x

VOLUME 1: Platform Initialization Specification

Pre-EFI Initialization Core Interface

Version 1.5

7/29/2016

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 • 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	02/23/09
1.2	• 407 Comment: additional change to LMA Pseudo-Register • 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	05/13/09

1.2 errata	• 345 PI1.0 errata • 468 Issues on proposed PI1.2 ACPI System Description Table Protocol • 492 Add Resource HOB Protectability Attributes • 494 Vol. 2 Appendix A Clean up • 495 Vol 1: update HOB reference • 380 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	12/16/09

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 • 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	10/27/11

1.2.1	• 527 PI Volume 2 DXE Security Architecture Protocol (SAP) clarification • 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	05/02/12
1.2.1 Errata A	• 922 Add a "Boot with Manufacturing" boot mode setting • 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 • 969Vol 1 errata: TE Header parameters	10/26/12
1.3	• 945 Integrated Circuit (I2C) Bus Protocol • 998 PI Status Code additions • 999 PCI enumeration complete GUID • 1005 NVMe Disk Info guid • 1006 Security Ppi Fixes • 1025 PI table revisions	3/29/13

1.3 Errata	• 1041 typo in HOB Overview • 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	2/19/15
1.4	• 1210 Adding persistence attribute to GCD • 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.	4/13/15

1.4 Errata A	• 1574 Fix artificial limitation in the PCD.SetSku support • 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	3/15/16
1.5	• 1315 SMM Environment to Support Newer Architecture/Platform Designs • 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	4/26/16

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

6

Architectural PPIs

1
Introduction

1.1 Overview

This specification defines the core code and services that are required for an implementation of the Pre-EFI Initialization (PEI) phase of the Platform Initialization (PI) specifications (hereafter referred to as the "PI Architecture"). This PEI core interface specification (CIS) does the following:

- Describes the basic components of the PEI phase
- Provides code definitions for services and functions that are architecturally required by the UEFI PI working group (PIWG)
- Describes the machine preparation that is required for subsequent phases of firmware execution
- Discusses state variables that describe the system restart type

See "Organization of the PEI CIS," below, for more information.

1.2 Organization of the PEI CIS

This PEI core interface specification is organized as shown in Table 1. Because the PEI Foundation is just one component of a PI Architecture-based firmware solution, there are a number of additional specifications that are referred to throughout this document.

Table 1. Organization of the PEI CIS

Section	Description
"Overview" on page 9	Describes the major components of PEI, including the PEI Services, boot mode, PEI Dispatcher, and PEIMs.
"PEI Services Table" on page 15	Describes the data structure that maintains the PEI Services.
"Services - PEI" on page 21	Details each of the functions that comprise the PEI Services.
"PEI Foundation" on page 61	Describes the PEI Foundation and its methods of operation and the PEI Dispatcher and its associated dependency expression grammar..
"PEIMs" on page 109	Describes the format and use of the Pre-EFI Initialization Module (PEIM).
"Architectural PPIs" on page 89	Contains PEIM-to-PEIM Interfaces (PPIs) that are used by the PEI Foundation.
"Additional PPIs" on page 121	Contains PPIs that can exist on a platform.
"PEI to DXE Handoff" on page 217	Describes the state of the machine and memory when the PEI phase invokes the DXE phase.
"Boot Paths" on page 219	Describes the restart modalities and behavior supported in the PEI phase.
"PEI Physical Memory Usage" on page 229	Describes the memory map and memory usage during the PEI phase.

Section	Description
"Special Paths Unique to the Itanium® Processor Family" on page 231	Contains flow during PEI that is unique to the Itanium® processor family.
"Security (SEC) Phase Information" on page 239	Contains an overview of the phase of execution that occurs prior to PEI.
"Dependency Expression Grammar" on page 245	Describes the BNF grammar for a tool that can convert a text file containing a dependency expression into a dependency section of a PEIM stored in a firmware volume.
"TE Image" on page 247	Describes the format of the TE executable.
"TE Image Creation" on page 251	Describes how TE executables are created from PE32+ executables.
"TE Image Loading" on page 253	Describes how TE executables are loaded into memory.

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 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.

1.3.3 Instruction Descriptions

A dependency expression instruction description generally has the following format:

InstructionName The formal name of the instruction.

Syntax: A brief description of the instruction.

Description: A description of the functionality provided by the instruction accompanied by a table that details the instruction encoding.

Operation: Details the operations performed on operands.

Behaviors and Restrictions:

An item-by-item description of the behavior of each operand involved in the instruction and any restrictions that apply to the operands or the instruction.

1.3.4 PPI Descriptions

A PEIM-to-PEIM Interface (PPI) description generally has the following format:

PPI Name: The formal name of the PPI.

Summary: A brief description of the PPI.

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

Protocol Interface Structure:

A "C-style" procedure template defining the PPI calling structure.

Parameters: A brief description of each field in the PPI 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 only by this interface.

Status Codes Returned: A description of any codes returned by the interface. The PPI 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.

1.3.5 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).

1.3.6 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.

1.5 Conventions used in this document

1.5.1 Number formats

A binary number is represented in this standard by any sequence of digits consisting of only the Western-Arabic numerals 0 and 1 immediately followed by a lower-case b (e.g., 0101b). Underscores or spaces may be included between characters in binary number representations to increase readability or delineate field boundaries (e.g., 0 0101 1010b or 0_0101_1010b).

A hexadecimal number is represented in this standard by 0x preceding any sequence of digits consisting of only the Western-Arabic numerals 0 through 9 and/or the upper-case English letters A through F (e.g., 0xFA23). Underscores or spaces may be included between characters in hexadecimal number representations to increase readability or delineate field boundaries (e.g., 0xB FD8C FA23 or 0xB_FD8C_FA23).

A decimal number is represented in this standard by any sequence of digits consisting of only the Arabic numerals 0 through 9 not immediately followed by a lower-case b or lower-case h (e.g., 25).

This standard uses the following conventions for representing decimal numbers:

- the decimal separator (i.e., separating the integer and fractional portions of the number) is a period;
- the thousands separator (i.e., separating groups of three digits in a portion of the number) is a comma;
- the thousands separator is used in the integer portion and is not used in the fraction portion of a number.

1.5.2 Binary prefixes

This standard uses the prefixes defined in the International System of Units (SI) (see http://www.bipm.org/en/si/si_brochure/chapter3/prefixes.html) for values that are powers of ten.

Table 2. SI prefixes

Factor	Factor	Name	Symbol
10^3	1,000	kilo	K
10^6	1,000,000	mega	M
10^9	1,000,000,000	giga	G

This standard uses the binary prefixes defined in ISO/IEC 80000-13 *Quantities and units -- Part 13: Information science and technology* and IEEE 1514 *Standard for Prefixes for Binary Multiples* for values that are powers of two.

Table 3. Binary prefixes

Factor	Factor	Name	Symbol
2^{10}	1,024	kibi	Ki
2^{20}	1,048,576	mebi	Mi
2^{30}	1,073,741,824	gibi	Gi

For example, 4 KB means 4,000 bytes and 4 KiB means 4,096 bytes.

2
Overview

2.1 Introduction

The Pre-EFI Initialization (PEI) phase of the PI Architecture specifications (hereafter referred to as the “PI Architecture”) is invoked quite early in the boot flow. Specifically, after some preliminary processing in the Security (SEC) phase, any machine restart event will invoke the PEI phase.

The PEI phase will initially operate with the platform in a nascent state, leveraging only on-processor resources, such as the processor cache as a call stack, to dispatch Pre-EFI Initialization Modules (PEIMs). These PEIMs are responsible for the following:

- Initializing some permanent memory complement
- Describing the memory in Hand-Off Blocks (HOBs)
- Describing the firmware volume locations in HOBs
- Passing control into the Driver Execution Environment (DXE) phase

Philosophically, the PEI phase is intended to be the thinnest amount of code to achieve the ends listed above. As such, any more sophisticated algorithms or processing should be deferred to the DXE phase of execution.

The PEI phase is also responsible for crisis recovery and resuming from the S3 sleep state. For crisis recovery, the PEI phase should reside in some small, fault-tolerant block of the firmware store. As a result, it is imperative to keep the footprint of the PEI phase as small as possible. In addition, for a successful S3 resume, the speed of the resume is of utmost importance, so the code path through the firmware should be minimized. These two boot flows also speak to the need to keep the processing and code paths in the PEI phase to a minimum.

The implementation of the PEI phase is more dependent on the processor architecture than any other phase. In particular, the more resources the processor provides at its initial or near initial state, the richer the interface between the PEI Foundation and PEIMs. As such, there are several parts of the following discussion that note requirements on the architecture but are otherwise left architecturally dependent.

2.2 Design Goals

The PI Architecture requires the PEI phase to configure a system to meet the minimum prerequisites for the Driver Execution Environment (DXE) phase of the PI Architecture architecture. In general, the PEI phase is required to initialize a linear array of RAM large enough for the successful execution of the DXE phase elements.

The PEI phase provides a framework to allow vendors to supply separate initialization modules for each functionally distinct piece of system hardware that must be initialized prior to the DXE phase of execution in the PI Architecture. The PEI phase provides a common framework through which the separate initialization modules can be independently designed, developed, and updated. The PEI phase was developed to meet the following goals in the PI architecture:

- Enable maintenance of the "chain of trust." This includes protection against unauthorized updates to the PEI phase or its modules, as well as a form of authentication of the PEI Foundation and its modules during the PEI phase.
- Provide a core PEI module (the PEI Foundation) that will remain more or less constant for a particular processor architecture but that will support add-in modules from various vendors, particular for processors, chipsets, RAM initialization, and so on.
- Allow independent development of early initialization modules.

2.3 Pre-EFI Initialization (PEI) Phase

The design for the Pre-EFI Initialization (PEI) phase of a PI Architecture-compliant boot is as an essentially miniature version of the DXE phase of the PI Architecture and addresses many of the same issues. The PEI phase is designed to be developed in several parts. The PEI phase consists of the following:

- Some core code known as the PEI Foundation
- Specialized plug-ins known as Pre-EFI Initialization Modules (PEIMs)

Unlike DXE, the PEI phase cannot assume the availability of reasonable amounts of RAM, so the richness of the features in DXE does not exist in PEI. The PEI phase limits its support to the following actions:

- Locating, validating, and dispatching PEIMs
- Facilitating communication between PEIMs
- Providing handoff data to subsequent phases

Figure 1 below shows a diagram of the process completed during the PEI phase.

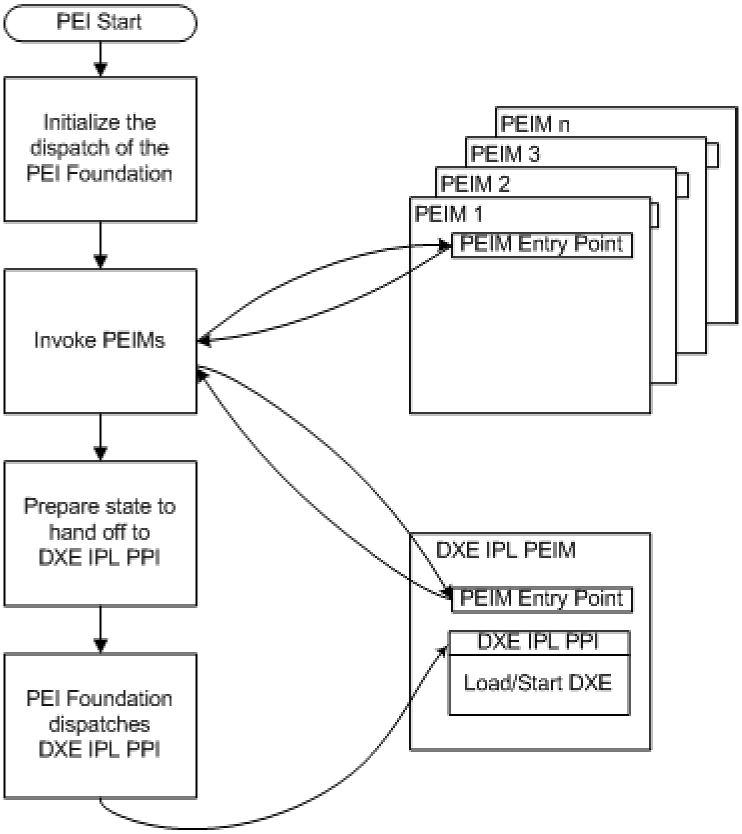

PEI Start
Initialize the dispatch of the PEI Foundation
Invoke PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

Figure 1. PEI Operations Diagram

2.4 PEI Services

The PEI Foundation establishes a system table named the PEI Services Table that is visible to all Pre-EFI Initialization Modules (PEIMs) in the system. A PEI Service is defined as a function, command, or other capability manifested by the PEI Foundation when that service's initialization requirements are met. Because the PEI phase has no permanent memory available until nearly the end of the phase, the range of services created during the PEI phase cannot be as rich as those created during later phases. Because the location of the PEI Foundation and its temporary RAM is not known at build time, a pointer to the PEI Services Table is passed into each PEIM's entry point and also to part of each PEIM-to-PEIM Interface (PPI).

The PEI Foundation provides the classes of services listed in Table 4.

Table 4. PEI Foundation Classes of Service

PPI Services:	Manages PPIs to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
Boot Mode Services:	Manages the boot mode (S3, S5, normal boot, diagnostics, etc.) of the system.
HOB Services:	Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the PI Architecture.
Firmware Volume Services:	Finds PEIMs and other firmware files in the firmware volumes.
PEI Memory Services:	Provides a collection of memory management services for use both before and after permanent memory has been discovered.
Status Code Services:	Provides common progress and error code reporting services (for example, port 080h or a serial port for simple text output for debug).
Reset Services:	Provides a common means by which to initiate a warm or cold restart of the system.

2.5 PEI Foundation

The PEI Foundation is the entity that is responsible for the following:

- Successfully dispatching Pre-EFI Initialization Modules (PEIMs)
- Maintaining the boot mode
- Initializing permanent memory
- Invoking the Driver Execution Environment (DXE) loader

The PEI Foundation is written to be portable across all platforms of a given instruction-set architecture. As such, a binary for 32-bit Intel® architecture (IA-32) should work across all Pentium® processors, from the Pentium II processor with MMX™ technology through the latest Pentium 4 processors. Similarly, the PEI Foundation binary for the Itanium® processor family should work across all Itanium processors.

Regardless of the processor microarchitecture, the set of services exposed by the PEI Foundation should be the same. This uniform surface area around the PEI Foundation allows PEIMs to be written in the C programming language and compiled across any microarchitecture.

2.6 PEI Dispatcher

The PEI Dispatcher is essentially a state machine that is implemented in the PEI Foundation. The PEI Dispatcher evaluates the dependency expressions in Pre-EFI Initialization Modules (PEIMs) that are in the firmware volume(s) being examined.

The dependency expressions are logical combinations of PEIM-to-PEIM Interfaces (PPIs). These expressions describe the PPIs that must be available before a given PEIM can be invoked. To evaluate the dependency expression for the PEIM, the PEI Dispatcher references the PPI database in the PEI Foundation to determine which PPIs have been installed. If the PPI has been installed, the dependency expression will evaluate to **TRUE**, which tells the PEI Dispatcher it can run the PEIM. At this point, the PEI Foundation passes control to the PEIM with a true dependency expression.

Once the PEI Dispatcher has evaluated all of the PEIMs in all of the exposed firmware volumes and no more PEIMs can be dispatched (i.e., the dependency expressions do not evaluate from **FALSE** to **TRUE**), the PEI Dispatcher will exit. It is at this point that the PEI Dispatcher cannot invoke any additional PEIMs. The PEI Foundation then reassumes control from the PEI Dispatcher and invokes the DXE IPL PPI to pass control to the DXE phase of execution.

2.7 Pre-EFI Initialization Modules (PEIMs)

Pre-EFI Initialization Modules (PEIMs) are specialized drivers that personalize the PEI Foundation to the platform. They are analogous to DXE drivers and generally correspond to the components being initialized. It is the responsibility of the PEI Foundation code to dispatch the PEIMs in a sequenced order and provide basic services. The PEIMs are intended to mirror the components being initialized.

Communication between PEIMs is not easy in a “memory poor” environment. Nonetheless, PEIMs cannot be coded without some interaction between one another and, even if they could, it would be inefficient to do so. The PEI phase provides mechanisms for PEIMs to locate and invoke interfaces from other PEIMs.

Because the PEI phase exists in an environment where minimal hardware resources are available and execution is performed from the boot firmware device, it is strongly recommended that PEIMs do the minimum necessary work to initialize the system to a state that meets the prerequisites of the DXE phase.

It is expected that, in the future, common practice will be that the vendor of a software or hardware component will provide the PEIM (possibly in source form) so the customer can debug integration problems quickly.

2.8 PEIM-to-PEIM Interfaces (PPIs)

PEIMs communicate with each other using a structure called a PEIM-to-PEIM Interface (PPI). PPIs are contained in a **EFI_PEI_PPI_DESCRIPTOR** data structure, which is composed of a GUID/ pointer pair. The GUID “names” the interface and the associated pointer provides the associated data structure and/or service set for that PPI. A consumer of a PPI must use the PEI Service **LocatePpi()** to discover the PPI of interest. The producer of a PPI publishes the available PPIs in its PEIM using the PEI Services **InstallPpi()** or **ReinstallPpi()**.

All PEIMs are registered and located in the same fashion, namely through the PEI Services listed above. Within this name space of PPIs, there are two classes of PPIs:

- Architectural PPIs
- Additional PPIs

An *architectural PPI* is a PPI whose GUID is described in the PEI CIS and is a GUID known to the PEI Foundation. These architectural PPIs typically provide a common interface to the PEI Foundation of a service that has a platform-specific implementation, such as the PEI Service **ReportStatusCode()**.

Additional PPIs are PPIs that are important for interoperability but are not depended upon by the PEI Foundation. They can be classified as mandatory or optional. Specifically, to have a large class of interoperable PEIMs, it would be good to signal that the final boot mode was installed in some standard fashion so that PEIMs could use this PPI in their dependency expressions. The alternative to defining these additional PPIs in the PEI CIS would be to have a proliferation of similar services under different names.

2.9 Firmware Volumes

Pre-EFI Initialization Modules (PEIMs) reside in firmware volumes (FVs). The PEI Foundation, defined here, must reside in the Boot Firmware Volume (BFV). The PEI phase supports the ability for PEIMs to reside in multiple firmware volumes.. Other PEIMs can expose firmware volumes for use by the PEI Foundation.

3
PEI Services Table

3.1 Introduction

The PEI Foundation establishes a system table named the PEI Services Table that is visible to all Pre-EFI Initialization Modules (PEIMs) in the system. A PEI Service is defined as a function, command, or other capability manifested by the PEI Foundation when that service's initialization requirements are met. Because the PEI phase has no permanent memory available until nearly the end of the phase, the range of services created during the PEI phase cannot be as rich as those created during later phases. Because the location of the PEI Foundation and its temporary RAM is not known at build time, a pointer to the PEI Services Table is passed into each PEIM's entry point and also to part of each PEIM-to-PEIM Interface (PPI).

Note: *In the PEI Foundation use of the* **EFI_TABLE_HEADER** *for the PEI Services Table, there is special treatment of the CRC32 field. This value is ignorable for PEI and should be set to zero.*

3.2 PEI Services Table

3.2.1 EFI_PEI_SERVICES

Summary

The PEI Services Table includes a list of function pointers in a table. The table is located in the temporary or permanent memory, depending upon the capabilities and phase of execution of PEI. The functions in this table are defined in "Services - PEI" on page 21.

Related Definitions

```
//
// PEI Specification Revision information
//
#define PEI_SPECIFICATION_MAJOR_REVISION 1
#define PEI_SPECIFICATION_MINOR_REVISION 50

//
// UEFI PEI Services Table
//
#define PEI_SERVICES_SIGNATURE     0x5652455320494550
#define ((PEI_SPECIFICATION_MAJOR_REVISION<<16) |
(PEI_SPECIFICATION_MINOR_REVISION))

typedef EFI_PEI_SERVICES {
  EFI_TABLE_HEADER                     Hdr;
```

```
//
// PPI Functions
//
EFI_PEI_INSTALL_PPI                    InstallPpi;
EFI_PEI_REINSTALL_PPI                  ReInstallPpi;
EFI_PEI_LOCATE_PPI                     LocatePpi;
EFI_PEI_NOTIFY_PPI                     NotifyPpi;

//
// Boot Mode Functions
//
EFI_PEI_GET_BOOT_MODE                  GetBootMode;
EFI_PEI_SET_BOOT_MODE                  SetBootMode;

//
// HOB Functions
//
EFI_PEI_GET_HOB_LIST                   GetHobList;
EFI_PEI_CREATE_HOB                     CreateHob;

//
// Firmware Volume Functions
//
EFI_PEI_FFS_FIND_NEXT_VOLUME2           FfsFindNextVolume;
EFI_PEI_FFS_FIND_NEXT_FILE2             FfsFindNextFile;
EFI_PEI_FFS_FIND_SECTION_DATA2          FfsFindSectionData;

//
// PEI Memory Functions
//
EFI_PEI_INSTALL_PEI_MEMORY             InstallPeiMemory;
EFI_PEI_ALLOCATE_PAGES                 AllocatePages;
EFI_PEI_ALLOCATE_POOL                  AllocatePool;
EFI_PEI_COPY_MEM                       CopyMem;
EFI_PEI_SET_MEM                        SetMem;

//
// Status Code
EFI_PEI_REPORT_STATUS_CODE             ReportStatusCode;

//
// Reset
//
EFI_PEI_RESET_SYSTEM                   ResetSystem;

//
```

```
  // (the following interfaces are installed by publishing PEIM)
  //
  // I/O Abstractions
  //
  EFI_PEI_CPU_IO_PPI                *CpuIo;
  EFI_PEI_PCI_CFG2_PPI              *PciCfg;

  //
  // Additional File System-Related Services
  //
  EFI_PEI_FFS_FIND_BY_NAME          FfsFindFileByName;
  EFI_PEI_FFS_GET_FILE_INFO         FfsGetFileInfo;
  EFI_PEI_FFS_GET_VOLUME_INFO       FfsGetVolumeInfo;
  EFI_PEI_REGISTER_FOR_SHADOW       RegisterForShadow;

  EFI_PEI_FFS_FIND_SECTION_DATA3    FindSectionData3;
  EFI_PEI_FFS_GET_FILE_INFO2        FfsGetFileInfo2;
  EFI_PEI_RESET2_SYSTEM             ResetSystem2;

} EFI_PEI_SERVICES;
```

Parameters

Hdr

The table header for the PEI Services Table. This header contains the **PEI_SERVICES_SIGNATURE** and **PEI_SERVICES_REVISION** values along with the size of the **EFI_PEI_SERVICES** structure and a 32-bit CRC to verify that the contents of the PEI Foundation Services Table are valid.

InstallPpi

Installs an interface in the PEI PEIM-to-PEIM Interface (PPI) database by GUID. See the **InstallPpi()** function description in this document.

ReInstallPpi

Reinstalls an interface in the PEI PPI database by GUID. See the **ReinstallPpi()** function description in this document.

LocatePpi

Locates an interface in the PEI PPI database by GUID. See the **LocatePpi()** function description in this document.

NotifyPpi

Installs the notification service to be called back upon the installation or reinstallation of a given interface. See the **NotifyPpi()** function description in this document.

GetBootMode

Returns the present value of the boot mode. See the **GetBootMode()** function description in this document.

SetBootMode

Sets the value of the boot mode. See the **SetBootMode()** function description in this document.

GetHobList

Returns the pointer to the list of Hand-Off Blocks (HOBs) in memory. See the **GetHobList()** function description in this document.

CreateHob

Abstracts the creation of HOB headers. See the **CreateHob()** function description in this document.

FfsFindNextVolume

Discovers instances of firmware volumes in the system. See the **FfsFindNextVolume()** function description in this document.

FfsFindNextFile

Discovers instances of firmware files in the system. See the **FfsFindNextFile()** function description in this document.

FfsFindSectionData

Searches for a section in a firmware file. See the **FfsFindSectionData()** function description in this document.

InstallPeiMemory

Registers the found memory configuration with the PEI Foundation. See the **InstallPeiMemory()** function description in this document.

AllocatePages

Allocates memory ranges that are managed by the PEI Foundation. See the **AllocatePages()** function description in this document.

AllocatePool

Frees memory ranges that are managed by the PEI Foundation. See the **AllocatePool()** function description in this document.

CopyMem

Copies the contents of one buffer to another buffer. See the **CopyMem()** function description in this document.

SetMem

Fills a buffer with a specified value. See the **SetMem()** function description in this document.

ReportStatusCode

Provides an interface that a PEIM can call to report a status code. See the **ReportStatusCode()** function description in this document. This is installed by provider PEIM by copying the interface into the PEI Service table.

ResetSystem

Resets the entire platform. See the **ResetSystem()** function description in this document. This is installed by provider PEIM by copying the interface into the PEI Service table.

ResetSystem2

Resets the entire platform. See the **ResetSystem2()** function description in this document. This is installed by provider PEIM by copying the interface into the PEI Service table.

CpuIo

Provides an interface that a PEIM can call to execute an I/O transaction. This interface is installed by provider PEIM by copying the interface into the PEI Service table.

PciCfg

Provides an interface that a PEIM can call to execute PCI Configuration transactions. This interface is installed by provider PEIM by copying the interface into the **EFI_PEI_SERVICES** table.

FfsFindFileByName

Discovers firmware files within a volume by name. See **FfsFindFileByName()** in this document.

FfsGetFileInfo

Return information about a particular file. See **FfsGetFileInfo()** in this document.

FfsGetFileInfo2

Return information about a particular file. See **FfsGetFileInfo2()** in this document.

FfsGetVolumeInfo

Return information about a particular volume. See **FfsGetVolumeInfo()** in this document.

RegisterForShadow

Register a driver to be re-loaded when memory is available. See **RegisterForShadow()** in this document.

FindSectionData3

Searches for a section in a firmware file. See the **FfsFindSectionData3()** function description in this document.

Description

EFI_PEI_SERVICES is a collection of functions whose implementation is provided by the PEI Foundation. These services fall into various classes, including the following:

- Managing the boot mode
- Allocating both early and permanent memory
- Supporting the Firmware File System (FFS)
- Abstracting the PPI database abstraction
- Creating Hand-Off Blocks (HOBs)

A pointer to the **EFI_PEI_SERVICES** table is passed into each PEIM when the PEIM is invoked by the PEI Foundation. As such, every PEIM has access to these services. Unlike the UEFI Boot Services, the PEI Services have no calling restrictions, such as the UEFI 2.0 Task Priority Level (TPL) limitations. Specifically, a service can be called from a PEIM or notification service.

Some of the services are also a proxy to platform-provided services, such as the Reset Services, Status Code Services, and I/O abstractions. This partitioning has been designed to provide a consistent interface to all PEIMs without encumbering a PEI Foundation implementation with platform-specific knowledge. Any callable services beyond the set in this table should be invoked using a PPI. The latter PEIM-installed services will return **EFI_NOT_AVAILABLE_YET** until a PEIM copies an instance of the interface into the **EFI_PEI_SERVICES** table.

4
Services - PEI

4.1 Introduction

A PEI Service is defined as a function, command, or other capability created by the PEI Foundation during a phase that remains available after the phase is complete. Because the PEI phase has no permanent memory available until nearly the end of the phase, the range of PEI Foundation Services created during the PEI phase cannot be as rich as those created during later phases.

Table 5 shows the PEI Services described in this section:

Table 5. PEI Services

PPI Services:	Manages PEIM-to-PEIM Interface (PPIs) to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
Boot Mode Services:	Manages the boot mode (S3, S5, normal boot, diagnostics, etc.) of the system.
HOB Services:	Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the PI Architecture.
Firmware Volume Services	Walks the Firmware File Systems (FFS) in firmware volumes to find PEIMs and other firmware files in the flash device.
PEI Memory Services:	Provides a collection of memory management services for use both before and after permanent memory has been discovered.
Status Code Services:	Provides common progress and error code reporting services (for example, port 080h or a serial port for simple text output for debug).
Reset Services:	Provides a common means by which to initiate a warm or cold restart of the system.

The calling convention for PEI Services is similar to PPIs. See "PEIM-to-PEIM Communication" on page 118 for more details on PPIs.

The means by which to bind a service call into a service involves a dispatch table, **EFI_PEI_SERVICES.** A pointer to the table is passed into the PEIM entry point.

4.2 PPI Services

The following services provide the interface set for abstracting the PPI database:

- InstallPpi()
- ReinstallPpi()
- LocatePpi()
- NotifyPpi()

InstallPpi()

Summary

This service is the first one provided by the PEI Foundation. This function installs an interface in the PEI PPI database by GUID. The purpose of the service is to publish an interface that other parties can use to call additional PEIMs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_INSTALL_PPI) (
  IN CONST EFI_PEI_SERVICES          **PeiServices,
  IN CONST EFI_PEI_PPI_DESCRIPTOR    *PpiList
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

PpiList

A pointer to the list of interfaces that the caller shall install. Type **EFI_PEI_PPI_DESCRIPTOR** is defined in “PEIM Descriptors” on page 113.

Description

This service enables a given PEIM to register an interface with the PEI Foundation. The interface takes a pointer to a list of records that adhere to the format of a **EFI_PEI_PPI_DESCRIPTOR**. Since the PEI Foundation maintains a pointer to the list rather than copying the list, the list must either be in the body of the PEIM or else allocated from temporary or permanent RAM.

The length of the list of described by the **EFI_PEI_PPI_DESCRIPTOR** that has the **EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST** flag set in its *Flags* field. There shall be at least one **EFI_PEI_PPI_DESCRIPTOR** in the list.

There are two types of **EFI_PEI_PPI_DESCRIPTOR**s that can be installed, including the **EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH** and **EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK**.

Status Codes Returned

EFI_SUCCESS	The interface was successfully installed.
EFI_INVALID_PARAMETER	The *PpiList* pointer is **NULL**.
EFI_INVALID_PARAMETER	Any of the PEI PPI descriptors in the list do not have the **EFI_PEI_PPI_DESCRIPTOR_PPI** bit set in the *Flags* field.
EFI_OUT_OF_RESOURCES	There is no additional space in the PPI database.

ReinstallPpi()

Summary

This function reinstalls an interface in the PEI PPI database by GUID. The purpose of the service is to publish an interface that other parties can use to replace an interface of the same name in the protocol database with a different interface.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_REINSTALL_PPI) (
  IN CONST EFI_PEI_SERVICES          **PeiServices,
  IN CONST EFI_PEI_PPI_DESCRIPTOR    *OldPpi,
  IN CONST EFI_PEI_PPI_DESCRIPTOR    *NewPpi
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

OldPpi

A pointer to the former PPI in the database. Type **EFI_PEI_PPI_DESCRIPTOR** is defined in “PEIM Descriptors” on page 113.

NewPpi

A pointer to the new interfaces that the caller shall install.

Description

This service enables PEIMs to replace an entry in the PPI database with an alternate entry.

Status Codes Returned

EFI_SUCCESS	The interface was successfully installed.
EFI_INVALID_PARAMETER	The *OldPpi or NewPpi* pointer is **NULL**.
EFI_INVALID_PARAMETER	Any of the PEI PPI descriptors in the list do not have the **EFI_PEI_PPI_DESCRIPTOR_PPI** bit set in the *Flags* field.
EFI_OUT_OF_RESOURCES	There is no additional space in the PPI database.
EFI_NOT_FOUND	The PPI for which the reinstallation was requested has not been installed.

LocatePpi()

Summary

This function locates an interface in the PEI PPI database by GUID.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_LOCATE_PPI) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN CONST EFI_GUID              *Guid,
  IN UINTN                       Instance,
  IN OUT EFI_PEI_PPI_DESCRIPTOR  **PpiDescriptor  OPTIONAL,
  IN OUT VOID                    **Ppi
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** published by the PEI Foundation.

Guid

A pointer to the GUID whose corresponding interface needs to be found.

Instance

The N-th instance of the interface that is required.

PpiDescriptor

A pointer to instance of the **EFI_PEI_PPI_DESCRIPTOR**.

Ppi

A pointer to the instance of the interface.

Description

This service enables PEIMs to discover a given instance of an interface. This interface differs from the interface discovery mechanism in the UEFI 2.0 specification, namely **HandleProtocol()**, in that the PEI PPI database does not expose the handle's name space. Instead, PEI manages the interface set by maintaining a partial order on the interfaces such that the *Instance* of the interface, among others, can be traversed.

LocatePpi() provides the ability to traverse all of the installed instances of a given GUID-named PPI. For example, there can be multiple instances of a PPI named *Foo* in the PPI database. An *Instance* value of 0 will provide the first instance of the PPI that is installed. Correspondingly, an *Instance* value of 2 will provide the second, 3 the third, and so on. The *Instance* value designates when a PPI was installed. For an implementation that must reference all possible manifestations of a given GUID-named PPI, the code should invoke **LocatePpi()** with a monotonically increasing *Instance* number until **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	The interface was successfully returned.
EFI_NOT_FOUND	The PPI descriptor is not found in the database.

NotifyPpi()

Summary

This function installs a notification service to be called back when a given interface is installed or reinstalled. The purpose of the service is to publish an interface that other parties can use to call additional PPIs that may materialize later.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_NOTIFY_PPI) (
  IN CONST EFI_PEI_SERVICES              **PeiServices,
  IN CONST EFI_PEI_NOTIFY_DESCRIPTOR     *NotifyList
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

NotifyList

A pointer to the list of notification interfaces that the caller shall install. Type **EFI_PEI_NOTIFY_DESCRIPTOR** is defined in "PEIM Descriptors" on page 113.

Description

This service enables PEIMs to register a given service to be invoked when another service is installed or reinstalled. This service will fire notifications on PPIs installed prior to this service invocation. This is different behavior than the RegisterProtocolNotify of UEFI2.0, for example **EFI_PEI_NOTIFY_DESCRIPTOR** is defined in "PEIM Descriptors" on page 113.

In addition, the PPI pointer is passed back to the agent that registered for the notification so that it can deference private data, if so needed.

Status Codes Returned

EFI_SUCCESS	The interface was successfully installed.
EFI_INVALID_PARAMETER	The *NotifyList* pointer is **NULL**.
EFI_INVALID_PARAMETER	Any of the PEI notify descriptors in the list do not have the **EFI_PEI_PPI_DESCRIPTOR_NOTIFY_TYPES** bit set in the *Flags* field.
EFI_OUT_OF_RESOURCES	There is no additional space in the PPI database.

4.3 Boot Mode Services

These services provide abstraction for ascertaining and updating the boot mode:

- GetBootMode()
- SetBootMode()

See “Boot Paths” on page 219 for additional information on the boot mode.

GetBootMode()

Summary

This function returns the present value of the boot mode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_BOOT_MODE) (
  IN CONST EFI_PEI_SERVICES    **PeiServices,
  OUT EFI_BOOT_MODE            *BootMode
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

BootMode

A pointer to contain the value of the boot mode. Type **EFI_BOOT_MODE** is defined in “Related Definitions” below.

Description

This service enables PEIMs to ascertain the present value of the boot mode. The list of possible boot modes is described in “Related Definitions” below.

Related Definitions

```
//*************************************************************
// EFI_BOOT_MODE
//*************************************************************
typedef UINT32     EFI_BOOT_MODE;

#define BOOT_WITH_FULL_CONFIGURATION                       0x00
#define BOOT_WITH_MINIMAL_CONFIGURATION                    0x01
#define BOOT_ASSUMING_NO_CONFIGURATION_CHANGES             0x02
#define BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS      0x03
#define BOOT_WITH_DEFAULT_SETTINGS                         0x04
#define BOOT_ON_S4_RESUME                                  0x05
#define BOOT_ON_S5_RESUME                                  0x06
#define BOOT_WITH_MFG_MODE_SETTINGS                        0x07
#define BOOT_ON_S2_RESUME                                  0x10
```

```
#define BOOT_ON_S3_RESUME                                       0x11
#define BOOT_ON_FLASH_UPDATE                                    0x12
#define BOOT_IN_RECOVERY_MODE                                   0x20
0x21 – 0xF..F Reserved Encodings
```

Table 6 describes the bit values in the Boot Mode Register.

Table 6. Boot Mode Register

Register Bits	Values	Descriptions
MSBit-0	000000b	Boot with full configuration
	000001b	Boot with minimal configuration
	000010b	Boot assuming no configuration changes from last boot
	000011b	Boot with full configuration plus diagnostics
	000100b	Boot with default settings
	000101b	Boot on S4 resume
	000110b	Boot in S5 resume
	000111b	Boot with manufacturing mode settings
	000111b-001111b	Reserved for boot paths that configure memory
	010000b	Boot on S2 resume
	010001b	Boot on S3 resume
	010010b	Boot on flash update restart
	010011c-011111b	Reserved for boot paths that preserve memory context
	100000b	Boot in recovery mode
	100001b-111111b	Reserved for special boots

Status Codes Returned

EFI_SUCCESS	The boot mode was returned successfully.

SetBootMode()

Summary

This function sets the value of the boot mode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SET_BOOT_MODE) (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_BOOT_MODE              BootMode
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

BootMode

The value of the boot mode to set. Type **EFI_BOOT_MODE** is defined in **GetBootMode()**.

Description

This service enables PEIMs to update the boot mode variable. This would be used by either the boot mode PPIs described in "Architectural PPIs" on page 89 or by a PEIM that needs to engender a recovery condition. It is permissible to change the boot mode at any point during the PEI phase.

Status Codes Returned

EFI_SUCCESS	The value was successfully updated.

4.4 HOB Services

The following services describe the capabilities in the PEI Foundation for providing Hand-Off Block (HOB) manipulation:

- GetHobList()
- CreateHob()

The purpose of the abstraction is to automate the common case of HOB creation and manipulation. See the *Volume 3* for details on HOBs and their type definitions.

GetHobList()

Summary

This function returns the pointer to the list of Hand-Off Blocks (HOBs) in memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_HOB_LIST) (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN OUT VOID                   **HobList
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

HobList

A pointer to the list of HOBs that the PEI Foundation will initialize.

Description

This service enables a PEIM to ascertain the address of the list of HOBs in memory. This service should not be required by many modules in that the creation of HOBs is provided by the PEI Service **CreateHob()**.

Status Codes Returned

EFI_SUCCESS	The list was successfully returned.
EFI_NOT_AVAILABLE_YET	The HOB list is not yet published.

CreateHob()

Summary

This service published by the PEI Foundation abstracts the creation of a Hand-Off Block's (HOB's) headers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CREATE_HOB) (
  IN CONST EFI_PEI_SERVICES       **PeiServices,
  IN UINT16                       Type,
  IN UINT16                       Length,
  IN OUT VOID                     **Hob
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

Type

The type of HOB to be installed. See the *Volume 3* for a definition of this type.

Length

The length of the HOB to be added.

Hob

The address of a pointer that will contain the HOB header.

Description

This service enables PEIMs to create various types of HOBs. This service handles the common work of allocating memory on the HOB list, filling in the type and length fields, and building the end of the HOB list. The final aspect of this service is to return a pointer to the newly allocated HOB. At this point, the caller can fill in the type-specific data. This service is always available because the HOBs can also be created on temporary memory.

There will be no error checking on the *Length* input argument. Instead, the PI Architecture implementation of this service will round up the allocation size that is specified in the *Length* field to be a multiple of 8 bytes in length. This rounding is consistent with the requirement that all of the HOBs, including the PHIT HOB, begin on an 8-byte boundary. See the PHIT HOB definition in the *Platform Initialization Specification,* Volume 3, for more information.

Status Codes Returned

EFI_SUCCESS	The HOB was successfully created.
EFI_OUT_OF_RESOURCES	There is no additional space for HOB creation.

4.5 Firmware Volume Services

The following services abstract traversing the Firmware File System (FFS):

- FfsFindNextVolume()
- FfsFindNextFile()
- FfsFindSectionData()
- FfsFindFileByName()
- FfsGetFileInfo()
- FfsGetVolumeInfo()

The description of the FFS can be found in the *Platform Initialization Specification*, Volume 3.

FfsFindNextVolume()

Summary

The purpose of the service is to abstract the capability of the PEI Foundation to discover instances of firmware volumes in the system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_FIND_NEXT_VOLUME2) (
  IN CONST EFI_PEI_SERVICES          **PeiServices,
  IN UINTN                           Instance,
  OUT EFI_PEI_FV_HANDLE              *VolumeHandle
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

Instance

This instance of the firmware volume to find. The value 0 is the Boot Firmware Volume (BFV).

VolumeHandle

On exit, points to the next volume handle or **NULL** if it does not exist.

Description

This service enables PEIMs to discover additional firmware volumes. The core uses **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** to discover these volumes. The service returns a volume handle of type **EFI_PEI_FV_HANDLE**, which must be unique within the system.

Related Definitions

```
typedef VOID *EFI_PEI_FV_HANDLE;
```

Status Codes Returned

EFI_SUCCESS	The volume was found.
EFI_NOT_FOUND	The volume was not found.
EFI_INVALID_PARAMETER	*VolumeHandle* is **NULL**

FfsFindNextFile()

Summary

Searches for the next matching file in the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_FIND_NEXT_FILE2) (
  IN CONST EFI_PEI_SERVICES       **PeiServices,
  IN EFI_FV_FILETYPE              SearchType,
  IN CONST EFI_PEI_FV_HANDLE      FvHandle,
  IN OUT EFI_PEI_FILE_HANDLE      *FileHandle
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

SearchType

A filter to find files only of this type. Type **EFI_FV_FILETYPE** is defined in the *Platform Initialization Specification,* Volume 3. Type **EFI_FV_FILETYPE_ALL** causes no filtering to be done.

FvHandle

Handle of firmware volume in which to search. The type **EFI_PEI_FV_HANDLE** is defined in the PEI Services **FfsFindNextVolume()**.

FileHandle

On entry, points to the current handle from which to begin searching or NULL to start at the beginning of the firmware volume. On exit, points the file handle of the next file in the volume or NULL if there are no more files. The type **EFI_PEI_FILE_HANDLE** is defined in “Related Defintions” below.

Description

This service enables PEIMs to discover firmware files within a specified volume. To find the first instance of a firmware file, pass a *FileHandle* value of **NULL** into the service.

The service returns a file handle of type **EFI_PEI_FILE_HANDLE**, which must be unique within the system.

The behavior of files with file types **EFI_FV_FILETYPE_FFS_MIN** and **EFI_FV_FILETYPE_FFS_MAX** depends on the firmware file system. For more information on the specific behavior for the standard PI firmware file system, see section 1.1.4.1.6 of the PI Specification, Volume 3.

Related Definitions

```
typedef VOID *EFI_PEI_FILE_HANDLE;
```

Status Codes Returned

EFI_SUCCESS	The file was found.
EFI_NOT_FOUND	The file was not found.
EFI_NOT_FOUND	The header checksum was not zero.

FfsFindSectionData()

Summary

Searches for the next matching section within the specified file. Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_FIND_SECTION_DATA2) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN EFI_SECTION_TYPE            SectionType,
  IN EFI_PEI_FILE_HANDLE         FileHandle,
  OUT VOID                       **SectionData
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

SectionType

The value of the section type to find. Type **EFI_SECTION_TYPE** is defined in the *Platform Initialization Specification,* Volume 3.

FileHandle

Handle of the firmware file to search. Type **EFI_PEI_FILE_HANDLE** is defined in **FfsFindNextFile()**, “Related Definitions.” A pointer to the file header that contains the set of sections to be searched.

SectionData

A pointer to the discovered section, if successful.

Description

This service enables PEI modules to discover the first section of a given type within a valid file. This service will search within encapsulation sections (compression and GUIDed) as well. It will search inside of a GUIDed section or a compressed section, but may not, for example, search a GUIDed section inside a GUIDes section.

This service will not search within compression sections or GUIDed sections which require extraction if memory is not present.

Status Codes Returned

EFI_SUCCESS	The section was found.
EFI_NOT_FOUND	The section was not found.

FfsFindSectionData3()

Summary

Searches for the next matching section within the specified file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_FIND_SECTION_DATA3) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN EFI_SECTION_TYPE            SectionType,
  In UINTN                       SectionInstance
  IN EFI_PEI_FILE_HANDLE         FileHandle,
  OUT VOID                       **SectionData
  OUT UINT32                     *AuthenticationStatus
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

SectionType

The value of the section type to find. Type **EFI_SECTION_TYPE** is defined in the *Platform Initialization Specification,* Volume 3.

SectionInstance

Section instance to find.

FileHandle

Handle of the firmware file to search. Type **EFI_PEI_FILE_HANDLE** is defined in **FfsFindNextFile()**, "Related Definitions." A pointer to the file header that contains the set of sections to be searched.

SectionData

A pointer to the discovered section, if successful.

AuthenticationStatus

A pointer to the authentication status for this section.

Description

This service enables PEI modules to discover the section of a given type within a valid file. This service will search within encapsulation sections (compression and GUIDed) as well. It will search inside of a GUIDed section or a compressed section, but may not, for example, search a GUIDed section inside a GUIDes section.

This service will not search within compression sections or GUIDed sections which require extraction if memory is not present.

Status Codes Returned

EFI_SUCCESS	The section was found.
EFI_NOT_FOUND	The section was not found.

FfsFindFileByName()

Summary

Find a file within a volume by its name.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_FIND_BY_NAME) (
  IN  CONST EFI_GUID            *FileName,
  IN  EFI_PEI_FV_HANDLE         VolumeHandle,
  OUT EFI_PEI_FILE_HANDLE       *FileHandle
  );
```

Parameters

FileName

A pointer to the name of the file to find within the firmware volume.

VolumeHandle

The firmware volume to search

FileHandle

Upon exit, points to the found file's handle or **NULL** if it could not be found.

Description

This service searches for files with a specific name, within either the specified firmware volume or all firmware volumes.

The service returns a file handle of type **EFI_PEI_FILE_HANDLE**, which must be unique within the system.

The behavior of files with file types **EFI_FV_FILETYPE_FFS_MIN** and **EFI_FV_FILETYPE_FFS_MAX** depends on the firmware file system. For more information on the specific behavior for the standard PI firmware file system, see section 1.1.4.1.6 of the PI Specification, Volume 3.

Status Codes Returned

EFI_SUCCESS	File was found.
EFI_NOT_FOUND	File was not found.
EFI_INVALID_PARAMETER	*VolumeHandle* or *FileHandle* or *FileName* was **NULL**.

FfsGetFileInfo()

Summary

Returns information about a specific file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_GET_FILE_INFO) (
  IN EFI_PEI_FILE_HANDLE  FileHandle,
  OUT EFI_FV_FILE_INFO    *FileInfo
  );
```

Parameters

FileHandle

Handle of the file.

FileInfo

Upon exit, points to the file's information.

Description

This function returns information about a specific file, including its file name, type, attributes, starting address and size. If the firmware volume is not memory mapped then the *Buffer* member will be NULL.

Related Definitions

```
typedef struct {
  EFI_GUID                 FileName;
  EFI_FV_FILETYPE          FileType;
  EFI_FV_FILE_ATTRIBUTES   FileAttributes;
  VOID                     *Buffer;
  UINT32                   BufferSize;
 } EFI_FV_FILE_INFO;
```

FileName

Name of the file.

FileType

File type. See **EFI_FV_FILETYPE**, which is defined in the *Platform Initialization Firmware Storage Specification*.

FileAttributes

Attributes of the file. Type **EFI_FV_FILE_ATTRIBUTES** is defined in the *Platform Initialization Firmware Storage Specification*.

Buffer

Points to the file's data (not the header). Not valid if **EFI_FV_FILE_ATTRIB_MEMORY_MAPPED** is zero.

BufferSize

Size of the file's data.

Status Codes Returned

EFI_SUCCESS	File information returned.
EFI_INVALID_PARAMETER	If *FileHandle* does not represent a valid file.
EFI_INVALID_PARAMETER	If *FileInfo* is NULL

FfsGetFileInfo2()

Summary

Returns information about a specific file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_GET_FILE_INFO2) (
IN EFI_PEI_FILE_HANDLE     FileHandle,
OUT EFI_FV_FILE_INFO2      *FileInfo
);
```

Parameters

FileHandle

Handle of the file.

FileInfo

Upon exit, points to the file's information.

Description

This function returns information about a specific file, including its file name, type, attributes, starting address, size and authentication status. If the firmware volume is not memory mapped then the *Buffer* member will be NULL.

Related Definitions

```
typedef struct {
  EFI_GUID FileName;
  EFI_FV_FILETYPE         FileType;
  EFI_FV_FILE_ATTRIBUTES  FileAttributes;
  VOID                    *Buffer;
  UINT32                   BufferSize;
  UINT32                   AuthenticationStatus;
} EFI_FV_FILE_INFO2;
```

FileName

Name of the file.

FileType

File type. See **EFI_FV_FILETYPE**, which is defined in the *Platform Initialization Firmware Storage Specification*.

FileAttributes

Attributes of the file. Type **EFI_FV_FILE_ATTRIBUTES** is defined in the P*latform Initialization Firmware Storage Specification*.

Buffer

Points to the file's data (not the header). Not valid if **EFI_FV_FILE_ATTRIB_MEMORY_MAPPED** is zero.

BufferSize

Size of the file's data.

AuthenticationStatus

Authentication status for this file.

Status Codes Returned

EFI_SUCCESS	File information returned.
EFI_INVALID_PARAMETER	If *FileHandle* does not represent a valid file.
EFI_INVALID_PARAMETER	If *FileInfo* is NULL

FfsGetVolumeInfo()

Summary

Returns information about the specified volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FFS_GET_VOLUME_INFO) (
  IN EFI_PEI_FV_HANDLE   VolumeHandle,
  OUT EFI_FV_INFO        *VolumeInfo
  );
```

Parameters

VolumeHandle

Handle of the volume.

VolumeInfo

Upon exit, points to the volume's information.

Related Definitions

```
typedef struct {
  EFI_FVB_ATTRIBUTES_2    FvAttributes;
  EFI_GUID                FvFormat;
  EFI_GUID                FvName;
  VOID*                   FvStart;
  UINT64                  FvSize;
} EFI_FV_INFO;
```

FvAttributes

Attributes of the firmware volume. Type **EFI_FVB_ATTRIBUTES_2** is defined in the *Platform Initialization Firmware Storage Specficiation*.

FvFormat

Format of the firmware volume. For PI Architecture Firmware Volumes, this can be copied from *FileSystemGuid* in **EFI_FIRMWARE_VOLUME_HEADER**.

FvName

Name of the firmware volume. For PI Architecture Firmware Volumes, this can be copied from *VolumeName* in the extended header of **EFI_FIRMWARE_VOLUME_HEADER**.

FvStart

Points to the first byte of the firmware volume, if bit **EFI_FVB_MEMORY_MAPPED** is set in *FvAttributes*.

FvSize

Size of the firmware volume.

Description

This function returns information about a specific firmware volume, including its name, type, attributes, starting address and size.

Status Codes Returned

EFI_SUCCESS	Volume information returned.
EFI_INVALID_PARAMETER	If *VolumeHandle* does not represent a valid volume.
EFI_INVALID_PARAMETER	If *VolumeInfo* is **NULL**.
EFI_SUCCESS	Information successfully returned
EFI_INVALID_PARAMETER	The volume designated by the VolumeHandle is not available

RegisterForShadow()

Summary

Register a PEIM so that it will be shadowed and called again.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_REGISTER_FOR_SHADOW) (
  IN  EFI_PEI_FILE_HANDLE            FileHandle
  );
```

Parameters

FileHandle

PEIM's file handle. Must be the currently executing PEIM.

Description

This service registers a file handle so that after memory is available, the PEIM will be re-loaded into permanent memory and re-initialized. The PEIM registered this way will always be initialized twice. The first time, this function call will return **EFI_SUCCESS**. The second time, this function call will return **EFI_ALREADY_STARTED**.

Depending on the order in which PEIMs are dispatched, the PEIM making this call may be initialized after permanent memory is installed, even the first time.

Status Codes Returned

EFI_SUCCESS	The PEIM was successfully registered for shadowing.
EFI_ALREADY_STARTED	The PEIM was previously registered for shadowing.
EFI_NOT_FOUND	The *FileHandle* does not refer to a valid file handle.

4.6 PEI Memory Services

The following services are a collection of memory management services for use both before and after permanent memory has been discovered:

- InstallPeiMemory()
- AllocatePages()
- AllocatePool()
- CopyMem()
- SetMem()

InstallPeiMemory()

Summary

This function registers the found memory configuration with the PEI Foundation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_INSTALL_PEI_MEMORY) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN EFI_PHYSICAL_ADDRESS        MemoryBegin,
  IN UINT64                      MemoryLength
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

MemoryBegin

The value of a region of installed memory.

MemoryLength

The corresponding length of a region of installed memory.

Description

This service enables PEIMs to register the permanent memory configuration that has been initialized with the PEI Foundation. The result of this call-set is the creation of the appropriate Hand-Off Block (HOB) describing the physical memory.

The usage model is that the PEIM that discovers the permanent memory shall invoke this service. The memory reported is a single contiguous run. It should be enough to allocate a PEI stack and some HOB list. The full memory map will be reported using the appropriate memory HOBs. The PEI Foundation will follow up with an installation of **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI**.

Any invocations of this service after the first invocation which returns EFI_SUCCESS will be ignored.

Status Codes Returned

EFI_SUCCESS	The region was successfully installed in a HOB or this service was successfully invoked earlier and no HOB modification will occur.
EFI_INVALID_PARAMETER	*MemoryBegin* and *MemoryLength* are illegal for this system.
EFI_OUT_OF_RESOURCES	There is no additional space for HOB creation.

AllocatePages()

Summary

The purpose of the service is to publish an interface that allows PEIMs to allocate memory ranges that are managed by the PEI Foundation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ALLOCATE_PAGES) (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_MEMORY_TYPE            MemoryType,
  IN UINTN                      Pages,
  OUT EFI_PHYSICAL_ADDRESS      *Memory,
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

MemoryType

The type of memory to allocate. The only types allowed are **EfiLoaderCode**, **EfiLoaderData**, **EfiRuntimeServicesCode**, **EfiRuntimeServicesData**, **EfiBootServicesCode**, **EfiBootServicesData**, **EfiACPIReclaimMemory,** **EfiReservedMemoryType**, and **EfiACPIMemoryNVS**.

Pages

The number of contiguous 4 KiB pages to allocate. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Memory

Pointer to a physical address. On output, the address is set to the base of the page range that was allocated.

Description

This service enables PEIMs to allocate memory after the permanent memory has been installed by a PEIM. The purpose of this service is to allow more state-ful, later PEIMs to have a single set of memory allocation services upon which to rely. This is especially of interest for services like the recovery PEIMs that might have to allocate large buffers for disk transactions and file system metadata. The memory regions that the memory allocation primitives manage will be described in the appropriate HOB type from the *Platform Initialization Specification, Volume 3.*

This service is not usable prior to the installation of main memory. There is no free memory.

The expectation is that the implementation of this service will automate the creation of the Memory Allocation HOB types. As such, this is in the same spirit as the PEI Services to create the FV HOB, for example.
As opposed to the UEFI memory allocation service, there is no allocate "type" field; this field dictates location information in UEFI (i.e., allocate below a given address, at a given address, etc). Instead, PEI will allocate pages within the region of memory provided by InstallPeiMemory() service in a best-effort fashion. Location-specific allocations are not managed by the PEI foundation code.

The service also supports the creation of Memory Allocation HOBs that describe the stack, boot-strap processor (BSP) BSPStore ("Backing Store Pointer Store"), and the DXE Foundation allocation. This additional information is conveyed through the final two arguments in this API and the description of the appropriate HOB types can be found in the *Platform Initialization Specification*, Volume 3.

Status Codes Returned

EFI_SUCCESS	The memory range was successfully allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	Type is not equal to **EfiLoaderCode**, **EfiLoaderData**, **EfiRuntimeServicesCode**, **EfiRuntimeServicesData**, **EfiBootServicesCode**, **EfiBootServicesData**, **EfiACPIReclaimMemory**, **EfiReservedMemoryType,** or **EfiACPIMemoryNVS**.

AllocatePool()

Summary

The purpose of this service is to publish an interface that allows PEIMs to allocate memory ranges that are managed by the PEI Foundation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ALLOCATE_POOL) (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN UINTN                      Size,
  OUT VOID                      **Buffer
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

Size

The number of bytes to allocate from the pool.

Buffer

If the call succeeds, a pointer to a pointer to the allocated buffer; undefined otherwise.

Description

This service allocates memory from the Hand-Off Block (HOB) heap. Because HOBs can be allocated from either temporary or permanent memory, this service is available throughout the entire PEI phase.

This service allocates memory in multiples of eight bytes to maintain the required HOB alignment. The early allocations from temporary memory will be migrated to permanent memory when permanent main memory is installed; this migration shall occur when the HOB list is migrated to permanent memory.

Status Codes Returned

EFI_SUCCESS	The allocation was successful.
EFI_OUT_OF_RESOURCES	There is not enough heap to allocate the requested size.

CopyMem()

Summary

This service copies the contents of one buffer to another buffer.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_COPY_MEM) (
  IN VOID                               *Destination,
  IN VOID                               *Source,
  IN UINTN                              Length
  );
```

Parameters

Destination

Pointer to the destination buffer of the memory copy.

Source

Pointer to the source buffer of the memory copy.

Length

Number of bytes to copy from *Source* to *Destination*.

Description

This function copies *Length* bytes from the buffer *Source* to the buffer *Destination*.

Status Codes Returned

None.

SetMem()

Summary

The service fills a buffer with a specified value.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_SET_MEM) (
  IN VOID                                *Buffer,
  IN UINTN                               Size,
  IN UINT8                               Value
  );
```

Parameters

Buffer

Pointer to the buffer to fill.

Size

Number of bytes in *Buffer* to fill.

Value

Value to fill *Buffer* with.

Description

This function fills *Size* bytes of *Buffer* with *Value*.

Status Codes Returned

None.

4.7 Status Code Service

The PEI Foundation publishes the following status code service:

- ReportStatusCode()

This service will report **EFI_NOT_AVAILABLE_YET** until a PEIM publishes the services for other modules. For the GUID of the PPI, see **EFI_PEI_PROGRESS_CODE_PPI**.

ReportStatusCode()

Summary

This service publishes an interface that allows PEIMs to report status codes.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_REPORT_STATUS_CODE) (
  IN CONST EFI_PEI_SERVICES        **PeiServices,
  IN EFI_STATUS_CODE_TYPE          Type,
  IN EFI_STATUS_CODE_VALUE         Value,
  IN UINT32                        Instance,
  IN CONST EFI_GUID                *CallerId  OPTIONAL,
  IN CONST EFI_STATUS_CODE_DATA    *Data      OPTIONAL
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

Type

Indicates the type of status code being reported. The type **EFI_STATUS_CODE_TYPE** is defined in “Related Definitions” below.

Value

Describes the current status of a hardware or software entity. This includes information about the class and subclass that is used to classify the entity as well as an operation. For progress codes, the operation is the current activity. For error codes, it is the exception. For debug codes, it is not defined at this time. Type **EFI_STATUS_CODE_VALUE** is defined in “Related Definitions” below.

Instance

The enumeration of a hardware or software entity within the system. A system may contain multiple entities that match a class/subclass pairing. The instance differentiates between them. An instance of 0 indicates that instance information is unavailable, not meaningful, or not relevant. Valid instance numbers start with 1.

CallerId

This optional parameter may be used to identify the caller. This parameter allows the status code driver to apply different rules to different callers.

Data

This optional parameter may be used to pass additional data. Type **EFI_STATUS_CODE_DATA** is defined in “Related Definitions“ below. The contents of this data type may have additional GUID-specific data.

Description

ReportStatusCode() is called by PEIMs that wish to report status information on their progress. The principal use model is for a PEIM to emit one of the standard 32-bit error codes. This will allow a platform owner to ascertain the state of the system, especially under conditions where the full consoles might not have been installed.

This is the entry point that PEIMs shall use. This service can use all platform PEI Services, and when main memory is available, it can even construct a GUIDed HOB that conveys the pre-DXE data. This service can also publish an interface that is usable only from the DXE phase. This entry point should not be the same as that published to the PEIMs, and the implementation of this code path should *not* do the following:

- Use any PEI Services or PPIs from other modules.
- Make any presumptions about global memory allocation.

It can only operate on its local stack activation frame and must be careful about using I/O and memory-mapped I/O resources. These concerns, including the latter warning, arise because this service could be used during the "blackout" period between the termination of PEI and the beginning of DXE, prior to the loading of the DXE progress code driver. As such, the ownership of the memory map and platform resource allocation is indeterminate at this point in the platform evolution.

Related Definitions

```
//
// Status Code Type Definition
//
typedef UINT32 EFI_STATUS_CODE_TYPE;

//
// A Status Code Type is made up of the code type and severity
// All values masked by EFI_STATUS_CODE_RESERVED_MASK are
// reserved for use by this specification.
//
#define EFI_STATUS_CODE_TYPE_MASK       0x000000FF
#define EFI_STATUS_CODE_SEVERITY_MASK   0xFF000000
#define EFI_STATUS_CODE_RESERVED_MASK   0x00FFFF00

//
// Definition of code types, all other values masked by
// EFI_STATUS_CODE_TYPE_MASK are reserved for use by
// this specification.
//
#define EFI_PROGRESS_CODE               0x00000001
#define EFI_ERROR_CODE                  0x00000002
#define EFI_DEBUG_CODE                  0x00000003

//
// Definitions of severities, all other values masked by
// EFI_STATUS_CODE_SEVERITY_MASK are reserved for use by
// this specification.
// Uncontained errors are major errors that could not contained
// to the specific component that is reporting the error
// For example, if a memory error was not detected early enough,
// the bad data could be consumed by other drivers.
//
#define EFI_ERROR_MINOR                         0x40000000
#define EFI_ERROR_MAJOR                         0x80000000
#define EFI_ERROR_UNRECOVERED                   0x90000000
#define EFI_ERROR_UNCONTAINED                   0xa0000000

//
// Status Code Value Definition
//
typedef UINT32 EFI_STATUS_CODE_VALUE;

//
// A Status Code Value is made up of the class, subclass, and
// an operation.
//
#define EFI_STATUS_CODE_CLASS_MASK      0xFF000000
#define EFI_STATUS_CODE_SUBCLASS_MASK   0x00FF0000
#define EFI_STATUS_CODE_OPERATION_MASK  0x0000FFFF

//
// Definition of Status Code extended data header.
// The data will follow HeaderSize bytes from the beginning of
// the structure and is Size bytes long.
//
typedef struct {
  UINT16    HeaderSize;
  UINT16    Size;
  EFI_GUID  Type;
} EFI_STATUS_CODE_DATA;
```

HeaderSize

The size of the structure. This is specified to enable future expansion.

Size

The size of the data in bytes. This does not include the size of the header structure.

Type

The GUID defining the type of the data.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_AVAILABLE_YET	No progress code provider has installed an interface in the system.

4.8 Reset Services

The PEI Foundation publishes the following reset service:

- ResetSystem()

ResetSystem()

Summary

Resets the entire platform.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_RESET_SYSTEM) (
  IN CONST EFI_PEI_SERVICES      **PeiServices
  );
```

Parameters

PeiServices

An indirect pointer to the **EFI_PEI_SERVICES** table published by the PEI Foundation.

Description

This service resets the entire platform, including all processors and devices, and reboots the system. It is important to have a standard variant of this function for cases such as the following:

- Resetting the processor to change frequency settings
- Restarting hardware to complete chipset initialization
- Responding to exceptions from a catastrophic errorReturned Status Codes

Status Codes Returned

EFI_NOT_AVAILABLE_YET	The service has not been installed yet.

4.9 I/O and PCI Services

- The PEI Foundation publishes CPU I/O and PCI Configuration services.

5
PEI Foundation

5.1 Introduction

The PEI Foundation centers around the PEI Dispatcher. The dispatcher's job is to hand control to the PEIMs in an orderly manner. The PEI Foundation also assists in PEIM-to-PEIM communication. The central resource for the module-to-module communication involves the PPI. The marshalling of references to PPIs can occur using the installable or notification interface.

The PEI Foundation is constructed as an autonomous binary image that is of file type **`EFI_FV_FILETYPE_PEI_CORE`** and is composed of the following:

- An authentication section
- A code image that is possibly PE32+

See the *Platform Initialization Specification,* Volume 3*,* for information on section and file types. If the code that comprises the PEI Foundation is not a PE32+ image, then it is a raw binary whose lowest address is the entry point to the PEI Foundation. The PEI Foundation is discovered and authenticated by the Security (SEC) phase.

5.1.1 Prerequisites

The PEI phase is handed control from the Security (SEC) phase of the PI Architecture-compliant boot process. The PEI phase must satisfy the following minimum prerequisites before it can begin execution:

- Processor execution mode
- Access to the Boot Firmware Volume (BFV) that contains the PEI Foundation

It is expected that the SEC infrastructure code and PEI Foundation are not linked together as a single ROMable executable image. The entry point from SEC into PEI is not architecturally fixed but is instead dependent on the PEI Foundation location within FV0, or the Boot Firmware Volume.

5.1.2 Processor Execution Mode

5.1.2.1 Processor Execution Mode in IA-32 Intel® Architecture

In IA-32 Intel architecture, the Security (SEC) phase of the PI Architecture is responsible for placing the processor in a native linear address mode by which the full address range of the processor is accessible for code, data, and stack. For example, "flat 32" is the IA-32 processor generation mode in which the PEI phase will execute. The processor must be in its most privileged "ring 0" mode, or equivalent, and be able to access all memory and I/O space.

This prerequisite is strictly dependent on the processor generation architecture.

5.1.2.2 Processor Execution Mode in Itanium® Processor Family

The PEI Foundation will begin executing after the Security (SEC) phase has completed. The SEC phase subsumed the System Abstraction Layer entry point (SALE_ENTRY) in Itanium® architecture. In addition, the SEC phase makes the appropriate Processor Abstraction Layer (PAL) calls or platform services to enable the temporary memory store. The SEC passes its handoff state to the PEI Foundation in physical mode with some configured memory stack, such as the processor cache configured as memory.

5.1.2.3 Access to the Boot Firmware Volume

The program that the Security (SEC) phase hands control to is known as the PEI Foundation. The firmware volume (FV) in which the PEI Foundation resides is known as the Boot Firmware Volume (BFV). PEIMs may reside in the BFV or other FVs. A “special” PEIM must be resident in the BFV to provide information about the location of the other FVs.

Each file contained in the BFV that is required to boot must be able to be discovered and validated by the PEI phase. This allows the PEI phase to determine if the FV has been corrupted.

The PEI Foundation and the PEIMs are expected to be stored in some reasonably tamper-proof (albeit not necessarily in the strict security-based definition of the term) nonvolatile storage (NVS). The storage is expected to be fairly analogous to a flat file system with the unique IDs substituting for names. Rules for using the particular NVS might affect certain storage considerations, but a standard data-only mechanism for locating PEIMs by ID is required. The PI Architecture architecture describes the PI Firmware Volume format and PI Firmware File System format, with the GUID convention of naming files. These standards are architectural for PEI inasmuch as the PEI phase needs to directly support this file system.

The BFV can only be constructed of type **`EFI_FIRMWARE_FILE_SYSTEM2_GUID`**.

The PEI Foundation and some PEIMs required for recovery must be either locked into a nonupdateable BFV or must be able to be updated via a “fault-tolerant” mechanism. The fault-tolerant mechanism is designed such that, if the system halts at any point, either the old (preupdate) PEIM or the newly updated PEIM is entirely valid and that the PEI phase can determine which is valid.

5.1.2.4 Access to the Boot Firmware Volume in IA-32 Intel Architecture

In IA-32 Intel architecture, the Security (SEC) file is at the top of the Boot Firmware Volume (BFV). This SEC file will have the 16-byte entry point for IA-32 and restarts at address 0xFFFFFFF0.

5.1.2.5 Access to the Boot Firmware Volume in Itanium Processor Family

In the Itanium processor family, the microcode starts up the Processor Abstraction Layer A (PAL-A) code, which is the first layer of PAL code and is provided by the processor vendor, that resides in the Boot Firmware Volume (BFV). This code minimally initializes the processor and then finds and authenticates the second layer of PAL code, called PAL-B. The location of both PAL-A and PAL-B can be found by consulting either of the following:

- The architected pointers in the ROM (near the 4 GiB region)
- The Firmware Interface Table (FIT) pointer in the ROM

The PAL layer communicates with the OEM boot firmware using a single entry point called the System Abstraction Layer entry point (SALE_ENTRY). The PEI Foundation will be located at the SALE_ENTRY point on the boot firmware device for an Itanium-based system. The Itanium processor family PEIMs, like other PEIMs, may reside in the BFV or other firmware volumes. A "special" PEIM must be resident in the BFV to provide information about the location of the other firmware volumes; this will be described in the context of the **EFI_PEI_FIND_FV_PPI** description. It must also be noted that in an Itanium-based system, all the processors in each node start up and execute the PAL code and subsequently enter the PEI Foundation. The BFV of a particular node must be accessible by all the processors running in that node. This also means that some of the PEIMs in the Itanium® architecture boot path will be multiprocessor (MP) aware.

In an Itanium-based system, it is also imperative that the organization of firmware modules in the BFV must be such that at least the PAL-A is contained in the fault-tolerant regions. This processor-specific PAL-A code authenticates the PAL-B code, which is usually contained in the non-fault-tolerant regions of the firmware system. The PAL-A and PAL-B binary components are always visible to all the processors in a node at the time of power-on; the system fabric should not need to be initialized.

5.2 PEI Foundation Entry Point

5.2.1 PEI Foundation Entry Point

The Security (SEC) phase calls the entry point to the PEI Foundation with the following information:

- A set of PPIs
- Size and location of the Boot Firmware Volume (BFV)
- Size and location of the temporary RAM
- Size and location of the temporary RAM available for use by the PEI Foundation
- Size and location of the stack

The entry point is described in "Code Definitions" below.

Prototype

```
typedef
VOID
EFIAPI
(*EFI_PEI_CORE_ENTRY_POINT)(
  IN CONST EFI_SEC_PEI_HAND_OFF   *SecCoreData,
  IN CONST EFI_PEI_PPI_DESCRIPTOR *PpiList
  );
```

Parameters

SecCoreData

Points to a data structure containing information about the PEI core's operating environment, such as the size and location of temporary RAM, the stack location and the BFV location. The type **EFI_SEC_PEI_HAND_OFF** is defined in "Related Definitions" below.

PpiList

Points to a list of one or more PPI descriptors to be installed initially by the PEI core. An empty PPI list consists of a single descriptor with the end-tag **EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST**. As part of its initialization phase, the PEI Foundation will add these SEC-hosted PPIs to its PPI database such that both the PEI Foundation and any modules can leverage the associated service calls and/or code in these early PPIs.

Description

This function is the entry point for the PEI Foundation, which allows the SEC phase to pass information about the stack, temporary RAM and the Boot Firmware Volume. In addition, it also allows the SEC phase to pass services and data forward for use during the PEI phase in the form of one or more PPIs. These PPI's will be installed and/or immediately signaled if they are notification type.

There is no limit to the number of additional PPIs that can be passed from SEC into the PEI Foundation. As part of its initialization phase, the PEI Foundation will add these SEC-hosted PPIs to its PPI database such that both the PEI Foundation and any modules can leverage the associated service calls and/or code in these early PPIs.

Finally, later phases of platform evolution might need many of the features and data that the SEC phase might possibly have. To support this, the SEC phase can construct a **EFI_PEI_PPI_DESCRIPTOR** and pass its address into the PEI Foundation as the final argument.

Among these PPIs, the SEC can pass an optional PPI, **EFI_SEC_PLATFORM_INFORMATION_PPI**, as part of the PPI list that is passed to the PEI Foundation entry point. This PPI abstracts platform-specific information that the PEI Foundation needs to discover where to begin dispatching PEIMs. Other possible values to pass into the PEI Foundation would include any security or verification services, such as the Trusted Computing Group (TCG) access services, because the SEC would constitute the Core Root-of-Trust Module (CRTM) in a TCG-conformant system.

Further, SEC can pass the **EFI_SEC_HOB_DATA_PPI** as a part of the PPI list. This PPI can retrieve zero or more HOBs to be added to the HOB list before any PEIMs are dispatched.

Related Definitions

```
typedef struct _EFI_SEC_PEI_HAND_OFF {
  UINT16   DataSize;
  VOID     *BootFirmwareVolumeBase;
  UINTN    BootFirmwareVolumeSize;
  VOID     *TemporaryRamBase;
  UINTN    TemporaryRamSize;
  VOID     *PeiTemporaryRamBase;
  UINTN    PeiTemporaryRamSize;
  VOID     *StackBase;
  UINTN    StackSize;
} EFI_SEC_PEI_HAND_OFF;
```

DataSize

Size of the data structure.

BootFirmwareVolumeBase

Points to the first byte of the boot firmware volume, which the PEI Dispatcher should search for PEI modules.

BootFirmwareVolumeSize

Size of the boot firmware volume, in bytes.

TemporaryRamBase

Points to the first byte of the temporary RAM.

TemporaryRamSize

Size of the temporary RAM, in bytes.

PeiTemporaryRamBase

Points to the first byte of the temporary RAM available for use by the PEI Foundation. The area described by *PeiTemporaryRamBase* and *PeiTemporaryRamSize* must not extend outside beyond the area described by *TemporaryRamBase* & *TemporaryRamSize*. This area should not overlap with the area reported by *StackBase* and *StackSize*.

PeiTemporaryRamSize

Size of the available temporary RAM available for use by the PEI Foundation, in bytes.

StackBase

Points to the first byte of the stack. This are may be part of the memory described by *TemporaryRamBase* and *TemporaryRamSize* or may be an entirely separate area.

StackSize

Size of the stack, in bytes.

The information from SEC is mandatory information that is placed on the stack by the SEC phase to invoke the PEI Foundation.

The SEC phase provides the required processor and/or platform initialization such that there is a temporary RAM region available to the PEI phase. This temporary RAM could be a particular configuration of the processor cache, SRAM, or other source. What is important with respect to this handoff is that the PEI ascertain the available amount of cache as RAM from this data structure.

Similarly, the PEI Foundation needs to receive *a priori* information about where to commence the dispatch of PEIMs. A platform can have various size BFVs. As such, the *BootFirmwareVolume* value tells the PEI Foundation where it can expect to discover a firmware volume header data structure, and it is this firmware volume that contains the PEIMs necessary to perform the basic system initialization.

5.3 PEI Calling Convention Processor Binding

Unless otherwise specified, the calling convention used for PEI functions is the same as the one specified in the UEFI specification. However, for certain processors, an alternate calling convention is recommended for new PPI definitions.

5.4 PEI Services Table Retrieval

This section describes processor-specific mechanisms for retrieving a pointer to a pointer to the PEI Services Table (**EFI_PEI_SERVICES****) such as is commonly used in PEIMs. The means of storage and retrieval are processor specific.

5.4.1 X86

For X86 processors, the **EFI_PEI_SERVICES**** is stored in the 4 bytes immediately preceding the Interrupt Descriptor Table.

The **EFI_PEI_SERVICES**** can be retrieved with the following code fragment, which should be placed in a library routine for portability between architectures:

```
IDTR32          STRUCT
Limit           DW 1 DUP (?)
BaseAddress     DD 1 DUP (?)
IDTR32          ENDS

sub       esp, SIZEOF IDTR32
sidt      FWORD PTR ss:[esp]
mov       eax, [esp].IDTR32.BaseAddress
mov       eax, DWORD PTR [eax – 4]
add       esp, SIZEOF IDTR32
```

5.4.1.1 Interrupt Descriptor Table Initialization and Ownership Rules.

1. The SEC Core must initialize the IDT using the lidt command and ensure that the four-bytes field immediately preceding the IDT base address resides within temporary memory.
2. The PEI Foundation initializes or updates the four-byte field immediately preceding the currently loaded IDT base address.

3. Any PEIM can reinitialize the IDT with the following restrictions:
    - The four-bytes field immediately prior to new IDT base address must reside within the temporary or permanent memory.
    - The four-byte field immediately preceding the old IDT base address must be copied to the four-byte field immediately preceding the new IDT base address.

5.4.2 x64

For x64 processors, the **EFI_PEI_SERVICES**** is stored in eight bytes immediately preceding the Interrupt Descriptor Table

The **EFI_PEI_SERVICES**** can be retrieved with the following code fragment, which should be placed in a library routine for portability between architectures:

```
IDTR64          STRUCT
Limit           DW 1 DUP (?)
BaseAddress     DQ 1 DUP (?)
IDTR64          ENDS

sub       rsp, SIZEOF IDTR64
sidt      [rsp]
mov       rax, [rsp].IDTR64.BaseAddress
mov       rax, QWORD PTR [rax – 8]
add       rsp, SIZEOF IDTR64
```

5.4.2.1 Interrupt Descriptor Table Initialization and Ownership Rules.

1. The SEC Core must initialize the IDT using the lidt command and ensure that the eight-bytes field immediately preceding the IDT base address resides within temporary memory.
2. The PEI initializes or updates the eight-byte field immediately preceding the currently loaded IDT base address.
3. Any PEIM can reinitialize the IDT with the following restrictions:
    - The eight-bytes field immediately prior to new IDT base address must reside within the temporary or permanent memory
    - The eight-byte field immediately preceding the old IDT base address must be copied to the eight-byte field immediately preceding the new IDT base address.

5.4.3 Itanium Processor Family – Register Mechanism

For Itanium Processor Family processors, the **EFI_PEI_SERVICES**** is stored in kernel register 7 (ar.kr7). Information on the kernel registers for IPF can be found at http://www.intel.com/design/itanium/downloads/245358.htm.

The **EFI_PEI_SERVICES**** can be retrieved with the following code fragment, which may be placed in a library routine for portability between architectures:

```
AsmReadKr7
    mov    r8, ar.kr7;;
    br.ret b0;;
```

```
EFI_PEI_SERVICES **
GetPeiServicesTablePointer (
  VOID
  )
{
  return (EFI_PEI_SERVICES **)(UINTN)AsmReadKr7 ();
}
```

Note: *Compilers should not be using KRs, they are reserved for OS use (i.e.,this is the overlap w/ the Software Development Manual). Also, priv. level 3 code can only read KRs and not write them anyway, only PL0 code can write these.*

5.4.4 ARM Processor Family – Register Mechanism

For the ARM Processor Family processors, the **EFI_PEI_SERVICES**** is stored in a the TPIDRURW read/write Software Thread ID register defined in the *ARMv7-A Architectural Reference Manual*.

The **EFI_PEI_SERVICES**** can be retrieved with the following code fragment, which may be placed in a library routine for portability between architectures:

```
CpuReadTPIDRURW:
    MRC p15, 0, r0, c13, c0, 2
    bx lr


EFI_PEI_SERVICES **
GetPeiServicesTablePointer (
  VOID
   )
{
  return (EFI_PEI_SERVICES **)(UINTN)CpuReadTPIDRURW ();
}
```

5.4.4.1 ARM Vector Table

For ARM processors the vector table entries are instructions, and thus are limited to 24-bit relative offset of a branch instruction. The PI specification requires that the 8 defined vectors contain the following instruction LDR pc, [pc, #0x20]. This means the 32-bit address of the handler is contained at a 32-byte offset from the address of the vector. When PI code hooks into the vector table it must ensure that the 32-bit absolute address offset 32-bytes from the vector is what is updated. The first code in the platform that initializes the vector table must fill it with 8 LDR pc, [pc, #0x20] instructions.

5.4.5 AArch64 Processor Family – Register Mechanism

For AArch64 architecture processors, the **EFI_PEI_SERVICES**** is stored in the TPIDR_EL0 register. Information on this register can be found in the "*ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*".

5.5 PEI Dispatcher Introduction

The PEI Dispatcher's job is to hand control to the PEIMs in an orderly manner. The PEI Dispatcher consists of a single phase. It is during this phase that the PEI Foundation will examine each file in the firmware volumes that contain files of type **EFI_FV_FILETYPE_PEIM** or **EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER** (see the *Platform Initialization Specification*, Volume 3, for file type definitions). It will examine the dependency expression (depex) and the optional *a priori* file within each firmware file to decide when a PEIM is eligible to be dispatched. The binary encoding of the depex will be the same as that of a depex associated with a PEIM.

5.6 Ordering

5.6.1 Requirements

Except for the order imposed by an *a priori* file, it is not reasonable to expect PEIMs to be executed in any order. A chipset initialization PEIM usually requires processor initialization and a memory initialization PEIM usually requires chipset initialization. On the other hand, the PEIMs that satisfy these requirements might have been authored by different organizations and might reside in different FVs. The requirement is thus to, without memory, create a mechanism to allow for the definition of ordering among the different PEIMs so that, by the time a PEIM executes, all of the requirements for it to execute have been met.

Although the update and build processes assist in resolving ordering issues, they cannot be relied upon completely. Consider a system with a removable processor card containing a processor and firmware volume that plugs into a main system board. If the processor card is upgraded, it is entirely reasonable that the user should expect the system to work even though no update program was executed.

5.6.2 Requirement Representation and Notation

Requirements are represented by GUIDs, with each GUID representing a particular requirement. The requirements are represented by two sets of data structures:

- The dependency expression (depex) of a given PEIM
- The installed set of PPIs maintained by the PEI Foundation in the PPI database

This mechanism provides for a "weak ordering" among PEIMs. If PEIMs A and B consume X (written AcX and BcX), once a PEIM (C) that produces X (CpX) is executed, A and B can be executed. There is no definition about the order in which A and B are executed.

5.6.3 PEI *a priori* File Overview

The PEI *a priori* file is a special file that may optionally be present in a firmware volume, and its main purpose is to provide a greater degree of flexibility in the firmware design of a platform. Specifically, the *a priori* file complements the dependency expression mechanism of PEI by stipulating a series of modules which need be dispatched in a prescribed order.

There may be at most one PEI *a priori* file per firmware volume present in a platform. The *a priori* file has a known GUID file name **PEI_APRIORI_FILE_NAME_GUID**, enabling the PEI Foundation dispatch behavior to find the *a priori* file if it is present. The contents of the file shall contain data of the format **PEI_APRIORI_FILE_CONTENTS**, with possibly zero entries. Every time the PEI Dispatcher discovers a firmware volume, it first looks for the *a priori* file. The PEIM's enumerated in a an *a priori* file must exist in the same firmware volume as the *a priori* file iteself; no cross-volume mapping is allowed. The PEI Foundation will invoke the PEIM's listed in the **PEI_APRIORI_FILE_CONTENTS** in the order found in this file.

Without the *a priori* file, PEIMs executed solely because of their dependency expressions are weakly ordered. This means that the execution order is not completely deterministic between boots or between platforms. In some cases a deterministic execution order is required. The PEI a priori file provides a deterministic execution order of PEIMs using the following two implementation methods.

The *a priori* model must be supported by all PEI Foundation implementations, but it does not preclude additional *a priori* dispatch methodologies, as long as the latter models use a different mechanism and/or file name GUID for the alternate *a priori* module listing. The *a priori* file format follows below.

PEI_APRIORI_FILE_NAME_GUID

Summary

The GUID **PEI_APRIORI_FILE_NAME_GUID** definition is the file name of the PEI *a priori* file that is stored in a firmware volume.

GUID

```
#define PEI_APRIORI_FILE_NAME_GUID \
  {0x1b45cc0a,0x156a,0x428a,0xaf62,0x49,0x86,\
  0x4d,0xa0,0xe6,0xe6}

typedef struct {
EFI_GUID  FileNamesWithinVolume[NumberOfModulesInVolume];
       // Optional list of file-names
} PEI_APRIORI_FILE_CONTENTS;
```

Parameters

FileNamesWithinVolume[]

An array of zero or more EFI_GUID type entries that match the file names of PEIM modules in the same Firmware Volume. The maximum number of entries *NumberOfModulesInVolume* is determined by the number of modules in the FV.

Description

This file must be of type **EFI_FV_FILETYPE_FREEFORM** and must contain a single section of type **EFI_SECTION_RAW**. For details on firmware volumes, firmware file types, and firmware file section types, see the *Platform Initialization Specification*, Volume 3.

5.6.3.1 Dispatch Behavior

The *a priori* file can contain a list of the EFI_GUIDs, which are the names of the PEIM files within the same firmware volume. Herein, the PEI Foundation dispatch logic reads the list of names from the *a priori* file and invokes the appropriately named module in the order enumerated in the *a priori* file. This value can be calculated by means of the size of **PEI_APRIORI_FILE_CONTENTS**. This shall be an integral number of GUID sizes.

If there is a file name within **PEI_APRIORI_FILE_CONTENTS** which is in the deleted state or does not exist, the specific file name shall be ignored by the PEI Foundation dispatch logic and the successive entry invoked.

During dispatch of PEIM's in the *a priori* file, any PEIMs in newly published firmware volumes will be ignored until completion of the *a priori* file dispatch. These interfaces would be assessed during subsequent module dispatch, though.

In addition to ignoring any additional volumes published during *a priori* dispatch, any dependency expressions associated with PEIMs listed within **PEI_APRIORI_FILE_CONTENTS** are ignored.

During dispatch of the *a priori* PEIM list, the PEI Dispatcher shall invoke the **EFI_PEI_SECURITY2_PPI AuthenticationState** service, if it exists, to qualify the dispatch of each module. This is the same behavior as the normal dependency-based dispatch. For the *a priori* file in the boot firmware volume, for example, the **EFI_PEI_SECURITY2_PPI** could be passed by the SEC into the PEI Foundation via the optional **EFI_PEI_PPI_DESCRIPTOR** list. This latter scenario allows authentication of PEIMs in the *a priori* file.

After executing all of the PEIMs specified in the *a priori* file, the PEI Dispatcher searches the firmware volume for any additional PEIMs and executes them according to their dependency expressions.

5.6.4 Firmware Volume Image Files

For PEI, while processing a firmware volume, if a file of type **EFI_FV_FIRMWARE_VOLUME_IMAGE** is found, the PEI Dispatcher will check whether this firmware volume image file was already processed. If it was, then the file is ignored.

Otherwise, the PEI Dispatcher will search the file for a section with the type **EFI_SECTION_PEI_DEPEX**, and if found, evaluate the expression against the presently installed entries in the PPI database. If the file has a dependency expression that evaluates to TRUE (or no **EFI_SECTION_PEI_DEPEX** section), then the PEI Dispatcher will search the file for a section with the type **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**, copy its contents into memory, and install the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** and **EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI** for the firmware volume image, and add HOBs of type **EFI_HOB_FIRMWARE_VOLUME** and **EFI_HOB_FIRMWARE_VOLUME2** to the hob list for the firmware volume image.

5.6.5 PEIM Dependency Expressions

The sequencing of PEIMs is determined by evaluating a *dependency expression* associated with each PEIM. This expression describes the requirements necessary for that PEIM to run, which imposes a weak ordering on the PEIMs. Within this weak ordering, the PEIMs may be initialized in any order.

5.6.6 Types of Dependencies

The base unit of the dependency expression is a dependency. A representative syntax (used in this document for descriptive purposes) for each dependency is shown in the following section. The syntax is case-insensitive and mnemonics are used in place of non-human-readable data such as GUIDs. White space is optional.

The operands are GUIDs of PPIs. The operand becomes “true” when a PPI with the GUID is registered.

5.7 Dependency Expressions

5.7.1 Introduction

A PEIM is stored in a firmware volume as a file with one or more sections. One of the sections must be a PE32+ image. If a PEIM has a dependency expression, then it is stored in a dependency section. A PEIM may contain additional sections for compression and security wrappers. The PEI Dispatcher can identify the PEIMs by their file type. In addition, the PEI Dispatcher can look up the dependency expression for a PEIM by looking for a dependency section in a PEIM file. The dependency section contains a section header followed by the actual dependency expression that is composed of a packed byte stream of opcodes and operands.

Dependency expressions stored in dependency sections are designed to meet the following goals:

- Be small to conserve space.
- Be simple and quick to evaluate to reduce execution overhead.

These two goals are met by designing a small, stack-based instruction set to encode the dependency expressions. The PEI Dispatcher must implement an interpreter for this instruction set to evaluate dependency expressions. The instruction set is defined in the following topics.

See “Dependency Expression Grammar” on page 245 for an example BNF grammar for a dependency expression compiler. There are many possible methods of specifying the dependency expression for a PEIM. This example grammar demonstrates one possible design for a tool that can be used to help build PEIM images.

5.7.1.1 Dependency Expression Instruction Set

The following topics describe each of the dependency expression (depex) opcodes in detail. Information includes a description of the instruction functionality, binary encoding, and any limitations or unique behaviors of the instruction.

Several of the opcodes require a GUID operand. The GUID operand is a 16-byte value that matches the type **EFI_GUID** that is described in Chapter 2 of the UEFI 2.0 specification. These GUIDs represent PPIs that are produced by PEIMs and the file names of PEIMs stored in firmware volumes. A dependency expression is a packed byte stream of opcodes and operands. As a result, some of the GUID operands will not be aligned on natural boundaries. Care must be taken on processor architectures that do allow unaligned accesses.

The dependency expression is stored in a packed byte stream using postfix notation. As a dependency expression is evaluated, the operands are pushed onto a stack. Operands are popped off the stack to perform an operation. After the last operation is performed, the value on the top of the stack represents the evaluation of the entire dependency expression. If a push operation causes a stack overflow, then the entire dependency expression evaluates to **FALSE**. If a pop operation causes a stack underflow, then the entire dependency expression evaluates to **FALSE**. Reasonable implementations of a dependency expression evaluator should not make arbitrary assumptions about the maximum stack size it will support. Instead, it should be designed to grow the dependency expression stack as required. In addition, PEIMs that contain dependency expressions should make an effort to keep their dependency expressions as small as possible to help reduce the size of the PEIM.

All opcodes are 8-bit values, and if an invalid opcode is encountered, then the entire dependency expression evaluates to **FALSE**.

If an END opcode is not present in a dependency expression, then the entire dependency expression evaluates to **FALSE**.

The final evaluation of the dependency expression results in either a **TRUE** or **FALSE** result.

Note: *NoteThe PEI Foundation will only support the evaluation of dependency expressions that are less than or equal to 256 terms.*

Table 7 is a summary of the opcodes that are used to build dependency expressions. The following sections describe each of these instructions in detail.

Table 7. Dependency Expression Opcode Summary

Opcode	Description
0x02	PUSH <PPI GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END

PUSH

Syntax

```
PUSH <PPI GUID>
```

Description

Pushes a Boolean value onto the stack. If the GUID is present in the handle database, then a **TRUE** is pushed onto the stack. If the GUID is not present in the handle database, then a **FALSE** is pushed onto the stack. The test for the GUID in the handle database may be performed with the Boot Service **LocatePpi()**.

Operation

```
Status = (*PeiServices)->LocatePpi (PeiServices, GUID, 0, NULL,
&Interface);
if (EFI_ERROR (Status)) {
  PUSH FALSE;
} Else {
  PUSH TRUE;
}
```

The following table defines the **PUSH** instruction encoding.

Table 8. PUSH Instruction Encoding

Byte	Description
0	0x02
1..16	A 16-byte GUID that represents a protocol that is produced by a different PEIM. The format is the same at type **EFI_GUID**.

Behaviors and Restrictions

None.

AND

Syntax

```
AND
```

Description

Pops two Boolean operands off the stack, performs a Boolean **AND** operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 AND Operand2
PUSH Result
```

Table 9 defines the **AND** instruction encoding.

Table 9. AND Instruction Encoding

Byte	Description
0	0x03

Behaviors and Restrictions

None.

OR

Syntax

```
OR
```

Description

Pops two Boolean operands off the stack, performs a Boolean **OR** operation between the two operands, and pushes the result back onto the stack.

Operation

```
Operand1 <= POP Boolean stack element
Operand2 <= POP Boolean stack element
Result <= Operand1 OR Operand2
PUSH Result
```

Table 10 defines the **OR** instruction encoding.

Table 10. OR Instruction Encoding

Byte	Description
0	0x04

Behaviors and Restrictions

None.

NOT

Syntax

```
NOT
```

Description

Pops a Boolean operands off the stack, performs a Boolean **NOT** operation on the operand, and pushes the result back onto the stack.

Operation

```
Operand <= POP Boolean stack element
Result <= NOT Operand
PUSH Result
```

Table 11 defines the **NOT** instruction encoding.

Table 11. NOT Instruction Encoding

Byte	Description
0	0x05

Behaviors and Restrictions

None.

TRUE

Syntax

```
TRUE
```

Description

Pushes a Boolean **TRUE** onto the stack.

Operation

```
PUSH TRUE
```

Table 12 defines the **TRUE** instruction encoding.

Table 12. TRUE Instruction Encoding

Byte	Description
0	0x06

Behaviors and Restrictions

None.

FALSE

Syntax

```
FALSE
```

Description

Pushes a Boolean **FALSE** onto the stack.

Operation

```
PUSH FALSE
```

Table 13 defines the **FALSE** instruction encoding.

Table 13. FALSE Instruction Encoding

Byte	Description
0	0x07

Behaviors and Restrictions

None.

END

Syntax

```
END
```

Description

Pops the final result of the dependency expression evaluation off the stack and exits the dependency expression evaluator.

Operation

```
POP Result
RETURN Result
```

Table 14 defines the **END** instruction encoding.

Table 14. END Instruction Encoding

Byte	Description
0	0x08

Behaviors and Restrictions

This opcode must be the last one in a dependency expression.

5.7.2 Dependency Expression with No Dependencies

A PEIM that does not have any dependencies will have a dependency expression that evaluates to **TRUE** with no dependencies on any PPI GUIDs.

5.7.3 Empty Dependency Expressions

If a PEIM file does not contain a dependency section, then the PEIM has an empty dependency expression.

5.7.4 Dependency Expression Reverse Polish Notation (RPN)

The actual equations will be presented by the PEIM in a simple-to-evaluate form, namely postfix.

The following is a BNF encoding of this grammar. See "Dependency Expression Instruction Set" on page 73 for definitions of the dependency expressions.

```
<statement> ::= <expression> END

<expression> ::= PUSH <guid> |
                 TRUE |
                 FALSE |
                 <expression> NOT |
                 <expression> <expression> OR |
                    <expression> <expression> AND
```

5.8 Dispatch Algorithm

5.8.1 Overview

5.8.1.1 Ordering Algorithm

The dispatch algorithm repeatedly scans through the PEIMs to find those that have not been dispatched. For each PEIM that is found, it scans through the PPI database of PPIs that have been published, searching for elements in the yet-to-be-dispatched PEIM's depex. If all of the elements in the depex are in the PEI Foundation's PPI database, the PEIM is dispatched. The phase terminates when all PEIMs are scanned and none dispatched.

Note: *The PEIM may be dispatched without a search if its depex is NULL.*

5.8.1.2 Multiple Firmware Volume Support

In order to expose a new firmware volume, a PEIM should install an instance of **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** containing the firmware volume format GUID, the starting address and the size of the firmware volume's window. PEIMs exposing firmware volumes which have a firmware volume format other than the PI Architecture Firmware Volume format should include the firmware volume format GUID in their dependency expression.

PEIMs exposing memory-mapped firmware volumes should create a memory resource descriptor HOB for the memory occupied by the firmware volume if it is outside of the PEI memory.

For each new exposed firmware volume, the PEI Foundation will take the following steps:

1. Create a new firmware volume handle. The firmware volume handle may be created by the PEI Foundation or by the optional **EFI_PEI_FIRMWARE_VOLUME_PPI.**
2. Create a new firmware volume HOB.
3. If the firmware volume's format (identified by its GUID) is not supported directly by the PEI Foundation and it is not supported by any installed **EFI_PEI_FIRMWARE_VOLUME_PPI**, the firmware volume is skipped.
4. Otherwise, all PEIMs in the firmware volume are scheduled for dispatching.
5. Find the *a priori* file, if it exists, and dispatch any PEIMs listed in it.

5.8.2 Requirements

5.8.2.1 Requirements of a Dispatching Algorithm

The dispatching algorithm must meet the following requirements:

1. Preserve the dispatch weak ordering.
2. Prevent an infinite loop.
3. Control processor resources.
4. Preserve proper dispatch order.
5. Make use of available memory.
6. Invoke each PEIM's entry point.
7. Know when the PEI Dispatcher tasks are finished.

5.8.2.2 Preserving Weak Ordering

The algorithm must preserve the weak ordering implied by the depex.

5.8.2.3 Preventing Infinite Loops

It is illegal for AcXpY (A consumes X and produces Y) and BcYpX. This is known as a cycle and is unresolvable even if memory is available. At a minimum, the dispatching algorithm must not end up in an infinite loop in such a scenario. With the algorithm described above, neither PEIM would be executed.

5.8.2.4 Controlling Processor Register Resources

The algorithm must require that a minimum of the processor's register resources be preserved while PEIMs are dispatched.

5.8.2.5 Preserving Proper Dispatch Order

The algorithm must preserve proper dispatch order in cases such as the following:

```
AcQpZ BcLpR CpL DcRpQ
```

The issue with the above scenario is that A and B are not obviously related until D is processed. If A and B were in one firmware volume and C and D were in another, the ordering could not be resolved until execution. The proper dispatch order in this case is CBDA. The algorithm must resolve this type of case.

5.8.2.6 Using Available Memory

The PEI Foundation begins operation using a temporary memory store that contains the initial call stack from the Security (SEC) phase. The SEC phase must pass the size and location of the stack and the size and location of the temporary memory store.

The PEI stack will be available for subsequent PEIM invocations, and the PEI heap will be used for PEIM memory allocations and Hand-Off Block (HOB) creation.

There can be no memory writes to the address space beyond this initial temporary memory until a PEIM registers a permanent memory range using the PEI Service **InstallPeiMemory()**. When permanent memory is installed, the PEI Foundation will copy the call stack that is located in temporary memory into a segment of permanent memory. If necessary, the size of the call stack can be expanded to to support the subsequent transition into DXE.

In addition to the call stack, the PEI Foundation will copy the following from temporary to permanent memory:

- PEI Foundation private data
- PEI Foundation heap
- HOB list

Any permanent memory consumed in this fashion by the PEI Foundation will be described in a HOB, which the PEI Foundation will create.

In addition, if there were any **EFI_PEI_PPI_DESCRIPTOR**s created in the temporary memory heap, their respective locations have been translated by an offset equal to the difference between the original heap location in temporary memory and the destination location in permanent memory. In addition to this heap copy, the PEI Foundation will traverse the PEI PPI database. Any references to **EFI_PEI_PPI_DESCRIPTOR**s that are in temporary memory will be fixed up by the PEI Foundation to reflect the location of the **EFI_PEI_PPI_DESCRIPTOR**s destination in permanent memory.

The PEI Foundation will invoke the DXE IPL PPI after dispatching all candidate PEIMs. The DXE IPL PPI may have to allocate additional regions from permanent memory to be able to load and relocate the DXE Foundation from its firmware store. The DXE IPL PPI will describe these memory allocations in the appropriate HOB such that when control is passed to DXE, an accurate record of the memory usage will be known to the DXE Foundation.

5.8.2.7 Invoking the PEIM's Entry Point

The entry point of a PEIM uses the calling conventions specified in the UEFI 2.0 specification, which detail how parameters are passed to a function. After assessing a PEIM's dependency expression to see if it can be invoked, the PEI Foundation will pass control to the PEIM's entry point. This entry point is a value described in the PEIM's image header.

The PEI Foundation will pass an indirect pointer to the PEI Services Table and the handle of the firmware file when it invokes the PEIM.

In the entry point of the PEIM, the PEIM has the opportunity do the following:

- Locate other PPIs
- Install PPIs that reference services within the body of this PEIM
- Register for a notification

- Upon return from the PEIM's entry point, it returns back to the PEI Foundation.
- See the *Microsoft Portable Executable and Common Object File Format Specification* for information on PE/COFF images; see "TE Image" on page 247 for information on TE images.

5.8.2.8 Knowing When Dispatcher Tasks Are Finished

The PEI Dispatcher is finished with a pass when it has finished dispatching all the PEIMs that it can. During a pass, some PEIMs might not have been dispatched if they had requirements that no other PEIM has met.

However, with the weak ordering defined in previous requirements, system RAM could possibly be initialized before all PEIMs are given a chance to run. This situation can occur because the system RAM initialization PEIM is not required to consume all resources provided by all other PEIMs. The PEI Dispatcher must recognize that its tasks are not complete until all PEIMs have been given an opportunity to run.

5.8.2.9 Reporting PEI Core Location

If the **EFI_PEI_LOADED_IMAGE_PPI** is supported by the PEI Dispatcher, then the PEI Foundation must first report its own location by using the PEI Service **InstallPpi()** and the **EFI_PEI_LOADED_IMAGE_PPI**. If the *FileHandle* is unknown, then **NULL** can be used. PEI Foundation must also report the location of the PEIM loaded by creating the **EFI_PEI_LOADED_IMAGE_PPI** and call the PEI Service **ReinstallPpi()**.

5.8.3 Example Dispatch Algorithm

The following pseudo code is an example of an algorithm that uses few registers and implements the requirements listed in the previous section. The pseudo code uses simple C-like statements but more assembly-like flow-of-control primitives.

The dispatch algorithm's main data structure is the DispatchedBitMap as described in Table 15.

Table 15. Example Dispatch Map

PEIM#	Item	PEIM#	Item
	FV0	4	FV1
	PEI Foundation		<non PEIM>
	<non PEIM>		<non PEIM>
0	PEIM		<non PEIM>
1	PEIM	5	PEIM
2	PEIM with **EFI_PEI_FIRMWARE_VOLUME_PPI**		<non PEIM>
	<non PEIM>	6	PEIM
3	PEIM	7	PEIM

Table 15 is an example of a dispatch in a given set of firmware volumes (FVs). Following are the steps in this dispatch:

1. The algorithm scans through the PEIMs that it knows about.

2. When it comes to a PEIM that has not been dispatched, it verifies that all of the required PPIs listed in the dependency expression (depex) are in the PPI database.
3. If all of the GUIDed interfaces listed in the depex are available, the PEIM is invoked.
4. Create the **EFI_PEI_LOADED_IMAGE_PPI** and call the PEI Service **ReinstallPpi()**
5. Iterations continue through all known PEIMs in all known FVs until a pass is made with no PEIMs dispatched, thus signifying completion.
6. After the dispatch completes, the PEI Foundation locates and invokes the GUID for the DXE IPL PPI, passing in the HOB address and a valid stack. Failing to discover the GUID for the DXE IPL PPI shall be an error.

5.8.4 Dispatching When Memory Exists

The purpose of the PEI phase of execution is to discover and initialize main memory. As such, a large number of the modules execute from the nonvolatile firmware store and cannot be shadowed. However, there are several circumstances in which the shadowing of a PEIM and the relocation of this image into memory are of interest. This can include but is not limited to compressing PEIMs, such as the DXE IPL PPI, and those modules that are required for crisis recovery.

The PEI architecture shall not dictate what compression mechanism is to be used, but there will be a Decompress service that is published by some PEIM that the PEI Foundation will discover and use when it becomes available. In addition, loading images also requires a full image-relocation service and the ability to flush the cache. The former will allow the PEIM that was relocated into RAM to have its relocations adjust pursuant to the new load address. The latter service will be invoked by the PEI Foundation so that this relocated code can be run, especially on Itanium-based platforms that do not have a coherent data and code cache.

A compressed section shall have an implied dependency on permanent memory having been installed. To speed up boot time, however, there can be an explicit annotation of this dependency.

5.8.5 PEIM Dispatching

When the PEI Dispatcher has decided to invoke a PEIM, the following steps are taken:

1. If any instances of **EFI_PEI_LOAD_FILE_PPI** are installed, they are called, one at a time, until one reports **EFI_SUCCESS**.
2. If no instance reports **EFI_SUCCESS** or there are no instances installed, then the built-in support for (at least) the PE32+/TE XIP image formats is used.
3. If any instances of **EFI_PEI_SECURITY2_PPI** are installed, they are called, one at a time, as long as none returns an **EFI_SECURITY_VIOLATION** error. If such an error is returned, then the PEIM is marked as dispatched, but is never invoked.
4. The PEIM’s entry point is invoked with the file’s handle and the PEI Services Table pointer.
5. The PEIM is marked as dispatched.

The PEI Core may decide, because of memory constraints or performance reasons, to dispatch XIP instead of shadowing into memory.

5.8.6 PEIM Authentication

The PEI specification provides three methods which the PEI Foundation can use to authenticate a PEIM:

1. The authentication information could be encoded as part of a GUIDed section. In this case, the provider of the **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI** (see the *Platform Initialization Specification*, Volume 3) can check the authentication data and return the results in *AttestationState*.
2. The authentication information can be checked by the provider of the **EFI_PEI_LOAD_FILE_PPI** (see the *Platform Initialization Specification*, Volume 3) and the results returned in *AttestationState*.
3. The PEI Foundation may implement the digital signing as described in the UEFI 2.0 specification.

In all cases, the result of the authentication must be passed to any instances of the **EFI_PEI_SECURITY2_PPI**.

6
Architectual PPIs

6.1 Introduction

The PEI Foundation and PEI Dispatcher rely on the following PEIM-to-PEIM Interfaces (PPIs) to perform its work. The abstraction provided by these interfaces allows dispatcher algorithms to be improved over time or have some platform variability without affecting the rest of PEI.

The key to these PPIs is that they are architecturally defined interfaces consumed by the PEI Foundation, but they may not be published by the PEI Foundation.

6.2 Required Architectural PPIs

6.2.1 Master Boot Mode PPI (Required)

EFI_PEI_MASTER_BOOT_MODE_PPI (Required)

Summary

The Master Boot Mode PPI is installed by a PEIM to signal that a final boot has been determined and set. This signal is useful in that PEIMs with boot-mode-specific behavior (for example, S3 versus normal) can put this PPI in their dependency expression.

GUID

```
#define EFI_PEI_MASTER_BOOT_MODE_PEIM_PPI \
  {0x7408d748, 0xfc8c, 0x4ee6, 0x92, 0x88, 0xc4, 0xbe, \
  0xc0, 0x92, 0xa4, 0x10}
```

PPI Interface Structure

None.

Description

The Master Boot Mode PPI is a PPI GUID and must be in the dependency expression of every PEIM that modifies the basic hardware. The dispatch, or entry point, of the module that installs the Master Boot Mode PPI modifies the boot path value in the following ways:

- Directly, through the PEI Service **SetBootMode()**
- Indirectly through its optional subordinate boot path modules

The PEIM that publishes the Master Boot Mode PPI has a non-null dependency expression if there are subsidiary modules that publish alternate boot path PPIs. The primary reason for this PPI is to be the root of dependencies for any child boot mode provider PPIs.

Status Codes Returned

None.

6.2.2 DXE IPL PPI (Required)

EFI_DXE_IPL_PPI (Required)

Summary

Final service to be invoked by the PEI Foundation.

GUID

```
#define EFI_DXE_IPL_PPI_GUID \
  { 0xae8ce5d, 0xe448, 0x4437, 0xa8, 0xd7, 0xeb, 0xf5, \
  0xf1, 0x94, 0xf7, 0x31 }
```

PPI Interface Structure

```
typedef struct _EFI_DXE_IPL_PPI {
  EFI_DXE_IPL_ENTRY  Entry;
} EFI_DXE_IPL_PPI;
```

Parameters

Entry

The entry point to the DXE IPL PPI. See the **Entry()** function description.

Description

After completing the dispatch of all available PEIMs, the PEI Foundation will invoke this PPI through its entry point using the same handoff state used to invoke other PEIMs. This special treatment by the PEI Foundation effectively makes the DXE IPL PPI the last PPI to execute during PEI. When this PPI is invoked, the system state should be as follows:

- Single thread of execution
- Interrupts disabled
- Processor mode as defined for PEI

The DXE IPL PPI is responsible for locating and loading the DXE Foundation. The DXE IPL PPI may use PEI services to locate and load the DXE Foundation. As long as the DXE IPL PPI is using PEI Services, it must obey all PEI interoperability rules of memory allocation, HOB list usage, and PEIM-to-PEIM communication mechanisms.

For S3 resume boot modes DXE IPL must be prepared to execute without permanent memory installed and invoke the S3 resume modules.

EFI_DXE_IPL_PPI.Entry()

Summary

The architectural PPI that the PEI Foundation invokes when there are no additional PEIMs to invoke.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DXE_IPL_ENTRY) (
  IN CONST EFI_DXE_IPL_PPI    *This,
  IN EFI_PEI_SERVICES         **PeiServices,
  IN EFI_PEI_HOB_POINTERS     HobList
  );
```

Parameters

This

Pointer to the DXE IPL PPI instance.

PeiServices

Pointer to the PEI Services Table.

HobList

Pointer to the list of Hand-Off Block (HOB) entries.

Related Definitions

```
//
// Union of all the possible HOB Types
//
typedef union {
  EFI_HOB_GENERIC_HEADER              *Header;
  EFI_HOB_HANDOFF_INFO_TABLE          *HandoffInformationTable;
  EFI_HOB_MEMORY_ALLOCATION           *MemoryAllocation;
  EFI_HOB_MEMORY_ALLOCATION_BSP_STORE *MemoryAllocationBspStore;
  EFI_HOB_MEMORY_ALLOCATION_STACK     *MemoryAllocationStack;
  EFI_HOB_MEMORY_ALLOCATION_MODULE    *MemoryAllocationModule;
  EFI_HOB_RESOURCE_DESCRIPTOR         *ResourceDescriptor;
  EFI_HOB_GUID_TYPE                   *Guid;
  EFI_HOB_FIRMWARE_VOLUME             *FirmwareVolume;
  EFI_HOB_CPU                         *Cpu;
  EFI_HOB_MEMORY_POOL                 *Pool;
  UINT8                               *Raw;
} EFI_PEI_HOB_POINTERS;
```

Description

This function is invoked by the PEI Foundation. The PEI Foundation will invoke this service when there are no additional PEIMs to invoke in the system. If this PPI does not exist, it is an error condition and an ill-formed firmware set. The DXE IPL PPI should never return after having been invoked by the PEI Foundation. The DXE IPL PPI can do many things internally, including the following:

- Invoke the DXE entry point from a firmware volume.
- Invoke the recovery processing modules.
- Invoke the S3 resume modules.

Status Codes Returned

EFI_SUCCESS	Upon this return code, the PEI Foundation should enter some exception handling. Under normal circumstances, the DXE IPL PPI should not return.

6.2.3 Memory Discovered PPI (Required)

EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI (Required)

Summary

This PPI is published by the PEI Foundation when the main memory is installed. It is essentially a PPI with no associated interface. Its purpose is to be used as a signal for other PEIMs who can register for a notification on its installation.

GUID

```
#define EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI_GUID \
  {0xf894643d, 0xc449, 0x42d1, 0x8e, 0xa8, 0x85, 0xbd, \
  0xd8, 0xc6, 0x5b, 0xde}
```

PPI Interface Structure

None.

Description

This PPI is installed by the PEI Foundation at the point of system evolution when the permanent memory size has been registered and waiting PEIMs can use the main memory store. Using this GUID allows PEIMs to do the following:

- Be notified when this PPI is installed.
- Include this PPI's GUID in the **EFI_DEPEX**.

The expectation is that a compressed PEIM would depend on this PPI, for example. The PEI Foundation will relocate the temporary cache to permanent memory prior to this installation.

Status Codes Returned

None.

6.3 Optional Architectural PPIs

6.3.1 Boot in Recovery Mode PPI (Optional)

EFI_PEI_BOOT_IN_RECOVERY_MODE_PPI (Optional)

Summary

This PPI is installed by the platform PEIM to designate that a recovery boot is in progress.

GUID

```
#define EFI_PEI_BOOT_IN_RECOVERY_MODE_PEIM_PPI \
  {0x17ee496a, 0xd8e4, 0x4b9a, 0x94, 0xd1, 0xce, 0x82, \
  0x72, 0x30, 0x8, 0x50}
```

PPI Interface Structure

None.

Description

This optional PPI is installed by the platform PEIM to designate that a recovery boot is in progress. Its purpose is to allow certain PEIMs that wish to be dispatched **only during a recovery boot** to include this PPI in their dependency expression (depex). Including this PPI in the depex allows the PEI Dispatcher to skip recovery-specific PEIMs during normal restarts and thus save on boot time. This PEIM has no associated PPI and is used only to designate the system state as being “in a crisis recovery dispatch.”

Status Codes Returned

None.

6.3.2 End of PEI Phase PPI (Optional)

EFI_PEI_END_OF_PEI_PHASE_PPI (Optional)

Summary

This PPI will be installed at the end of PEI for all boot paths, including normal, recovery, and S3. It allows for PEIMs to possibly quiesce hardware, build handoff information for the next phase of execution, or provide some terminal processing behavior.

GUID

```
#define EFI_PEI_END_OF_PEI_PHASE_PPI_GUID \
  {0x605EA650, 0xC65C, 0x42e1, 0xBA, 0x80, 0x91, 0xA5, \
  0x2A, 0xB6,0x18, 0xC6}
```

PPI Interface Structure

None.

Description

This PPI is installed by the DXE IPL PPI to indicate the end of the PEI usage of memory and ownership of memory allocation by the DXE phase.

For the **BOOT_ON_S3_RESUME** boot mode, this PPI is installed by the **EFI_PEI_S3_RESUME_PPI.S3RestoreConfig()** (Section 8.6 of the PI1.2 Specification, Volume 5) just before jump to OS waking vector.

The intended use model is for any agent that needs to do cleanup, such as memory services to convert internal metadata for tracking memory allocation into HOBs, to have some distinguished point in which to do so. The PEI Memory Services would register for a callback on the installation of this PPI.

Status Codes Returned

None.

6.3.3 PEI Reset PPI

EFI_PEI_RESET_PPI (Optional)

Summary

This PPI is installed by some platform- or chipset-specific PEIM that abstracts the Reset Service to other agents.

GUID

```
#define EFI_PEI_RESET_PPI_GUID \
  {0xef398d58, 0x9dfd, 0x4103, 0xbf, 0x94, 0x78, 0xc6, \
  0xf4, 0xfe, 0x71, 0x2f}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RESET_PPI {
  EFI_PEI_RESET_SYSTEM  ResetSystem;
} EFI_PEI_RESET_PPI;
```

Parameters

ResetSystem

A service to reset the platform. See the **ResetSystem()** function description in "Reset Services" on page 59.

Description

These services provide a simple reset service. See the **ResetSystem()** function description for a description of this service.

6.3.4 PEI Reset2 PPI

EFI_PEI_RESET2_PPI (Optional)

Summary

This PPI is installed by some platform- or chipset-specific PEIM that abstracts the ability to reset the platform.

GUID

```
#define EFI_PEI_RESET2_PPI_GUID \
{0x6cc45765, 0xcce4, 0x42fd, \
 {0xbc, 0x56, 0x1, 0x1a,0xaa, 0xc6, 0xc9, 0xa8}}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RESET2_PPI {
EFI_PEI_RESET2_SYSTEM ResetSystem;
} EFI_PEI_RESET_PPI;
```

Parameters

ResetSystem

A service to reset the platform.

Description

These services provide a simple reset service. This is equivalent to the **ResetSystem()** API call in the UEFI2.4 specification.

ResetSystem()

Summary

Resets the entire platform.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_RESET2_SYSTEM) (
IN EFI_RESET_TYPE ResetType,
IN EFI_STATUS      ResetStatus,
IN UINTN           DataSize,
IN VOID           *ResetData OPTIONAL
);
```

Parameters.

ResetType

The type of reset to perform. Type **EFI_RESET_TYPE** is defined in "Related Definitions" below.

ResetStatus

The status code for the reset. If the system reset is part of a normal operation, the status code would be **EFI_SUCCESS**. If the system reset is due to some type of failure the most appropriate EFI Status code would be used.

DataSize

The size, in bytes, of *ResetData*.

ResetData

For a *ResetType* of **EfiResetCold**, **EfiResetWarm**, or **EfiResetShutdown** the data buffer starts with a Null-terminated string, optionally followed by additional binary data. The string is a description that the caller may use to further indicate the reason for the system reset. *ResetData* is only valid if *ResetStatus* is something other than **EFI_SUCCESS** unless the *ResetType* is *EfiResetPlatformSpecific* where a minimum amount of *ResetData* is always required.

Related Definitions

```
//*******************************************************
// EFI_RESET_TYPE   //
*******************************************************

typedef enum {
    EfiResetCold,
    EfiResetWarm,
    EfiResetShutdown,
    EfiResetPlatformSpecific
} EFI_RESET_TYPE;
```

Description

The **ResetSystem()** function resets the entire platform, including all processors and devices, and reboots the system.

Calling this interface with *ResetType* of **EfiResetCold** causes a system-wide reset. This sets all circuitry within the system to its initial state. This type of reset is asynchronous to system operation and operates without regard to cycle boundaries. **EfiResetCold** is tantamount to a system power cycle.

Calling this interface with *ResetType* of **EfiResetWarm** causes a system-wide initialization. The processors are set to their initial state, and pending cycles are not corrupted. If the system does not support this reset type, then an **EfiResetCold** must be performed.

Calling this interface with *ResetType* of **EfiResetShutdown** causes the system to enter a power state equivalent to the ACPI G2/S5 or G3 states. If the system does not support this reset type, then when the system is rebooted, it should exhibit the **EfiResetCold** attributes.

Calling this interface with *ResetType* of EfiResetPlatformSpecific causes a system-wide reset. The exact type of the reset is defined by the **EFI_GUID** that follows the Null-terminated Unicode string passed into *ResetData*. If the platform does not recognize the **EFI_GUID** in *ResetData* the platform must pick a supported reset type to perform.The platform may optionally log the parameters from any non-normal reset that occurs.

The **ResetSystem()** function does not return.

6.3.5 Status Code PPI (Optional)

EFI_PEI_PROGRESS_CODE_PPI (Optional)

Summary

This service is published by a PEIM. There can be only one instance of this service in the system. If there are multiple variable access services, this PEIM must multiplex these alternate accessors and provide this single, read-only service to the other PEIMs and the PEI Foundation. This singleton nature is important because the PEI Foundation will notify when this service is installed.

GUID

```
#define EFI_PEI_REPORT_PROGRESS_CODE_PPI_GUID  \
  {0x229832d3, 0x7a30, 0x4b36, 0xb8, 0x27, 0xf4, 0xc, \
  0xb7, 0xd4, 0x54, 0x36);
```

PPI Interface Structure

```
typedef struct _EFI_PEI_PROGRESS_CODE_PPI {
  EFI_PEI_REPORT_STATUS_CODE      ReportStatusCode;
} EFI_PEI_PROGRESS_CODE_PPI;
```

Parameters

ReportStatusCode

Service that allows PEIMs to report status codes. See the **ReportStatusCode()** function description in “Status Code Service” on page 54.

Description

See the **ReportStatusCode()** function description for a description of this service.

6.3.6 Security PPI (Optional)

EFI_PEI_SECURITY2_PPI (Optional)

Summary

This PPI is installed by some platform PEIM that abstracts the security policy to the PEI Foundation, namely the case of a PEIM's authentication state being returned during the PEI section extraction process.

GUID

```
#define EFI_PEI_SECURITY2_PPI_GUID \
  { 0xdcd0be23, 0x9586, 0x40f4, 0xb6, 0x43, 0x6, 0x52, \
  0x2c, 0xed, 0x4e, 0xde }
```

PPI Interface Structure

```
typedef struct _EFI_PEI_SECURITY2_PPI {
  EFI_PEI_SECURITY_AUTHENTICATION_STATE  AuthenticationState;
} EFI_PEI_SECURITY2_PPI;
```

Parameters

AuthenticationState

Allows the platform builder to implement a security policy in response to varying file authentication states. See the **AuthenticationState()** function description.

Description

This PPI is a means by which the platform builder can indicate a response to a PEIM's authentication state. This can be in the form of a requirement for the PEI Foundation to skip a module using the *DeferExecution* Boolean output in the **AuthenticationState()** member function. Alternately, the Security PPI can invoke something like a cryptographic PPI that hashes the PEIM contents to log attestations, for which the *FileHandle* parameter in **AuthenticationState()** will be useful. If this PPI does not exist, PEIMs will be considered trusted.

EFI_PEI_SECURITY2_PPI.AuthenticationState()

Summary

Allows the platform builder to implement a security policy in response to varying file authentication states.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SECURITY_AUTHENTICATION_STATE) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN CONST EFI_PEI_SECURITY2_PPI *This,
  IN UINT32                       AuthenticationStatus,
  IN EFI_PEI_FV_HANDLE            FvHandle,
  IN EFI_PEI_FILE_HANDLE          FileHandle,
  IN OUT BOOLEAN                  *DeferExecution
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Interface pointer that implements the particular **EFI_PEI_SECURITY2_PPI** instance.

AuthenticationStatus

Authentication status of the file.

FvHandle

Handle of the volume in which the file resides. Type **EFI_PEI_FV_HANDLE** is defined in **FfsFindNextVolume**. This allows different policies depending on different firmware volumes.

FileHandle

Handle of the file under review. Type **EFI_PEI FILE HANDLE** is defined in **FfsFindNextFile**.

DeferExecution

Pointer to a variable that alerts the PEI Foundation to defer execution of a PEIM.

Description

This service is published by some platform PEIM. The purpose of this service is to expose a given platform's policy-based response to the PEI Foundation. For example, if there is a PEIM in a GUIDed encapsulation section and the extraction of the PEI file section yields an authentication failure, there is no *a priori* policy in the PEI Foundation. Specifically, this situation leads to the question whether PEIMs that are either not in GUIDed sections or are in sections whose authentication fails should still be executed.

In fact, it is the responsibility of the platform builder to make this decision. This platform-scoped policy is a result that a desktop system might not be able to skip or not execute PEIMs because the skipped PEIM could be the agent that initializes main memory. Alternately, a system may require that unsigned PEIMs not be executed under any circumstances. In either case, the PEI Foundation simply multiplexes access to the Section Extraction PPI and the Security PPI. The Section Extraction PPI determines the contents of a section, and the Security PPI tells the PEI Foundation whether or not to invoke the PEIM.

The PEIM that publishes the **AuthenticationState()** service uses its parameters in the following ways:

- *AuthenticationStatus* conveys the source information upon which the PEIM acts.
- The *DeferExecution* value tells the PEI Foundation whether or not to dispatch the PEIM.

In addition, between receiving the **AuthenticationState()** from the PEI Foundation and returning with the *DeferExecution* value, the PEIM that publishes **AuthenticationState()** can do the following:

- Log the file state.
- Lock the firmware hubs in response to an unsigned PEIM being discovered.

These latter behaviors are platform- and market-specific and thus outside the scope of the PEI CIS.

Status Codes Returned

EFI_SUCCESS	The service performed its action successfully.
EFI_SECURITY_VIOLATION	The object cannot be trusted

6.3.7 Temporary RAM Support PPI (Optional)

EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI (Optional)

Summary

This service allows for migrating from some contents of Temporary RAM store, which is instantiated during the SEC phase, into permanent RAM. The latter store will persist unmodified into the subsequent phase of execution, such as DXE. This service may be published by the SEC as part of the SEC-to-PEI handoff or published by any other PEIM.

GUID

```
#define EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI_GUID \
  {0xdbe23aa9, 0xa345, 0x4b97,\
  0x85, 0xb6, 0xb2, 0x26, 0xf1, 0x61, 0x73, 0x89}
```

Prototype

```
typedef struct _EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI {
  TEMPORARY_RAM_MIGRATION         TemporaryRamMigration;
} EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI;
```

Parameters

TemporaryRamMigration

Perform the migration of contents of Temporary RAM to Permanent RAM. This service may terminate the Temporary RAM, for example, if it cannot coexist with the Permanent RAM. See the **TemporaryRamMigration()** function description.

Description

This service abstracts the ability to migrate contents of the platform early memory store. This is an optional PPI that is only required for platforms that may have side effects when both Temporary RAM and Permanent RAM are enabled. This PPI provides a service that orchestrates the complete transition from Temporary RAM to Permanent RAM that avoids side effects. This includes the migration of all data, a stack switch action, and possibly the disabling of Temporary RAM.

If a platform does not have any side effects when both Temporary RAM and Permanent RAM are enabled, and the platform is required to disable the use of Temporary RAM, then **EFI_PEI_TEMPORARY_RAM_DONE** should be produced.

If a platform does not have any side effects when both Temporary RAM and Permanent RAM are enabled, and the platform is not required to disable the use of Temporary RAM, then neither **EFI_PEI_TEMPORARY_RAM_DONE** nor **EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI** should be produced.

EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI.TemporaryRamMigration ()

Summary

This service of the **EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI** that migrates temporary RAM into permanent memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * TEMPORARY_RAM_MIGRATION) (
  IN CONST EFI_PEI_SERVICES     **PeiServices,
  IN EFI_PHYSICAL_ADDRESS       TemporaryMemoryBase,
  IN EFI_PHYSICAL_ADDRESS       PermanentMemoryBase,
  IN UINTN                      CopySize
);
```

Parameters

PeiServices

Pointer to the PEI Services Table.

TemporaryMemoryBase

Source Address in temporary memory from which the SEC or PEIM will copy the Temporary RAM contents.

PermanentMemoryBase

Destination Address in permanent memory into which the SEC or PEIM will copy the Temporary RAM contents.

CopySize

Amount of memory to migrate from temporary to permanent memory.

Description

This service is published by the SEC module or a PEIM. It migrates the Temporary RAM contents into Permanent RAMRAM and performs all actions required to switch the active stack from Temporary RAM to Permanent RAM. The address range from *PermanentMemoryBase* to *PermanentMemoryBase* + *CopySize* should fix within the range of memory provided to the PEI Foundation as part of the **InstallPeiMemory()** core services. Also, since the SEC may have sequestered some of the Temporary RAM for its own data storage and PPI's, the SEC handoff now includes addresses and sizes of both the "available" (*PeiTemporaryRamBase*/ *PeiTemporaryRamSize*) and "total" (*TemporaryRamBase*/*TemporaryRamSize*) Temporary RAM as separate numbers.

PeiTemporaryRamBase is used by the PEI foundation for its resource management; *TemporaryRamBase* is used by the foundation as an input to this **TemporaryRamMigration()** service call. As such, the PEI foundation is the only agent who knows the full extent of the Temporary RAM store that needs migration to Permanent RAM. It will use this full extent as the *CopySize* argument in this PPI invocation. At minimum, the *CopySize* must include the portion of the Temporary RAM used by the SEC.

The PEI Foundation implementation will invoke this PPI service **TemporaryRamMigration()**, if present, after **InstallPeiMemory()** is invoked. **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI** is installed after the PPI service **TemporaryRamMigration()** is invoked, providing a signal to PEIMs that permanent memory is available.

If the **EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI** service is not available, a PEI foundation implementation shall copy the contents of the Temporary RAM to Permanent RAM directly and perform the stack switch action. The lack of this PPI is not an error condition.

The stack switch action, namely the beginning of usage of the permanent RAM as stack in lieu of the temporary RAM stack, is an integral capability of any PEI foundation implementation and need not have an API in this PPI or any other to externally-installed abstraction.

Status Codes Returned

EFI_SUCCESS	The data was successfully returned.
EFI_INVALID_PARAMETER	PermanentMemoryBase + CopySize > TemporaryMemoryBase when TemporaryMemoryBase > PermanentMemoryBase.

6.3.8 Temporary RAM Done PPI (Optional)

EFI_PEI_TEMPORARY_RAM_DONE_PPI (Optional)

Summary

The PPI that provides a service to disable the use of Temporary RAM.

GUID

```
#define EFI_PEI_TEMPORARY_RAM_DONE_PPI_GUID \
  { 0xceab683c, 0xec56, 0x4a2d, \
  { 0xa9, 0x6, 0x40, 0x53, 0xfa, 0x4e, 0x9c, 0x16 } }
```

Protocol Interface Structure

```
typedef struct _EFI_PEI_TEMPORARY_RAM_DONE_PPI {
  EFI_PEI_TEMPORARY_RAM_DONE  TemporaryRamDone;
} EFI_PEI_TEMPORARY_RAM_DONE_PPI;
```

Parameters

TemporaryRamDone

Disable the use of Temporary RAM.

Description

This is an optional PPI that may be produced by SEC or a PEIM. If present, it provide a service to disable the use of Temporary RAM. This service may only be called by the PEI Foundation after the transition from Temporary RAM to Permanent RAM is complete. This PPI provides an alternative to the Temporary RAM Migration PPI for system architectures that allow Temporary RAM and Permanent RAM to be enabled and accessed at the same time with no side effects.

EFI_PEI_TEMPORARY_RAM_DONE_PPI.TemporaryRamDone ()

Summary

Disable the use of Temporary RAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PEI_TEMPORARY_RAM_DONE) (
  VOID
  );
```

Description

TemporaryRamDone() disables the use of Temporary RAM. If present, this service is invoked by the PEI Foundation after the **EFI_PEI_PERMANANT_MEMORY_INSTALLED_PPI** is installed.

Status Codes Returned

EFI_SUCCESS	Use of Temporary RAM was disabled.
EFI_DEVICE_ERROR	Temporary RAM could not be disabled.

7
PEIMs

7.1 Introduction

A Pre-EFI Initialization Module (PEIM) represents a unit of code and/or data. It abstracts domain-specific logic and is analogous to a DXE driver. As such, a given group of PEIMs for a platform deployment might include a set of the following:

- Platform-specific PEIMs
- Processor-specific PEIMs
- Chipset-specific PEIMs
- PEI CIS–prescribed architectural PEIMs
- Miscellaneous PEIMs

The PEIM encapsulation allows for a platform builder to use services for a given hardware technology without having to build the source of this technology or necessarily understand its implementation. A PEIM-to-PEIM Interface (PPI) is the means by which to abstract hardware-specific complexities to a platform builder's PEIM. As such, PEIMs can work in concert with other PEIMs using PPIs.

In addition, PEIMs can ascertain a fixed set of services that are always available through the PEI Services Table.

Finally, because the PEIM represents the basic unit of execution beyond the Security (SEC) phase and the PEI Foundation, there will always be some non-zero-sized collection of PEIMs in a platform.

7.2 PEIM Structure

7.2.1 PEIM Structure Overview

Each PEI Module (PEIM) is stored in a file. It consists of the following:

- Standard header
- Execute-in-place code/data section
- Optional relocation information
- Authentication information, if present

The PEIM binary image can be executed in place from its location in the firmware volume (FV) or from a compressed component that will be shadowed after permanent memory has been installed. The executable section of the PEIM may be either position-dependent or position-independent code. If the executable section of the PEIM is position-dependent code, relocation information must be provided in the PEIM image to allow FV store software to relocate the image to a different location than it is compiled.

Figure 2 depicts the typical layout of a PEIM.

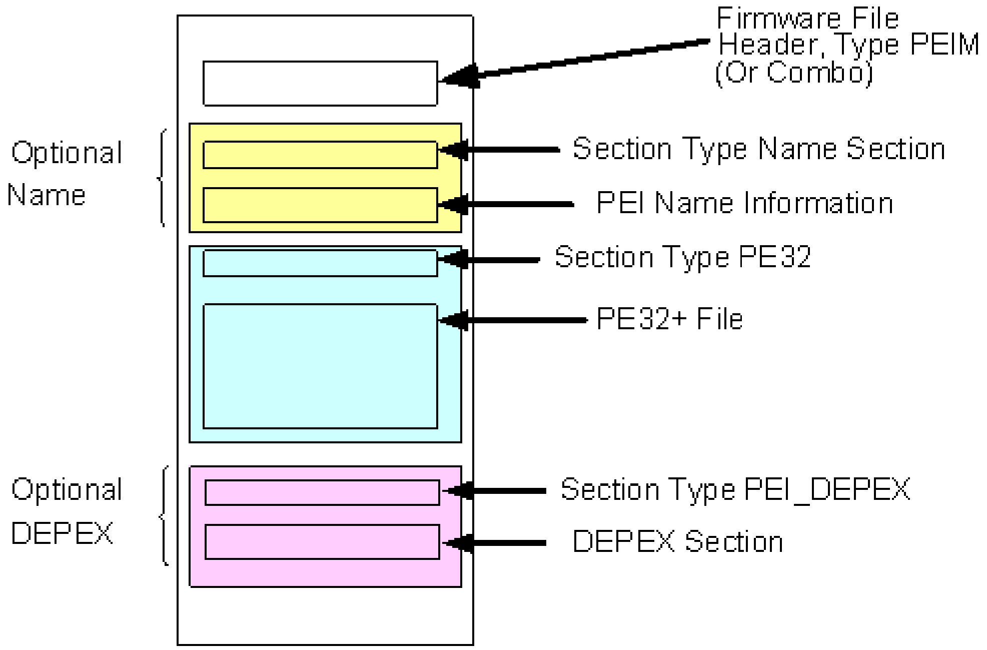


Figure 2. Typical PEIM Layout in a Firmware File

7.2.2 Relocation Information

7.2.2.1 Position-Dependent Code

PEIMs that are developed using position-dependent code require relocation information. When an image in a firmware volume (FV) is updated, the update software will use the relocation information to fix the code image according to the module's location in the FV. The relocation is done on the authenticated image; therefore, software verifying the integrity of the image must undo the relocation during the verification process.

There is no explicit pointer to this data. Instead, the update and verification tool will know that the image is actually stored as PE32 if the *Pe32Image* bit is set in the header **EFI_COMMON_SECTION_HEADER** or **EFI_COMMON_SECTION_HEADER2**; types **EFI_COMMON_SECTION_HEADER** and **EFI_COMMON_SECTION_HEADER2** are defined in the *Platform Initialization Specification, Volume 3.* The PE32 specification, in turn, will be used to ascertain the relocation records.

7.2.2.2 Position-Independent Code

If the PEIM is written in position-independent code, then its entry point shall be at the lowest address in the section. This method is useful for creating PEIMs for the Itanium® processor family.

7.2.2.3 Relocation Information Format

The relocations will be contained in a TE or PE32+ image. See the *Microsoft Portable Executable and Common Object File Format Specification* for more information. The determination of whether the image subscribes to the PE32 image format or is position-independent assembly language is provided by the firmware volume section type. The PEIM that is formatted as PE/COFF will always be linked against a base address of zero. This allows for support of signature checking.

The section may also be compressed if there is a compression encapsulation section.

7.2.3 Authentication Information

This section describes in more detail, the means by which authentication information could be contained in a section of type **EFI_SECTION_GUID_DEFINED** (see the *Platform Initialization Specification*, Volume 3, for more information on section types). The information contained in this section could be one of the following:

- A cryptographic-quality hash computed across the PEIM image
- A simple checksum
- A CRC

The GUID defines the meaning of the associated encapsulated data. The relocation section is needed to undo the fix-ups done on the image so the hash that was computed at build time can be confirmed. In other words, the build of a PEIM image is linked against zero, but the update tool will relocate the PEIM image for its execute-in-place address (at least for images that are not position-independent code). Any signing information is calculated on the image after the image has been linked against an address of zero. The relocations on the image will have to be “undone” to determine if the image has been modified.

The image must be linked against address zero by the PEIM provider. The build or update tool will apply the appropriate relocations. The linkage against address zero is key because it allows a subsequent undoing of the relocations.

7.3 PEIM Invocation Entry Point

7.3.1 EFI_PEIM_ENTRY_POINT2

Summary

The PEI Dispatcher will invoke each PEIM one time.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEIM_ENTRY_POINT2) (
  IN EFI_PEI_FILE_HANDLE          FileHandle,
  IN CONST EFI_PEI_SERVICES       **PeiServices
  );
```

Parameters

FileHandle

Handle of the file being invoked. Type **EFI_PEI_FILE_HANDLE** is defined in **FfsFindNextFile()**.

PeiServices

Describes the list of possible PEI Services.

Description

This function is the entry point for a PEIM. **EFI_IMAGE_ENTRY_POINT2** is the equivalent of this state in the UEFI/DXE environment; see the DXE CIS for its definition.

The motivation behind this definition is that the firmware file system has the provision to mark a file as being both a PEIM and DXE driver. The result of this name would be that both the PEI Dispatcher and the DXE Dispatcher would attempt to execute the module. In doing so, it is incumbent upon the code in the entry point of the driver to decide what services are exposed, namely whether to make boot service and runtime calls into the UEFI System Table or to make calls into the PEI Services Table. The means by which to make this decision entail examining the second argument on entry, which is a pointer to the respective foundation's exported service-call table. Both PEI and UEFI/DXE have a common header, **EFI_TABLE_HEADER**, for the table. The code in the PEIM or DXE driver will examine the *Arg2->Hdr->Signature*. If it is **EFI_SYSTEM_TABLE_SIGNATURE**, the code will assume DXE driver behavior; if it is **PEI_SERVICES_SIGNATURE**, the code will assume PEIM behavior.

Status Codes Returned

EFI_SUCCESS	The service completed successfully
< 0	There was an error

7.4 PEIM Descriptors

7.4.1 PEIM Descriptors Overview

A PEIM descriptor is the data structure used by PEIMs to export service entry points and data. The descriptor contains the following:

- Flags
- A pointer to a GUID
- A pointer to data

The latter data can include a list of pointers to functions and/or data. It is the function pointers that are commonly referred to as PEIM-to-PEIM Interfaces (PPIs), and the PPI is the unit of software across which PEIMs can invoke services from other PEIMs.

A PEIM also uses a PEIM descriptor to export a service to the PEI Foundation into which the PEI Foundation will pass control in response to an event, namely "notifying" the callback when a PPI is installed or reinstalled. As such, PEIM descriptors serve the dual role of exposing the following:

- A callable interface/data for other PEIMs
- A callback interface from the perspective of the PEI Foundation

EFI_PEI_DESCRIPTOR

Summary

This data structure is the means by which callable services are installed and notifications are registered in the PEI phase.

Prototype

```
typedef union {
  EFI_PEI_NOTIFY_DESCRIPTOR   Notify;
  EFI_PEI_PPI_DESCRIPTOR      Ppi;
} EFI_PEI_DESCRIPTOR;
```

Parameters

Notify

The typedef structure of the notification descriptor. See the **EFI_PEI_NOTIFY_DESCRIPTOR** type definition.

Ppi

The typedef structure of the PPI descriptor. See the **EFI_PEI_PPI_DESCRIPTOR** type definition.

Description

EFI_PEI_DESCRIPTOR is a data structure that can be either a PPI descriptor or a notification descriptor. A PPI descriptor is used to expose callable services to other PEIMs. A notification descriptor is used to register for a notification or callback when a given PPI is installed.

EFI_PEI_NOTIFY_DESCRIPTOR

Summary

The data structure in a given PEIM that tells the PEI Foundation where to invoke the notification service.

Prototype

```
typedef struct _EFI_PEI_NOTIFY_DESCRIPTOR {
  UINTN                        Flags;
  EFI_GUID                     *Guid;
  EFI_PEIM_NOTIFY_ENTRY_POINT  Notify;
} EFI_PEI_NOTIFY_DESCRIPTOR;
```

Parameters

Flags

Details if the type of notification is callback or dispatch.

Guid

The address of the **EFI_GUID** that names the interface.

Notify

Address of the notification callback function itself within the PEIM. Type **EFI_PEIM_NOTIFY_ENTRY_POINT** is defined in "Related Definitions" below.

Description

EFI_PEI_NOTIFY_DESCRIPTOR is a data structure that is used by a PEIM that needs to be called back when a PPI is installed or reinstalled. The notification is similar to the **RegisterProtocolNotify()** function in the UEFI 2.0 Specification. The use model is complementary to the dependency expression (depex) and is as follows:

- A PEIM expresses the PPIs that it *must* have to execute in its depex list.
- A PEIM expresses any other PEIMs that it needs, perhaps at some later time, in **EFI_PEI_NOTIFY_DESCRIPTOR**.

The latter data structure includes the GUID of the PPI for which the PEIM publishing the notification would like to be reinvoked.

Following is an example of the notification use model for **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI**. In this example, a PEIM called *SamplePeim* executes early in the PEI phase before main memory is available. However, SamplePeim also needs to create some large data structure later in the PEI phase. As such, SamplePeim has a NULL depex, but after its entry point is processed, it needs to call **NotifyPpi()** with a **EFI_PEI_NOTIFY_DESCRIPTOR**, where the notification descriptor includes the following:

- A reference to **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI**
- A reference to a function within this same PEIM called *SampleCallback*

When the PEI Foundation finally migrates the system from temporary to permanent memory and installs the **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI**, the PEI Foundation assesses if there are any pending notifications on this PPI. After the PEI Foundation discovers the descriptor from SamplePeim, the PEI Foundation invokes SampleCallback.

With respect to the *Flags* parameter, the difference between callback and dispatch mode is as follows:

- **Callback mode:** Invokes all of the agents that are registered for notification immediately after the PPI is installed.
- **Dispatch mode:** Calls the agents that are registered for notification only after the PEIM that installs the PPI in question has returned to the PEI Foundation.

The callback mechanism will give a better quality of service, but it has the downside of possibly deepening the use of the stack (i.e., the agent that installed the PPI that engenders the notification is a PEIM itself that has used the stack already). The dispatcher mode, however, is better from a stack-usage perspective in that when the PEI Foundation invokes the agents that want notification, the stack has returned to the minimum stack usage of just the PEI Foundation.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEIM_NOTIFY_ENTRY_POINT) (
  IN EFI_PEI_SERVICES           **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  );
```

PeiServices

Indirect reference to the PEI Services Table.

NotifyDescriptor

Address of the notification descriptor data structure. Type **EFI_PEI_NOTIFY_DESCRIPTOR** is defined above.

Ppi

Address of the PPI that was installed.

The status code returned from this function is ignored.

EFI_PEI_PPI_DESCRIPTOR

Summary

The data structure through which a PEIM describes available services to the PEI Foundation.

Prototype

```
typedef struct {
  UINTN                         Flags;
  EFI_GUID                      *Guid;
  VOID                          *Ppi;
} EFI_PEI_PPI_DESCRIPTOR;
```

Parameters

Flags

This field is a set of flags describing the characteristics of this imported table entry. See “Related Definitions” below for possible flag values.

Guid

The address of the **EFI_GUID** that names the interface.

Ppi

A pointer to the PPI. It contains the information necessary to install a service.

Description

EFI_PEI_PPI_DESCRIPTOR is a data structure that is within the body of a PEIM or created by a PEIM. It includes the following:

- Information about the nature of the service
- A reference to a GUID naming the service
- An associated pointer to either a function or data related to the service

There can be a catenation of one or more of these **EFI_PEI_PPI_DESCRIPTOR**s. The final descriptor will have the **EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST** flag set to indicate to the PEI Foundation how many of the descriptors need to be added to the PPI database within the PEI Foundation. The PEI Services that references this data structure include **InstallPpi()**, **ReinstallPpi()**, and **LocatePpi()**.

Related Definitions

```
//
// PEI PPI Services List Descriptors
//

#define EFI_PEI_PPI_DESCRIPTOR_PIC             0x00000001
#define EFI_PEI_PPI_DESCRIPTOR_PPI             0x00000010
#define EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK 0x00000020
#define EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH 0x00000040
#define EFI_PEI_PPI_DESCRIPTOR_NOTIFY_TYPES    0x00000060
```

```
#define EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST  0x80000000
```

Table 16 provides descriptions of the fields in the above definition:

Table 16. PEI PPI Services List Descriptors

Descriptor	Description
EFI_PEI_PPI_DESCRIPTOR_PIC	When set to 1, this designates that the PPI described by the structure is position-independent code (PIC).
EFI_PEI_PPI_DESCRIPTOR_PPI	When set to 1, this designates that the PPI described by this structure is a normal PPI. As such, it should be callable by the conventional PEI infrastructure.
EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK	When set to 1, this flag designates that the service registered in the descriptor is to be invoked at callback. This means that if the PPI is installed for which the listener registers a notification, then the callback routine will be immediately invoked. The danger herein is that the callback will inherit whatever depth had been traversed up to and including this call.
EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH	When set to 1, this flag designates that the service registered in the descriptor is to be invoked at dispatch. This means that if the PPI is installed for which the listener registers a notification, then the callback routine will be deferred until the PEIM calling context returns to the PEI Foundation. Prior to invocation of the next PEIM, the notifications will be dispatched. The advantage herein is that the callback will have the maximum available stack depth as any other PEIM.
EFI_PEI_PPI_DESCRIPTOR_NOTIFY_TYPES	When set to 1, this flag designates that this is a notification-style PPI.
EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST	This flag is set to 1 in the last structure entry in the list of PEI PPI descriptors. This flag is used by the PEI Foundation Services to know that there are no additional interfaces to install.

7.5 PEIM-to-PEIM Communication

7.5.1 Overview

PEIMs may invoke other PEIMs. The interfaces themselves are named using GUIDs. Because the PEIMs may be authored by different organizations at different times and updated at different times, references to these interfaces cannot be resolved during their execution by referring to the PEI PPI database. The database is loaded and queried using PEI Services such as **InstallPpi()** and **LocatePpi()**.

7.5.2 Dynamic PPI Discovery

7.5.2.1 PPI Database

The PPI database is a data structure that PEIMs can use to discover what interfaces are available or to manage a specific interface. The actual layout of the PPI database is opaque to a PEIM but its contents can be queried and manipulated using the following PEI Services:

- **`InstallPpi()`**
- **`ReinstallPpi()`**
- **`LocatePpi()`**
- **`NotifyPpi()`**

7.5.2.2 Invoking a PPI

When the PEI Foundation examines a PEIM for dispatch eligibility, it examines the dependency expression section of the firmware file. If there are non-NULL contents, the Reverse Polish Notation (RPN) expression is evaluated. Any requested PPI GUIDs in this data structure are queried in the PPI database. The existence in the database of the particular PUSH_GUID depex opcode leads to this expression evaluating to true.

7.5.2.3 Address Resolution

When a PEIM needs to leverage a PPI, it uses the PEI Foundation Service **`LocatePpi()`** to discover if an instance of the interface exists. The PEIM could do either of the following:

- Install the PPI in its depex to ensure that its entry point will not be invoked until the needed PPI is already installed
- Have a very thin set of code in its entry point that simply registers a notification on the desired PPI.

In the case of either the depex or the notification, the **`LocatePpi()`** call will then succeed and the pointer returned on this call references the **`EFI_PEI_PPI_DESCRIPTOR`**. It is through this data structure that the actual code entry point can be discovered. If this PEIM is being loaded before permanent memory is available, it will not have resources to cache this discovered interface and will have to search for this interface every time it needs to invoke the service.

It should also be noted that you cannot uninstall a PPI, so the services will be left in the database. If a PPI needs to be shrouded, a version can be "reinstalled" that just returns failure.

Also, there is peril in caching a PPI. For example, if you cache a PPI and the producer of the PPI "reinstalls" it to be something else (i.e., shadows to memory), then you have the possibility that the agent who cached the data will have "stale" or "illegal" data. For example, imagine the Stall PPI, **`EFI_PEI_STALL_PPI`**, relocating itself to memory using the Load File PPI, **`EFI_PEI_LOAD_FILE_PPI`**, and reinstalling the interface for performance considerations. A way to solve the latter issue, as a platform builder, is by having a different stall PPI for the memory-based one versus that of the Execute In Place (XIP) one.

8
Additional PPIs

8.1 Introduction

Architectural PPIs described a collection of architecturally required PPIs. These were interfaces consumed by the PEI Foundation and are not intended to be consumed by other PEIMs.

In addition to these architectural PPIs, however, there is another name space of PPIs that are optional or mandatory for a given platform. This section describes these additional PPIs:

- Required PPIs:
  - CPU I/O PPI
  - PCI Configuration PPI
  - Stall PPI
  - PEI Variable Services
- Optional PPIs:
  - Security (SEC) Platform Information PPI

These shall be referred to as first-class PEIMs in some contexts.

8.2 Required Additional PPIs

8.2.1 PCI Configuration PPI (Required)

The PEI phase provides limited support for initializing and configuring PCI devices through the EFI_PEI_PCI_CFG2_PPI. The PEI module which supports a PCI root bridge may install this PPI to allow access to the PCI configuration space for a particular PCI segment. The PEI module responsible for the PCI root bridge representing segment 0 should also install a pointer to the PPI in the PEI Services Table.

The PEI modules which control devices on segment 0 may use the pointer provided in the PEI Services Table. The PEI modules for devices residing on other segments may find the correct PPI by iterating through PPI instances using the LocatePpi() function. For example:

```
EFI_STATUS                Status;
UINTN                     Instance = 0;
EFI_PEI_PPI_DESCRIPTOR    *PciCfgDescriptor = NULL;
EFI_PEI_PCI_CFG2_PPI      *PciCfg = NULL;

/* Loop through all instances of the PPI */
for (;;) {
     Status = PeiServices->LocatePpi(PeiServices,
          &gPeiPciCfg2PpiGuid,
          Instance,
          &PciCfgDescriptor,
          (VOID**) &PciCfg
          );
     if (Status != EFI_SUCCESS ||
          PciCfg->Segment == MySegment) {
          break;
     }
     Instance++;
}
if (Status == EFI_SUCCESS) {
     ...PciCfg contains pointer…
}
```

EFI_PEI_PCI_CFG2_PPI

Summary

Provides platform or chipset-specific access to the PCI configuration space for a specific PCI segment.

Guid

```
static const EFI_GUID EFI_PEI_PCI_CFG2_PPI_GUID = \
{ 0x57a449a, 0x1fdc, 0x4c06, \
{ 0xbf, 0xc9, 0xf5, 0x3f, 0x6a, 0x99, 0xbb, 0x92 }}
```

Prototype

```
typedef struct _EFI_PEI_PCI_CFG2_PPI {
    EFI_PEI_PCI_CFG2_PPI_IO     Read;
    EFI_PEI_PCI_CFG2_PPI_IO     Write;
    EFI_PEI_PCI_CFG2_PPI_RW     Modify;
    UINT16                     Segment;
} EFI_PEI_PCI_CFG2_PPI
```

Parameters

Read

PCI read services. See the **Read()** function description.

Write

PCI write services. See the **Write()** function description.

Modify

PCI read-modify-write services. See the **Modify()** function description.

Segment

The PCI bus segment which the specified functions will access.

Description

The **EFI_PEI_PCI_CFG2_PPI** interfaces are used to abstract accesses to the configuration space of PCI controllers behind a PCI root bridge controller. There can be multiple instances of this PPI in the system, one for each segment. The pointer to the instance which describes segment 0 is installed in the PEI Services Table.

The assignment of segment numbers is implementation specific.

The **Modify()** service allows for space-efficient implementation of the following common operations:

- Reading a register
- Changing some bit fields within the register
- Writing the register value back into the hardware

The **Modify()** service is a composite of the **Read()** and **Write()** services.

Parameters

Register

Register number in PCI configuration space.

Function

Function number in the PCI device (0-7).

Device

Device number in the PCI device (0-31).

Bus

PCI bus number (0-255).

ExtendedRegister

Register number in PCI configuration space. If this field is zero, then *Register* is used for the register number. If this field is non-zero, then *Register* is ignored and this field is used for the register number.

EFI_PEI_PCI_CFG2_PPI.Read()

Summary

PCI read operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCI_CFG_PPI_IO) (
    IN CONST EFI_PEI_SERVICES       **PeiServices,
    IN CONST EFI_PEI_PCI_CFG2_PPI   *This,
    IN EFI_PEI_PCI_CFG_PPI_WIDTH    Width,
    IN UINT64                       Address,
    IN OUT VOID                     *Buffer
    );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Width

The width of the access. Enumerated in bytes. Type **EFI_PEI_PCI_CFG_PPI_WIDTH** is defined in "Related Definitions" below.

Address

The physical address of the access. The format of the address is described by **EFI_PEI_PCI_CFG_PPI_PCI_ADDRESS**, which is defined in "Related Definitions" below.

Buffer

A pointer to the buffer of data.

Description

The **Read()** function reads from a given location in the PCI configuration space.

Related Definitions

```
//*************************************************************
// EFI_PEI_PCI_CFG_PPI_WIDTH
//*************************************************************
typedef enum {
    EfiPeiPciCfgWidthUint8 = 0,
    EfiPeiPciCfgWidthUint16 = 1,
    EfiPeiPciCfgWidthUint32 = 2,
    EfiPeiPciCfgWidthUint64 = 3,
    EfiPeiPciCfgWidthMaximum
} EFI_PEI_PCI_CFG_PPI_WIDTH;

//*************************************************************
// EFI_PEI_PCI_CFG_PPI_PCI_ADDRESS
//*************************************************************
typedef struct {
    UINT8             Register;
    UINT8             Function;
    UINT8             Device;
    UINT8             Bus;
    UINT32            ExtendedRegister;
} EFI_PEI_PCI_CFG_PPI_PCI_ADDRESS;
```

Register

8-bit register offset within the PCI configuration space for a given device's function space.

Function

Only the 3 least-significant bits are used to encode one of 8 possible functions within a given device.

Device

Only the 5 least-significant bits are used to encode one of 32 possible devices.

Bus

8-bit value to encode between 0 and 255 buses.

ExtendedRegister

Register number in PCI configuration space. If this field is zero, then *Register* is used for the register number. If this field is non-zero, then *Register* is ignored and this field is used for the register number.

```
#define EFI_PEI_PCI_CFG_ADDRESS(bus,dev,func,reg) \
               (((bus) << 24) | \
               ((dev) << 16) | \
               ((func) << 8) | \
               ((reg) < 256 ? (reg) : ((UINT64) (reg) << 32))
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully
EFI_DEVICE_ERROR	There was a problem with the transaction.
EFI_DEVICE_NOT_READY	The device is not capable of supporting the operation at this time.

EFI_PEI_PCI_CFG2_PPI.Write()

Summary

PCI write operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCI_CFG_PPI_IO) (
    IN CONST EFI_PEI_SERVICES        **PeiServices,
    IN CONST EFI_PEI_PCI_CFG2_PPI    *This,
    IN EFI_PEI_PCI_CFG_PPI_WIDTH     Width,
    IN UINT64                        Address,
    IN OUT VOID                      *Buffer
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Width

The width of the access. Enumerated in bytes. Type **EFI_PEI_PCI_CFG_PPI_WIDTH** is defined in **Read()**.

Address

The physical address of the access.

Buffer

A pointer to the buffer of data.

Description

The **Write()** function writes to a given location in the PCI configuration space.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_DEVICE_ERROR	There was a problem with the transaction.
EFI_DEVICE_NOT_READY	The device is not capable of supporting the operation at this time.

EFI_PEI_PCI_CFG2_PPI.Modify()

Summary

PCI read-modify-write Operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCI_CFG_PPI_RW) (
    IN CONST EFI_PEI_SERVICES     **PeiServices,
    IN CONST EFI_PEI_PCI_CFG_PPI  *This,
    IN EFI_PEI_PCI_CFG_PPI_WIDTH  Width,
    IN UINT64                     Address,
    IN VOID                       *SetBits,
    IN VOID                       *ClearBits
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Width

The width of the access. Enumerated in bytes. Type **EFI_PEI_PCI_CFG_PPI_WIDTH** is defined in **Read()**.

Address

The physical address of the access.

SetBits

Points to value to bitwise-OR with the read configuration value. The size of the value is determined by *Width*.

ClearBits

Points to the value to negate and bitwise-AND with the read configuration value. The size of the value is determined by *Width*.

Description

The **Modify()** function performs a read-modify-write operation on the contents from a given location in the PCI configuration space.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_DEVICE_ERROR	There was a problem with the transaction.
EFI_DEVICE_NOT_READY	The device is not capable of supporting the operation at this time.

8.2.2 Stall PPI (Required)

EFI_PEI_STALL_PPI (Required)

Summary

This PPI is installed by some platform or chipset-specific PEIM that abstracts the blocking stall service to other agents.

GUID

```
#define EFI_PEI_STALL_PPI_GUID \
  { 0x1f4c6f90, 0xb06b, 0x48d8, {0xa2, 0x01, 0xba, 0xe5, \
  0xf1, 0xcd, 0x7d, 0x56} }
```

PPI Interface Structure

```
typedef
struct _EFI_PEI_STALL_PPI {
  UINTN                                Resolution;
  EFI_PEI_STALL                        Stall;
} EFI_PEI_STALL_PPI;
```

Parameters

Resolution

The resolution in microseconds of the stall services.

Stall

The actual stall procedure call. See the **Stall()** function description.

Description

This service provides a simple, blocking stall with platform-specific resolution.

EFI_PEI_STALL_PPI.Stall()

Summary

Blocking stall.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_STALL) (
  IN CONST EFI_PEI_SERVICES      **PeiServices,
  IN CONST EFI_PEI_STALL_PPI     *This,
  IN UINTN                        Microseconds
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to the local data for the interface.

Microseconds

Number of microseconds for which to stall.

Description

The **Stall()** function provides a blocking stall for at least the number of microseconds stipulated in the final argument of the API.

Status Codes Returned

EFI_SUCCESS	The service provided at least the required delay.

8.2.3 Variable Services PPI (Required)

EFI_PEI_READ_ONLY_VARIABLE2_PPI

Summary

Permits read-only access to the UEFI variable store during the PEI phase.

GUID

```
#define EFI_PEI_READ_ONLY_VARIABLE2_PPI_GUID \
  { 0x2ab86ef5, 0xecb5, 0x4134, \
  0xb5, 0x56, 0x38, 0x54, 0xca, 0x1f, 0xe1, 0xb4 }
```

Prototype

```
typedef struct _EFI_PEI_READ_ONLY_VARIABLE2_PPI {
  EFI_PEI_GET_VARIABLE2             GetVariable;
  EFI_PEI_GET_NEXT_VARIABLE_NAME2   NextVariableName;
} EFI_PEI_READ_ONLY_VARIABLE2_PPI;
```

Parameters

GetVariable

A service to read the value of a particular variable using its name.

NextVariableName

Find the next variable name in the variable store.

Description

These services provide a light-weight, read-only variant of the full UEFI variable services.

EFI_PEI_READ_ONLY_VARIABLE2_PPI.GetVariable

Summary

This service retrieves a variable's value using its name and GUID.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_VARIABLE2)(
  IN      CONST EFI_PEI_READ_ONLY_VARIABLE2_PPI *This,
  IN      CONST CHAR16                          *VariableName,
  IN      CONST EFI_GUID                        *VariableGuid,
  OUT     UINT32                                *Attributes,
  IN OUT  UINTN                                 *DataSize,
  OUT     VOID                                  *Data OPTIONAL
  );
```

Parameters

This

A pointer to this instance of the **EFI_PEI_READ_ONLY_VARIABLE2_PPI**.

VariableName

A pointer to a null-terminated string that is the variable's name.

VariableGuid

A pointer to an **EFI_GUID** that is the variable's GUID. The combination of *VariableGuid* and *VariableName* must be unique.

Attributes

If non-NULL, on return, points to the variable's attributes. See "Related Definitons" below for possible attribute values.

DataSize

On entry, points to the size in bytes of the *Data* buffer. On return, points to the size of the data returned in *Data*.

Data

Points to the buffer which will hold the returned variable value. May be **NULL** with a zero *DataSize* in order to determine the size of the buffer needed.

Description

Read the specified variable from the UEFI variable store. If the *Data* buffer is too small to hold the contents of the variable, the error **EFI_BUFFER_TOO_SMALL** is returned and *DataSize* is set to the required buffer size to obtain the data.

Status Codes Returned

EFI_SUCCESS	The variable was read successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The *DataSize* is too small for the result. *DataSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*VariableName* is **NULL**.
EFI_INVALID_PARAMETER	*DataSize* is **NULL**.
EFI_INVALID_PARAMETER	The *DataSize* is not too small and Data is **NULL**.
EFI_DEVICE_ERROR	The variable could not be retrieved because of a device error.
EFI_INVALID_PARAMETER	*VariableGuid* is **NULL**.

EFI_PEI_READ_ONLY_VARIABLE2_PPI.NextVariableName

Summary

Return the next variable name and GUID.

Prototype

```
typedef
EFI_STATUS
(EFIAPI EFI_PEI_GET_NEXT_VARIABLE_NAME2) (
  IN CONST EFI_PEI_READ_ONLY_VARIABLE2_PPI *This,
  IN OUT UINTN                             *VariableNameSize,
  IN OUT CHAR16                            *VariableName,
  IN OUT EFI_GUID                          *VariableGuid
  );
```

Parameters

This

A pointer to this instance of the **EFI_PEI_READ_ONLY_VARIABLE2_PPI**.

VariableNameSize

On entry, points to the size of the buffer pointed to by *VariableName*. On return, the size of the variable name buffer

VariableName

On entry, a pointer to a null-terminated string that is the variable's name. On return, points to the next variable's null-terminated name string.

VariableGuid

On entry, a pointer to an **EFI_GUID** that is the variable's GUID. On return, a pointer to the next variable's GUID.

Description

This function is called multiple times to retrieve the *VariableName* and *VariableGuid* of all variables currently available in the system. On each call, the previous results are passed into the interface, and, on return, the interface returns the data for the next interface. When the entire variable list has been returned, **EFI_NOT_FOUND** is returned.

Note: *If* **EFI_BUFFER_TOO_SMALL** *is returned, the* VariableName *buffer was too small for the name of the next variable. When such an error occurs,* VariableNameSize *is updated to reflect the size of the buffer needed. In all cases when calling* GetNextVariableName*() the* VariableNameSize *must not exceed the actual buffer size that was allocated for* VariableName*.*

To start the search, a null-terminated string is passed in *VariableName*; that is, *VariableName* is a pointer to a null Unicode character. This is always done on the initial call. When *VariableName* is a pointer to a null Unicode character, *VariableGuid* is ignored.

Status Codes Returned

EFI_SUCCESS	The variable was read successfully.
EFI_NOT_FOUND	The variable could not be found.
EFI_BUFFER_TOO_SMALL	The *VariableNameSize* is too small for the resulting data. *VariableNameSize* is updated with the size required for the specified variable.
EFI_INVALID_PARAMETER	*VariableName*, *VariableGuid* or *VariableNameSize* is **NULL**
EFI_DEVICE_ERROR	The variable could not be retrieved because of a device error.

8.3 Optional Additional PPIs

8.3.1 SEC Platform Information PPI (Optional)

EFI_SEC_PLATFORM_INFORMATION_PPI (Optional)

Summary

This service is the platform information for the PEI Foundation.

GUID

```
#define EFI_SEC_PLATFORM_INFORMATION_GUID \
  {0x6f8c2b35, 0xfef4, 0x448d, 0x82, 0x56, 0xe1, \
  0x1b, 0x19, 0xd6,   0x10, 0x77}
```

Prototype

```
typedef struct _EFI_SEC_PLATFORM_INFORMATION_PPI {
  EFI_SEC_PLATFORM_INFORMATION           PlatformInformation;
} EFI_SEC_PLATFORM_INFORMATION_PPI;
```

Parameters

PlatformInformation

Conveys state information out of the SEC phase into PEI. See the **PlatformInformation()** function description.

Description

This service abstracts platform-specific information.

EFI_SEC_PLATFORM_INFORMATION_PPI.PlatformInformation()

Summary

This service is the single member of the **EFI_SEC_PLATFORM_INFORMATION_PPI** that conveys state information out of the Security (SEC) phase into PEI.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SEC_PLATFORM_INFORMATION) (
  IN CONST EFI_PEI_SERVICES              **PeiServices,
  IN OUT UINT64                          *StructureSize,
  OUT EFI_SEC_PLATFORM_INFORMATION_RECORD
                                     *PlatformInformationRecord
  );
```

Parameters

PeiServices

Pointer to the PEI Services Table.

StructureSize

Pointer to the variable describing size of the input buffer.

PlatformInformationRecord

Pointer to the **EFI_SEC_PLATFORM_INFORMATION_RECORD**. Type **EFI_SEC_PLATFORM_INFORMATION_RECORD** is defined in "Related Definitions" below.

Description

This service is published by the SEC phase. The SEC phase handoff has an optional **EFI_PEI_PPI_DESCRIPTOR** list as its final argument when control is passed from SEC into the PEI Foundation. As such, if the platform supports the built-in self test (BIST) on IA-32 Intel architecture or the PAL-A handoff state for Itanium® architecture, this information is encapsulated into the data structure abstracted by this service. This information is collected for the boot-strap processor (BSP) on IA-32, and for Itanium architecture, it is available on all processors that execute the PEI Foundation.

The motivation for this service is that a specific processor register contains this information for each microarchitecture, but the PEI CIS avoids using specific processor registers. Instead, the PEI CIS describes callable interfaces across which data is conveyed. As such, this processor state information that is collected at the reset of the machine is mapped into a common interface. The expectation is that a manageability agent, such as a platform PEIM that logs information for the platform, would use this interface to determine the viability of the BSP and possibly select an alternate BSP if there are significant errors.

Related Definitions

```
//*******************************************************
// EFI_SEC_PLATFORM_INFORMATION_RECORD
//*******************************************************
typedef union {
  IA32_HANDOFF_STATUS           IA32HealthFlags;
  X64_HANDOFF_STATUS            x64HealthFlags;
  ITANIUM_HANDOFF_STATUS        ItaniumHealthFlags;
} EFI_SEC_PLATFORM_INFORMATION_RECORD;
```

HealthFlags

Contains information generated by microcode, or hardware, about the state of the processor upon reset. Type **EFI_HEALTH_FLAGS** is defined below.

```
//*******************************************************
// EFI_HEALTH_FLAGS
//*******************************************************
typedef union {
  struct {
    UINT32   Status                    : 2;
    UINT32   Tested                    : 1;
    UINT32   Reserved1                 :13;
    UINT32   VirtualMemoryUnavailable  : 1;
    UINT32   Ia32ExecutionUnavailable  : 1;
    UINT32   FloatingPointUnavailable  : 1;
    UINT32   MiscFeaturesUnavailable   : 1;
    UINT32   Reserved2                 :12;
  }                    Bits;
  UINT32               Uint32;
} EFI_HEALTH_FLAGS;
```

IA-32 and X64 have the BIST. See "Health Flag Bit Format" on page 240 for more information on **EFI_HEALTH_FLAGS**.

The following two structures are for IA32 and x64.

```
typedef   EFI_HEALTH_FLAGS    X64_HANDOFF_STATUS;
typedef   EFI_HEALTH_FLAGS   IA32_HANDOFF_STATUS;
```

There is no instance of an **EFI_SEC_PLATFORM_INFORMATION_RECORD** for the ARM PI binding.

For Itanium, the structure is as follows:

For details, see the *Itanium Software Developers Manual*, Volume 2, Rev 2.2, Document Number: 245318-005 (SwDevMan) Section 11.2.2.1 "Definition of **SALE_ENTRY** State Parameter" as indicated below.

```
typedef struct {
  UINT8 BootPhase; // SALE_ENTRY state : 3 = Recovery_Check
                   // and 0 = RESET or Normal_Boot phase.
                   // See 'function' in SwDevMan Fig 11-8 and
                   // Table 11-3.
  UINT8 FWStatus;  // Firmware status on entry to SALE.
                   // See 'Status' in SwDevMan Fig 11-8 and
                   // Table 11-4.
  UINT16 Reserved1;
  UINT32 Reserved2;
  UINT16 ProcId;   // Geographically significant unique
                   // processor ID assigned by PAL.
                   // See 'proc_id' in SwDevMan Fig 11-9
                   // and Table 11-5.
  UINT16 Reserved3;
  UINT8  IdMask;               // See 'id_mask' in SwDevMan
                               // Fig 11-9 and Table 11-5.
  UINT8  EidMask;              // See 'eid_mask' in SwDevMan
                               // Fig 11-9 and Table 11-5
  UINT16 Reserved4;
  UINT64 PalCallAddress;       // Address to make PAL calls.
  UINT64 PalSpecialAddress;    // If the entry state is
                               // RECOVERY_CHECK, this
                               // contains the PAL_RESET
                               // return address, and if entry
                               // state is RESET, this contains
                               // address for PAL_authentication
                               // call.
  UINT64 SelfTestStatus;       // GR35 from PALE_EXIT state,
                               // See 'Self Test State' in
                               // SwDevMan Fig 11-10 and
                               // Table 11-6.
  UINT64 SelfTestControl;      // GR37 from PALE_EXIT state:
                               // See 'Self Test Control' in
                               // SwDevMan Fig 11-11.
  UINT64 MemoryBufferRequired; // See GR38 Reset Layout
                               // in SwDevMan Table 11-2.
} ITANIUM_HANDOFF_STATUS;
```

Consult the **PALE_RESET** Exit State in Software Development Manual for Itanium regarding an interpretation of these fields.

Status Codes Returned

EFI_SUCCESS	The data was successfully returned.
EFI_BUFFER_TOO_SMALL	The buffer was too small. The current buffer size needed to hold the record is returned in *StructureSize*.

8.3.1.1 SEC Platform Information 2 PPI (Optional)

EFI_SEC_PLATFORM_INFORMATION2_PPI (Optional)

Summary

This service is the primary handoff state into the PEI Foundation. The Security (SEC) component creates the early, transitory memory environment and also encapsulates knowledge of at least the location of the Boot Firmware Volume (BFV).

GUID

```
#define EFI_SEC_PLATFORM_INFORMATION2_GUID \
{0x9e9f374b, 0x8f16, 0x4230,
{ 0x98, 0x24, 0x58, 0x46, 0xee, 0x76, 0x6a, 0x97}};
```

Prototype

```
typedef struct _EFI_SEC_PLATFORM_INFORMATION2_PPI {
  EFI_SEC_PLATFORM_INFORMATION2 PlatformInformation2;
} EFI_SEC_PLATFORM_INFORMATION2_PPI;
```

Parameters

PlatformInformation2

Conveys state information out of the SEC phase into PEI for many CPU's. See the **PlatformInformation2()** function description.

Description

This service abstracts platform-specific information for many CPU's. It is the multi-processor equivalent of *PlatformInformation* for implementations that synchronize some, if not all CPU's in the SEC phase.

EFI_SEC_PLATFORM_INFORMATION2_PPI.PlatformInformation2()

Summary

This service is the single member of the **EFI_SEC_PLATFORM_INFORMATION2_PPI** that conveys state information out of the Security (SEC) phase into PEI.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SEC_PLATFORM_INFORMATION2) (
  IN CONST EFI_PEI_SERVICES   **PeiServices,
  IN OUT UINT64               *StructureSize,
  OUT EFI_SEC_PLATFORM_INFORMATION_RECORD2
                              *PlatformInformationRecord2
);
```

Parameters

PeiServices

Pointer to the PEI Services Table.

StructureSize

Pointer to the variable describing size of the input buffer.

PlatformInformationRecord2

Pointer to the **EFI_SEC_PLATFORM_INFORMATION_RECORD2**. Type **EFI_SEC_PLATFORM_INFORMATION_RECORD2** is defined in “Related Definitions” below.

Description

This service is published by the SEC phase.

Related Definitions

```
//*******************************************************
// EFI_SEC_PLATFORM_INFORMATION_RECORD2
//*******************************************************

typedef struct {
        UINT32                              CpuLocation;
        EFI_SEC_PLATFORM_INFORMATION_RECORD InfoRecord;
    } EFI_SEC_PLATFORM_INFORMATION_CPU;
```

```
typedef struct {
    UINT32                              NumberOfCpus.
    EFI_SEC_PLATFORM_INFORMATION_CPU    CpuInstance [1];
} EFI_SEC_PLATFORM_INFORMATION_RECORD2;
```

The CPU location would be the local API ID.

Status Codes Returned

EFI_SUCCESS	The data was successfully returned.
EFI_BUFFER_TOO_SMALL	The buffer was too small. The current buffer size needed to hold the record is returned in *StructureSize*.

8.3.2 Loaded Image PPI (Optional)

EFI_PEI_LOADED_IMAGE_PPI

Summary

Notifies other drivers of the PEIM being initialized by the PEI Dispatcher.

GUID

```
#define EFI_PEI_LOADED_IMAGE_PPI_GUID \
  { 0xc1fcd448, 0x6300, 0x4458, \
  0xb8, 0x64, 0x28, 0xdf, 0x1, 0x53, 0x64, 0xbc }
```

Prototype

```
typedef struct _EFI_PEI_LOADED_IMAGE_PPI {
  EFI_PHYSICAL_ADDRESS    ImageAddress,
  UINT64                  ImageSize,
  EFI_PEI_FILE_HANDLE     FileHandle
} EFI_PEI_LOADED_IMAGE_PPI;
```

Parameters

ImageAddress

Address of the image at the address where it will be executed.

ImageSize

Size of the image as it will be executed.

FileHandle

File handle from which the image was loaded. Can be NULL, indicating the image was not loaded from a handle.

Description

This interface is installed by the PEI Dispatcher after the image has been loaded and after all security checks have been performed, to notify other PEIMs of the files which are being loaded.

Note: *The same PEIM may be initialized twice.*

8.3.3 SEC HOB PPI

EFI_SEC_HOB_DATA_PPI

Summary

This PPI allows the SEC code to install HOBs into the HOB list.

GUID

```
#define EFI_SEC_HOB_DATA_PPI_GUID \
{0x3ebdaf20, 0x6667, 0x40d8,\
{0xb4, 0xee, 0xf5, 0x99, 0x9a, 0xc1, 0xb7, 0x1f}};
```

Protocol Interface Structure

```
typedef struct _EFI_SEC_HOB_DATA_PPI {
 EFI_SEC_HOB_DATA_GET GetHobs;
} EFI_SEC_HOB_DATA_PPI;
```

Parameters

GetHobs

Retrieves a list of HOBs to install into the PEI HOB list.

Description

This PPI provides a way for the SEC code to pass zero or more HOBs in a HOB list.

EFI_SEC_HOB_DATA_PPI.GetHobs()

Summary

Return a pointer to a buffer containing zero or more HOBs that will be installed into the PEI HOB List.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_SEC_HOB_DATA_GET) (
  IN     CONST EFI_SEC_HOB_DATA_PPI *This,
  OUT     EFI_HOB_GENERIC_HEADER    **HobList
  );
```

Parameters

This

Pointer to this PPI structure.

HobList

A pointer to a returned pointer to zero or more HOBs. If no HOBs are to be returned, then the returned pointer is a pointer to a HOB of type **EFI_HOB_TYPE_END_OF_HOB_LIST**.

Description

This function returns a pointer to a pointer to zero or more HOBs, terminated with a HOB of type **EI_HOB_TYPE_END_OF_HOB_LIST**.

Note: *The HobList must not contain a* **EFI_HOB_HANDOFF_INFO_TABLE** *HOB (PHIT) HOB.*

Note: *The HOBs pointed to by HobList must be formed as described in section 4.5.2 of Volume 3, "HOB Construction Rules" including the requirement that the list start on an 8-byte boundary.*

Status Codes

EFI_SUCCESS	This function completed successfully.
EFI_UNSUPPORTED	No HOBS are available.

8.3.4 Recovery

This section contains the definitions of the PPIs that are required on platforms that support firmware recovery. The table below explains the organization of this section and lists the PPIs that are defined in this section.

Table 17. Organization of the Code Definitions Section

Section	Summary	PPI Definition
Recovery Module PPI	Describes the main Recovery Module PPI.	EFI_PEI_RECOVERY_MODULE_PPI
Device Recovery Module PPI	Describes the Device Recovery Module PPI.	EFI_PEI_DEVICE_RECOVERY_MODULE_PPI
Device Recovery Block I/O PPI	Describes the Device Recovery Block I/O PPI. This section is device specific and addresses the most common form of recovery media—block I/O devices such as legacy floppy, CD-ROM, or IDE devices.	EFI_PEI_RECOVERY_BLOCK_IO_PPI

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_PEI_BLOCK_IO_MEDIA**
- **EFI_PEI_BLOCK_DEVICE_TYPE**
- **EFI_PEI_LBA**

8.3.4.1 Recovery Module PPI

EFI_PEI_RECOVERY_MODULE_PPI

Summary

Finds and loads the recovery files.

GUID

```
#define EFI_PEI_RECOVERY_MODULE_PPI_GUID \
  {0xFB6D9542, 0x612D, 0x4f45, 0x87, 0x2F, 0x5C, \
  0xFF, 0x52, 0xE9, 0x3D, 0xCF}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RECOVERY_MODULE_PPI {
  EFI_PEI_LOAD_RECOVERY_CAPSULE         LoadRecoveryCapsule;
} EFI_PEI_RECOVERY_MODULE_PPI;
```

Parameters

LoadRecoveryCapsule

Loads a DXE binary capsule into memory.

Description

This module has many roles and is responsible for the following:

1. Calling the driver recovery PPI **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetNumberRecoveryCapsules()** to determine if one or more DXE recovery entities exist.
2. If no capsules exist, then performing appropriate error handling.
3. Allocating a buffer of *MaxRecoveryCapsuleSize* as determined by **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetRecoveryCapsuleInfo()** or larger.
4. Determining the policy in which DXE recovery capsules are loaded.
5. Calling the driver recovery PPI **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** for capsule number x.
6. If the load failed, performing appropriate error handling.
7. Performing security checks for a loaded DXE recovery capsule.
8. If the security checks failed, then logging the failure in a data HOB.
9. If the security checks failed, then determining the next **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** capsule number; otherwise, go to step 11.
10. If more DXE recovery capsules exist, then go to step 5; otherwise, perform error handling.
11. Decomposing the capsule loaded by **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** into its components. It is assumed that the path parameters are redundant for recovery and Setup parameters are either redundant or canned.
12. Invalidating all HOB entries for updateable firmware volume entries. This invalidation prevents possible errant drivers from being executed.
13. Updating the HOB table with the recovery DXE firmware volume information generated from the capsule decomposition.
14. Returning to the PEI Dispatcher.

EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()

Summary

Loads a DXE capsule from some media into memory and updates the HOB table with the DXE firmware volume information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_LOAD_RECOVERY_CAPSULE) (
  IN EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_RECOVERY_MODULE_PPI      *This
  );
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1*.

This

Indicates the **EFI_PEI_RECOVERY_MODULE_PPI** instance.

Description

This function, by whatever mechanism, retrieves a DXE capsule from some device and loads it into memory. Note that the published interface is device neutral.

Status Codes Returned

EFI_SUCCESS	The capsule was loaded correctly.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

8.3.4.2 Device Recovery Module PPI

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI

Summary

Presents a standard interface to **EFI_PEI_RECOVERY_MODULE_PPI**, regardless of the underlying device(s).

GUID

```
#define EFI_PEI_DEVICE_RECOVERY_MODULE_PPI_GUID    \
  { 0x0DE2CE25, 0x446A, 0x45a7, 0xBF, 0xC9, 0x37, 0xDA, \
  0x26, 0x34, 0x4B, 0x37}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI {
  EFI_PEI_DEVICE_GET_NUMBER_RECOVERY_CAPSULE
                                GetNumberRecoveryCapsules;
  EFI_PEI_DEVICE_GET_RECOVERY_CAPSULE_INFO
                                GetRecoveryCapsuleInfo;
  EFI_PEI_DEVICE_LOAD_RECOVERY_CAPSULE
                                LoadRecoveryCapsule;
} EFI_PEI_DEVICE_RECOVERY_MODULE_PPI;
```

Parameters

GetNumberRecoveryCapsules

Returns the number of DXE capsules that were found. See the **GetNumberRecoveryCapsules()** function description.

GetRecoveryCapsuleInfo

Returns the capsule image type and the size of a given image. See the **GetRecoveryCapsuleInfo()** function description.

LoadRecoveryCapsule

Loads a DXE capsule into memory. See the **LoadRecoveryCapsule()** function description.

Description

The role of this module is to present a standard interface to **EFI_PEI_RECOVERY_MODULE_PPI**, regardless of the underlying device(s). The interface does the following:

- Reports the number of recovery DXE capsules that exist on the associated device(s)
- Finds the requested firmware binary capsule
- Loads that capsule into memory

A device can be either a group of devices, such as a block device, or an individual device. The module determines the internal search order, with capsule number 1 as the highest load priority and number N as the lowest priority.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. GetNumberRecoveryCapsules()

Summary

Returns the number of DXE capsules residing on the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_GET_NUMBER_RECOVERY_CAPSULE) (
  IN EFI_PEI_SERVICES                           **PeiServices,
  IN struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI   *This,
  OUT UINTN                                    *NumberRecoveryCapsules
  );
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1*.

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

NumberRecoveryCapsules

Pointer to a caller-allocated **UINTN**. On output, **NumberRecoveryCapsules* contains the number of recovery capsule images available for retrieval from this PEIM instance.

Description

This function, by whatever mechanism, searches for DXE capsules from the associated device and returns the number and maximum size in bytes of the capsules discovered. Entry 1 is assumed to be the highest load priority and entry N is assumed to be the lowest priority.

Status Codes Returned

EFI_SUCCESS	One or more capsules were discovered.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. GetRecoveryCapsuleInfo()

Summary

Returns the size and type of the requested recovery capsule.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_GET_RECOVERY_CAPSULE_INFO) (
  IN  EFI_PEI_SERVICES                        **PeiServices,
  IN  struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI *This,
  IN  UINTN                                  CapsuleInstance,
  OUT UINTN                                   *Size,
  OUT EFI_GUID                                *CapsuleType
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1*.

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

CapsuleInstance

Specifies for which capsule instance to retrieve the information. This parameter must be between one and the value returned by **GetNumberRecoveryCapsules()** in *NumberRecoveryCapsules*.

Size

A pointer to a caller-allocated **UINTN** in which the size of the requested recovery module is returned.

CapsuleType

A pointer to a caller-allocated **EFI_GUID** in which the type of the requested recovery capsule is returned. The semantic meaning of the value returned is defined by the implementation. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

Description

This function returns the size and type of the capsule specified by CapsuleInstance.

Status Codes Returned

EFI_SUCCESS	One or more capsules were discovered.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. LoadRecoveryCapsule()

Summary

Loads a DXE capsule from some media into memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_LOAD_RECOVERY_CAPSULE) (
  IN EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI *This,
  IN UINTN                                   CapsuleInstance,
  OUT VOID                                   *Buffer
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1.*

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

CapsuleInstance

Specifies which capsule instance to retrieve.

Buffer

Specifies a caller-allocated buffer in which the requested recovery capsule will be returned.

Description

This function, by whatever mechanism, retrieves a DXE capsule from some device and loads it into memory. Note that the published interface is device neutral.

Status Codes Returned

EFI_SUCCESS	The capsule was loaded correctly.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	The requested recovery DXE capsule cannot be found.

8.3.4.3 Device Recovery Block I/O PPI

The Recovery Module PPI and the Device Recovery Module PPI subsections earlier in Code Definitions are device neutral. This section is device specific and addresses the most common form of recovery media-block I/O devices such as legacy floppy, CD-ROM, or IDE devices.

The Recovery Block I/O PPI is used to access block devices. Because the Recovery Block I/O PPIs that are provided by the PEI ATAPI driver and PEI legacy floppy driver are the same, here we define a set of general PPIs for both drivers to use.

EFI_PEI_RECOVERY_BLOCK_IO_PPI

Summary

Provides the services required to access a block I/O device during PEI recovery boot mode.

GUID

```
#define EFI_PEI_RECOVERY_BLOCK_IO_PPI_GUID \
  { 0x695d8aa1, 0x42ee, 0x4c46, 0x80, 0x5c,0x6e, 0xa6, \
  0xbc, 0xe7, 0x99, 0xe3 }
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI {
  EFI_PEI_GET_NUMBER_BLOCK_DEVICES   GetNumberOfBlockDevices;
  EFI_PEI_GET_DEVICE_MEDIA_INFORMATION
                                     GetBlockDeviceMediaInfo;
  EFI_PEI_READ_BLOCKS                ReadBlocks;
} EFI_PEI_RECOVERY_BLOCK_IO_PPI;
```

Parameters

GetNumberOfBlockDevices

Gets the number of block I/O devices that the specific block driver manages. See the **GetNumberOfBlockDevices()** function description.

GetBlockDeviceMediaInfo

Gets the specified media information. See the **GetBlockDeviceMediaInfo()** function description.

ReadBlocks

Reads the requested number of blocks from the specified block device. See the **ReadBlocks()** function description.

Description

This function provides the services that are required to access a block I/O device during PEI recovery boot mode.

EFI_PEI_RECOVERY_BLOCK_IO_PPI. GetNumberOfBlockDevices()

Summary

Gets the count of block I/O devices that one specific block driver detects.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_NUMBER_BLOCK_DEVICES) (
  IN  EFI_PEI_SERVICES                     **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI   *This,
  OUT UINTN                                 *NumberBlockDevices
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1*.

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

NumberBlockDevices

The number of block I/O devices discovered.

Description

This function is used for getting the count of block I/O devices that one specific block driver detects. To the PEI ATAPI driver, it returns the number of all the detected ATAPI devices it detects during the enumeration process. To the PEI legacy floppy driver, it returns the number of all the legacy devices it finds during its enumeration process. If no device is detected, then the function will return zero.

Status Codes Returned

EFI_SUCCESS	Operation performed successfully

EFI_PEI_RECOVERY_BLOCK_IO_PPI.GetBlockDeviceMediaInfo()

Summary

Gets a block device's media information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_DEVICE_MEDIA_INFORMATION) (
  IN  EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI    *This,
  IN UINTN                                    DeviceIndex,
  OUT EFI_PEI_BLOCK_IO_MEDIA                  *MediaInfo
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in section 3.2.1.

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

MediaInfo

The media information of the specified block media. Type **EFI_PEI_BLOCK_IO_MEDIA** is defined in "Related Definitions" below. The caller is responsible for the ownership of this data structure.

Note: *This structure describes an enumeration of possible block device types. This enumeration exists because no device paths are actually passed across interfaces that describe the type or class of hardware that is publishing the block I/O interface. This enumeration will allow for policy decisions in the Recovery PEIM, such as "Try to recover from legacy floppy first, LS-120 second, CD-ROM third." If there are multiple partitions abstracted by a given device type, they should be reported in ascending order; this order also applies to nested partitions, such as legacy MBR, where the outermost partitions would have precedence in the reporting order. The same logic applies to systems such as IDE that have precedence relationships like "Master/Slave" or "Primary/ Secondary"; the master device should be reported first, the slave second.*

Description

This function will provide the caller with the specified block device's media information. If the media changes, calling this function will update the media information accordingly.

Related Definitions

```
//****************************************************
// EFI_PEI_BLOCK_IO_MEDIA
//****************************************************

typedef struct {
  EFI_PEI_BLOCK_DEVICE_TYPE   DeviceType;
  BOOLEAN                     MediaPresent;
  UINTN                       LastBlock;
  UINTN                       BlockSize;
} PEI_BLOCK_IO_MEDIA;
```

DevType

The type of media device being referenced by DeviceIndex. Type **EFI_PEI_BLOCK_DEVICE_TYPE** is defined below.

MediaPresent

A flag that indicates if media is present. This flag is always set for nonremovable media devices.

LastBlock

The last logical block that the device supports.

BlockSize

The size of a logical block in bytes.

```
//************************************************************
// EFI_PEI_BLOCK_DEVICE_TYPE
//************************************************************
typedef enum {
  LegacyFloppy    =   0,
  IdeCDROM        =   1,
  IdeLS120        =   2,
  UsbMassStorage  =   3,
  SD              =   4,
  EMMC            =   5,
  UfsDevice       =   6,
  MaxDeviceType
} EFI_PEI_BLOCK_DEVICE_TYPE;
```

Status Codes Returned

EFI_SUCCESS	Media information about the specified block device was obtained successfully.
EFI_DEVICE_ERROR	Cannot get the media information due to a hardware error.

EFI_PEI_RECOVERY_BLOCK_IO_PPI.ReadBlocks()

Summary

Reads the requested number of blocks from the specified block device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_READ_BLOCKS) (
  IN  EFI_PEI_SERVICES                      **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI  *This,
  IN UINTN                                  DeviceIndex,
  IN EFI_PEI_LBA                            StartLBA,
  IN UINTN                                  BufferSize,
  OUT VOID                                  *Buffer
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in *section 3.2.1*.

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

StartLBA

The starting logical block address (LBA) to read from on the device. Type **EFI_PEI_LBA** is defined in "Related Definitions" below.

BufferSize

The size of the *Buffer* in bytes. This number must be a multiple of the intrinsic block size of the device.

Buffer

A pointer to the destination buffer for the data. The caller is responsible for the ownership of the buffer.

Description

The function reads the requested number of blocks from the device. All the blocks are read, or an error is returned. If there is no media in the device, the function returns **EFI_NO_MEDIA**.

Related Definitions

```
//*****************************************************
// EFI_PEI_LBA
//*****************************************************

typedef UINT64             EFI_PEI_LBA;
```

EFI_PEI_LBA is the **UINT64** LBA number.

Status Codes Returned

EFI_SUCCESS	The data was read correctly from the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to perform the read operation.
EFI_INVALID_PARAMETER	The read request contains LBAs that are not valid, or the buffer is not properly aligned.
EFI_NO_MEDIA	There is no media in the device.
EFI_BAD_BUFFER_SIZE	The *BufferSize* parameter is not a multiple of the intrinsic block size of the device.

8.3.5 EFI PEI Recovery Block IO2 PPI

EFI_PEI_RECOVERY_BLOCK_IO2_PPI

Summary

Provides the services required to access a block I/O device during PEI recovery boot mode.

GUID

```
#define EFI_PEI_RECOVERY_BLOCK_IO2_PPI_GUID \
    { 0x26cc0fad, 0xbeb3, 0x478a,\
     { 0x91, 0xb2, 0xc, 0x18, 0x8f, 0x72, 0x61, 0x98 } }
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RECOVERY_BLOCK_IO2_PPI {
  UINT64                             Revision;
  EFI_PEI_GET_NUMBER_BLOCK_DEVICES2  GetNumberOfBlockDevices;
  EFI_PEI_GET_DEVICE_MEDIA_INFORMATION2
                                    GetBlockDeviceMediaInfo;
  EFI_PEI_READ_BLOCKS2               ReadBlocks;
} EFI_PEI_RECOVERY_BLOCK_IO2_PPI;
```

Parameters

Revision

The revision to which the interface adheres. All future revisions must be backwards compatible.

GetNumberOfBlockDevices

Gets the number of block I/O devices that the specific block driver manages. See the **GetNumberOfBlockDevices()** function description.

GetBlockDeviceMediaInfo

Gets the specified media information. See the **GetBlockDeviceMediaInfo()** function description.

ReadBlocks

Reads the requested number of blocks from the specified block device. See the **ReadBlocks()** function description.

Related Definitions

```
#define EFI_PEI_RECOVERY_BLOCK_IO2_PPI_REVISION 0x00010000
```

Description

This function provides the services that are required to access a block I/O device during PEI recovery boot mode.

EFI_PEI_RECOVERY_BLOCK_IO2_PPI.GetNumberOfBlockDevices()

Summary

Gets the count of block I/O devices that one specific block driver detects.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_NUMBER_BLOCK_DEVICES2) (
  IN   EFI_PEI_SERVICES                        **PeiServices,
  IN   EFI_PEI_RECOVERY_BLOCK_IO2_PPI          *This,
  OUT  UINTN                                   *NumberBlockDevices
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

NumberBlockDevices

The number of block I/O devices discovered.

Description

This function is used for getting the count of block I/O devices that one specific block driver detects. To the PEI ATAPI driver, it returns the number of all the detected ATAPI devices it detects during the enumeration process. To the PEI legacy floppy driver, it returns the number of all the legacy devices it finds during its enumeration process. If no device is detected, then the function will return zero.

Status Codes Returned

EFI_SUCCESS	Operation performed successfully

EFI_PEI_RECOVERY_BLOCK_IO2_PPI.GetBlockDeviceMediaInfo()

Summary

Gets a block device's media information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_DEVICE_MEDIA_INFORMATION2) (
  IN  EFI_PEI_SERVICES                          **PeiServices,
  IN EFI_PEI_RECOVERY_BLOCK_IO2_PPI             *This,
  IN UINTN                                      DeviceIndex,
  OUT EFI_PEI_BLOCK_IO2_MEDIA                   *MediaInfo
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

MediaInfo

The media information of the specified block media. Type **EFI_PEI_BLOCK_IO2_MEDIA** is defined in "Related Definitions" below. The caller is responsible for the ownership of this data structure.

Note that this structure describes an enumeration of possible block device types. This enumeration exists because no device paths are actually passed across interfaces that describe the type or class of hardware that is publishing the block I/O interface. This enumeration will allow for policy decisions in the Recovery PEIM, such as "Try to recover from legacy floppy first, USB mass storage device second, CD-ROM third." If there are multiple partitions abstracted by a given device type, they should be reported in ascending order; this order also applies to nested partitions, such as legacy MBR, where the outermost partitions would have precedence in the reporting order. The same logic applies to systems such as IDE that have precedence relationships like "Master/Slave" or "Primary/Secondary"; the master device should be reported first, the slave second.

Description

This function will provide the caller with the specified block device's media information. If the media changes, calling this function will update the media information accordingly.

Related Definitions

```
//***************************************************
// EFI_PEI_BLOCK_IO2_MEDIA
//***************************************************

typedef struct {
  UINT8          InterfaceType;
  BOOLEAN        RemovablaMedia;
  BOOLEAN        MediaPresent;
  BOOLEAN        ReadOnly;
  UINT32         BlockSize;
  EFI_PEI_LBA    LastBlock;
} PEI_BLOCK_IO2_MEDIA;
```

InterfaceType

A type of interface that the device being referenced by *DeviceIndex* is attached to. This field re-uses Messaging Device Path Node sub-type values as defined by Section "9.3.5 Messaging Device Path" of *UEFI Specification*. When more than one sub-type is associated with the interface, sub-type with the smallest number must be used. For example, *InterfaceType* must be set to 5 for USB devices.

RemovablaMedia

A flag that indicates if media is removable.

MediaPresent

A flag that indicates if media is present. This flag is always set for non-removable media devices.

ReadOnly

A flag that indicates if media is read-only.

LastBlock

The last logical block that the device supports.

BlockSize

The size of a logical block in bytes. Type **EFI_PEI_LBA** is defined below.

Related Definitions

```
//*****************************************************
// EFI_PEI_LBA
//*****************************************************
typedef UINT64             EFI_PEI_LBA;
```

EFI_PEI_LBA is the **UINT64** LBA number.

Status Codes Returned

EFI_SUCCESS	Media information about the specified block device was obtained successfully.
EFI_DEVICE_ERROR	Cannot get the media information due to a hardware error.

EFI_PEI_RECOVERY_BLOCK_IO2_PPI.ReadBlocks()

Summary

Reads the requested number of blocks from the specified block device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_READ_BLOCKS2) (
  IN  EFI_PEI_SERVICES                       **PeiServices,
  IN EFI_PEI_RECOVERY_BLOCK_IO2_PPI          *This,
  IN UINTN                                   DeviceIndex,
  IN EFI_PEI_LBA                             StartLBA,
  IN UINTN                                   BufferSize,
  OUT VOID                                   *Buffer
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

StartLBA

The starting logical block address (LBA) to read from on the device. Type **EFI_PEI_LBA** is defined in in the **GetBlockDeviceMediaInfo()** function description.

BufferSize

The size of the *Buffer* in bytes. This number must be a multiple of the intrinsic block size of the device.

Buffer

A pointer to the destination buffer for the data. The caller is responsible for the ownership of the buffer.

Description

The function reads the requested number of blocks from the device. All the blocks are read, or an error is returned. If there is no media in the device, the function returns **EFI_NO_MEDIA**.

Status Codes Returned

EFI_SUCCESS	The data was read correctly from the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to perform the read operation.
EFI_INVALID_PARAMETER	The read request contains LBAs that are not valid, or the buffer is not properly aligned.
EFI_NO_MEDIA	There is no media in the device.
EFI_BAD_BUFFER_SIZE	The *BufferSize* parameter is not a multiple of the intrinsic block size of the device.

8.3.6 EFI PEI Vector Handoff Info PPI

EFI_PEI_VECTOR_HANDOFF_INFO_PPI (Optional)

Summary

The PPI that describes an array of interrupt and/or exception vectors that are in use and need to persist.

GUID

```
#define EFI_PEI_VECTOR_HANDOFF_INFO_PPI_GUID \
  { 0x3cd652b4, 0x6d33, 0x4dce, \
  { 0x89, 0xdb, 0x83, 0xdf, 0x97, 0x66, 0xfc, 0xca } }
```

Protocol Interface Structure

```
typedef struct _EFI_PEI_VECTOR_HANDOFF_INFO_PPI {
  EFI_VECTOR_HANDOFF_INFO   *Info;
} EFI_PEI_VECTOR_HANDOFF_INFO_PPI;
```

Parameters

Info

Pointer to an array of interrupt and /or exception vectors.

Description

This is an optional PPI that may be produced by SEC. If present, it provides a description of the interrupt and/or exception vectors that were established in the SEC Phase and need to persist into PEI and DXE. This PPI is an array of entries that is terminated by an entry whose *Attribute* is set to **EFI_VECTOR_HANDOFF_LAST_ENTRY**.

If *Attribute* is set to **EFI_VECTOR_HANDOFF_DO_NOT_HOOK**, then the associated handler for *VectorNumber* must be preserved in PEI and DXE.

If *Attribute* is set to **EFI_VECTOR_HANDOFF_HOOK_BEFORE**, then *VectorNumber* may be used in PEI and DXE, but new handlers must be invoked prior to when the existing handler is called.

If *Attribute* is set to **EFI_VECTOR_HANDOFF_HOOK_AFTER**, then the associated *VectorNumber* may be used in PEI and DXE, but new handlers must be called after the existing handler is called.

EFI_PEI_VECTOR_HANDOFF_INFO_PPI_GUID can also be used in the PEI Phase to build a GUIDed HOB that contains an array of **EFI_VECTOR_HANDOFF_INFO** entries that describes the interrupt and/or exception vectors in use in the PEI Phase. This may be identical to the array passed up from SEC, or it could be an array that is augmented with additional vectors used in PEI Phase.

Related Definitions

```
//
// System configuration table entry that points to the table
// in case an entity in DXE wishes to update/change the vector
// table contents.
//
#define EFI_VECTOR_HANDOFF_TABLE_GUID \
{0x996ec11c, 0x5397, 0x4e73, \
 {0xb5, 0x8f, 0x82, 0x7e, 0x52, 0x90, 0x6d, 0xef}}

typedef struct {
  UINT32     VectorNumber;
  UINT32     Attribute;
  EFI_GUID   Owner;
} EFI_VECTOR_HANDOFF_INFO;
```

Parameters

VectorNumber

The interrupt or exception vector that is in use and must be preserved.

Attribute

A bitmask that describes the attributes of the interrupt or exception vector.

Owner

The GUID identifies the party who created the entry.   For the **EFI_VECTOR_HANDOFF_DO_NOT_HOOK** case, this establishes the single owner.

8.3.7 CPU I/O PPI (Optional)

EFI_PEI_CPU_IO_PPI (Optional)

If the service is not available, the PEI Core service **EFI_PEI_CPU_IO_PPI** **CpuIo* member functions will have a dummy function that return **EFI_NOT_AVAILABLE_YET**;

Summary

This PPI is installed by some platform or chipset-specific PEIM that abstracts the processor-visible I/O operations.

GUID

```
#define EFI_PEI_CPU_IO_PPI_INSTALLED_GUID  \
  {0xe6af1f7b, 0xfc3f, 0x46da, 0xa8, 0x28, 0xa3, 0xb4, \
  0x57, 0xa4, 0x42, 0x82}
```

This is an indicator GUID without any data. It represents the fact that a PEIM has written the address of the **EFI_PEI_CPU_IO_PPI** into the **EFI_PEI_SERVICES** table.

PPI Interface Structure

```
typedef
struct _EFI_PEI_CPU_IO_PPI {
  EFI_PEI_CPU_IO_PPI_ACCESS          Mem;
  EFI_PEI_CPU_IO_PPI_ACCESS          Io;
  EFI_PEI_CPU_IO_PPI_IO_READ8        IoRead8;
  EFI_PEI_CPU_IO_PPI_IO_READ16       IoRead16;
  EFI_PEI_CPU_IO_PPI_IO_READ32       IoRead32;
  EFI_PEI_CPU_IO_PPI_IO_READ64       IoRead64;
  EFI_PEI_CPU_IO_PPI_IO_WRITE8       IoWrite8;
  EFI_PEI_CPU_IO_PPI_IO_WRITE16      IoWrite16;
  EFI_PEI_CPU_IO_PPI_IO_WRITE32      IoWrite32;
  EFI_PEI_CPU_IO_PPI_IO_WRITE64      IoWrite64;
  EFI_PEI_CPU_IO_PPI_MEM_READ8       MemRead8;
  EFI_PEI_CPU_IO_PPI_MEM_READ16      MemRead16;
  EFI_PEI_CPU_IO_PPI_MEM_READ32      MemRead32;
  EFI_PEI_CPU_IO_PPI_MEM_READ64      MemRead64;
  EFI_PEI_CPU_IO_PPI_MEM_WRITE8      MemWrite8;
  EFI_PEI_CPU_IO_PPI_MEM_WRITE16     MemWrite16;
  EFI_PEI_CPU_IO_PPI_MEM_WRITE32     MemWrite32;
  EFI_PEI_CPU_IO_PPI_MEM_WRITE64     MemWrite64;
} EFI_PEI_CPU_IO_PPI;
```

Parameters

Mem

Collection of memory-access services. See the **Mem()** function description. Type **EFI_PEI_CPU_IO_PPI_ACCESS** is defined in "Related Definitions" below.

Io

Collection of I/O-access services. See the **Io()** function description. Type **EFI_PEI_CPU_IO_PPI_ACCESS** is defined in "Related Definitions" below.

IoRead8

8-bit read service. See the **IoRead8()** function description.

IoRead16

16-bit read service. See the **IoRead16()** function description.

IoRead32

32-bit read service. See the **IoRead32()** function description.

IoRead64

64-bit read service. See the **IoRead64()** function description.

IoWrite8

8-bit write service. See the **IoWrite8()** function description.

IoWrite16

16-bit write service. See the **IoWrite16()** function description.

IoWrite32

32-bit write service. See the **IoWrite32()** function description.

IoWrite64

64-bit write service. See the **IoWrite64()** function description.

MemRead8

8-bit read service. See the **MemRead8()** function description.

MemRead16

16-bit read service. See the **MemRead16()** function description.

MemRead32

32-bit read service. See the **MemRead32()** function description.

MemRead64

64-bit read service. See the **MemRead64()** function description.

MemWrite8

8-bit write service. See the **MemWrite8()** function description.

MemWrite16

16-bit write service. See the **MemWrite16()** function description.

MemWrite32

32-bit write service. See the **MemWrite32()** function description.

MemWrite64

64-bit write service. See the **MemWrite64()** function description.

Description

This PPI provides a set of memory- and I/O-based services. The perspective of the services is that of the processor, not the bus or system.

Related Definitions

```
//*******************************************************
// EFI_PEI_CPU_IO_PPI_ACCESS
//*******************************************************

typedef
struct {
  EFI_PEI_CPU_IO_PPI_IO_MEM           Read;
  EFI_PEI_CPU_IO_PPI_IO_MEM           Write;
} EFI_PEI_CPU_IO_PPI_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_PEI_CPU_IO_PPI.Mem()

Summary

Memory-based access services.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  CONST EFI_PEI_SERVICES         **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI       *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH        Width,
  IN  UINT64                          Address,
  IN  UINTN                           Count,
  IN  OUT VOID                        *Buffer
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Width

The width of the access. Enumerated in bytes. Type **EFI_PEI_CPU_IO_PPI_WIDTH** is defined in "Related Definitions" below.

Address

The physical address of the access.

Count

The number of accesses to perform.

Buffer

A pointer to the buffer of data.

Description

The **Mem()** function provides a list of memory-based accesses.

Related Definitions

```
//*******************************************************
// EFI_PEI_CPU_IO_PPI_WIDTH
//*******************************************************

typedef enum {
  EfiPeiCpuIoWidthUint8,
  EfiPeiCpuIoWidthUint16,
  EfiPeiCpuIoWidthUint32,
  EfiPeiCpuIoWidthUint64,
  EfiPeiCpuIoWidthFifoUint8,
  EfiPeiCpuIoWidthFifoUint16,
  EfiPeiCpuIoWidthFifoUint32,
  EfiPeiCpuIoWidthFifoUint64,
  EfiPeiCpuIoWidthFillUint8,
  EfiPeiCpuIoWidthFillUint16,
  EfiPeiCpuIoWidthFillUint32,
  EfiPeiCpuIoWidthFillUint64,
  EfiPeiCpuIoWidthMaximum
} EFI_PEI_CPU_IO_PPI_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_YET_AVAILABLE	The service has not been installed.

EFI_PEI_CPU_IO_PPI.Io()

Summary

I/O-based access services.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  CONST EFI_PEI_SERVICES          **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH        Width,
  IN  UINT64                          Address,
  IN  UINTN                           Count,
  IN  OUT VOID                        *Buffer
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Width

The width of the access. Enumerated in bytes. Type **EFI_PEI_CPU_IO_PPI_WIDTH** is defined in **Mem()**.

Address

The physical address of the access.

Count

The number of accesses to perform.

Buffer

A pointer to the buffer of data.

Description

The **Io()** function provides a list of I/O-based accesses. Input or output data can be found in the last argument.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_NOT_YET_AVAILABLE	The service has not been installed.

EFI_PEI_CPU_IO_PPI.IoRead8()

Summary

8-bit I/O read operations.

Prototype

```
typedef
UINT8
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_READ8) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                      Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **IoRead8()** function returns an 8-bit value from the I/O space.

EFI_PEI_CPU_IO_PPI.IoRead16()

Summary

16-bit I/O read operations.

Prototype

```
typedef
UINT16
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_READ16) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **IoRead16()** function returns a 16-bit value from the I/O space.

EFI_PEI_CPU_IO_PPI.IoRead32()

Summary

32-bit I/O read operations.

Prototype

```
typedef
UINT32
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_READ32) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **IoRead32()** function returns a 32-bit value from the I/O space.

EFI_PEI_CPU_IO_PPI.IoRead64()

Summary

64-bit I/O read operations.

Prototype

```
typedef
UINT64
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_READ64) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST_EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **IoRead64()** function returns a 64-bit value from the I/O space.

EFI_PEI_CPU_IO_PPI.IoWrite8()

Summary

8-bit I/O write operations.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_WRITE8) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address,
  IN  UINT8                       Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **IoWrite8()** function writes an 8-bit value to the I/O space.

EFI_PEI_CPU_IO_PPI.IoWrite16()

Summary

16-bit I/O write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_WRITE16) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                       Address,
  IN  UINT16                       Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **IoWrite16()** function writes a 16-bit value to the I/O space.

EFI_PEI_CPU_IO_PPI.IoWrite32()

Summary

32-bit I/O write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_WRITE32) (
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                      Address,
  IN  UINT32                      Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **IoWrite32()** function writes a 32-bit value to the I/O space.

EFI_PEI_CPU_IO_PPI.IoWrite64()

Summary

64-bit I/O write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_WRITE64) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address,
  IN  UINT64                      Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **IoWrite64()** function writes a 64-bit value to the I/O space.

EFI_PEI_CPU_IO_PPI.MemRead8()

Summary

8-bit memory read operations.

Prototype

```
typedef
UINT8
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_READ8) (
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                     Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **MemRead8()** function returns an 8-bit value from the memory space.

EFI_PEI_CPU_IO_PPI.MemRead16()

Summary

16-bit memory read operations.

Prototype

```
typedef
UINT16
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_READ16) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **MemRead16()** function returns a 16-bit value from the memory space.

EFI_PEI_CPU_IO_PPI.MemRead32()

Summary

32-bit memory read operations.

Prototype

```
typedef
UINT32
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_READ32) (
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                     Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **MemRead32()** function returns a 32-bit value from the memory space.

EFI_PEI_CPU_IO_PPI.MemRead64()

Summary

64-bit memory read operations.

Prototype

```
typedef
UINT64
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_READ64) (
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                     Address
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Description

The **MemRead64()** function returns a 64-bit value from the memory space.

EFI_PEI_CPU_IO_PPI.MemWrite8()

Summary

8-bit memory write operations.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_WRITE8) (
  IN  CONST EFI_PEI_SERVICES      **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI    *This,
  IN  UINT64                      Address,
  IN  UINT8                       Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **MemWrite8()** function writes an 8-bit value to the memory space.

EFI_PEI_CPU_IO_PPI.MemWrite16()

Summary

16-bit memory write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_WRITE16) (
  IN  CONST EFI_PEI_SERVICES    **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI  *This,
  IN  UINT64                     Address,
  IN  UINT16                     Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **MemWrite16()** function writes a 16-bit value to the memory space.

EFI_PEI_CPU_IO_PPI.MemWrite32()

Summary

32-bit memory write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_MEM_WRITE32) (
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI   *This,
  IN  UINT64                      Address,
  IN  UINT32                      Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **MemWrite32()** function writes a 32-bit value to the memory space.

EFI_PEI_CPU_IO_PPI.MemWrite64()

Summary

64-bit memory write operation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_WRITE64) (
  IN  CONST EFI_PEI_SERVICES       **PeiServices,
  IN  CONST EFI_PEI_CPU_IO_PPI     *This,
  IN  UINT64                       Address,
  IN  UINT64                       Data
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to local data for the interface.

Address

The physical address of the access.

Data

The data to write.

Description

The **MemWrite64()** function writes a 64-bit value to the memory space.

```
//
// Vector Handoff Info Attributes
//
#define EFI_VECTOR_HANDOFF_DO_NOT_HOOK  0x00000000
#define EFI_VECTOR_HANDOFF_HOOK_BEFORE  0x00000001
#define EFI_VECTOR_HANDOFF_HOOK_AFTER   0x00000002
#define EFI_VECTOR_HANDOFF_LAST_ENTRY   0x80000000
```

8.3.8 EFI Pei Capsule PPI

EFI_PEI_CAPSULE_PPI (Optional)

Summary

This PPI is installed by some platform or chipset-specific PEIM that abstracts handling of UEFI Capsule processing.

GUID

```
#define EFI_PEI_CAPSULE_PPI_GUID \
{0x3acf33ee, 0xd892, 0x40f4, \
 {0xa2, 0xfc, 0x38, 0x54, 0xd2, 0xe1, 0x32, 0x3d } }
```

PPI Interface Structure

```
typedef
struct _EFI_PEI_CAPSULE_PPI {
   EFI_PEI_CAPSULE_COALESCE Coalesce;
   EFI_PEI_CAPSULE_CHECK_CAPSULE_UDPATE CheckCapsuleUpdate;
   EFI_PEI_CAPSULE_CREATE_STATE         CreateState;
} EFI_PEI_CAPSULE_PPI;
```

Parameters

Coalesce

Upon determining that there is a capsule to operate on, this service will use a series of **EFI_CAPSULE_BLOCK_DESCRIPTOR** entries to determine the current location of the various capsule fragments and coalesce them into a contiguous region of system memory.

CheckCapsuleUpdate

Determine if a capsule needs to be processed. The means by which the presence of a capsule is determined is platform specific. For example, an implementation could be driven by the presence of a Capsule EFI Variable containing a list of **EFI_CAPSULE_BLOCK_DESCRIPTOR** entries. If present, return **EFI_SUCCESS**, otherwise return **EFI_NOT_FOUND**.

CreateState

The Capsule PPI service that gets called after memory is available. The capsule coalesce function, which must be called first, returns a base address and size. Once the memory init PEIM has discovered memory, it should call this function and pass in the base address and size returned by the Coalesce() function. Then this function can create a capsule HOB and return.

Description

This PPI provides several services in PEI to work with the underlying capsule capabilities of the platform. These services include the ability for PEI to coalesce a capsule from a scattered set of memory locations into a contiguous space in memory, detect if a capsule is present for processing, and once memory is available, create a HOB for the capsule.

EFI_PEI_CAPSULE_PPI.Coalesce

Summary

Coalesce the capsule

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CAPSULE_COALESCE)(
  IN EFI_PEI_SERVICES   **PeiServices,
  IN OUT VOID           **MemoryBase,
  IN OUT UINTN          *MemSize
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

MemoryBase

Pointer to the base of a block of memory into which the buffers will be coalesced. On output, this variable will hold the base address of a coalesced capsule.

MemorySize

Pointer to local data for the interface.

Description

Upon determining that there is a capsule to operate on, this service will use a series of **EFI_CAPSULE_BLOCK_DESCRIPTOR** entries to determine the current location of the various capsule fragments and coalesce them into a contiguous region of system memory.

Status Codes Returned

EFI_SUCCESS	There was no capsule, or the capsule was processed successfully.
EFI_NOT_FOUND	If: boot mode could not be determined, or the boot mode is not flash-update, or the capsule descriptors were not found.
EFI_BUFFER_TOO_SMALL	The capsule could not be coalesced in the provided memory region.

EFI_PEI_CAPSULE_CHECK_CAPSULE_UDPATE.CheckCapsuleUpdate()

Summary

Check the Capsule Update.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CAPSULE_CHECK_CAPSULE_UPDATE)(
  IN EFI_PEI_SERVICES   **PeiServices
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

Description

Determine if a capsule needs to be processed. The means by which the presence of a capsule is determined is platform specific. For example, an implementation could be driven by the presence of a Capsule EFI Variable containing a list of **EFI_CAPSULE_BLOCK_DESCRIPTOR** entries. If present, return **EFI_SUCCESS**, otherwise return **EFI_NOT_FOUND**.

Status Codes Returned

EFI_SUCCESS	If a capsule is available.
EFI_NOT_FOUND	No capsule detected.

EFI_PEI_CAPSULE_CHECK_CAPSULE_UDPATE.CapsuleCreateState( )

Summary

Create the Capsule state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CAPSULE_CREATE_STATE)(
  IN EFI_PEI_SERVICES   **PeiServices,
  IN VOID               *CapsuleBase,
  IN UINTN              CapsuleSize
  );
```

Parameters

PeiServices

Pointer to the PEI Services Table.

CapsuleBase

Address returned by the capsule coalesce function.

CapsuleSize

Value returned by the capsule coalesce function.

Description

The Capsule PPI service that gets called after memory is available. The capsule coalesce function, which must be called first, returns a base address and size. Once the memory init PEIM has discovered memory, it should call this function and pass in the base address and size returned by the Coalesce() function. Then this function can create a capsule HOB and return.

Status Codes Returned

EFI_VOLUME_CORRUPTED	*CapsuleBase* does not appear to point to a coalesced capsule.
EFI_SUCCESS	Capsule HOB was created successfully.

8.3.9 EFI MP Services PPI

EFI_MP_SERVICES_PPI (Optional)

Summary

This PPI is installed by some platform or chipset-specific PEIM that abstracts handling multiprocessor support.

GUID

```
#define EFI_MP_SERVICES_PPI_GUID \
{0xee16160a, 0xe8be, 0x47a6,\
 {0x82, 0xa, 0xc6, 0x90, 0xd, 0xb0, 0x25, 0xa } }
```

PPI Interface Structure

```
typedef
struct _EFI_MP_SERVICES_PPI {
   PEI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS
GetNumberOfProcessors;
   PEI_MP_SERVICES_GET_PROCESSOR_INFO
GetProcessorInfo;
   PEI_MP_SERVICES_STARTUP_ALL_APS
StartupAllAPs;
   PEI_MP_SERVICES_STARTUP_THIS_AP
StartupThisAP;
   PEI_MP_SERVICES_SWITCH_BSP
SwitchBSP;
   PEI_MP_SERVICES_ENABLEDISABLEAP
EnableDisableAP;
   PEI_MP_SERVICES_WHOAMI
WhoAmI;
} EFI_MP_SERVICES_PPI;
```

Parameters

GetNumberOfProcessors

Discover the number of CPU’s

GetProcessorInfo

Ascertain information on the CPU’s.

StartupAllAPs

Startup all of the application processors.

StartupThisAP

Startup the specific application processor.

SwitchBSP

Swtich the boot strap processor.

WhoAmI

Identify the currently executing processor.

Description

When installed, the MP Services Ppi produces a collection of services that are needed for MP management.

Before the PI event **END_OF_PEI** is signaled, the module that produces this protocol is required to place all APs into an idle state whenever the APs are disabled or the APs are not executing code as requested through the **StartupAllAPs()** or **StartupThisAP()** services. The idle state of an AP before the PI event **END_OF_PEI** is signaled is implementation dependent.

After the PI event **END_OF_PEI** is signaled, all the APs must be placed in the OS compatible CPU state as defined by the *UEFI Specification*. Implementations of this Ppi may use the PI event **END_OF_PEI** to force APs into the OS compatible state as defined by the *UEFI Specification.*

The support for **SwitchBSP()** and **EnableDisableAP()** may no longer be supported after the PEI event **END_OF_PEI** is signaled.

EFI_MP_SERVICES_PPI.GetNumberOfProcessors()

Summary

Get the number of CPU's

Prototype

```
typedef
EFI_STATUS
(EFIAPI PEI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS)(
  IN  CONST EFI_PEI_SERVICES    **PeiServices,
  IN  EFI_MP_SERVICES_PPI       *This,
  OUT UINTN                     *NumberOfProcessors,
  OUT UINTN                     *NumberOfEnabledProcessors
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

Pointer to this instance of the PPI.

NumberOfProcessors

Pointer to the total number of logical processors in the system, including the BSP and disabled APs.

NumberOfEnabledProcessors

Number of processors in the system that are enabled.

Description

This service retrieves the number of logical processor in the platform and the number of those logical processors that are enabled on this boot. This service may only be called from the BSP.

This function is used to retrieve the following information:

- The number of logical processors that are present in the system.
- The number of enabled logical processors in the system at the instant this call is made.

Because MP Service Ppi provides services to enable and disable processors dynamically, the number of enabled logical processors may vary during the course of a boot session.

If this service is called from an AP, then **EFI_DEVICE_ERROR** is returned.

If *NumberOfProcessors* or *NumberOfEnabledProcessors* is NULL, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the total number of processors is returned in *NumberOfProcessors*, the number of currently enabled processor is returned in *NumberOfEnabledProcessors*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The number of logical processors and enabled logical processors was retrieved.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_INVALID_PARAMETER	*NumberOfProcessors* is NULL.
EFI_INVALID_PARAMETER	*NumberOfEnabledProcessors* is NULL.

EFI_MP_SERVICES_PPI.GetProcessorInfo()

Summary

Get information on a specific CPU.

Prototype

```
typedef
EFI_STATUS
(EFIAPI PEI_MP_SERVICES_GET_PROCESSOR_INFO)(
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  EFI_MP_SERVICES_PPI        *This,
  IN  UINTN                       ProcessorNumber,
  OUT EFI_PROCESSOR_INFORMATION   *ProcessorInfoBuffer
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

ProcessorNumber

The handle number of the processor.

ProcessorInfoBuffer

A pointer to the buffer where the processor information is stored.

Description

Gets detailed MP-related information on the requested processor at the instant this call is made. This service may only be called from the BSP.

This service retrieves detailed MP-related information about any processor on the platform. Note the following:

- The processor information may change during the course of a boot session.
- The information presented here is entirely MP related.

Information regarding the number of caches and their sizes, frequency of operation, slot numbers is all considered platform-related information and is not provided by this service.

Status Codes Returned

EFI_SUCCESS	Processor information was returned.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_INVALID_PARAMETER	*ProcessorInfoBuffer* is NULL.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist in the platform.

EFI_MP_SERVICES_PPI.StartupAllAPs ()

Summary

Activate all of the application proessors.

Prototype

```
typedef
EFI_STATUS
 (EFIAPI *PEI_MP_SERVICES_STARTUP_ALL_APS)(
    IN  CONST EFI_PEI_SERVICES    **PeiServices,
    IN  EFI_MP_SERVICES_PPI        *This,
    IN  EFI_AP_PROCEDURE           Procedure,
    IN  BOOLEAN                    SingleThread,
    IN  UINTN                      TimeoutInMicroSeconds,
    IN  VOID                      *ProcedureArgument      OPTIONAL
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

Procedure

A pointer to the function to be run on enabled APs of the system. See type **EFI_AP_PROCEDURE**.

SingleThread

If **TRUE**, then all the enabled APs execute the function specified by *Procedure* one by one, in ascending order of processor handle number. If **FALSE**, then all the enabled APs execute the function specified by *Procedure* simultaneously.

TimeoutInMicroseconds

Indicates the time limit in microseconds for APs to return from Procedure, for blocking mode only. Zero means infinity. If the timeout expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. All enabled APs are available for next function assigned by **EFI_MP_SERVICES_PPI.StartupAllAPs()** or **EFI_MP_SERVICES_PPI.StartupThisAP()**.

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

ProcedureArgument

The parameter passed into *Procedure* for all APs.

Description

This service executes a caller provided function on all enabled APs. APs can run either simultaneously or one at a time in sequence. This service supports both blocking requests only. This service may only be called from the BSP.

This function is used to dispatch all the enabled APs to the function specified by *Procedure*. If any enabled AP is busy, then **EFI_NOT_READY** is returned immediately and *Procedure* is not started on any AP.

If *SingleThread* is **TRUE**, all the enabled APs execute the function specified by *Procedure* one by one, in ascending order of processor handle number. Otherwise, all the enabled APs execute the function specified by *Procedure* simultaneously.

If the timeout specified by *TimeoutInMicroseconds* expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. All enabled APs are always available for further calls to **EFI_MP_SERVICES_PPI.StartupAllAPs()** and **EFI_MP_SERVICES_PPI.StartupThisAP()**. If *FailedCpuList* is not NULL, its content points to the list of processor handle numbers in which *Procedure* was terminated.

Note: *It is the responsibility of the consumer of the* **EFI_MP_SERVICES_PPI.StartupAllAPs()** *to make sure that the nature of the code that is executed on the BSP and the dispatched APs is well controlled. The MP Services Ppi does not guarantee that the* Procedure *function is MP-safe. Hence, the tasks that can be run in parallel are limited to certain independent tasks and well-controlled exclusive code. PEI services and Ppis may not be called by APs unless otherwise specified.*

In blocking execution mode, BSP waits until all APs finish or *TimeoutInMicroSeconds* expires.

Status Codes Returned

EFI_SUCCESS	In blocking mode, all APs have finished before the timeout expired.
EFI_DEVICE_ERROR	Caller processor is AP.
EFI_NOT_STARTED	No enabled APs exist in the system.
EFI_NOT_READY	Any enabled APs are busy.
EFI_TIMEOUT	In blocking mode, the timeout expired before all enabled APs have finished.
EFI_INVALID_PARAMETER	*Procedure* is NULL

EFI_MP_SERVICES_PPI.StartupThisAP ()

Summary

Activate a specific application processor

Prototype

```
typedef
EFI_STATUS
(EFIAPI *PEI_MP_SERVICES_STARTUP_THIS_AP)(
  IN  CONST EFI_PEI_SERVICES    **PeiServices,
  IN  EFI_MP_SERVICES_PPI       *This,
  IN  EFI_AP_PROCEDURE          Procedure,
  IN  UINTN                     ProcessorNumber,
  IN  UINTN                     TimeoutInMicroseconds,
  IN  VOID                      *ProcedureArgument      OPTIONAL
);
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

Procedure

A pointer to the function to be run on enabled APs of the system. See type **EFI_AP_PROCEDURE**.

ProcessorNumber

The handle number of the AP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PPI.GetNumberOfProcessors()**.

TimeoutInMicrosecsond

Indicates the time limit in microseconds for APs to return from *Procedure*, for blocking mode only. Zero means infinity. If the timeout expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. All enabled APs are available for next function assigned by **EFI_MP_SERVICES_PPI.StartupAllAPs()** or **EFI_MP_SERVICES_PPI.StartupThisAP()**.

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

ProcedureArgument

The parameter passed into Procedure for all APs.

Description

This service lets the caller get one enabled AP to execute a caller-provided function. The caller can request the BSP to wait for the completion of the AP. This service may only be called from the BSP.

This function is used to dispatch one enabled AP to the function specified by *Procedure* passing in the argument specified by *ProcedureArgument*.

The execution is in blocking mode. The BSP waits until the AP finishes or *TimeoutInMicroSecondss* expires.

If the timeout specified by *TimeoutInMicroseconds* expires before the AP returns from *Procedure*, then execution of *Procedure* by the AP is terminated. The AP is available for subsequent calls to **EFI_MP_SERVICES_PPI.StartupAllAPs()** and **EFI_MP_SERVICES_PPI.StartupThisAP()**.

Status Codes Returned

EFI_SUCCESS	In blocking mode, specified AP finished before the timeout expires.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_TIMEOUT	In blocking mode, the timeout expired before the specified AP has finished.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the current BSP or a disabled AP.
EFI_INVALID_PARAMETER	*Procedure* is NULL

EFI_MP_SERVICES_PPI.SwitchBSP ()

Summary

Switch the boot strap processor

Prototype

```
typedef
(EFIAPI *PEI_MP_SERVICES_SWITCH_BSP)(
  IN  CONST EFI_PEI_SERVICES    **PeiServices,
  IN  EFI_MP_SERVICES_PPI       *This,
  IN  UINTN                     ProcessorNumber,
  IN  BOOLEAN                   EnableOldBSP
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

ProcessorNumber

The handle number of AP that is to become the new BSP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PPI.GetNumberOfProcessors()**.

EnableOldBSP

If **TRUE**, then the old BSP will be listed as an enabled AP. Otherwise, it will be disabled.

Description

This service switches the requested AP to be the BSP from that point onward.

This service changes the BSP for all purposes.This call can only be performed by the current BSP.

This service switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes. The new BSP can take over the execution of the old BSP and continue seamlessly from where the old one left off.

If the BSP cannot be switched prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

Status Codes Returned

EFI_SUCCESS	BSP successfully switched.
EFI_UNSUPPORTED	Switching the BSP cannot be completed prior to this service returning.
EFI_UNSUPPORTED	Switching the BSP is not supported.
EFI_SUCCESS	The calling processor is an AP.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the current BSP or a disabled AP.
EFI_NOT_READY	The specified AP is busy.

Summary

Switch the boot strap processor

Prototype

```
typedef
 (EFIAPI *PEI_MP_SERVICES_ENABLEDISABLEAP)(
  IN  CONST EFI_PEI_SERVICES     **PeiServices,
  IN  EFI_MP_SERVICES_PPI        *This,
  IN  UINTN                      ProcessorNumber,
  IN  BOOLEAN                    EnableAP,
  IN  UINT32                     *HealthFlag OPTIONAL
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

ProcessorNumber

The handle number of AP that is to become the new BSP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PPI.GetNumberOfProcessors()**.

EnableAP

Specifies the new state for the processor for enabled, FALSE for disabled.

HealthFlag

If not NULL, a pointer to a value that specifies the new health status of the AP. This flag corresponds to *StatusFlag* defined in **EFI_MP_SERVICES_PPI.GetProcessorInfo()**. Only the

PROCESSOR_HEALTH_STATUS_BIT is used. All other bits are ignored. If it is NULL, this parameter is ignored.

Description

This service lets the caller enable or disable an AP from this point onward.

This service may only be called from the BSP.

This service allows the caller enable or disable an AP from this point onward. The caller can optionally specify the health status of the AP by Health. If an AP is being disabled, then the state of the disabled AP is implementation dependent. If an AP is enabled, then the implementation must guarantee that a complete initialization sequence is performed on the AP, so the AP is in a state that is compatible with an MP operating system.

If the enable or disable AP operation cannot be completed prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

Status Codes Returned

EFI_SUCCESS	The specified AP was enabled or disabled successfully.
EFI_UNSUPPORTED	Enabling or disabling an AP cannot be completed prior to this service returning.
EFI_UNSUPPORTED	Enabling or disabling an AP is not supported.
EFI_DEVICE_ERROR	The calling processor is an AP.
EFI_NOT_FOUND	Processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the BSP.

EFI_MP_SERVICES_PPI.WhoAmI ()

Summary

Identify the currently executing processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *PEI_MP_SERVICES_WHOAMI)(
  IN  CONST EFI_PEI_SERVICES   **PeiServices,
  IN  EFI_MP_SERVICES_PPI      *This,
  OUT UINTN                    *ProcessorNumber
  );
```

Parameters

PeiServices

An indirect pointer to the PEI Services Table published by the PEI Foundation.

This

A pointer to the **EFI_MP_SERVICES_PPI** instance.

ProcessorNumber

The handle number of AP that is to become the new BSP. The range is from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved by **EFI_MP_SERVICES_PPI.GetNumberOfProcessors()**.

Description

This services returns the handle number for the calling processor. This service may be called from the BSP and APs.

This service returns the processor handle number for the calling processor.

The returned value is in the range from 0 to the total number of logical processors minus 1. The total number of logical processors can be retrieved with **EFI_MP_SERVICES_PPI.GetNumberOfProcessors()**. This service may be called from the BSP and APs. If *ProcessorNumber* is NULL, then **EFI_INVALID_PARAMETER** is returned. Otherwise, the current processors handle number is returned in *ProcessorNumber*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The current processor handle number was returned in *ProcessorNumber*.
EFI_INVALID_PARAMETER	*ProcessorNumber* is NULL.

8.4 Graphics PEIM Interfaces

There is one PEI to PEI Interfaces (PPI) that is required to provide graphics functionality in the PEI phase.

The **PeiGraphicsPpi** is the PPI produced by the Graphics PEI Module and provides interfaces to the platform code to complete the basic initialization of the graphics subsystem to enable console output.

8.4.1 Pei Graphics PPI

The **PeiGraphicsPpi** is the main interface exposed by the Graphics PEIM to be used by the other firmware modules.

The following sections cover the individual APIs in detail.

GUID

```
#define EFI_PEI_GRAPHICS_PPI_GUID \
{ 0x6ecd1463, 0x4a4a, 0x461b,
 {0xaf, 0x5f, 0x5a, 0x33, 0xe3, 0xb2, 0x16, 0x2b }};
```

Prototype

```
struct _EFI_PEI_GRAPHICS_PPI {
  EFI_PEI_GRAPHICS_INIT      GraphicsPpiInit;
  EFI_PEI_GRAPHICS_GET_MODE  GraphicsPpiGetMode;
} EFI_PEI_GRAPHICS_PPI;
```

GraphicsPpiInit

Description

The *GraphicsPpiInit* initializes the graphics subsystem in phases.

Calling Condition

There are certain conditions to be met before the *GraphicsPpiInit* can be called; Memory has been initialized.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GRAPHICS_INIT) (
     IN VOID *GraphicsPolicyPtr;
);
```

Parameters

GraphicsPolicyPtr

GraphicsPolicyPtr points to a configuration data block of policy settings required by Graphics PEIM.

Return

EFI_SUCCESS	The invocation was successful.
EFI_INVALID_PARAMETER	The phase parameter is not valid.
EFI_NOT_ABORTED	The stages were not called in the proper order.
EFI_NOT_FOUND	The **PeiGraphicsPlatformPolicyPpi** is not located.
EFI_DEVICE_ERROR	The initialization failed due to device error.
EFI_NOT_READY	The previous init stage is still in progress and not ready for the current initialization phase yet. The platform code should call this again sometime later..

GraphicsPpiGetMode

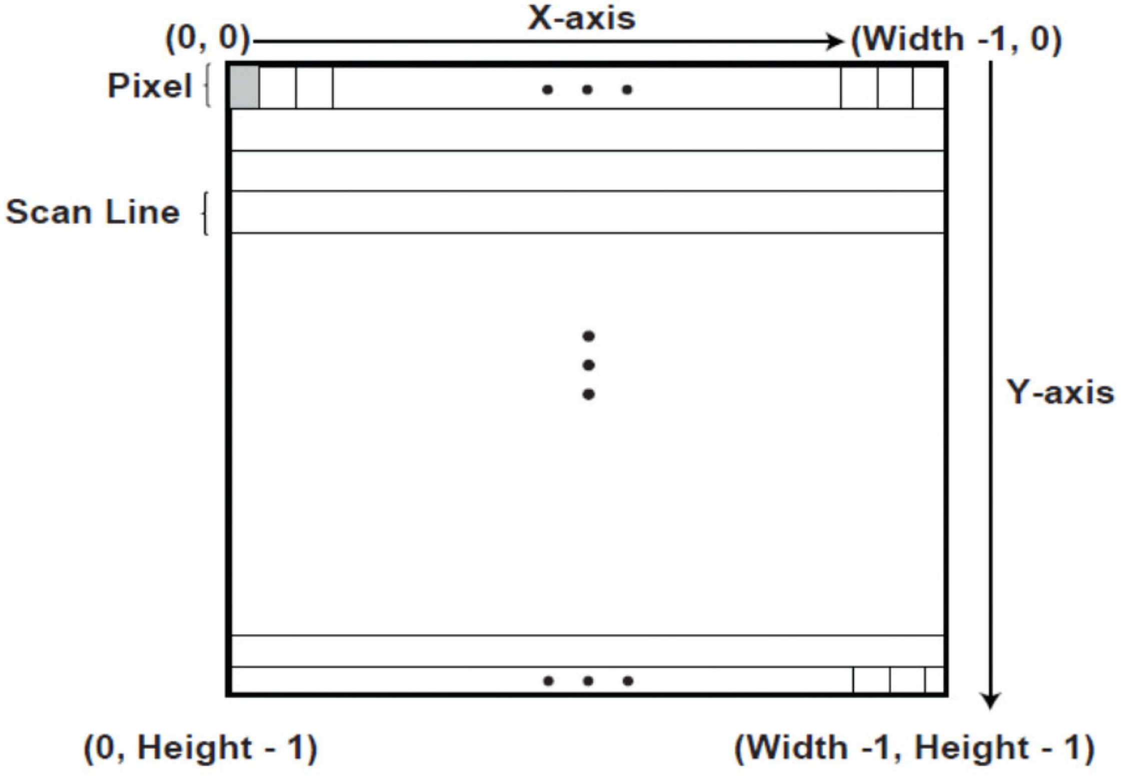
Description

The **GraphicsPpiGetMode** returns the mode information supported by the Graphics PEI Module.

The frame buffer abstracts the video display as an array of pixels. Each pixels location on the video display is defined by its X and Y coordinates. The X coordinate represents a scan line. A scan line is a horizontal line of pixels on the display. The Y coordinate represents a vertical line on the display. The upper left hand corner of the video display is defined as (0, 0) where the notation (X, Y) represents the X and Y coordinate of the pixel. The lower right corner of the video display is represented by (Width –1, Height -1).

A pixel is comprised of a 32-bit quantity. The first three bytes for each pixel represents the intensity for Red, Blue and Green colors. The fourth byte is reserved and must be zero. The byte values for the red, green, and blue components represent the color intensity. This color intensity value range from a minimum intensity of 0 to maximum intensity of 255.

The mode information returned by this PPI is similar to the GOP's **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** structure.



Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GRAPHICS_GET_MODE) (
IN OUT EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode
);
```

Parameters

Mode

Pointer to **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** data. Type **EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE** is defined in the *UEFI Specification* and in “Related Definitions” below.

Return

EFI_SUCCESS	Valid mode information was returned.
EFI_INVALID_PARAMETER	The *Mode* parameter is not valid.
EFI_NOT_FOUND	The **PeiGraphicsPlatformPolicyPpi** is not located.
EFI_DEVICE_ERROR	A hardware error occurred trying to retrieve the video mode.
EFI_NOT_READY	The Graphics Initialization is not compete, and *Mode* information is not yet available.The platform code should call this again after the Graphics initialization is done.

```
typedef struct {
  UINT32 MaxMode;
  UINT32 Mode;
  EFI_GRAPHICS_OUTPUT_MODE_INFORMATION *Info;
  UINTN SizeOfInfo;
  EFI_PHYSICAL_ADDRESS FrameBufferBase;
  UINTN FrameBufferSize;
} EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE;
```

Related Definition – EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE

MaxMode

The number of modes that is supported by this module.

Mode

Current mode of the graphics device. If the *MaxMode* is 1, then this field will be 0.

Info

Pointer to **EFI_GRAPHICS_OUTPUT_MODE_INFORMATION** data. See Related Definition below.

SizeOfInfo

Size of the *Info* structure in bytes.

FrameBufferBase

Base address of graphics linear frame buffer. *Info* contains information required to allow software to draw directly to the frame buffer.

FrameBufferSize

Size of the frame buffer represented by *FrameBufferBase* in bytes.

Related Definition – EFI_GRAPHICS_OUTPUT_MODE_INFORMATION

```
typedef struct {
  UINT32                       Version;
  UINT32                       HorizontalResolution;
  UINT32                       VerticalResolution;
  EFI_GRAPHICS_PIXEL_FORMAT    PixelFormat;
  EFI_PIXEL_BITMASK            PixelInformation;
  UINT32                       PixelsPerScanLine;
} EFI_GRAPHICS_OUTPUT_MODE_INFORMATION;
```

Version

The version of this data structure. A value of zero represents the structure as defined in this specification. Future version of this specification may extend this data structure in a backwards compatible way and increase the value of Version.

HorizontalResolution

The size of video screen in pixels in the X dimension.

VerticalResolution

The size of video screen in pixels in the Y dimension.

PixelFormat

Enumeration that defines the physical format of the pixel. A value of *PixelBltOnly*

implies that a linear frame buffer is not available for this mode.

PixelInformation

This bit-mask is only valid if *PixelFormat* is set to *PixelPixelBitMask*. A bit being set

defines what bits are used for what purpose such as Red, Green, Blue, or Reserved.

PixelsPerScanLine

Defines the number of pixel elements per video memory line. For performance reasons, or due to hardware restrictions, scan lines may be padded to an amount of memory alignment. These padding pixel elements are outside the area covered by *HorizontalResolution* and are not visible. For direct frame buffer access, this number is used as a span between starts of pixel lines in video memory. Based on the size of an individual pixel element and *PixelsPerScanline*, the offset in video memory from pixel element (x, y) to pixel element (x, y+1) has to be calculated as "sizeof(PixelElement) * *PixelsPerScanLine*", not "*sizeof*(PixelElement) *

HorizontalResolution", though in many cases those values can coincide. This value can depend on video hardware and mode resolution.

Related Definition – EFI_GRAPHICS_OUTPUT_MODE_INFORMATION

```
typedef enum {
  PixelRedGreenBlueReserved8BitPerColor,
  PixelBlueGreenRedReserved8BitPerColor,
  PixelBitMask,
  PixelBltOnly,
  PixelFormatMax
} EFI_GRAPHICS_PIXEL_FORMAT;
```

PixelRedGreenBlueReserved8BitPerColor

A pixel is 32-bits and byte zero represents red, byte one represents green, byte two represents blue, and byte three is reserved. This is the definition for the physical frame buffer. The byte values for the red, green, and blue components represent the color intensity. This color intensity value range from a minimum intensity of 0 to maximum intensity of 255.

PixelBlueGreenRedReserved8BitPerColor

A pixel is 32-bits and byte zero represents blue, byte one represents green, byte two represents red, and byte three is reserved. This is the definition for the physical frame buffer. The byte values for the red, green, and blue components represent the color intensity. This color intensity value range from a minimum intensity of 0 to maximum intensity of 255.

PixelBitMask

The pixel definition of the physical frame buffer is defined by **EFI_PIXEL_BITMASK**

PixelBltOnly

This mode does not support a physical frame buffer.

Related Definition – EFI_PIXEL_BITMASK

```
typedef struct {
  UINT32 RedMask;
  UINT32 GreenMask;
  UINT32 BlueMask;
  UINT32 ReservedMask;
} EFI_PIXEL_BITMASK;
```

If a bit is set in *RedMask*, *GreenMask*, or *BlueMask* then those bits of the pixel represent the corresponding color. Bits in *RedMask*, *GreenMask*, *BlueMask*, and *ReserveredMask* must not overlap bit positions. The values for the red, green, and blue components in the bit mask represent the color intensity. The color intensities must increase as the color values for each color mask increase with a minimum intensity of all bits in a color mask clear to a maximum intensity of all bits in a color mask set.

8.4.2 EFI PEI Graphics INFO HOB

EFI_PEI_GRAPHICS_INFO_HOB

```
#define EFI_PEI_GRAPHICS_INFO_HOB_GUID \
{ 0x39f62cce, 0x6825, 0x4669,\
 { 0xbb, 0x56, 0x54, 0x1a, 0xba, 0x75, 0x3a, 0x07 }}

typedef struct {
   EFI_PHYSICAL_ADDRESS                  FrameBufferBase;
   UINT32                                FrameBufferSize;
   EFI_GRAPHICS_OUTPUT_MODE_INFORMATION  GraphicsMode;
} EFI_PEI_GRAPHICS_INFO_HOB;
```

EFI_PEI_GRAPHICS_DEVICE_INFO_HOB

```
#define EFI_PEI_GRAPHICS_DEVICE_INFO_HOB_GUID \
{ 0xe5cb2ac9, 0xd35d, 0x4430,\
 { 0x93, 0x6e, 0x1d, 0xe3, 0x32, 0x47, 0x8d, 0xe7 }}

typedef struct {
  UINT16                                VendorId
  UINT16                                DeviceId
  UINT16                                SubsystemVendorId
  UINT16                                SubsystemId;
  UINT8                                 RevisionId;
  UINT8                                 BarIndex;
 } EFI_PEI_GRAPHICS_DEVICE_INFO_HOB;
```

When graphics capability is included in PEI, it may optionally provide a splash screen capability as well.

When graphics capability is included in PEI, it produces a **EFI_PEI_GRAPHICS_INFO_HOB** which provides information about the graphics mode and the framebuffer, and may optionally produce a **EFI_PEI_GRAPHICS_DEVICE_INFO_HOB** which provides information about the graphics device characteristics. The **EFI_GRAPHICS_OUTPUT_MODE_INFORMATION** structure is defined in the *UEFI specification*. This information can be used by the HOB-consumer phase, such as DXE, to provide display support of its own, or elide the need to do graphics initialization again in the UEFI GOP driver, for example.

It is to be noted that the PEI phase may program a temporary framebuffer address to complete its initialization and the framebuffer address at the time of building the **EFI_PEI_GRAPHICS_INFO_HOB** will reflect the current assignment. The post-PEI phase consuming this HOB should be aware that a generic PCI enumeration logic could reprogram the temporary resources assigned by the PEI phase and it is the responsibility of the post-PEI phase to update its internal data structures with the new framebuffer address after the enumeration is complete.

The EFI_PEI_GRAPHICS_DEVICE_INFO_HOB is optional. When it exists, the DXE module which provides display support uses the **VendorId**, **DeviceId**, **RevisionId**, **SubsystemVendorId**, and **SubsystemDeviceId** in the HOB to match the graphics device. It's useful when system has multiple graphics devices and the DXE module cannot know which one to manage without the information provided by this HOB. If **VendorId**, **DeviceId**, **SubsystemVendorId** or **SubsystemDeviceId** is set to MAX_UINT16, or **RevisionId** is set to MAX_UINT8, that field will be ignored. The ID values that are assigned to other values will be used to identify the graphics device. The **BarIndex** tells DXE module which PCI MMIO BAR is used to hold the frame buffer. BAR 0 is used if the **BarIndex** is set to MAX_UINT8 or the HOB doesn't exist.

9
PEI to DXE Handoff

9.1 Introduction

The PEI phase of the system firmware boot process performs rudimentary initialization of the system to meet specific minimum system state requirements of the DXE Foundation. The PEI Foundation must have a mechanism of locating and passing off control of the system to the DXE Foundation. PEI must also provide a mechanism for components of DXE and the DXE Foundation to discover the state of the system when the DXE Foundation is invoked. Certain aspects of the system state at handoff are architectural, while other system state information may vary and hence must be described to DXE components.

9.2 Discovery and Dispatch of the DXE Foundation

The PEI Foundation uses a special PPI named the DXE Initial Program Load (IPL) PPI to discover and dispatch the DXE Foundation and components that are needed to run the DXE Foundation

The final action of the PEI Foundation is to locate and pass control to the DXE IPL PPI. To accomplish this, the PEI Foundation scans all PPIs by GUID for the GUID matching the DXE IPL PPI. The GUID for this PPI is defined in **EFI_DXE_IPL_PPI**.

9.3 Passing the Hand-Off Block (HOB) List

The DXE IPL PPI passes the Hand-Off Block (HOB) list from PEI to the DXE Foundation when it invokes the DXE Foundation. The handoff state is described in the form of HOBs in the HOB list. The HOB list must contain at least the HOBs listed in Table 18.

Table 18. Required HOB Types in the HOB List

Required HOB Type	Usage
Phase Handoff Information Table (PHIT) HOB	This HOB is required.
One or more Resource Descriptor HOB(s) describing physical system memory	The DXE Foundation will use this physical system memory for DXE.
Boot-strap processor (BSP) Stack HOB	The DXE Foundation needs to know the current stack location so that it can move it if necessary, based upon its desired memory address map. This HOB will be of type EfiConventionalMemory
BSP BSPStore (“Backing Store Pointer Store”) HOB **Note:** Itanium processor family only	The DXE Foundation needs to know the current store location so that it can move it if necessary, based upon its desired memory address map.
One or more Resource Descriptor HOB(s) describing firmware devices	The DXE Foundation will place this into the GCD.

One or more Firmware Volume HOB(s)	The DXE Foundation needs this information to begin loading other drivers in the platform.
A Memory Allocation Module HOB	This HOB tells the DXE Foundation where it is when allocating memory into the initial system address map.

The above HOB types are defined in volume 3 of this specification.

9.4 Handoff Processor State to the DXE IPL PPI

Table 19 defines the state that processors must be in at handoff to the DXE IPL PPI, for the following processors:

- IA-32 processors
- Itanium processor family
- Intel® processors using Intel® XScale™ technology

Table 19. Handoff Processor State to the DXE IPL PPI

Processor	State at Handoff
IA-32	In 32-bit flat mode
Itanium	In Itanium processor family physical mode
Intel XScale	In SuperVisor Mode with a one-to-one virtual-to-physical mapping if there is a memory management unit (MMU) in the system

10
Boot Paths

10.1 Introduction

The PEI Foundation is unaware of the boot path required by the system. It relies on the PEIMs to determine the boot mode (e.g. R0, R1, S3, etc.) and take appropriate action depending on the mode.

To implement this, each PEIM has the ability to manipulate the boot mode using the PEI Service **SetBootMode()** described in Services - PEI.

The PEIM does not change the order in which PEIMs are dispatched depending on the boot mode.

10.2 Code Flow

The normal code flow in PI firmware passes through a succession of phases, in the following order:

1. SEC
2. PEI
3. DXE
4. BDS
5. Runtime
6. Afterlife

This section describes alternatives to this ordering.

10.2.1 Reset Boot Paths

The following sections describe the boot paths that are followed when a system encounters several different types of reset.

10.2.1.1 Intel Itanium Processor Reset

Itanium architecture contains enough hooks to authenticate PAL-A and PAL-B code that is distributed by the processor vendor. The internal microcode on the processor silicon, which starts up on a PowerGood reset, finds the first layer of processor abstraction code (called PAL-A) that is located in the boot firmware volume (BFV), or the volume that has SEC and the PEI core, using architecturally defined pointers in the BFV. It is the responsibility of this microcode to authenticate that the PAL-A code layer from the processor vendor has not been tampered. If the authentication of the PAL-A layer passes, control then passes to the PAL-A layer, which then authenticates the next layer of processor abstraction code (called PAL-B) before passing control to it. In addition to this microarchitecture-specific authentication, the SEC phase of UEFI is still responsible for locating the PEI Foundation and verifying its authenticity.

In an Itanium-based system, it is also imperative that the firmware modules in the BFV be organized such that at least the PAL-A is contained in the fault-tolerant regions. This processor-specific PAL-A authenticates the PAL-B code, which usually is contained in the non-fault-tolerant regions of the firmware system. The PAL A and PAL B binary components are always visible to all the processors in a node at the time of power-on; the system fabric should not need to be initialized.

10.2.1.2 Non-Power-on Resets

Non-power-on resets can occur for many reasons. There are PEI and DXE system services that reset and reboot the entire platform, including all processors and devices. It is important to have a standard variant of this boot path for cases such as the following:

- Resetting the processor to change frequency settings
- Restarting hardware to complete chipset initialization
- Responding to an exception from a catastrophic error

This reset is also used for Configuration Values Driven through Reset (CVDR) configuration.

10.3 Normal Boot Paths

A traditional BIOS executes POST from a cold boot (G3 to S0 state), on resumes, or in special cases like INIT. UEFI covers all those cases but provides a richer and more standardized operating environment

The basic code flow of the system needs to be changeable due to different circumstances. The boot path variable satisfies this need. The initial value of the boot mode is defined by some early PEIMs, but it can be altered by other, later PEIM(s). All systems must support a basic S0 boot path. Typically a system has a more rich set of boot paths, including S0 variations, S-state boot paths, and one or more special boot paths.

The architecture for multiple boot paths presented here has several benefits, as follows:

- The PEI Foundation is not required to be aware of system-specific requirements such as MP and various power states. This lack of awareness allows for scalability and headroom for future expansion.
- Supporting the various paths only minimally impacts the size of the PEI Foundation.
- The PEIMs that are required to support the paths scale with the complexity of the system.

Note that the Boot Mode Register becomes a variable upon transition to the DXE phase. The DXE phase can have additional modifiers that affect the boot path more than the PEI phase.

These additional modifiers can indicate if the system is in manufacturing mode, chassis intrusion, or AC power loss or if silent boot is enabled.

10.3.1 Basic G0-to-S0 and S0 Variation Boot Paths

The basic S0 boot path is "boot with full configuration." This path setting informs all PEIMs to do a full configuration. The basic S0 boot path must be supported.

The Framework architecture also defines several optional variations to the basic S0 boot path. The variations that are supported depend on the following:

- Richness of supported features
- If the platform is open or closed
- Platform hardware

For example, a closed system or one that has detected a chassis intrusion could support a boot path that assumes no configuration changes from last boot option, thus allowing a very rapid boot time. Unsupported variations default to basic S0 operation. The following are the defined variations to the basic boot path:

- Boot with minimal configuration:

This path is for configuring the minimal amount of hardware to boot the system.

- Boot assuming no configuration changes:

This path uses the last configuration data.

- Boot with full configuration plus diagnostics:

This path also causes any diagnostics to be executed.

- Boot with default settings: This path uses a known set of safe values for programming hardware.

10.3.2 S-State Boot Paths

The following optional boot paths allow for different operation for a resume from S3, S4, and S5:

- S3 (Save to RAM Resume): Platforms that support S3 resume must take special care to preserve/restore memory and critical hardware.
- S4 (Save to Disk): Some platforms may want to perform an abbreviated PEI and DXE phase on a S4 resume.
- S5 (Soft Off): Some platforms may want an S5 system state boot to be differentiated from a normal boot-for example, if buttons other than the power button can wake the system.

An S3 resume needs to be explained in more detail because it requires cooperation between a G0-to-S0 boot path and an S3 resume boot path. The G0-to-S0 boot path needs to save hardware programming information that the S3 resume path needs to retrieve.

This information is saved in the Hardware Save Table using predefined data structures to perform I/O or memory writes. The data is stored in an UEFI equivalent of the INT15 E820 type 4 (firmware reserved memory) area or a firmware device area that is reserved for use by UEFI. The S3 resume boot path code can access this region after memory has been restored.

10.4 Recovery Paths

All of the above boot paths can be modified or aborted if the system detects that recovery is needed. Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes on nonvolatile storage devices (flash, disk) are managed as blocks. If the system loses power while a block, or semantically bound blocks, are being updated, the storage might become invalid. On the other hand, the device might become corrupted by an errant program or by errant hardware. The system designers must determine the level of support for recovery based on their perceptions of the probabilities of these events occurring and their consequences.

The following are some reasons why system designers may choose to not support recovery:

- A system's firmware volume storage media might not support modification after being manufactured. It might be the functional equivalent of a ROM.

- Most mechanisms of implementing recovery require additional firmware volume space, which might be too expensive for a particular application.
- A system may have enough firmware volume space and hardware features that the firmware volume can be made sufficiently fault tolerant to make recovery unnecessary.

10.4.1 Discovery

Discovering that recovery is done using a PEIM (for example, by checking a "force recovery" jumper).

10.4.2 General Recovery Architecture

The concept behind recovery is to preserve enough of the system firmware so that the system can boot to a point where it can do the following:

- Read a copy of the data that was lost from chosen peripherals.
- Reprogram the firmware volume with that data.

Preserving the recovery firmware is a function of the way the firmware volume store is managed, which is generally beyond the scope of this document.

The PI recovery architecture allows for one or many PEIMs to be built to handle the portion of the recovery that would initialize the recovery peripherals (and the buses they reside on) and then to read the new images from the peripherals and update the firmware volumes.

It is considered far more likely that the PEI will transition to DXE because DXE is designed to handle access to peripherals. This transition has the additional benefit that, if DXE then discovers that a device has become corrupted, it may institute recovery without transferring control back to the PEI.

10.5 Defined Boot Modes

The list of possible boot modes is described in the **`GetBootMode()`** function description. PI architecture specifically does not define an upgrade path if new boot modes are defined. This is necessary as the nature of those additional boot modes may work in conjunction with or may conflict with the previously defined boot modes.

10.6 Priority of Boot Paths

Within a given PEIM, the priority ordering of the sources of boot mode should be as follows (from highest priority to lowest):

1. **`BOOT_IN_RECOVERY_MODE`**
2. **`BOOT_ON_FLASH_UPDATE`**
3. **`BOOT_ON_S3_RESUME`**
4. **`BOOT_WITH_MINIMAL_CONFIGURATION`**
5. **`BOOT_WITH_FULL_CONFIGURATION`**
6. **`BOOT_ASSUMING_NO_CONFIGURATION_CHANGES`**

7. **BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS**
8. **BOOT_WITH_DEFAULT_SETTINGS**
9. **BOOT_ON_S4_RESUME**
10. **BOOT_ON_S5_RESUME**
11. **BOOT_ON_S2_RESUME**

The boot modes listed above are defined in the PEI Service **SetBootMode()**.

10.7 Assumptions

Table 20 lists the assumptions that can be made about the system for each sleep state.

Table 20. Boot Path Assumptions

System State	Description	Assumptions
R0	Cold Boot	Cannot assume that the previously stored configuration data is valid.
R1	Warm Boot	May assume that the previously stored configuration data is valid.
S3	ACPI Save to RAM Resume	The previously stored configuration data is valid and RAM is valid. RAM configuration must be restored from nonvolatile storage (NVS) before RAM may be used. The firmware may only modify previously reserved RAM. There are two types of reserved memory. One is the equivalent of the BIOS INT15h, E820 type-4 memory and indicates that the RAM is reserved for use by the firmware. The suggestion is to add another type of memory that allows the OS to corrupt the memory during runtime but that may be overwritten during resume.
S4, S5	Save to Disk Resume, “Soft Off”	S4 and S5 are identical from a PEIM's point of view. The two are distinguished to support follow-on phases. The entire system must be reinitialized but the PEIMs may assume that the previous configuration is still valid.
Boot on Flash Update		This boot mode can be either an INIT, S3, or other means by which to restart the machine. If it is an S3, for example, the flash update cause will supersede the S3 restart. It is incumbent upon platform code, such as the Memory Initialization PEIM, to determine the exact cause and perform correct behavior (i.e., S3 state restoration versus INIT behavior).
Boot with Manufacturing Mode settings		PEIM's and/or DXE drivers may parameterize based upon actions that should only occur in the factory or a manufacturer approved setting.

10.8 Architectural Boot Mode PPIs

There is a possible hierarchy of boot mode PPIs that abstracts the various producers of this variable. It is a hierarchy in that there should be an order of precedence in which each mode can be set. The PPIs and their respective GUIDs are described in “Required Architectural PPIs” on page 89 and “Optional Architectural PPIs” on page 94. The hierarchy includes the master PPI, which publishes a PPI that will be depended upon by the appropriate PEIMs, and some subsidiary PPI. For PEIMs that require that the boot mode is finally known, the Master Boot Mode PPI can be used as a dependency.

Table 21 lists the architectural boot mode PPIs.

Table 21. Architectural Boot Mode PPIs

PPI Name	Required or Optional?	PPI Definition in Section...
Master Boot Mode PPI	Required	Architectural PPIs: Required Architectural PPIs
Boot in Recovery Mode PPI	Optional	Architectural PPIs: Optional Architectural PPIs

10.9 Recovery

10.9.1 Scope

Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes (FVs) in nonvolatile storage (NVS) devices (flash or disk, for example) are managed as blocks. If the system loses power while a block, or semantically bound blocks, are being updated, the storage might become invalid. On the other hand, an errant program or hardware could corrupt the device. The system designers must determine the level of support for recovery based on their perceptions of the probabilities of these events occurring and the consequences.

The designers of a system may choose not to support recovery for the following reasons:

- A system's FV storage media might not support modification after being manufactured. It might be the functional equivalent of a ROM.
- Most mechanisms of implementing recovery require additional FV space that might be too expensive for a particular application.
- A system may have enough FV space and hardware features that the FV can be made sufficiently fault tolerant to make recovery unnecessary.

10.9.2 Discovery

Discovering that recovery is required may be done using a PEIM (for example, by checking a "force recovery" jumper) or the PEI Foundation itself. The PEI Foundation might discover that a particular PEIM has not validated correctly or that an entire firmware has become corrupted.

10.9.3 General Recovery Architecture

The concept behind recovery is to preserve enough of the system firmware so that the system can boot to a point where it can do the following:

- Read a copy of the data that was lost from chosen peripherals.
- Reprogram the firmware volume (FV) with that data.

Preserving the recovery firmware is a function of the way the FV store is managed, which is generally beyond the scope of this document.

If the PEI Dispatcher encounters PEIMs that have been corrupted (for example, by receiving an incorrect hash value), it must change the boot mode to "recovery." Once set to recovery, other PEIMs must not change it to one of the other states.

A PEIM can also detect a catastrophic condition or a forced-recovery event and alert the PEI 10.6.4 Finding and Loading the Recovery DXE Image.

10.9.4 Finding and Loading the Recovery DXE Image

10.9.4.1 Finding the Recovery DXE Image: Overview

The PEI Dispatcher specifically invokes the DXE Initial Program Load (IPL) PEIM, regardless of normal or recovery mode. The DXE IPL PEIM detects that a recovery is in process and invokes a recovery-specific PPI, the Recovery Module PPI. The Recovery Module PPI, **`EFI_PEI_RECOVERY_MODULE_PPI,`** does the following:

- Loads a binary capsule that includes a recovery DXE image into memory
- Updates the Hand-Off Block (HOB) table with the DXE firmware volume
- Installs or Reinstalls instance of the Firmware Volume Info PPI (**`EFI_PEI_FIRMWARE_VOLUME_INFO_PPI)`** for the DXE firmware volume

See Section 8.6.3 for the PPIs that are needed to load the DXE image.

Note: *The Recovery Module PPI is device and content neutral. The DXE IPL PEIM uses the Recovery Module PPI to load a DXE image and invokes the DXE image normally. The DXE IPL PEIM does not know or care about the capsule's internal structure or from which device the capsule was loaded.*

The internals of the recovery PEIM normally fall within four phases:

- Searching the supported devices for recovery capsules
- Deciding which capsule to load
- Loading the capsule into memory
- Loading the resulting DXE firmware volume

The Recovery Module PPI encompasses the first three phases and the DXE IPL PEIM encompasses the last phase. See the next topic, Recovery Sequence: Detailed Steps, for the details of these four phases.

10.9.4.2 Recovery Sequence

The normal, non-recovery sequence is that after completion of the PEI phase, the PEI Dispatcher specifically invokes the DXE Initial Program Load (IPL) PEIM. The recovery sequence is identical to the non-recovery sequence in that the PEI Dispatcher also specifically invokes the DXE IPL PEIM. After invoking the DXE IPL PEIM, the recovery sequence is as follows:

1. The DXE IPL PEIM detects that a recovery is in process, searches for the Recovery Module PPI, and invokes the recovery function **`EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()`**.
2. **`EFI_PEI_RECOVERY_MODULE_PPI`** searches for one or more instances of the Device Recovery Module PPI, **`EFI_PEI_DEVICE_RECOVERY_MODULE_PPI`**. For each instance found, the

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetNumberRecoveryCapsules() function is invoked to determine the following:

- The number of recovery DXE capsules detected by the specified device
- The maximum buffer size required to load a capsule

3. **EFI_PEI_RECOVERY_MODULE_PPI** then decides the following:
   - The device search order, if more than one Device Recovery Module PPI was discovered
   - The individual search order, if the device reported more than one recovery DXE capsule was found generating a search order list
4. **EFI_PEI_RECOVERY_MODULE_PPI** invokes the device recovery function **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** to load a capsule that includes a recovery DXE image into memory. The capsule that is returned from the device recovery module is a capsule that contains the recovery DXE image.
5. The **EFI_PEI_RECOVERY_MODULE_PPI** security does the following:
   - Verifies the capsule
   - Generates a data Hand-Off Block (HOB) entry for a security failure
   - Tries the next entry in the search order list
6. Once a valid capsule has been loaded, **EFI_PEI_RECOVERY_MODULE_PPI** does the following:
   - Decomposes the capsule and updates the HOB table with the recovery DXE firmware volume information. The path parameters are assumed to be redundant for recovery. The Setup parameters are either redundant or fixed.
   - Invalidates all HOB entries for updateable firmware volume entries.

The DXE capsule that is loaded by the Device Recovery Module PPI makes no assumptions about contents or format other than assuming that the recovery DXE image is somewhere in the returned capsule.

The following subsections describe the different recovery PPIs.

10.9.4.3 Recovery PPIs: Recovery Module PPI

The Recovery Module PPI, **EFI_PEI_RECOVERY_MODULE_PPI**, invokes the Device Recovery Module **PPI EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** to do the following:

- Determine the number of DXE recovery capsules found by each device
- Determine capsule information
- Load a specific DXE recovery capsule from the indicated device
- Determine the device load order

The capsule is security verified and decomposed and the HOB table is updated with the DXE recovery firmware volume.

There are two general categories of recovery PPIs:

- Device recovery PPI
- Device recovery block I/O PPI

The Device Recovery Module PPI is device neutral. The Device Recovery Block I/O PPI is device specific and used to access the physical media. The following subsections describe the PPI associated with each category. See Code Definitions for the definitions of these PPIs.

10.9.4.3.1 Device Recovery Module PPI

The table below lists the device recovery functions in the Device Recovery Module PPI, **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI**.

Table 22. Device Recovery Module Functions

Function	Description
GetNumberRecoveryCapsules()	Scans the devices that are supported by the PPI for DXE recovery capsules and reports the number found. The internal ordering should reflect the priority in the load order, with the highest priority capsule number set to one and the lowest priority number set to *N*.
GetRecoveryCapsuleInfo()	Provides the size of the indicated capsule and a *CapsuleType* Globally Unique Identifier (GUID). The recovery module uses this information to allow an alternate priority scheme based on the *CapsuleType* information.
LoadRecoveryCapsule()	Loads the indicated DXE recovery capsule instance and returns a capsule with the actual number of bytes loaded.

10.9.4.3.2 Device Recovery Block I/O PPI

The Device Recovery Block I/O PPI, **EFI_PEI_RECOVERY_BLOCK_IO_PPI**, differs from the Device Recovery Module PPI in that the Device Recovery Block I/O PPI is used for physical media access. The Device Recovery Module PPI uses this PPI to search for capsules. This PPI is included with the recovery PEIMs because a block I/O is the most common recovery media.

The table below lists the functions in the Device Recovery Block I/O PPI.

Table 23. Device Recovery Block I/O Functions

Function	Description
GetNumberOfBlockDevices()	Returns the number of block I/O devices supported. There is no ordering priority.
GetBlockDeviceMediaInfo()	Indicates the type of block I/O device found, such as a legacy floppy or CD-ROM. The block size and last block number are also returned.
ReadBlocks()	Reads the indicated block I/O device starting at the given logical block address (LBA) and for buffer size/block size.

11
PEI Physical Memory Usage

11.1 Introduction

This section describes how physical system memory is used during PEI. The rules for using physical system memory are different before and after permanent memory registration within the PEI execution.

11.2 Before Permanent Memory Is Installed

11.2.1 Discovering Physical Memory

Before permanent memory is installed, the minimum exit condition for the PEI phase is that it has enough physical system memory to run PEIMs and the DXE IPL PPI that require permanent memory. These memory-aware PEIMs may discover and initialize additional system memory, but in doing so they must not cause loss of data in the physical system memory initialized during the earlier phase. The required amount of memory initialized and tested by PEIMs in these two phases is platform dependent.

Before permanent memory is installed, a PEIM may not assume any area of physical memory is present and initialized. During this early phase, a PEIM—usually one specific to the chipset memory controller—will initialize and test physical memory. When this PEIM has initialized and tested the physical memory, it will register the memory using the PEI Memory Service **`InstallPeiMemory()`**, which in turn will cause the PEI Foundation to create an initial Hand-Off Block (HOB) list and describe the memory. The memory that is present, initialized, and tested will reside in resource descriptor HOBs in the initial HOB list (see *Volume 3* for more information). This memory allocation PEIM may also choose to allocate some of this physical memory by doing the following:

- Creating memory allocation HOBs, as described in "Allocating Memory Using GUID Extension HOBs" on page 230.
- Using the memory allocation services **`AllocatePages()`** and **`AllocatePool()`**

Once permanent memory has been installed, the resources described in the HOB list are considered permanent system memory.

11.2.2 Using Physical Memory

A PEIM that requires permanent, fixed memory allocation must schedule itself to run after **`EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI`** is installed. To schedule itself, the PEIM can do one of the following:

- Put this PPI's GUID into the depex of the PEIM.
- Register for a notification.

The PEIM can then allocate Hand-Off Blocks (HOBs) and other memory using the same mechanisms described in “Allocating Physical Memory” on page 230.

The **AllocatePool()** service can be invoked at any time during the boot phase to discover temporary memory that will have its location translated, even before permanent memory is installed.

11.3 After Permanent Memory Is Installed

11.3.1 Allocating Physical Memory

After permanent memory is installed, PEIMs may allocate memory in four ways:

- Using a GUID Extension HOB
- Within the PEI free memory space

11.3.2 Allocating Memory Using GUID Extension HOBs

A PEIM may allocate memory for its private use by constructing a GUID Extension HOB and using the private data area defined by the GUIDed name of the HOB for private data storage.

See *Volume 3* for HOB construction rules.

11.3.3 Allocating Memory Using PEI Service

A PEIM may allocate memory using the PEI Service **AllocatePages()**. Use the **EFI_MEMORY_TYPE** values to specify the type of memory to allocate; type **EFI_MEMORY_TYPE** is defined is defined in **AllocatePages()** in the UEFI 2.0 specification.

12
Special Paths Unique to the Itanium® Processor Family

12.1 Introduction

The Itanium processor family supports the full complement of boot modes listed in the PEI CIS. In addition, however, Itanium® architecture requires an augmented flow. This flow includes a "recovery check call" in which all processors execute the PEI Foundation when an Itanium platform restarts. Each processor has its own version of temporary memory such that there are as many concurrent instances of PEI execution as there are Itanium processors.

There is a point in the multiprocessor flow, however, when all processors have to call back into the Processor Abstraction Layer A (PAL-A) component to assess whether the processor revisions and PAL-B binaries are compatible. This callback into the PAL-A does not preserve the state of the temporary memory, however. When the PAL-A returns control back to the various processors, the PEI Foundation and its associated data structures have to be reinstantiated.

At this point, however, the flow of the PEI phase is the same as for IA-32 Intel architecture in that all processors make forward progress up through invoking the DXE IPL PPI.

12.2 Unique Boot Paths for Itanium Architecture

Intel® Itanium processors possess two unique boot paths that also invoke the dispatcher located at the System Abstraction Layer entry point (SALE_ENTRY):

- Processor INIT
- Machine Check (MCHK)

INIT and MCHK are two asynchronous events that start up the Security (SEC) code/dispatcher in an Itanium®-based system. The PI Architecture security module is transparent during all the code paths except for the recovery check call that happens during a cold boot. The PEIMs that handle these events are architecture aware and do not return control to the PEI Dispatcher. They call their respective architectural handlers in the operating system.

Figure 3 shows the boot path for INIT and MCHK events.

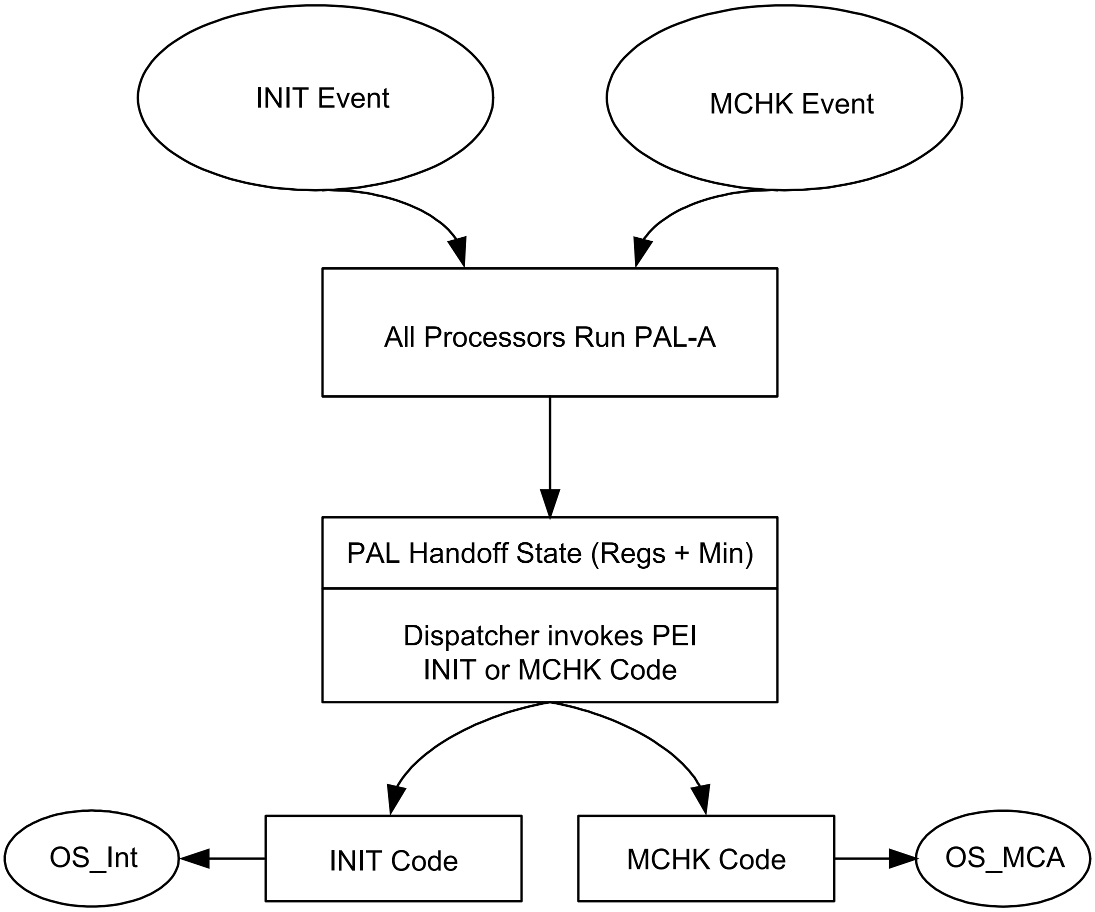


Figure 3. Itanium Processor Boot Path (INIT and MCHK)

12.3 Min-State Save Area

When the Processor Abstraction Layer (PAL) hands control to the dispatcher, it will supply the following:

- Unique handoff state in the registers
- A pointer, called the *min-state pointer,* to the minimum-state saved buffer area

This buffer is a unique per-processor save area that is registered to each processor during the normal OS boot path. The PI Architecture defines a unique, PI Architecture-specific data pointer, **EFI_PEI_MIN_STATE_DATA**, that is attached to this min-state pointer. This data structure is defined in the next topic.

Figure 4 shows a typical organization of a min-state buffer. The PEI Data Pointer references **EFI_PEI_MIN_STATE_DATA**.

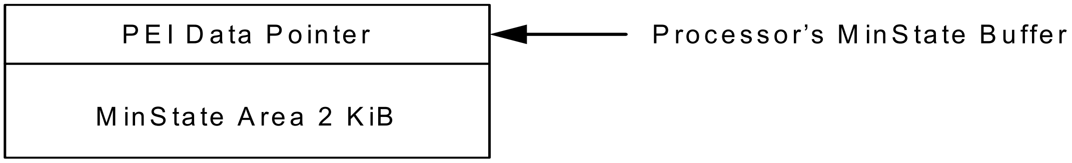


Figure 4. Min-State Buffer Organization

EFI_PEI_MIN_STATE_DATA

Note: *This data structure is for the Itanium® processor family only.*

Summary

A structure that encapsulates the Processor Abstraction Layer (PAL) min-state data structure for purposes of firmware state storage and reference.

Prototype

```
typedef struct {
  UINT64      OsInitHandlerPointer;
  UINT64      OsInitHandlerGP;
  UINT64      OsInitHandlerChecksum;
  UINT64      OSMchkHandlerPointer;
  UINT64      OSMchkHandlerGP;
  UINT64      OSMchkHandlerChecksum;
  UINT64      PeimInitHandlerPointer;
  UINT64      PeimInitHandlerGP;
  UINT64      PeimInitHandlerChecksum;
  UINT64      PeimMchkHandlerPointer;
  UINT64      PeimMchkHandlerGP;
  UINT64      PeimMckhHandlerChecksum;
  UINT64      TypeOfOSBooted;
  UINT8       MinStateReserved[0x400];
  UINT8       OEMReserved[0x400];
} EFI_PEI_MIN_STATE_DATA;
```

Parameters

OsInitHandlerPointer

The address of the operating system's INIT handler. The INIT is a restart type for the Itanium processor family.

OsInitHandlerGP

The value of the operating system's INIT handler's General Purpose (GP) register. Per the calling conventions for the Itanium processor family, the GP must be set before invoking the function.

OsInitHandlerChecksum

A 64-bit checksum across the contents of the operating system's INIT handler. This can be used by the PEI firmware to corroborate the integrity of the INIT handler prior to invocation.

OSMchkHandlerPointer

The address of the operating system's Machine Check (MCHK) handler. MCHK is a restart type for the Itanium processor family.

OSMchkHandlerGP

The value of the operating system's MCHK handler's GP register. Per the calling conventions for the Itanium processor family, the GP must be set before invoking the function.

OSMchkHandlerChecksum

A 64-bit checksum across the contents of the operating system's MCHK handler. This can be used by the PEI firmware to corroborate the integrity of the MCHK handler prior to invocation.

PeimInitHandlerPointer

The address of the PEIM's INIT handler.

PeimInitHandlerGP

The value of the PEIM's INIT handler's GP register. Per the calling conventions for the Itanium processor family, the GP must be set before invoking the function.

PeimInitHandlerChecksum

A 64-bit checksum across the contents of the PEIM's INIT handler. This can be used by the PEI firmware to corroborate the integrity of the INIT handler prior to invocation.

PeimMchkHandlerPointer

The address of the PEIM's MCHK handler.

PeimMchkHandlerGP

The value of the PEIM's MCHK handler's GP register. Per the calling conventions for the Itanium processor family, the GP must be set before invoking the function.

PeimMckhHandlerChecksum

A 64-bit checksum across the contents of the PEIM's MCHK handler. This can be used by the PEI firmware to corroborate the integrity of the MCHK handler prior to invocation.

TypeOfOSBooted

Details the type of operating system that was originally booted. This allows for different preliminary processing in firmware based upon the target OS.

MinStateReserved

Reserved bytes that must not be interpreted by OEM firmware. Future versions of PEI may choose to expand in this range.

OEMReserved

Reserved bytes for the OEM. PEI core components should not attempt to interpret the contents of this region.

Description

A 64-bit PEI data pointer is defined at the beginning of the Itanium processor family min-state data structure. This data pointer references an **EFI_PEI_MIN_STATE_DATA** structure that is defined above. This latter structure contains the entry points of INIT and MCHK code blocks. The pointers are defined such that the INIT and MCHK code can be either written as ROM-based PEIMs or as DXE drivers. The distinction between PEIM and DXE driver are at the OEM's discretion.

In Itanium® architecture, the PEI firmware must register a min-state with the PAL. This min-state is memory when the PAL code can deposit processor-specific information upon various restart events (INIT, RESET, Machine Check). Upon receipt of INIT or MCHK, the PEI firmware shall first invoke the PEIM INIT or MCHK handlers, respectively, and then the OS INIT or MCHK handler. The min-state data structure is a natural location from which to reference the PEI data structure that contains these latter entry points.

12.4 Dispatching Itanium Processor Family PEIMs

The Itanium processor family dispatcher starts dispatching all the PEIMs as it resolves the dependency grammar contained within their headers. Because all Itanium processors enter into SALE_ENTRY for a recovery check, some of the PEIMs will contain multiprocessor (MP) code and will work on all processors. The behavior of a particular PEIM that is dispatched depends on the following:

- Handoff state given by the Processor Abstraction Layer (PAL)
- The boot mode flag

Once the processor runs some code and one of the recovery check PEIM determines that the firmware needs to be recovered, it flips the boot flag to recovery and invokes the dispatcher again in recovery mode.

If it is a nonrecovery situation (normal boot), then the recovery check PEIM wakes up all the processors and returns them to PAL-A for further initialization. Note that when control for a normal boot returns back to the PAL to run PAL-B code, all of the register contents are lost. When control returns to the dispatcher, the PEIMs gain control in the dispatched order and can determine the memory topology (if needed in a platform implementation) by reading the memory controller registers of the chipset. The PEIMs can then build Hand-Off Blocks (HOBs).

When the first phase is done, there will be coherent memory on the system that all the node processors can see. The system then begins to execute the dispatcher in a second phase, during which it builds HOBs. On a multinode system with many processors, the configuration of memory may take several steps and therefore quite a bit of code.

When the second phase is done, the last PEIM will build DXE as described in “PEI to DXE Handoff” on page 217 and hand control to the PI Architecture DXE phase for further initialization of the platform.

Figure 5 depicts the initial flow between PAL-A , PAL-B, and the PEI Foundation located at SALE_ENTRY point.

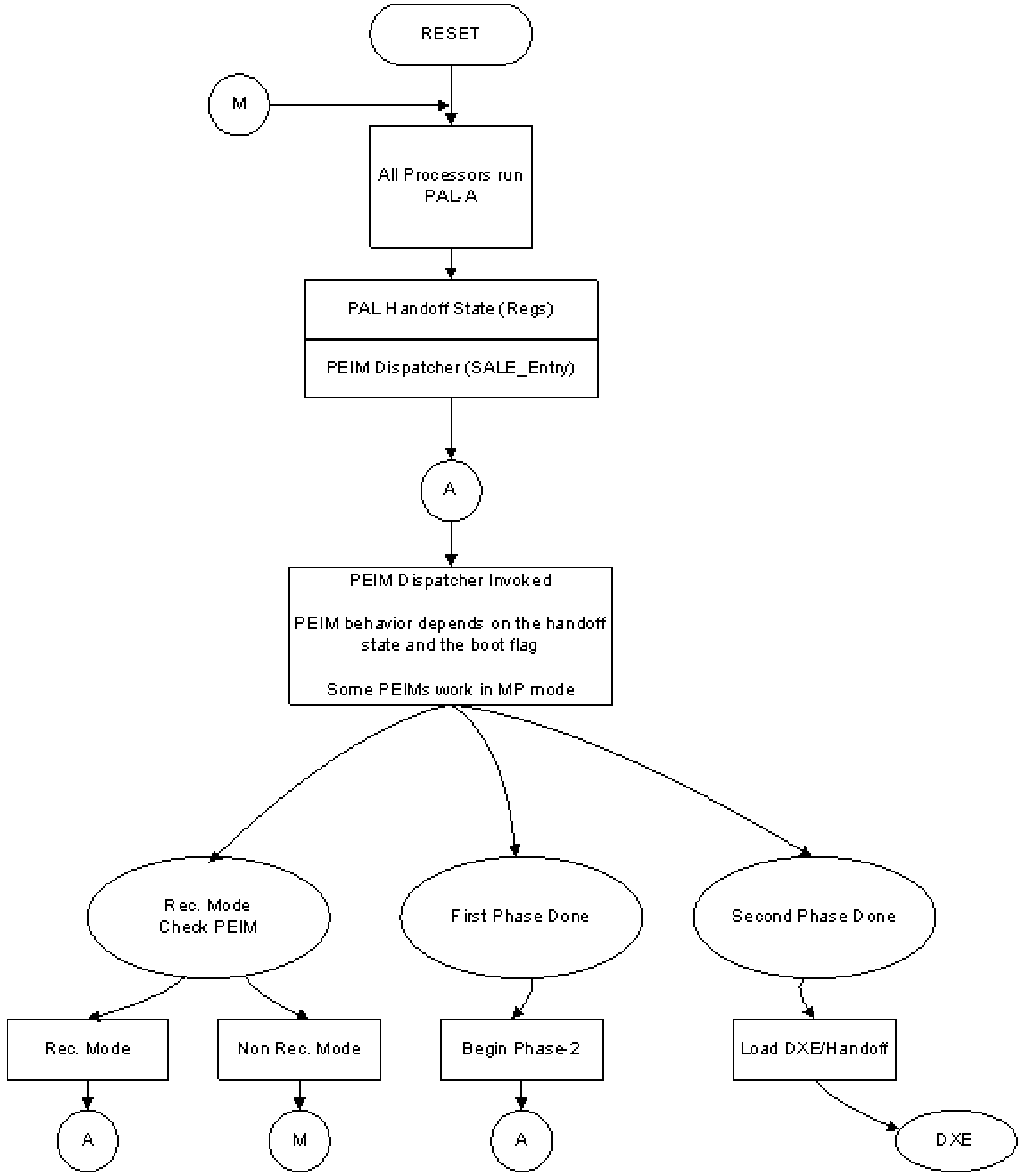


Figure 5. Boot Path in Itanium Processors

13
Security (SEC) Phase Information

13.1 Introduction

The Security (SEC) phase is the first phase in the PI Architecture architecture and is responsible for the following:

- Handling all platform restart events
- Creating a temporary memory store
- Serving as the root of trust in the system
- Passing handoff information to the PEI Foundation

In addition to the minimum architecturally required handoff information, the SEC phase can pass optional information to the PEI Foundation, such as the SEC Platform Information PPI or information about the health of the processor.

The tasks listed above are common to all processor microarchitectures. However, there are some additions or differences between IA-32 and Itanium processors, which are discussed in “Processor-Specific Details” on page 243.

13.2 Responsibilities

13.2.1 Handling All Platform Restart Events

The Security (SEC) phase is the unit of processing that handles all platform restart events, including the following:

- Applying power to the system from an unpowered state
- Restarting the system from an active state
- Receiving various exception conditions

The SEC phase is responsible for aggregating any state information so that some PEIM can deduce the health of the processor upon the respective restart.

13.2.2 Creating a Temporary Memory Store

The Security (SEC) phase is also responsible for creating some temporary memory store. This temporary memory store can include but is not limited to programming the processor cache to behave as a linear store of memory. This cache behavior is referred to as “no evictions mode” in that access to the cache should always represent a hit and not engender an eviction to the main memory backing store; this “no eviction” is important in that during this early phase of platform evolution, the main memory has not been configured and such as eviction could engender a platform failure.

13.2.3 Serving As the Root of Trust in the System

Finally, the Security (SEC) phase represents the root of trust in the system. Any inductive security design in which the integrity of the subsequent module to gain control is corroborated by the caller must have a root, or "first," component. For any PI Architecture deployment, the SEC phase represents the initial code that takes control of the system. As such, a platform or technology deployment may choose to authenticate the PEI Foundation from the SEC phase before invoking the PEI Foundation.

13.2.4 Passing Handoff Information to the PEI Foundation

Regardless of the other responsibilities listed in this section, the Security (SEC) phase's final responsibility is to convey the following handoff information to the PEI:

- State of the platform
- Location and size of the Boot Firmware Volume (BFV)
- Location and size of the temporary RAM
- Location and size of the stack
- Optionally, one or more HOBs via the **EFI_SEC_HOB_DATA_PPI**.

This handoff information listed above is passed to the PEI as arguments to the PEI Foundation entry point described in section 5.2.

13.3 SEC Platform Information PPI

Handoff information is passed from the Security (SEC) phase to the PEI Foundation using the **EFI_SEC_PEI_HAND_OFF** structure and the list of PPI descriptors passed to the PEI entry point. One of these PPIs, **EFI_SEC_PLATFORM_INFORMATION_PPI**, can be used to pass handoff information from SEC to the PEI Foundation. This PPI abstracts platform-specific information that the PEI Foundation needs to discover where to begin dispatching PEIMs.

13.4 SEC HOB Data PPI

HOB data can be passed forward from the SEC phase to PEI or DXE consumers using HOBs. If the **EFI_SEC_HOB_DATA_PPI** is in the list of PPIs passed to the PEI entry point, the PEI Foundation will call the **GetHobs()** member function and installed all HOBs returned into the HOB list. It does this after installing all PPIs passed from SEC into the PPI database and before dispatching any PEIMs.

13.5 Health Flag Bit Format

The Health flag contains information that is generated by microcode, hardware, and/or the Itanium processor Processor Abstraction Layer (PAL) code about the state of the processor upon reset. Type **EFI_HEALTH_FLAGS** is defined in **SEC_PLATFORM_INFORMATION_PPI.PlatformInformation()**.

In an Itanium®-based system, the Health flag is passed from PAL-A after restarting. It is the means by which the PAL conveys the state of the processor to the firmware, such as PI. The handoff state is separated between the PAL and PI because the code is provided by different vendors; Intel provides the PAL and various OEMs design the PI firmware.

The Health flag is used by both IA-32 and Itanium architectures, but `Tested` (Te) is the only common bit. IA-32 has the built-in self-test (BIST), but none of the other capabilities.

Figure 6 depicts the bit format in the Health flag.

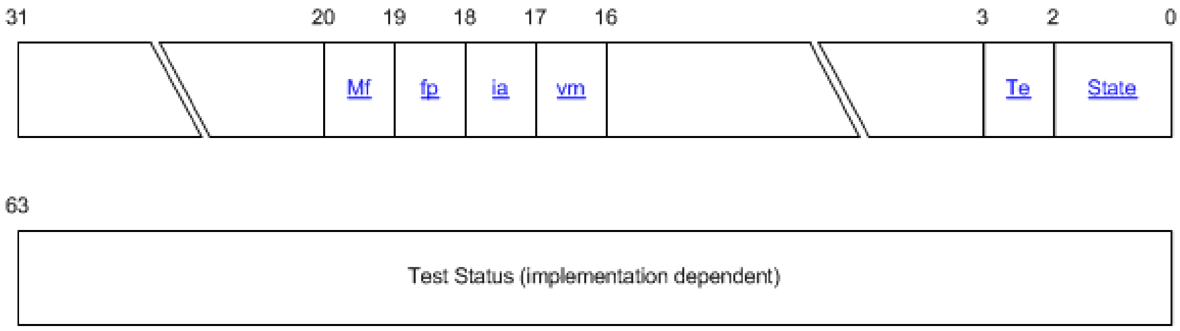


Figure 6. Health Flag Bit Format

Table 24 explains the bit fields in the Health flag. IA-32 ignores all bits except `Tested` (Te).

Table 24. Health Flag Bit Field Description

Field	Parameter Name in `EFI_HEALTH_FLAGS`	Bit #	Description
State	`Status`	0:1	A 2-bit field indicating self-test state after reset. For more information, see "Self-Test State Parameter" on page 242.
Te	`Tested`	2	A 1-bit field indicating whether testing has occurred. If this field is zero, the processor has not been tested, and no further fields in the self-test State parameter are valid.
Vm	`VirtualMemoryUnavailable`	16	A 1-bit field. If set to 1, indicates that virtual memory features are not available.
Ia	`Ia32ExecutionUnavailable`	17	A 1-bit field. If set to 1, indicates that IA-32 execution is not available.
Fp	`FloatingPointUnavailable`	18	A 1-bit field. If set to 1, indicates that the floating point unit is not available.

Mf	`MiscFeaturesUnavailable`	19	A 1-bit field. If set to 1, indicates miscellaneous functional failure other than vm, ia, or fp. The test status field provides additional information on test failures when the State field returns a value of performance restricted or functionally restricted. The value returned is implementation dependent.

13.5.1 Self-Test State Parameter

Self-test state parameters are defined in the same format for IA-32 Intel® processors and the Intel® Itanium® processor family. Some of the test status bits may not be relevant to IA-32 processors. In that case, these bits will read **NULL** on IA-32 processors.

Table 25 indicates the meanings for various values of the self-test State parameter (bits 0:1) of the Health flag.

Table 25. Self-Test State Bit Values

State	Value	Description
Catastrophic Failure	N/A	Processor is not executing.
Healthy	00	No failure in functionality or performance.
Performance Restricted	01	No failure in functionality but performance is restricted.
Functionally Restricted	10	Some code may run but functionality is restricted and performance may also be affected.

If the state field indicates that the processor is functionally restricted, then the vm, ia, and fp fields in the Health flag specify additional information about the functional failure. See Table 24 for a description of these fields.

To further qualify "Functionally Restricted," the following requirements will be met:

- The processor or PAL (for the Itanium processor family) has detected and isolated the failing component so that it will not be used.
- The processor must have at least one functioning memory unit, arithmetic logic unit (ALU), shifter, and branch unit.
- The floating-point unit may be disabled.
- For the Itanium processor family, the Register Stack Engine (RSE) is not required to work, but register renaming logic must work properly.
- The paths between the processor-controlled caches and the register files must work during the tests.
- Loads from the firmware address space must work correctly.

13.6 Processor-Specific Details

13.6.1 SEC Phase in IA-32 Intel Architecture

In 32-bit Intel® architecture (IA-32), the Security (SEC) phase of the PI Architecture is responsible for several activities:

- Locating the PEI Foundation
- Passing control directly to PEI using an architecturally defined handoff state
- Initializing processor-controlled memory resources, such as the processor data cache, that can be used as a linear extent of memory for a call stack (if supported)

Figure 7 below shows the steps completed during PEI initialization for IA-32.

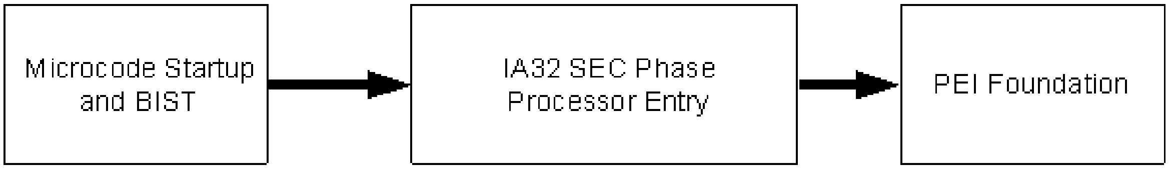


Figure 7. PEI Initialization Steps in IA-32

13.6.2 SEC Phase in the Itanium Processor Family

Itanium architecture contains enough hooks to authenticate the PAL-A and PAL-B code distributed by the processor vendor.

The internal microcode on the processor silicon that starts up on a power-good reset finds the first layer of processor abstraction code (called PAL-A) located in the Boot Firmware Volume (BFV) using architecturally defined pointers in the BFV. It is the responsibility of this microcode to authenticate that the PAL-A code layer from the processor vendor has not been tampered.

If the authentication of the PAL-A layer passes, then control passes on to the PAL-A layer. The PAL-A layer then authenticates the next layer of processor abstraction code (called PAL-B) before passing control to it.

In addition, the SEC phase of the PI Architecture is also responsible for locating the PEI Foundation and verifying its authenticity.

Figure 8 summarizes the SEC phase in the Itanium® processor family.

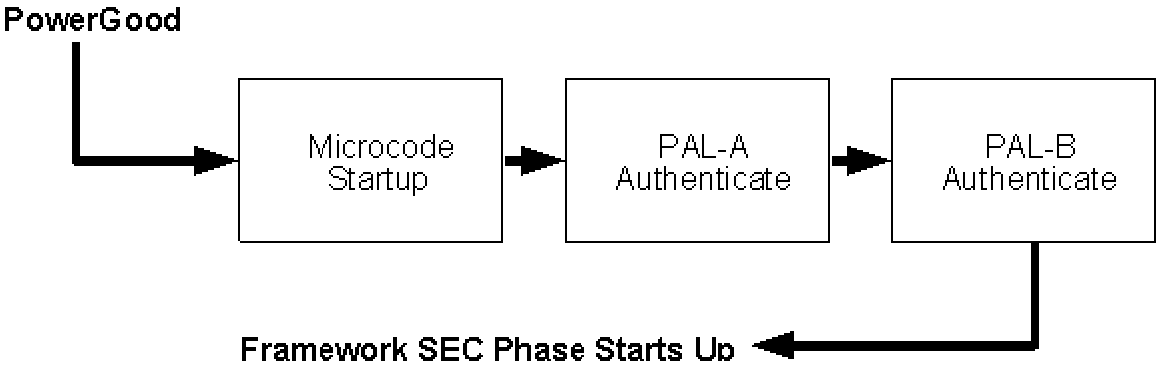


Figure 8. Security (SEC) Phase in the Itanium Processor Family

14
Dependency Expression Grammar

14.1 Dependency Expression Grammar

This topic contains an example BNF grammar for a PEIM dependency expression compiler that converts a dependency expression source file into a dependency section of a PEIM stored in a firmware volume.

14.1.1 Example Dependency Expression BNF Grammar

```
<depex>    ::= <bool>
<bool>     ::= <bool> AND <term>
             | <bool> OR <term>
             | <term>
<term>     ::= NOT <factor>
             | <factor>
<factor>   ::= <bool>
             | TRUE
             | FALSE
             | GUID
             | END
<guid>     ::= ‘{‘ <hex32> ‘,’ <hex16> ‘,’ <hex16> ‘,’
               <hex8> ‘,’ <hex8> ‘,’ <hex8> ‘,’ <hex8> ‘,’
               <hex8> ‘,’ <hex8> ‘,’ <hex8> ‘,’ <hex8> ’}’
<hex32>    ::= <hexprefix> <hexvalue>
<hex16>    ::= <hexprefix> <hexvalue>
<hex8>     ::= <hexprefix> <hexvalue>
<hexprefix>::= ‘0’ ‘x’
             | ‘0’ ‘X’
<hexvalue> ::= <hexdigit> <hexvalue>
             | <hexdigit>
<hexdigit> ::= [0-9]
             | [a-f]
             | [A-F]
```

14.1.2 Sample Dependency Expressions

The following contains three examples of source statements using the BNF grammar from above along with the opcodes, operands, and binary encoding that a dependency expression compiler would generate from these source statements.

```
//
// Source
//
EFI_PEI_CPU_IO_PPI_GUID AND EFI_PEI_READ_ONLY_VARIABLE_ACCESS_PPI_GUID
END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   =======================    =======================
0x00 : 02                          PUSH
0x01 : 26 25 73 b0 c8 38 40 4b    EFI_PEI_CPU_IO_PPI_GUID
       88 77 61 c7 b0 6a ac 45
0x11 : 02                          PUSH
0x12 : b1 cc ba 26 42 6f d4 11
EFI_PEI_READ_ONLY_VARIABLE_ACCESS_PPI_GUID
       bc e7 00 80 c7 3c 88 81
0x22 : 03                          AND
0x23 : 08                          END
```

15
TE Image

15.1 Introduction

The *Terse Executable* (TE) image format was created as a mechanism to reduce the overhead of the PE/COFF headers in PE32/PE32+ images, resulting in a corresponding reduction of image sizes for executables running in the PI Architecture environment. Reducing image size provides an opportunity for use of a smaller system flash part.

TE images, both drivers and applications, are created as PE32 (or PE32+) executables. PE32 is a generic executable image format that is intended to support multiple target systems, processors, and operating systems. As a result, the headers in the image contain information that is not necessarily applicable to all target systems. In an effort to reduce image size, a new executable image header (TE) was created that includes only those fields from the PE/COFF headers required for execution under the PI Architecture. Since this header contains the information required for execution of the image, it can replace the PE/COFF headers from the original image. This specification defines the TE header, the fields in the header, and how they are used in the PI Architecture's execution environment.

15.2 PE32 Headers

A PE file header, as described in the *Microsoft Portable Executable and Common Object File Format Specification*, contains an MS-DOS* stub, a PE signature, a COFF header, an optional header, and section headers. For successful execution, PEIMs in the PI Architecture require very little of the data from these headers, and in fact the MS-DOS stub and PE signature are not required at all.

See Table 26 and Table 27 for the necessary fields and their descriptions.

Table 26. COFF Header Fields Required for TE Images

COFF Header	Description
Machine	Target machine identifier. 2 bytes in both COFF header and TE header
NumberOfSections	Number of sections/section headers. 2 bytes in COFF header, 1 byte in TE header

Table 27. Optional Header Fields Required for TE Images

OPTIONAL Header	Description
AddressOfEntryPoint	Address of entry point relative to image base. 4 bytes in both optional header and TE header
BaseOfCode	Offset from image base to the start of the code section. 4 bytes in both optional header and TE header

ImageBase	Image’s linked address. 4 bytes in OptionalHeader32, 8 bytes in OptionalHeader64, and 8 bytes in TE header
Subsystem	Subsystem required to run the image. 2 bytes in optional header, 1 byte in TE header

TE Header

Summary

To reduce the overhead of PE/COFF headers in the PI Architecture's environment, a minimal (TE) header can be defined that includes only those fields required for execution in the PI Architecture. This header can then be used to replace the original headers at the start of the original image.

Prototype

```
typedef struct {
  UINT16                        Signature;
  UINT16                        Machine;
  UINT8                         NumberOfSections;
  UINT8                         Subsystem;
  UINT16                        StrippedSize;
  UINT32                        AddressOfEntryPoint;
  UINT32                        BaseOfCode;
  UINT64                        ImageBase;
  EFI_IMAGE_DATA_DIRECTORY      DataDirectory[2];
} EFI_TE_IMAGE_HEADER;
```

Parameters

Signature

TE image signature

Machine

Target machine, as specified in the original image’s file header

NumberOfSections

Number of sections, as specified in the original image’s file header

Subsystem

Target subsystem, as specified in the original optional header

StrippedSize

Number of bytes removed from the base of the original image

AddressOfEntryPoint

Address of the entry point to the driver, as specified in the original image’s optional header

BaseOfCode

Base of the code, as specified in the original image’s optional header

ImageBase

Image base, as specified in the original image’s optional header (0-extended to 64-bits for PE32 images)

DataDirectory

Directory entries for base relocations and the debug directory from the original image’s corresponding directory entries. See “Related Definitions” below.

Field Descriptions

In the **EFI_TE_IMAGE HEADER**, the *Machine, NumberOfSections, Subsystem, AddressOfEntryPoint, BaseOfCode*, and *ImageBase* all come directly from the original PE headers to enable partial reconstitution of the original headers if necessary.

The 2-byte *Signature* should be set to **EFI_TE_IMAGE_HEADER_SIGNATURE** to designate the image as TE, as opposed to the “MZ” signature at the start of standard PE/COFF images.

The *StrippedSize* should be set to the number of bytes removed from the start of the original image, which will typically include the MS-DOS, COFF, and optional headers, as well as the section headers. This size can be used by image loaders and tools to make appropriate adjustments to the other fields in the TE image header. Note that *StrippedSize* does not take into account the size of the TE image header that will be added to the image. That is to say, the delta in the total image size when converted to TE is *StrippedSize* – sizeof (**EFI_TE_IMAGE_HEADER**). This will typically need to be taken into account by tools using the fields in the TE header.

The *DataDirectory* array contents are copied directly from the base relocations and debug directory entries in the original optional header data directories. This image format also assumes that file alignment is equal to section alignment.

Related Definitions

```
//*****************************************************
//EFI_IMAGE_DATA_DIRECTORY
//*****************************************************
typedef struct {
  UINT32   VirtualAddress;
  UINT32   Size;
} EFI_IMAGE_DATA_DIRECTORY;
#define EFI_TE_IMAGE_DIRECTORY_ENTRY_BASERELOC      0
#define EFI_TE_IMAGE_DIRECTORY_ENTRY_DEBUG          1

#define EFI_TE_IMAGE_HEADER_SIGNATURE     0x5A56  // “VZ”
```

16
TE Image Creation

16.1 Introduction

This section describes the tool requirements to create a TE image.

16.2 TE Image Utility Requirements

A utility that creates TE images from standard PE/COFF images must be able to do the following:

- Create an **EFI_TE_IMAGE_HEADER** in memory
- Parse the PE/COFF headers in an existing image and extract the necessary fields to fill in the **EFI_TE_IMAGE_HEADER**
- Fill in the signature and stripped size fields in the **EFI_TE_IMAGE_HEADER**
- Write out the **EFI_TE_IMAGE_HEADER** to a new binary file
- Write out the contents of the original image, less the stripped headers, to the output file

Since some fields from the PE/COFF headers have a smaller corresponding field in the TE image header, the utility must be able to recognize if the original value from the PE/COFF header does not fit in the TE header. In this case, the original image is not a candidate for conversion to TE image format.

16.3 TE Image Relocations

Relocation fix ups in TE images are not modified by the TE image creation process. Therefore, if a TE image is to be relocated, the loader/relocator must take into consideration the stripped size and size of a TE image header when applying fix ups.

17
TE Image Loading

17.1 Introduction

This section describes the use of the TE image and how embedded, execute-in-place environments can invoke these images.

17.2 XIP Images

For execute-in-place (XIP) images that do not require relocations, loading a TE image simply requires that the loader adjust the image's entry point from the value specified in the **EFI_TE_IMAGE_HEADER**. For example, if the image (and thus the TE header) resides at memory location *LoadedImageAddress*, then the actual entry for the driver is computed as follows:

```
EntryPoint = LoadedImageAddress + sizeof (EFI_TE_IMAGE_HEADER)
+
               ((EFI_TE_IMAGE_HEADER *)LoadedImageAddress)->
               AddressOfEntryPoint – ((EFI_TE_IMAGE_HEADER *)
               LoadedImageAddress)->StrippedSize;
```

17.3 Relocated Images

To successfully load and relocate a TE image requires the same operations as required for XIP code. However, for images that can be relocated, the image loader must make adjustments for all the relocation fix ups performed. Details on this operation are beyond the scope of this document, but suffice it to say that the adjustments will be computed in a manner similar to the *EntryPoint* adjustment made in XIP Images.

17.4 PIC Images

A TE Image is Position Independent Code (PIC) if it can be executed in flash and shadowed to memory without any fix ups. In this case, the TE Image Relocation Data Directory Entry Virtual Address is non-zero, but the Relocation Data Directory Size is zero.