL  *This,
  IN UINTN                        I2cBusConfiguration,
  IN UINTN                        SlaveAddress,
  IN EFI_EVENT                    Event             OPTIONAL,
  IN EFI_I2C_REQUEST_PACKET       *RequestPacket,
  OUT EFI_STATUS                  *I2cStatus        OPTIONAL
  );
```

Parameters

This

> Pointer to an **EFI_I2C_HOST_PROTOCOL** structure.

I2cBusConfiguration

> I^2C bus configuration to access the I^2C device

SlaveAddress

> Address of the device on the I^2C bus. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^2C slave addresses and bits 0-9 are used for 10-bit I^2C slave addresses.

Event

> Event to signal for asynchronous transactions, NULL for synchronous transactions

RequestPacket

> Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^2C transaction

I2cStatus

> Optional buffer to receive the I^2C transaction completion status

Description

Queue an I^2C transaction for execution on the I^2C controller.

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

The I^2C host protocol uses the concept of I^2C bus configurations to describe the I^2C bus. An I^2C bus configuration is defined as a unique setting of the multiplexers and switches in the I^2C bus which

enable access to one or more I^{2}C devices. When using a switch to divide a bus, due to bus frequency differences, the I^{2}C bus configuration management protocol defines an I^{2}C bus configuration for the I^{2}C devices on each side of the switch. When using a multiplexer, the I^{2}C bus configuration management defines an I^{2}C bus configuration for each of the selector values required to control the multiplexer. See Figure 1 in the I^{2}C -bus specification and user manual for a complex I^{2}C bus configuration.

The I^{2}C host protocol processes all transactions in FIFO order. Prior to performing the transaction, the I^{2}C host protocol calls *EnableI2cBusConfiguration* to reconfigure the switches and multiplexers in the I^{2}C bus enabling access to the specified I^{2}C device. The EnableI2cBusConfiguration also selects the I^{2}C bus frequency for the I^{2}C device. After the I^{2}C bus is configured, the I^{2}C host protocol calls the I^{2}C master protocol to start the I^{2}C transaction.

When Event is NULL, *QueueRequest()* operates synchronously and returns the I^{2}C completion status as its return value.

When Event is not NULL, *QueueRequest()* synchronously returns **EFI_SUCCESS** indicating that the asynchronously I^{2}C transaction was queued. The values above are returned in the buffer pointed to by I2cStatus upon the completion of the I^{2}C transaction when I2cStatus is not NULL.

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully queued when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.3 I^{2}C I/O Protocol

EFI_I2C_IO_PROTOCOL

Summary

The EFI I^{2}C I/O protocol enables the user to manipulate a single I^{2}C device independent of the host controller and I^{2}C design.

GUID

```
#define EFI_I2C_IO_PROTOCOL_GUID \
{ 0xb60a3e6b, 0x18c4, 0x46e5, { 0xa2, 0x9a, 0xc9, 0xa1, \
0x06, 0x65, 0xa2, 0x8e }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_IO_PROTOCOL {
  EFI_I2C_IO_PROTOCOL_QUEUE_REQUEST QueueRequest;
  CONST EFI_GUID                    *DeviceGuid;
  UINT32                            DeviceIndex;
  UINT32                            HardwareRevision;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES
*I2cControllerCapabilities;
} EFI_I2C_IO_PROTOCOL;
```

Parameters

QueueRequest

Queue an I^{2}C transaction for execution on the I^{2}C device.

DeviceGuid

Unique value assigned by the silicon manufacture or the third party I^{2}C driver writer for the I^{2}C part. This value logically combines both the manufacture name and the I^{2}C part number into a single value specified as a GUID.

DeviceIndex

Unique ID of the I^{2}C part within the system

HardwareRevision

Hardware revision - ACPI _HRV value. See the *Advanced Configuration and Power Interface Specification, Revision 5.0* for the field format and the *Plug and play support for I2C* web-page for restriction on values.

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

- The I^{2}C IO protocol enables access to a specific device on the I^{2}C bus.

- Each I^{2}C device is identified uniquely in the system by the tuple *DeviceGuid*:*DeviceIndex*. The *DeviceGuid* represents the manufacture and part number and is provided by the silicon vendor or the third party I^{2}C device driver writer. The *DeviceIndex* identifies the part within the system by using a unique number and is created by the board designer or the writer of the **EFI_I2C_ENUMERATE_PROTOCOL**.

I^2C slave addressing is abstracted to validate addresses and limit operation to the specified I^2C device. The third party providing the I^2C device support provides an ordered list of slave addresses for the I^2C device required to implement the **EFI_I2C_ENUMERATE_PROTOCOL**. The order of the list must be preserved.

EFI_I2C_IO_PROTOCOL.QueueRequest()

Summary

Queue an I^{2}C transaction for execution on the I^{2}C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_PROTOCOL_QUEUE_REQUEST) (
  IN CONST EFI_I2C_IO_PROTOCOL  *This,
  IN UINTN                      SlaveAddressIndex,
  IN EFI_EVENT                  Event               OPTIONAL,
  IN EFI_I2C_REQUEST_PACKET     *RequestPacket,
  OUT EFI_STATUS                *I2cStatus          OPTIONAL
  );
```

Parameters

This

> Pointer to an **EFI_I2C_IO_PROTOCOL** structure.

SlaveAddressIndex

> Index value into an array of slave addresses for the I^{2}C device. The values in the array are specified by the board designer, with the third party I^{2}C device driver writer providing the slave address order.

> For devices that have a single slave address, this value must be zero. If the I^{2}C device uses more than one slave address then the third party (upper level) I^{2}C driver writer needs to specify the order of entries in the slave address array.

Event

> Event to signal for asynchronous transactions, NULL for synchronous transactions

RequestPacket

> Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

I2cStatus

> Optional buffer to receive the I^{2}C transaction completion status

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

This routine queues an I^{2}C transaction to the I^{2}C controller for execution on the I^{2}C bus.

When *Event* is NULL, *QueueRequest()* operates synchronously and returns the I^{2}C completion status as its return value.

When *Event* is not NULL, *QueueRequest()* synchronously returns **EFI_SUCCESS** indicating that the asynchronous I^{2}C transaction was queued. The values above are returned in the buffer pointed to by I2cStatus upon the completion of the I^{2}C transaction when I2cStatus is not NULL.

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully queued when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NO_MAPPING	The **EFI_I2C_HOST_PROTOCOL** could not set the bus configuration required to access this I^{2}C device.
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address selected by *SlaveAddressIndex*. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.4 I^{2}C Bus Configuration Management Protocol

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL

Summary

The EFI I^{2}C bus configuration management protocol provides platform specific services that allow the I^{2}C host protocol to reconfigure the switches and multiplexers and set the clock frequency for the I^{2}C bus. This protocol also enables the I^{2}C host protocol to reset an I^{2}C device which may be locking up the I^{2}C bus by holding the clock or data line low.

GUID

```
#define EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_GUID \
{ 0x55b71fb5, 0x17c6, 0x410e, { 0xb5, 0xbd, 0x5f, 0xa2, \
0xe3, 0xd4, 0x46, 0x6b }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL {

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_ENABLE_I2C_BUS_CON
FIGURATION  EnableI2cBusConfiguration;
} EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL;
```

Parameters

EnableI2cBusConfiguration

> Enable an I^{2}C bus configuration for use.

Description

The I^{2}C protocol stack uses the concept of an I^{2}C bus configuration as a way to describe a particular state of the switches and multiplexers in the I^{2}C bus.

A simple I^{2}C bus does not have any multiplexers or switches is described to the I^{2}C protocol stack with a single I^{2}C bus configuration which specifies the I^{2}C bus frequency.

An I^{2}C bus with switches and multiplexers use an I^{2}C bus configuration to describe each of the unique settings for the switches and multiplexers and the I^{2}C bus frequency. However the I^{2}C bus configuration management protocol only needs to define the I^{2}C bus configurations that the software uses, which may be a subset of the total.

The I^{2}C bus configuration description includes a list of I^{2}C devices which may be accessed when this I^{2}C bus configuration is enabled. I^{2}C devices before a switch or multiplexer must be included in one I^{2}C bus configuration while I^{2}C devices after a switch or multiplexer are on another I^{2}C bus configuration.

The I^{2}C bus configuration management protocol is an optional protocol. When the I^{2}C bus configuration protocol is not defined the I^{2}C host protocol does not start and the I^{2}C master protocol may be used for other purposes such as SMBus traffic. When the I^{2}C bus configuration protocol is available, the I^{2}C host protocol uses the I^{2}C bus configuration protocol to call into the platform specific code to set the switches and multiplexers and set the maximum I^{2}C bus frequency.

The platform designers determine the maximum I^{2}C bus frequency by selecting a frequency which supports all of the I^{2}C devices on the I^{2}C bus for the setting of switches and multiplexers. The platform designers must validate this against the I^{2}C device data sheets and any limits of the I^{2}C controller or bus length.

During I^{2}C device enumeration, the I^{2}C bus driver retrieves the I^{2}C bus configuration that must be used to perform I^{2}C transactions to each I^{2}C device. This I^{2}C bus configuration value is passed into the I^{2}C host protocol to identify the I^{2}C bus configuration required to access a specific I^{2}C device. The I^{2}C host protocol calls **EnableBusConfiguration()** to set the switches and multiplexers in the I^{2}C bus and the I^{2}C clock frequency. The I^{2}C host protocol may optimize calls to **EnableBusConfiguration()** by only making the call when the I^{2}C bus configuration value changes between I^{2}C requests.

When I^{2}C transactions are required on the same I^{2}C bus to change the state of multiplexers or switches, the I^{2}C master protocol must be used to perform the necessary I^{2}C transactions.

It is up to the platform specific code to choose the proper I^{2}C bus configuration when **ExitBootServices()** is called. Some operating systems are not able to manage the I^{2}C bus configurations and must use the I^{2}C bus configuration that is established by the platform firmware before **ExitBootServices()** returns.

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL.
EnableI2cBusConfiguration()

Summary

Enable access to an I^{2}C bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_ENABLE_I2C_BUS_CO
NFIGURATION) (
  IN CONST EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL  *This,
  IN UINTN                                I2cBusConfiguration,
  IN EFI_EVENT                            Event      OPTIONAL,
  IN EFI_STATUS                           *I2cStatus  OPTIONAL
  );
```

Parameters

This

> Pointer to an **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL**
> structure.

I2cBusConfiguration

> Index of an I^{2}C bus configuration. All values in the range of zero to N-1 are valid
> where N is the total number of I^{2}C bus configurations for an I^{2}C bus.

Event

> Event to signal when the transaction is complete

I2cStatus

> Buffer to receive the transaction status.

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must
be called at or below **TPL_CALLBACK**.

Reconfigure the switches and multiplexers in the I^{2}C bus to enable access to a specific I^{2}C bus
configuration. Also select the maximum clock frequency for this I^{2}C bus configuration.

This routine uses the I^{2}C Master protocol to perform I^{2}C transactions on the local bus. This
eliminates any recursion in the I^{2}C stack for configuration transactions on the same I^{2}C bus. This
works because the local I^{2}C bus is idle while the I^{2}C bus configuration is being enabled.

If I^{2}C transactions must be performed on other I^{2}C busses, then the **EFI_I2C_HOST_PROTOCOL**,
the **EFI_I2C_IO_PROTCOL**, or a third party I^{2}C driver interface for a specific device must be

used. This requirement is because the I^{2}C host protocol controls the flow of requests to the I^{2}C controller. Use the **EFI_I2C_HOST_PROTOCOL** when the I^{2}C device is not enumerated by the **EFI_I2C_ENUMERATE_PROTOCOL**. Use a protocol produced by a third party driver when it is available or the **EFI_I2C_IO_PROTOCOL** when the third party driver is not available but the device is enumerated with the **EFI_I2C_ENUMERATE_PROTOCOL**.

When Event is NULL, *EnableI2cBusConfiguration* operates synchronously and returns the I^{2}C completion status as its return value. The values returned from *EnableI2cBusConfiguration* are:

Status Codes Returned

EFI_SUCCESS	The asynchronous bus configuration request was successfully started when *Event* is not NULL.
EFI_SUCCESS	The bus configuration request completed successfully when *Event* is NULL.
EFI_DEVICE_ERROR	The bus configuration failed.
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

17.2.5 I^{2}C Enumerate Protocol

EFI_I2C_ENUMERATE_PROTOCOL

Summary

Support the enumeration of the I^{2}C devices.

GUID

```
#define EFI_I2C_ENUMERATE_PROTOCOL_GUID \
{ 0xda8cd7c4, 0x1c00, 0x49e2, { 0x80, 0x3e, 0x52, 0x14, \
0xe7, 0x01, 0x89, 0x4c }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_ENUMERATE_PROTOCOL {
  EFI_I2C_ENUMERATE_PROTOCOL_ENUMERATE          Enumerate;
  EFI_I2C_ENUMERATE_PROTOCOL_GET_BUS_FREQUENCY  GetBusFrequency;
} EFI_I2C_ENUMERATE_PROTOCOL;
```

Parameters

Enumerate

> Traverse the set of I^{2}C devices on an I^{2}C bus. This routine returns the next I^{2}C device on an I^{2}C bus.

GetBusFrequency

> Get the requested I^{2}C bus frequency for a specified bus configuration.

Description

The I^{2}C bus driver uses this protocol to enumerate the devices on the I^{2}C bus.

Related Definitions

```
typedef struct {
  CONST EFI_GUID  *DeviceGuid;
  UINT32          DeviceIndex;
  UINT32          HardwareRevision;
  UINT32          I2cBusConfiguration;
  UINT32          SlaveAddressCount;
  CONST UINT32    *SlaveAddressArray;
} EFI_I2C_DEVICE;
```

Parameters

DeviceGuid

> Unique value assigned by the silicon manufacture or the third party I^{2}C driver writer for the I^{2}C part. This value logically combines both the manufacture name and the I^{2}C part number into a single value specified as a GUID.

DeviceIndex

> Unique ID of the I^{2}C part within the system

HardwareRevision

> Hardware revision - ACPI _HRV value. See the *Advanced Configuration and Power Interface Specification, Revision 5.0* for the field format and the *Plug and play support for I2C* web-page for restriction on values.

I2cBusConfiguration

> I^{2}C bus configuration for the I^{2}C device

SlaveAddressCount

> Number of slave addresses for the I^{2}C device.

SlaveAddressArray

> Pointer to the array of slave addresses for the I^{2}C device.

Description

The **EFI_I2C_ENUMERATE_PROTOCOL** uses the **EFI_I2C_DEVICE** to describe the platform specific details associated with an I^{2}C device. This description is passed to the I^{2}C bus driver during enumeration where it is made available to the third party I^{2}C device driver via the **EFI_I2C_IO_PROTOCOL**.

EFI_I2C_ENUMERATE_PROTOCOL.Enumerate()

Summary

Enumerate the I^{2}C devices

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_ENUMERATE_PROTOCOL_ENUMERATE) (
  IN CONST EFI_I2C_ENUMERATE_PROTOCOL  *This,
  IN OUT CONST EFI_I2C_DEVICE          **Device
  );
```

Parameters

This

> Pointer to an **EFI_I2C_ENUMERATE_PROTOCOL** structure.

Device

> Pointer to a buffer containing an **EFI_I2C_DEVICE** structure. Enumeration is started by setting the initial **EFI_I2C_DEVICE** structure pointer to NULL. The buffer receives an **EFI_I2C_DEVICE** structure pointer to the next I^{2}C device.

Description

This function enables the caller to traverse the set of I^{2}C devices on an I^{2}C bus.

Status Codes Returned

EFI_SUCCESS	The platform data for the next device on the I^{2}C bus was returned successfully.
EFI_INVALID_PARAMETER	*Device* is NULL
EFI_NO_MAPPING	**Device* does not point to a valid **EFI_I2C_DEVICE** structure returned in a previous call *Enumerate()*.

EFI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency()

Summary

Get the requested I^{2}C bus frequency for a specified bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_ENUMERATE_PROTOCOL_GET_BUS_FREQUENCY) (
  IN CONST EFI_I2C_ENUMERATE_PROTOCOL  *This,
  IN UINTN                             I2cBusConfiguration,
  OUT UINTN                            *BusClockHertz
  );
```

Parameters

This

> Pointer to an **EFI_I2C_ENUMERATE_PROTOCOL** structure.

I2cBusConfiguration

> I^{2}C bus configuration to access the I^{2}C device

BusClockHertz

> Pointer to a buffer to receive the I^{2}C bus clock frequency in Hertz

Description

This function returns the requested I^{2}C bus clock frequency for the *I2cBusConfiguration*. This routine is provided for diagnostic purposes and is meant to be called after calling *Enumerate* to get the *I2cBusConfiguration* value.

Status Codes Returned

EFI_SUCCESS	The I^{2}C bus frequency was returned successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* was **NULL**
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

17.3 PEI Code definitions

For the Pre-EFI Initialization environment a subset of the I^{2}C stack is defined to support basic hardware initialization in the PEI phase. The **EFI_PEI_I2C_MASTER** PPI is defined to standardize access to the I^{2}C controller.

17.3.1 I²C Master PPI

EFI_PEI_I2C_MASTER

Summary

This PPI manipulates the I2C host controller to perform transactions as a master on the I2C bus using the current state of any switches or multiplexers in the I2C bus.

GUID

```
#define EFI_PEI_I2C_MASTER_PPI_GUID \
{ 0xb3bfab9b, 0x9f9c, 0x4e8b, { 0xad, 0x37, 0x7f, 0x8c, \
0x51, 0xfc, 0x62, 0x80 }}
```

PEIM-to-PEIM Interface Structure

```
typedef struct _EFI_PEI_I2C_MASTER_PPI {
  EFI_PEI_I2C_MASTER_PPI_SET_BUS_FREQUENCY  SetBusFrequency;
  EFI_PEI_I2C_MASTER_PPI_RESET              Reset;
  EFI_PEI_I2C_MASTER_PPI_START_REQUEST      StartRequest;
  CONST EFI_PEI_I2C_CONTROLLER_CAPABILITIES *
I2cControllerCapabilities;
  EFI_GUID                                  Identifier;
} EFI_PEI_I2C_MASTER_PPI;
```

Parameters

SetBusFrequency

Set the clock frequency in Hertz for the I²C bus.

Reset

Reset the I²C host controller.

StartRequest

Start an I²C transaction in master mode on the host controller.

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I²C host controller.

Identifier

Identifier which uniquely identifies thisI²C controller in the system.

Description

The **EFI_PEI_I2C_MASTER** PPI enables the platform code to perform transactions on the I²C bus.

EFI_PEI_I2C_MASTER_PPI.SetBusFrequency()

Summary

Set the frequency for the I^{2}C clock line.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_SET_BUS_FREQUENCY) (
  IN EFI_PEI_I2C_MASTER   *This,
  IN UINTN                *BusClockHertz
  );
```

Parameters

This

> Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

BusClockHertz

> Pointer to the requested I^{2}C bus clock frequency in Hertz. Upon return this value contains the actual frequency in use by the I^{2}C controller.

Description

The software and controller do a best case effort of using the specified frequency for the I2C bus. If the frequency does not match exactly then the I2C master protocol selects the next lower frequency to avoid exceeding the operating conditions for any of the I2C devices on the bus. For example if 400 KHz was specified and the controller's divide network only supports 402 KHz or 398 KHz then the controller would be set to 398 KHz. If there are no lower frequencies available, then return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The bus frequency was set successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* is NULL
EFI_UNSUPPORTED	The controller does not support this frequency.

EFI_PEI_I2C_MASTER_PPI.Reset()

Summary

Reset the I^{2}C controller and configure it for use.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_RESET) (
  IN CONST EFI_PEI_I2C_MASTER  *This
  );
```

Parameters

This

Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

Description

The I^{2}C controller is reset. The caller must call **SetBusFrequency()** after calling **Reset()**.

Status Codes Returned

EFI_SUCCESS	The reset completed successfully.
EFI_DEVICE_ERROR	The reset operation failed.

EFI_PEI_I2C_MASTER_PPI.StartRequest()

Summary

Start an I^2C transaction on the host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_START_REQUEST) (
  IN CONST EFI_PEI_I2C_MASTER  *This,
  IN UINTN                     SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET    *RequestPacket
  );
```

Parameters

This

> Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

SlaveAddress

> Address of the device on the I^2C bus. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^2C slave addresses and bits 0-9 are used for 10-bit I^2C slave addresses.

RequestPacket

> Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^2C transaction.

Description

This function initiates an I^2C transaction on the controller.

The transaction is performed by sending a start-bit and selecting the I^2C device with the specified I^2C slave address and then performing the specified I^2C operations. When multiple operations are requested they are separated with a repeated start bit and the slave address. The transaction is terminated with a stop bit. When the transaction completes, the status value is returned.

Status Codes Returned

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I²C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NO_RESPONSE	The I²C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_OUT_OF_RESOURCES	Insufficient memory for I²C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.2 I²C Host PPI

EFI_PEI_I2C_HOST

Summary

This PPI provides callers with the ability to do I/O transactions to all of the devices on the I²C bus.

GUID

```
#define EFI_PEI_I2C_HOST_GUID \
{ 0x3a12e52d, 0x3bd2, 0x482c, 0xa6, 0x80, 0x0f, 0xeb, \
0x61, 0x9a, 0xeb, 0xef }
```

PEIM-to-PEIM Interface Structure

```
typedef struct _EFI_I2C_HOST_PPI {
  EFI_I2C_HOST_START_REQUEST          StartRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES *
I2cControllerCapabilities;
  UINTN                               ControllerNumber;
};
```

Parameters

QueueRequest

> Queue a transaction for execution on the I²C bus

I2cControllerCapabilities

> The address of an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I²C host controller.

ControllerNumber

Unique number identifying the I^{2}C controller in the system

Description

Please use **EFI_PEI_I2C_IO** as **EFI_PEI_I2C_HOST** is only intended to be used by the I^{2}C bus driver. The **EFI_PEI_I2C_HOST** requires the tuple **ControllerNumber:BusConfiguration:SlaveAddress** which is platform specific data to identify the I^{2}C device. **EFI_PEI_I2C_IO** eliminates the platform specific details.

The upper layer driver locates the correct **EFI_PEI_I2C_HOST** interface (I2cHost) by comparing the following field:

•   I2cHost→ControllerNumber with the system unique value for the I^{2}C ControllerNumber

Prior to each transaction, the I^{2}C host driver ensures that the switches and multiplexers are properly configured. The I^{2}C host driver then starts the transaction on the I^{2}C controller calling the I^{2}C port driver interface (**EFI_PEI_I2C_MASTER**).

Related Definitions

The I^{2}C platform driver installs the following GUID after installing **EFI_PEI_I2C_ENUMERATE** and **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** PPIs for the necessary I^{2}C controllers. The following GUID resolves the dependency expressions for the I^{2}C port and host drivers enabling them to load and start their configuration.

Lack of **EFI_PEI_I2C_BUS_CONFIGURATION** prevents the I^{2}C host driver from loading, reserving the I^{2}C port driver for SMBus transactions.

GUID

```
#define EFI_PEI_I2C_BUS_CONFIGURED_GUID \
{ 0x9eade134, 0x6bb1, 0x421d, 0xac, 0xaf, 0x59, 0x0a, \
0x5d, 0x2e, 0xa6, 0x3a }
```

EFI_PEI_I2C_HOST.StartRequest()

Summary

Start a transaction on the I^{2}C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_HOST_START_REQUEST) (
  IN EFI_PEI_I2C_HOST      *This,
  IN UINTN                 I2cBusConfiguration,
  IN UINTN                 SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET *RequestPacket
  );
```

Parameters

This

> Address of an **EFI_PEI_I2C_HOST** structure.

I2cBusConfiguration

> I^{2}C bus configuration to access the I^{2}C device

SlaveAddress

> Address of the device on the I^{2}C bus. Or in the value **I2C_ADDRESSING_10_BIT** when using 10-bit addresses.

RequestPacket

> Address of an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

Description

Start an I2C transaction on the I2C controller.

N.B. The typical consumers of this API are the I^{2}C bus driver and on rare occasions I^{2}C test applications. Extreme care must be taken by other consumers of this API to prevent confusing the third party I^{2}C drivers due to a state change at the I^{2}C device which the third party I^{2}C drivers did not initiate. I^{2}C platform drivers may use this API within these guidelines.

This layer uses the concept of I^{2}C bus configurations to describe the I^{2}C bus. An I^{2}C bus configuration is defined as a unique setting of the multiplexers and switches in the I^{2}C bus which enable access to one or more I^{2}C devices. When using a switch to divide a bus, due to speed differences, the I^{2}C platform layer would define an I^{2}C bus configuration for the I^{2}C devices on each side of the switch. When using a multiplexer, the I^{2}C platform layer defines an I^{2}C bus configuration

for each of the selector values required to control the multiplexer. See Figure 1 in the *I2C -bus specification and user manual* for a complex I^{2}C bus configuration.

The I^{2}C host driver calls the I^{2}C platform driver to reconfigure the switches and multiplexers in the I^{2}C bus enabling access to the specified I^{2}C device. The I^{2}C platform driver also selects the maximum bus speed for the device. After the I^{2}C bus is configured, the I^{2}C host driver calls the I^{2}C port driver to initialize the I^{2}C controller and start the I^{2}C transaction.

In event of timeout, the I2C host driver calls the I2C platform driver in an attempt to reset the host controller and the I2C device.

Status Codes Returned

The values returned from *StartRequest* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction. One possible cause is that the slave device is not present.
EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I^{2}C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.3 I^{2}C I/O PPI

EFI_PEI_I2C_IO

Summary

The EFI I^{2}C I/O PPI enables the user to manipulate a single I^{2}C device independent of the host controller and I^{2}C bus design.

GUID

```
#define EFI_PEI_I2C_IO_GUID \
{ 0x42179ed0, 0x2fa2, 0x47c0, 0x85, 0x7e, 0x8b, 0xc0, \
0x18, 0x81, 0xea, 0x97 }
```

PEIM-to-PEIM Interface Structure

```
typedef struct {
  EFI_PEI_I2C_IO_GET_DEVICE_INFO          GetDeviceInfo;
  EFI_PEI_I2C_IO_GET_DEVICE_INFO_ID_LIST  GetDeviceInfoIdList;
  EFI_PEI_I2C_IO_START_REQUEST            StartRequest;
  EFI_I2C_DEVICE                          *I2cDevice;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES   *
I2cControllerCapabilities;
} EFI_PEI_I2C_IO;
```

Parameters

GetDeviceInfo

> Get a blob of data identified by a GUID.

GetDeviceInfoIdList

> Get a list of the GUIDs associated with this I^{2}C device.

StartRequest

> Start a transaction on the I^{2}C device.

I2cDevice

> A pointer to the **EFI_I2C_DEVICE** structure contained within the I^{2}C platform driver.

I2cControllerCapabilities

> The address of an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

The I^{2}C I/O PPI enables access to a specific device on the I^{2}C bus.

Each I^{2}C device is identified uniquely in the system by the tuple *DeviceGuid:DeviceIndex*. The *DeviceGuid* combines the manufacture and part number and is provided by the silicon vendor or the third party I^{2}C device driver writer. The *DeviceIndex* identifies the part within the system by using a unique number and is created by the board designer or the I^{2}C platform driver writer.

The upper layer I2C driver writer provides the following to the platform vendor:

- Vendor specific GUID for the I^{2}C part that is used to connect the upper layer driver to the device.

- Slave address array guidance when the I^{2}C device uses more than one slave address. This is used to access the blocks of hardware within the I^{2}C device.

The upper layer driver locates the correct **EFI_PEI_I2C_IO** interface (I2cIo) by comparing the following fields:

- I2cIo➔Device.DeviceGuid with the vendor supplied GUID

- I2cIo➔DeviceIndex with the system wide unique number assigned to the specific I^{2}C part.

I^{2}C slave addressing is abstracted to validate addresses and limit operation to the specified I^{2}C device. The third party providing the I^{2}C device support provides an ordered list of slave addresses for the I^{2}C device to the team building the platform layer. The platform team must preserve the order of the supplied list. *SlaveAddressCount* is the number of entries in this list or array within the platform layer. The third party device support references a slave address using an index into the list or array in the range of zero to *SlaveAddressCount* - 1.

EFI_I2C_IO_PROTOCOL.GetDeviceInfo()

Summary

Get a data blob associated with the I²C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_GET_DEVICE_INFO) (
  IN EFI_PEI_I2C_IO      *This,
  IN EFI_GUID            *DataGuid,
  IN OUT UINT32          *LengthInBytes,
  OUT VOID               *Buffer
  );
```

Parameters

This

>Address of an **EFI_PEI_I2C_IO** structure.

DataGuid

>Address of the GUID associated with the data

LengthInBytes

>Address of a value containing the length of the buffer in bytes on input and receiving the length of the data on output. If the input length was too small, the output length specifies the data length.

Buffer

>Buffer address to receive the data

Description

This routine locates the specified data blob associated with the I²C device.

Status Codes Returned

The values returned from *GetDeviceInfo* are:

EFI_SUCCESS	The data was returned successfully
EFI_BUFFER_TOO_SMALL	The specified buffer length is too small
EFI_INVALID_PARAMETER	*Buffer* is **NULL**
EFI_INVALID_PARAMETER	*DataGuid* is **NULL**
EFI_INVALID_PARAMETER	*LengthInBytes* is **NULL**
EFI_NOT_FOUND	Data blob was not found

EFI_I2C_IO_PROTOCOL.GetDeviceInfoIdList()

Summary

Get the list of data associated with the I²C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_GET_DEVICE_INFO_ID_LIST) (
  IN EFI_PEI_I2C_IO    *This,
  IN CONST EFI_GUID    ***GuidArray,
  IN UINTN             *GuidEntries
  );
```

Parameters

This

        Address of an **EFI_PEI_I2C_IO** structure.

GuidArray

        Address to receive the list of GUIDs

GuidEntries

        Address to receive the number of entries in the GUID array

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine returns an array of GUIDs identifying data associated with the I2C device. When the caller is done with the GUID array, the caller must call *FreePool* to return the GUID array to the heap.

Status Codes Returned

The values returned from *GetDeviceInfoIdList* are:

EFI_SUCCESS	The GUID array was returned successfully
EFI_INVALID_PARAMETER	*GuidArray* is **NULL**
EFI_INVALID_PARAMETER	*GuidEntries* is **NULL**
EFI_OUT_OF_RESOURCES	Memory allocation failure

EFI_PEI_I2C_IO.StartRequest()

Summary

Start an I²c transaction on the I²C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_BUS_START_REQUEST) (
  IN EFI_PEI_I2C_IO        *This,
  IN UINTN                 SlaveAddressIndex,
  IN EFI_I2C_REQUEST_PACKET *RequestPacket
  );
```

Parameters

This

> Address of an **EFI_PEI_I2C_IO** structure.

SlaveAddressIndex

> Index value into an array of slave addresses for the I²C device. The values in the array are specified by the board designer, with the I²C device driver writer providing the slave address order.

> For devices that have a single slave address, this value must be zero. If the I²C device uses more than one slave address then the third party (upper level) I²C driver writer needs to specify the order of entries in the slave address array.

RequestPacket

> Address of an **EFI_I2C_REQUEST_PACKET** structure describing the I²C transaction

Description

N.B. The typical consumers of this API are the third party I²C drivers. Extreme care must be taken by other consumers of this API to prevent confusing the third party I²C drivers due to a state change at the I²C device which the third party I²C drivers did not initiate. I²C platform drivers may use this API within these guidelines.

This routine starts a transaction on the I²C device.

Status Codes Returned

The values returned from *StartRequest* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.

EFI_DEVICE_ERROR	There was an I²C error (NACK) during the transaction. One possible cause is that the slave device is not present.
EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I²C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I²C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I²C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.4 I²C Bus Configuration Management PPI

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT

Summary

The EFI I²C bus configuration management PPI enables the host driver to interact with the platform layer to reconfigure the switches and multiplexers and set the clock speed for the I²C bus. This PPI also enables the I²C host driver to reset an I²C device which may be locking up the I²C bus by holding the clock or data line low.

GUID

```
#define EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_GUID \
{ 0xe721de6f, 0x145b, 0x4532, 0xbd, 0x78, 0x9b, 0x40, \
0x95, 0xc7, 0x46, 0x97 }
```

PEIM-to-PEIM Interface Structure

```
typedef struct {

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_ENABLE_I2C_BUS_CONFIGUR
ATION EnableI2cBusConfiguration;
  EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_I2C_DEVICE_RESET
I2cDeviceReset;
} EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT;
```

Parameters

EnableI2cBusConfiguration

Enable an I²C bus configuration for use.

I2cDeviceReset

Perform a platform specific reset for the specified I^{2}C part and the I^{2}C controller.

ControllerNumber

Unique number identifying the I^{2}C controller in the system.

Description

The I^{2}C driver stack uses the concept of an I^{2}C bus configuration as a way to describe a particular state of the switches and multiplexers in the I^{2}C bus.

A simple I^{2}C bus does not have any multiplexers or switches is described to the I^{2}C driver stack with a single I^{2}C bus configuration.

An I^{2}C bus with switches and multiplexers use an I^{2}C bus configuration to describe each of the unique settings for the switches and multiplexers. However the I^{2}C platform driver only needs to define the I^{2}C bus configurations that the software uses, which is a subset of the total.

The upper layer driver locates the correct **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** interface (I2cBusConfig) by comparing the following field:

• I2cBusConfigControllerNumber with the system unique value for the I^{2}C ControllerNumber

The I^{2}C bus configuration description includes a list of I^{2}C devices which may be accessed when this I^{2}C bus configuration is enabled. I^{2}C devices before a switch or multiplexer must be included in one I^{2}C bus configuration while I^{2}C devices after a switch or multiplexer are on another I^{2}C bus configuration.

The I^{2}C bus configuration management PPI is an optional PPI provided by the I^{2}C platform driver. The I^{2}C host driver only starts for this I^{2}C controller if the I^{2}C bus configuration management PPI is present. The I^{2}C host driver uses the I^{2}C bus configuration management PPI to call into the I^{2}C platform driver to set the switches and multiplexers and set the maximum I^{2}C bus frequency.

The platform designers determine the maximum I^{2}C bus frequency by selecting a frequency which supports all of the I^{2}C devices on the I^{2}C bus for the setting of switches and multiplexers. The platform designers must validate this against the I^{2}C device data sheets and any limits of the I^{2}C controller or bus length.

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT. EnableI2cBusConfiguration()

Summary

Enable access to an I^{2}C bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_ENABLE_I2C_BUS_CONFIGU
RATION) (
  IN EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT *This,
  IN UINTN                              I2cBusConfiguration
  );
```

Parameters

This

> Address of an **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** structure.

I2cBusConfiguration

> Index into a list or array of I^{2}C bus configurations

Description

Reconfigure the switches and multiplexers in the I^{2}C bus to enable access to a specific I^{2}C bus configuration. Also select the maximum clock frequency for this I^{2}C bus configuration.

This routine uses the I^{2}C Master PPI when the platform routine needs to perform I^{2}C transactions on the local bus. This eliminates any recursion in the I^{2}C stack for configuration transactions on the local bus.

The platform layer must perform I^{2}C transactions on other I^{2}C busses by using the **EFI_PEI_I2C_HOST** PPI or third party driver interface for the specific device. Use the **EFI_PEI_I2C_HOST** PPI when the device is not defined by the I^{2}C platform driver. Use the third party driver when it is available or **EFI_PEI_I2C_IO** when the third party driver is not available but the device is defined in the I^{2}C platform driver.

Status Codes Returned

The values returned from *EnableI2cBusConfiguration* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction. One possible cause is that the slave device is not present.

EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I^{2}C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT.I2cDeviceReset ()

Summary

Perform a platform specific reset for the specified I²C part and the I²C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_I2C_DEVICE_RESET) (
  IN CONST EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT *This,
  IN UINTN                                I2cBusConfiguration,
  IN UINTN                                SlaveAddress
  );
```

Parameters

This

>   Address of an **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** structure.

I2cBusConfiguration

>   Index into a list or array of I²C bus configurations

SlaveAddress

>   Address of the device on the I²C bus.

Description

This error handling routine is called by the I²C host driver when **EFI_TIMEOUT** status is returned by the I²C port driver for an **EFI_I2C_REQUEST_PACKET**. This routine attempts to reset the failing I²C device and the I²C controller.

Status Codes Returned

The values returned from *I2cDeviceReset* are:

EFI_SUCCESS	The transaction completed successfully. Only the requested I²C device and the I²C controller were reset.
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset! The host driver needs to return errors for the queue of pending **EFI_I2C_REQUEST_PACKETs**.

17.3.5 I^{2}C Enumerate PPI

EFI_PEI_I2C_ENUMERATE

Summary

Support the enumeration of the I^{2}C devices listed in the I^{2}C platform driver.

GUID

```
#define EFI_PEI_I2C_ENUMERATE_GUID \
{ 0xbe83f6f4, 0xe286, 0x4e70, 0xb4, 0x51, 0x1a, 0x2e, \
0x42, 0xdf, 0x31, 0x03 }
```

Protocol Interface Structure

```
typedef struct {
  EFI_PEI_I2C_ENUMERATE_ENUMERATE       Enumerate;
  EFI_PEI_I2C_ENUMERATE_GET_BUS_FREQUENCY GetBusFrequency;
} EFI_PEI_I2C_ENUMERATE;
```

Parameters

Enumerate

Walk the platform's list of I^{2}C devices on the bus. This routine returns the next I^{2}C device in the platform's list for this I^{2}C bus.

GetBusFrequency

Get the I^{2}C bus frequency for the bus configuration.

Description

The I^{2}C bus driver uses this PPI to enumerate the devices on the I^{2}C bus listed in the platform layer.

The upper layer driver locates the correct **EFI_PEI_I2C_ENUMERATE** interface (I2cEnumerate) by comparing the following field:

- *I2cEnumerateControllerNumber* with the system unique value for the I^{2}C *ControllerNumber*.

EFI_PEI_I2C_ENUMERATE_PROTOCOL.Enumerate()

Summary

Enumerate the I^{2}C devices

Prototype

```
typedef
  EFI_STATUS
  (EFIAPI *EFI_PEI_I2C_ENUMERATE_ENUMERATE) (
    IN EFI_PEI_I2C_ENUMERATE      *This,
    IN OUT CONST EFI_I2C_DEVICE   **Device
    );
```

Parameters

This

Address of an **EFI_PEI_I2C_ENUMERATE** structure.

Device

Address of a buffer containing an **EFI_I2C_DEVICE** structure. Enumeration is started by setting the initial **EFI_I2C_DEVICE** structure address to NULL. The buffer receives an **EFI_I2C_DEVICE** structure address for the next I^{2}C device.

Description

This function walks the platform specific data to enumerate the I^{2}C devices on an I^{2}C bus.

Status Codes Returned

The values returned from *Enumerate* are:

EFI_SUCCESS	The platform data for the next device on the I^{2}C bus was returned successfully.
EFI_INVALID_PARAMETER	Device was **NULL**
EFI_NO_MAPPING	Device does not point to a valid **EFI_I2C_DEVICE** structure.

EFI_PEI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency()

Summary

Get the I^{2}C bus frequency for the bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_ENUMERATE_GET_BUS_FREQUENCY) (
  IN EFI_PEI_I2C_ENUMERATE  *This,
  IN UNITN                  I2cBusConfiguration,
  IN OUT UINTN              *BusClockHertz
  );
```

Parameters

This

> Address of an **EFI_PEI_I2C_ENUMERATE** structure.

I2cBusConfiguration

> I^{2}C bus configuration to access the I^{2}C device

BusClockHertz

> Address to receive the I^{2}C bus clock frequency in Hertz

Description

This function returns the I^{2}C bus clock frequency for the specified I^{2}C bus configuration.

Status Codes Returned

The values returned from *GetBusFrequency* are:

EFI_SUCCESS	The I^{2}C bus frequency was returned successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* was **NULL**
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

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
```