Platform Initialization (PI) Specification

Volume 1:
Pre-EFI Initialization Core Interface

Version 1.7

January 2019

Revision History

Revision	Mantis ID / Description	Date
1.7	• 1848 PEI Core PEIM Migration Support Change • 1856 SecCore/PeiCore BFV Requirement Change • 1860 MM MP Protocol issues • 1885 Add extended data for EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR • 1889 New Status Codes • 1891 PEI delayed dispatch • 1892 new pi spec revision • 1955 set (*Attributes) when EFI_PEI_GET_VARIABLE2 returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL • 1967 New architectural PPI for PI PEI Core FV Location • 1969 Incompatible Pci Ignore Option Rom	January 2019
1.6 Errata A	• 1828 Add decorator 'OPTIONAL' for Attributes parameter of EFI_PEI_GET_VARIABLE2. • 1849 Issues in PI Spec Vol. 5 Ch. 18 • 1857 Specifies how notifications are passed from SEC to PEI. • 1884 Variadic API issue in S3 save API • 1907 Clarification of the EFI_MM_COMMUNICATION_PROTOCOL • 1940 Clarify EFI_MM_CONFIGURATION_PROTOCOL_GUID	December 2018
1.6	• 1567 Layered SPI bus • 1648 PI Binding for RISC-V • 1746 Add an FV Extended Header entry that contains the used size of the FV • 1763 MM Handler state notification protocol • 1764 Add additional alignment • 1768 Update the PI Spec to 1.6 • 1777 Update Revision History • 1778 Update front matter	April 2017
1.5 Errata A	• 1587 pre permanent memory page allocation • 1665 Incorrect status code for an AP calling EFI_MP_SERVICES_PROTOCOL.SwitchBSP() • 1734 Outdated EFI spec reference in Save State Write • 1735 Several copy & paste errors in Save State Write • 1747 Clarify that MM_ACCESS_PROTOCOL should cover all MMRAM region used by the platform	April 2017

Revision	Mantis ID / Description	Date
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 • 1593 coalesce language enhancements • 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 • 1607 Update MM guid def'n to match edkII impl • 1626 Add new Status Code for BDS Attempting UEFI BootOrder entries • 1628 Minor feedback for PI 1.5 Vol 4 SMM Draft • 1666 Graphics Device Info Hob	4/26/16
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

Revision	Mantis ID / Description	Date
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.	2/20/15
1.3 Errata A	• 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
1.3	• 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

Revision	Mantis ID / Description	Date
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 • 937SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 • 969Vol 1 errata: TE Header parameters	10/26/12
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
1.2.1	• 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

Revision	Mantis ID / Description	Date
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 • 737 Remove SMM Communication ACPI Table definition . • 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 • 753 SIO PEI and UEFI-Driver Model Architecture • 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

Revision	Mantis ID / Description	Date
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 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	• 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

Revision	Mantis ID / Description	Date
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	• 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

Revision	Mantis ID / Description	Date
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
1.1 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

Revision	Mantis ID / Description	Date
1.1 Errata	• 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	Revises typographical errors and minor omissions--see Errata for details	04/25/08
1.1 correction	Restore (missing) MP protocol	03/12/08
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

Revision	Mantis ID / Description	Date
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
1.0	Initial public release.	8/21/06

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.

Table of Contents

Revision History iii
Table of Contents xiv
List of Figures xxi
List of Tables xxii

1 Introduction 1
1.1 Overview 1
1.2 Organization of the PEI CIS 1
1.3 Conventions Used in this Document 2
1.3.1 Data Structure Descriptions 2
1.3.2 Procedure Descriptions 2
1.3.3 Instruction Descriptions 3
1.3.4 PPI Descriptions 3
1.3.5 Pseudo-Code Conventions 4
1.3.6 Typographic Conventions 4
1.4 Requirements 5
1.5 Conventions used in this document 6
1.5.1 Number formats 6
1.5.2 Binary prefixes 6

2 Overview 8
2.1 Introduction 8
2.2 Design Goals 8
2.3 Pre-EFI Initialization (PEI) Phase 9
2.4 PEI Services 10
2.5 PEI Foundation 11
2.6 PEI Dispatcher 11
2.7 Pre-EFI Initialization Modules (PEIMs) 12
2.8 PEIM-to-PEIM Interfaces (PPIs) 12
2.9 Firmware Volumes 13

3 PEI Services Table 14
3.1 Introduction 14
3.2 PEI Services Table 14
3.2.1 EFI_PEI_SERVICES 14

4 Services - PEI 20
4.1 Introduction 20
4.2 PPI Services 20
InstallPpi() 21
ReinstallPpi() 22
LocatePpi() 23
NotifyPpi() 25
4.3 Boot Mode Services 25

GetBootMode() 26
SetBootMode() 28
4.4 HOB Services 28
GetHobList() 29
CreateHob() 30
4.5 Firmware Volume Services 31
FfsFindNextVolume() 32
FfsFindNextFile() 33
FfsFindSectionData() 35
FfsFindSectionData3() 36
FfsFindFileByName() 38
FfsGetFileInfo() 39
FfsGetFileInfo2() 41
FfsGetVolumeInfo() 43
RegisterForShadow() 45
4.6 PEI Memory Services 45
InstallPeiMemory() 46
AllocatePages() 47
AllocatePool() 49
CopyMem() 50
FreePages() 51
SetMem() 53
4.7 Status Code Service 53
ReportStatusCode() 54
4.8 Reset Services 57
ResetSystem() 57
4.9 I/O and PCI Services 57

5 PEI Foundation 58

5.1 Introduction 58
5.1.1 Prerequisites 58
5.1.2 Processor Execution Mode 58
5.2 PEI Foundation Entry Point 60
5.2.1 PEI Foundation Entry Point 60
5.3 PEI Calling Convention Processor Binding 63
5.4 PEI Services Table Retrieval 63
5.4.1 X86 63
5.4.2 x64 64
5.4.3 Itanium Processor Family – Register Mechanism 64
5.4.4 ARM Processor Family – Register Mechanism 65
5.4.5 AArch64 Processor Family – Register Mechanism 66
5.4.6 RISC-V Processor Family – Register Mechanism 66
5.5 PEI Dispatcher Introduction 67
5.6 Ordering 67
5.6.1 Requirements 67
5.6.2 Requirement Representation and Notation 67
5.6.3 PEI a priori File Overview 68

PEI_APRIORI_FILE_NAME_GUID 69
5.6.4 Firmware Volume Image Files 70
5.6.5 PEIM Dependency Expressions 70
5.6.6 Types of Dependencies 70
5.7 Dependency Expressions 70
5.7.1 Introduction 70
PUSH 72
AND 74
OR 75
NOT 76
TRUE 77
FALSE 78
END 79
5.7.2 Dependency Expression with No Dependencies 80
5.7.3 Empty Dependency Expressions 80
5.7.4 Dependency Expression Reverse Polish Notation (RPN) 80
5.8 Dispatch Algorithm 80
5.8.1 Overview 80
5.8.2 Requirements 81
5.8.3 Example Dispatch Algorithm 83
5.8.4 Dispatching When Memory Exists 84
5.8.5 PEIM Dispatching 85
5.8.6 PEIM Authentication 85

6 Architectural PPIs 86
6.1 Introduction 86
6.2 Required Architectural PPIs 86
6.2.1 Master Boot Mode PPI (Required) 86
EFI_PEI_MASTER_BOOT_MODE_PPI (Required) 86
6.2.2 DXE IPL PPI (Required) 87
EFI_DXE_IPL_PPI (Required) 87
EFI_DXE_IPL_PPI.Entry() 88
6.2.3 Memory Discovered PPI (Required) 90
EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI (Required) 90
6.3 Optional Architectural PPIs 91
6.3.1 Boot in Recovery Mode PPI (Optional) 91
EFI_PEI_BOOT_IN_RECOVERY_MODE_PPI (Optional) 91
6.3.2 End of PEI Phase PPI (Optional) 92
EFI_PEI_END_OF_PEI_PHASE_PPI (Optional) 92
6.3.3 PEI Reset PPI 93
EFI_PEI_RESET_PPI (Optional) 93
6.3.4 PEI Reset2 PPI 93
EFI_PEI_RESET2_PPI (Optional) 93
ResetSystem() 95
6.3.5 Status Code PPI (Optional) 97
EFI_PEI_PROGRESS_CODE_PPI (Optional) 97
6.3.6 Security PPI (Optional) 98

EFI_PEI_SECURITY2_PPI (Optional) 98
EFI_PEI_SECURITY2_PPI.AuthenticationState() 99
6.3.7 Temporary RAM Support PPI (Optional).......... 100
EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI (Optional) 100
EFI_PEI_TEMPORARY_RAM_SUPPORT_PPI.TemporaryRamMigration().... 102
6.3.8 Temporary RAM Done PPI (Optional).......... 103
EFI_PEI_TEMPORARY_RAM_DONE_PPI (Optional) 103
EFI_PEI_TEMPORARY_RAM_DONE_PPI.TemporaryRamDone () 105
6.3.9 EFI_PEI_CORE_FV_LOCATION_PPI.......... 105

7 PEIMs 107

7.1 Introduction 107
7.2 PEIM Structure.......... 107
7.2.1 PEIM Structure Overview 107
7.2.2 Relocation Information 108
7.2.3 Authentication Information 109
7.3 PEIM Invocation Entry Point 110
7.3.1 EFI_PEIM_ENTRY_POINT2.......... 110
7.4 PEIM Descriptors 111
7.4.1 PEIM Descriptors Overview 111
EFI_PEI_DESCRIPTOR 112
EFI_PEI_NOTIFY_DESCRIPTOR 113
EFI_PEI_PPI_DESCRIPTOR.......... 115
7.5 PEIM-to-PEIM Communication 116
7.5.1 Overview 116
7.5.2 Dynamic PPI Discovery.......... 117

8 Additional PPIs 118

8.1 Introduction 118
8.2 Required Additional PPIs 118
8.2.1 PCI Configuration PPI (Required) 118
EFI_PEI_PCI_CFG2_PPI.......... 120
EFI_PEI_PCI_CFG2_PPI.Read().......... 122
EFI_PEI_PCI_CFG2_PPI.Write().......... 124
EFI_PEI_PCI_CFG2_PPI.Modify().......... 125
8.2.2 Stall PPI (Required) 126
EFI_PEI_STALL_PPI (Required) 126
EFI_PEI_STALL_PPI.Stall().......... 127
8.2.3 Variable Services PPI (Required) 128
EFI_PEI_READ_ONLY_VARIABLE2_PPI.......... 128
EFI_PEI_READ_ONLY_VARIABLE2_PPI.GetVariable 129
EFI_PEI_READ_ONLY_VARIABLE2_PPI.NextVariableName.......... 131
8.3 Optional Additional PPIs 133
8.3.1 SEC Platform Information PPI (Optional).......... 133
EFI_SEC_PLATFORM_INFORMATION_PPI (Optional) 133
EFI_SEC_PLATFORM_INFORMATION_PPI.PlatformInformation() 134
EFI_SEC_PLATFORM_INFORMATION2_PPI (Optional) 138
EFI_SEC_PLATFORM_INFORMATION2_PPI.PlatformInformation2() 139

8.3.2 Loaded Image PPI (Optional) ... 141
EFI_PEI_LOADED_IMAGE_PPI ... 141
8.3.3 SEC HOB PPI ... 141
EFI_SEC_HOB_DATA_PPI ... 141
EFI_SEC_HOB_DATA_PPI.GetHobs() ... 143
8.3.4 Recovery ... 143
EFI_PEI_RECOVERY_MODULE_PPI ... 144
EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule() ... 146
EFI_PEI_DEVICE_RECOVERY_MODULE_PPI ... 146
EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.
GetNumberRecoveryCapsules() ... 148
EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. GetRecoveryCapsuleInfo().. 149
EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. LoadRecoveryCapsule() ... 151
EFI_PEI_RECOVERY_BLOCK_IO_PPI ... 152
EFI_PEI_RECOVERY_BLOCK_IO_PPI. GetNumberOfBlockDevices() ... 153
EFI_PEI_RECOVERY_BLOCK_IO_PPI.GetBlockDeviceMediaInfo() ... 154
EFI_PEI_RECOVERY_BLOCK_IO_PPI.ReadBlocks() ... 156
8.3.5 EFI PEI Recovery Block IO2 PPI ... 157
EFI_PEI_RECOVERY_BLOCK_IO2_PPI ... 158
EFI_PEI_RECOVERY_BLOCK_IO2_PPI.GetNumberOfBlockDevices() ... 159
EFI_PEI_RECOVERY_BLOCK_IO2_PPI.GetBlockDeviceMediaInfo() ... 160
EFI_PEI_RECOVERY_BLOCK_IO2_PPI.ReadBlocks() ... 163
8.3.6 EFI PEI Vector Handoff Info PPI ... 164
EFI_PEI_VECTOR_HANDOFF_INFO_PPI (Optional) ... 165
8.3.7 CPU I/O PPI (Optional) ... 166
EFI_PEI_CPU_IO_PPI (Optional) ... 166
EFI_PEI_CPU_IO_PPI.Mem() ... 170
EFI_PEI_CPU_IO_PPI.Io() ... 172
EFI_PEI_CPU_IO_PPI.IoRead8() ... 173
EFI_PEI_CPU_IO_PPI.IoRead16() ... 174
EFI_PEI_CPU_IO_PPI.IoRead32() ... 175
EFI_PEI_CPU_IO_PPI.IoRead64() ... 176
EFI_PEI_CPU_IO_PPI.IoWrite8() ... 177
EFI_PEI_CPU_IO_PPI.IoWrite16() ... 178
EFI_PEI_CPU_IO_PPI.IoWrite32() ... 179
EFI_PEI_CPU_IO_PPI.IoWrite64() ... 180
EFI_PEI_CPU_IO_PPI.MemRead8() ... 181
EFI_PEI_CPU_IO_PPI.MemRead16() ... 182
EFI_PEI_CPU_IO_PPI.MemRead32() ... 183
EFI_PEI_CPU_IO_PPI.MemRead64() ... 184
EFI_PEI_CPU_IO_PPI.MemWrite8() ... 185
EFI_PEI_CPU_IO_PPI.MemWrite16() ... 186
EFI_PEI_CPU_IO_PPI.MemWrite32() ... 187
EFI_PEI_CPU_IO_PPI.MemWrite64() ... 188
8.3.8 EFI Pei Capsule PPI ... 188
EFI_PEI_CAPSULE_PPI (Optional) ... 189
EFI_PEI_CAPSULE_PPI.Coalesce ... 190

EFI_PEI_CAPSULE_CHECK_CAPSULE_UDPATE.CheckCapsuleUpdate().. 191
EFI_PEI_CAPSULE_CHECK_CAPSULE_UDPATE.CapsuleCreateState() 192
8.3.9 EFI MP Services PPI 192
EFI_MP_SERVICES_PPI (Optional)........ 193
EFI_MP_SERVICES_PPI.GetNumberOfProcessors()........ 195
EFI_MP_SERVICES_PPI.GetProcessorInfo() 197
EFI_MP_SERVICES_PPI.StartupAllAPs () 198
EFI_MP_SERVICES_PPI.StartupThisAP () 200
EFI_MP_SERVICES_PPI.SwitchBSP () 202
EFI_MP_SERVICES_PPI.WhoAmI () 205
8.4 Graphics PEIM Interfaces 206
8.4.1 Pei Graphics PPI 206
GraphicsPpiInit 207
GraphicsPpiGetMode 208
8.4.2 EFI PEI Graphics INFO HOB 212
EFI_PEI_GRAPHICS_INFO_HOB 213

9 PEI to DXE Handoff 215
9.1 Introduction 215
9.2 Discovery and Dispatch of the DXE Foundation 215
9.3 Passing the Hand-Off Block (HOB) List 215
9.4 Handoff Processor State to the DXE IPL PPI 216

10 Boot Paths 217
10.1 Introduction 217
10.2 Code Flow 217
10.2.1 Reset Boot Paths 217
10.3 Normal Boot Paths 218
10.3.1 Basic G0-to-S0 and S0 Variation Boot Paths 218
10.3.2 S-State Boot Paths 219
10.4 Recovery Paths 219
10.4.1 Discovery 220
10.4.2 General Recovery Architecture 220
10.5 Defined Boot Modes 220
10.6 Priority of Boot Paths 220
10.7 Assumptions 221
10.8 Architectural Boot Mode PPIs 221
10.9 Recovery 222
10.9.1 Scope 222
10.9.2 Discovery 222
10.9.3 General Recovery Architecture 222
10.9.4 Finding and Loading the Recovery DXE Image 223

11 PEI Physical Memory Usage 226
11.1 Introduction 226
11.2 Before Permanent Memory Is Installed 226
11.2.1 Discovering Physical Memory 226
11.2.2 Using Physical Memory 226

11.3 After Permanent Memory Is Installed ... 227
11.3.1 Allocating Physical Memory ... 227
11.3.2 Allocating Memory Using GUID Extension HOBs ... 227
11.3.3 Allocating Memory Using PEI Service ... 227

12 Special Paths Unique to the Itanium® Processor Family ... 228
12.1 Introduction ... 228
12.2 Unique Boot Paths for Itanium Architecture ... 228
12.3 Min-State Save Area ... 229
EFI_PEI_MIN_STATE_DATA ... 231
12.4 Dispatching Itanium Processor Family PEIMs ... 233

13 Security (SEC) Phase Information ... 235
13.1 Introduction ... 235
13.2 Responsibilities ... 235
13.2.1 Handling All Platform Restart Events ... 235
13.2.2 Creating a Temporary Memory Store ... 235
13.2.3 Serving As the Root of Trust in the System ... 236
13.2.4 Passing Handoff Information to the PEI Foundation ... 236
13.3 SEC Platform Information PPI ... 236
13.4 SEC HOB Data PPI ... 236
13.5 Health Flag Bit Format ... 236
13.5.1 Self-Test State Parameter ... 238
13.6 Processor-Specific Details ... 239
13.6.1 SEC Phase in IA-32 Intel Architecture ... 239
13.6.2 SEC Phase in the Itanium Processor Family ... 239

14 Dependency Expression Grammar ... 241
14.1 Dependency Expression Grammar ... 241
14.1.1 Example Dependency Expression BNF Grammar ... 241
14.1.2 Sample Dependency Expressions ... 242

15 TE Image ... 243
15.1 Introduction ... 243
15.2 PE32 Headers ... 243
TE Header ... 245

16 TE Image Creation ... 247
16.1 Introduction ... 247
16.2 TE Image Utility Requirements ... 247
16.3 TE Image Relocations ... 247

17 TE Image Loading ... 248
17.1 Introduction ... 248
17.2 XIP Images ... 248
17.3 Relocated Images ... 248
17.4 PIC Images ... 248

List of Figures

Figure 1. PEI Operations Diagram 10
Figure 2. Typical PEIM Layout in a Firmware File 108
Figure 3. Itanium Processor Boot Path (INIT and MCHK) 229
Figure 4. Min-State Buffer Organization 230
Figure 5. Boot Path in Itanium Processors 234
Figure 6. Health Flag Bit Format 237
Figure 7. PEI Initialization Steps in IA-32 239
Figure 8. Security (SEC) Phase in the Itanium Processor Family 240

List of Tables

Table 1. Organization of the PEI CIS 1
Table 2. SI prefixes 7
Table 3. Binary prefixes 7
Table 4. PEI Foundation Classes of Service 11
Table 5. PEI Services 20
Table 6. Boot Mode Register 27
Table 7. Dependency Expression Opcode Summary 72
Table 8. PUSH Instruction Encoding 73
Table 9. AND Instruction Encoding 74
Table 10. OR Instruction Encoding 75
Table 11. NOT Instruction Encoding 76
Table 12. TRUE Instruction Encoding 77
Table 13. FALSE Instruction Encoding 78
Table 14. END Instruction Encoding 79
Table 15. Example Dispatch Map 84
Table 16. PEI PPI Services List Descriptors 116
Table 17. Organization of the Code Definitions Section 144
Table 18. Required HOB Types in the HOB List 215
Table 19. Handoff Processor State to the DXE IPL PPI 216
Table 20. Boot Path Assumptions 221
Table 21. Architectural Boot Mode PPIs 222
Table 22. Device Recovery Module Functions 225
Table 23. Device Recovery Block I/O Functions 225
Table 24. Health Flag Bit Field Description 237
Table 25. Self-Test State Bit Values 238
Table 26. COFF Header Fields Required for TE Images 243
Table 27. Optional Header Fields Required for TE Images 243

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 8	Describes the major components of PEI, including the PEI Services, boot mode, PEI Dispatcher, and PEIMs.
“PEI Services Table” on page 14	Describes the data structure that maintains the PEI Services.
“Services - PEI” on page 20	Details each of the functions that comprise the PEI Services.
“PEI Foundation” on page 58	Describes the PEI Foundation and its methods of operation and the PEI Dispatcher and its associated dependency expression grammar..
“PEIMs” on page 107	Describes the format and use of the Pre-EFI Initialization Module (PEIM).
“Architectural PPIs” on page 86	Contains PEIM-to-PEIM Interfaces (PPIs) that are used by the PEI Foundation.
“Additional PPIs” on page 118	Contains PPIs that can exist on a platform.
“PEI to DXE Handoff” on page 215	Describes the state of the machine and memory when the PEI phase invokes the DXE phase.
“Boot Paths” on page 217	Describes the restart modalities and behavior supported in the PEI phase.
“PEI Physical Memory Usage” on page 226	Describes the memory map and memory usage during the PEI phase.

Section	Description
"Special Paths Unique to the Itanium® Processor Family" on page 228	Contains flow during PEI that is unique to the Itanium® processor family.
"Security (SEC) Phase Information" on page 235	Contains an overview of the phase of execution that occurs prior to PEI.
"Dependency Expression Grammar" on page 241	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 243	Describes the format of the TE executable.
"TE Image Creation" on page 247	Describes how TE executables are created from PE32+ executables.
"TE Image Loading" on page 248	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.

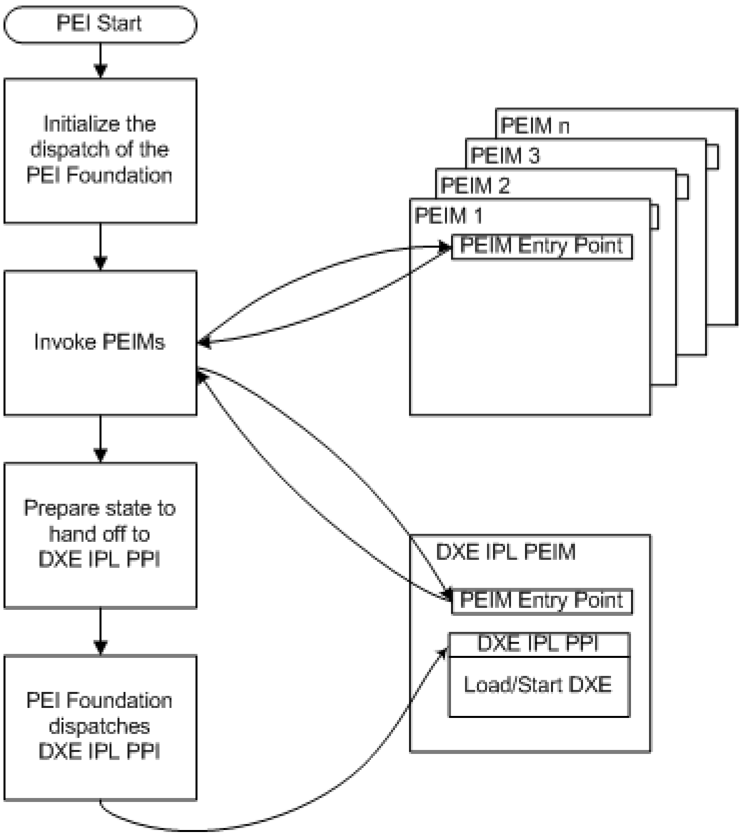

PEI Start
Initialize the dispatch of the PEI Foundation
Invoke PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

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 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 20.

Related Definitions

```
//
// PEI Specification Revision information
//
#define PEI_SPECIFICATION_MAJOR_REVISION 1
#define PEI_SPECIFICATION_MINOR_REVISION 70

//
// UEFI PEI Services Table
//
#define PEI_SERVICES_SIGNATURE     0x5652455320494550
#define ((PEI_SPECIFICATION_MAJOR_REVISION<<17) |
(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_FREE_PAGES FreePages;

  } 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.

FreePages

Releases memory previously allocated using AllocatePages().

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 116 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 111.

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 111.

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 111.

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 111.

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 217 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 86 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()
- FreePages()

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 allocates the requested number of pages and returns a pointer to the base address of the page range in the location referenced by Memory. The service scans the available memory to locate free pages. When it finds a physically contiguous block of pages that is large enough it creates a memory allocation HOB describing the region with the requested MemoryType.

Allocation made prior to permanent memory will be migrated to permanent memory and the HOB updated.

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.
Prior to InstallPeiMemory() being called, PEI will allocate pages from the heap. After InstallPeiMemory() is called, 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.

FreePages()

Summary

Frees memory pages.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FREE_PAGES) (
     IN CONST EFI_PEI_SERVICES **PeiServices,
     IN EFI_PHYSICAL_ADDRESS Memory
     IN UINTN Pages
 );
```

Parameters

PeiServices

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

Memory

The base physical address of the pages to be freed. Type EFI_PHYSICAL_ADDRESS is defined in the EFI_BOOT_SERVICES.AllocatePages()function description.

Pages

The number of contiguous 4KiB pages to free.

Description

The FreePages() function returns memory allocated by AllocatePages() to the firmware.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_NOT_FOUND	The requested memory pages were not allocated with AllocatePages().
EFI_INVALID_PARAMETER	Memory is not a page-aligned address or Pages is invalid.

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 firmware volume 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 (BFV) and other boot-critical FVs

The program that the Security (SEC) phase hands control to is known as the PEI Foundation. PEIMs may reside in the BFVor other FVs. A "special" PEIM must be resident in the BFV to provide information about the location of the other FVs.

Each file that is required to boot, in the BFV and other critical FVs (like where the PEI foundation is located), must be able to be discovered and validated by the PEI phase. This allows the PEI phase to determine if those FVs have 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 FVor 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 other boot-critical FVs, by adding the firmware volume into the PpiList with **EFI_PEI_FIRMWARE_VOLUME_PPI** type.
- 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. These PPI descriptors can be a combination of descriptors of type **EFI_PEI_PPI_DESCRIPTOR** for PPIs to be installed initially by the PEI core and descriptors of type **EFI_PEI_NOTIFY_DESCRIPTOR** for notifications in which the PEI Core will notify when the PPI service is installed. An empty PPI list consists of a single descriptor with the end-tag **EFI_PEI_PPI_DESCRIPTOR_TERMINATE_LIST**. Types **EFI_PEI_PPI_DESCRIPTOR** and **EFI_PEI_NOTIFY_DESCRIPTOR** are defined in "PEIM Descriptors." 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. This should contain all the boot critical FVs that would be passed from SEC to PEI Foundation thorough the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

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*".

5.4.6 RISC-V Processor Family – Register Mechanism

For the RISC-V processor, the **EFI_PEI_SERVICES **** is stored in the **RISCV_MACHINE_MODE_CONTEXT** structure and the pointer to this structure is stored in the Machine mode Control and Status register **MSCRATCH. EFI_PEI_SERVICES **** is one of the structure member in **RISCV_MACHINE_MODE_CONTEXT**. The contents in this structure will be kept across all UEFI execution phases during the entire system life. **MSCRATCH** is a readable and writable CSR which is initiated to maintain various pointers for each UEFI execution phase. The pointers in this structure includes **EFI_PEI_SERVICES **** and the interrupt handlers of each RISC-V privilege level.

```
/// Machine mode context used for saving hart-local context.
typedef struct _RISCV_MACHINE_MODE_CONTEXT {
  EFI_PHYSICAL_ADDRESS PeiService;/// PEI service.
  EFI_PHYSICAL_ADDRESS MachineModeTrapHandler;/// Machine mode
trap
         /// handler.
  EFI_PHYSICAL_ADDRESS HypervisorModeTrapHandler;/// Hypervisor
mode trap
         /// handler.
  EFI_PHYSICAL_ADDRESS SupervisorModeTrapHandler;/// Supervisor
mode trap
         /// handler.
  EFI_PHYSICAL_ADDRESS UserModeTrapHandler;/// User mode trap
handler.
  TRAP_HANDLER_CONTEXT MModeHandler;/// Handler for machine
         /// mode.
} RISCV_MACHINE_MODE_CONTEXT;

EFI_PEI_SERVICES ** can be retrieved through below function
which is provided by RISC-V library.

CONST EFI_PEI_SERVICES **
EFIAPI
GetPeiServicesTablePointer (
  VOID
  )
{
  RISCV_MACHINE_MODE_CONTEXT *Context;
  EFI_PEI_SERVICES **PeiServices;
  Context = (RISCV_MACHINE_MODE_CONTEXT *) UINTN)
RiscVGetScratch ();
  PeiServices = (EFI_PEI_SERVICES **) Context->PeiService;
  return (CONST EFI_PEI_SERVICES **)PeiServices;
}
```

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 241 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 71 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 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
- Installed Firmware Volumes

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

The PEI Foundation will copy any installed firmware volumes from the temporary memory location to a permanent memory location with the alignment specified in the firmware volume header. Any *uncompressed* PE32 or TE sections within PEIMs in these firmware volumes will be fixed up. This ensures any static **`EFI_PEI_PPI_DESCRIPTOR`**s or PPI interface pointers in these PEIMs point to the permanent memory addresses.

In addition, if there were any **`EFI_PEI_PPI_DESCRIPTOR`**s created in the temporary memory heap or declared statically in PEIMs, their respective locations have been translated by an offset equal to the difference between the original 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 243 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. 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, those modules that are required for crisis recovery, and platforms in which code is executed from temporary memory.

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 Architectural 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 57.

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 53.

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.

6.3.9 EFI_PEI_CORE_FV_LOCATION_PPI

Summary

Indicates to the PEI Foundation which Firmware Volume contains the PEI Foundation.

GUID

```
#define EFI_PEI_CORE_FV_LOCATION_GUID \
  {0x52888eae, 0x5b10, 0x47d0, \
  {0xa8, 0x7f, 0xb8, 0x22, 0xab, 0xa0, 0xca, 0xf4}}
```

Prototype

```
typedef struct {
  VOID           *PeiCoreFvLocation;
} EFI_PEI_CORE_FV_LOCATION_PPI;
```

Parameters

PeiCoreFvLocation	Points to the first byte of the firmware volume which contains the PEI Foundation.

Description

PI Specification 1.6 and earlier versions required that the PEI Foundation reside in the Boot Firmware Volume (BFV). This allowed the PEI Foundation to assume that it can locate itself within the BFV. This assumption is no longer valid. As the *PI Specification* no longer defines a mechanism for the PEI Foundation to locate itself, a new method is needed. Specifically, the PEI Core can use the **EFI_PEI_CORE_FV_LOCATION_PPI** to ascertain its containing firmware volume location.
Specifically, if the PEI Foundation does not reside in the BFV, then SEC must pass the **EFI_PEI_CORE_FV_LOCATION_PPI** as a part of the PPI list provided to the PEI Foundation Entry Point. If this PPI is not present in the PPI list, the PEI Foundation shall assume that it resides within the BFV.

The PEI Foundation will use the **EFI_PEI_CORE_FV_LOCATION_PPI** for purposes of module dispatch if it exists. If it does not exist, the **BootFirmwareVolume** base in **EFI_SEC_PEI_HAND_OFF** will be used.

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.

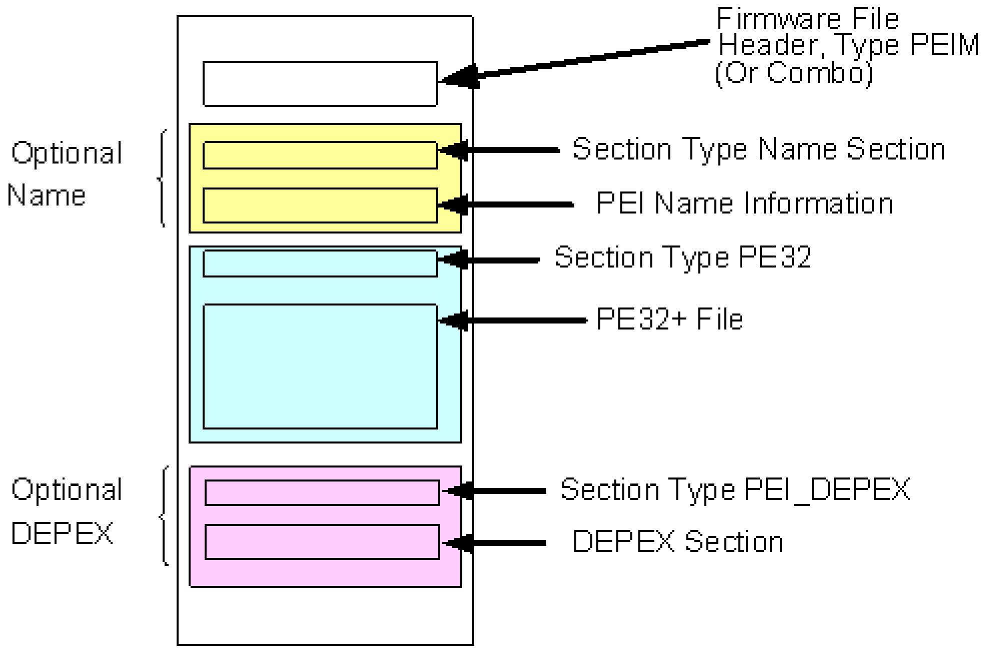


Figure 2. Typical PEIM Layout in a Firmware File

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, OPTIONAL
  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. If not NULL, then Attributes is set on output both when EFI_SUCCESS and when EFI_BUFFER_TOO_SMALL is returned.

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. If *Attributes* is not **NULL**, then the attributes bitmask for the variable has been stored to the memory location pointed-to by *Attributes*.
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 236 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_DEVICE_ERROR	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

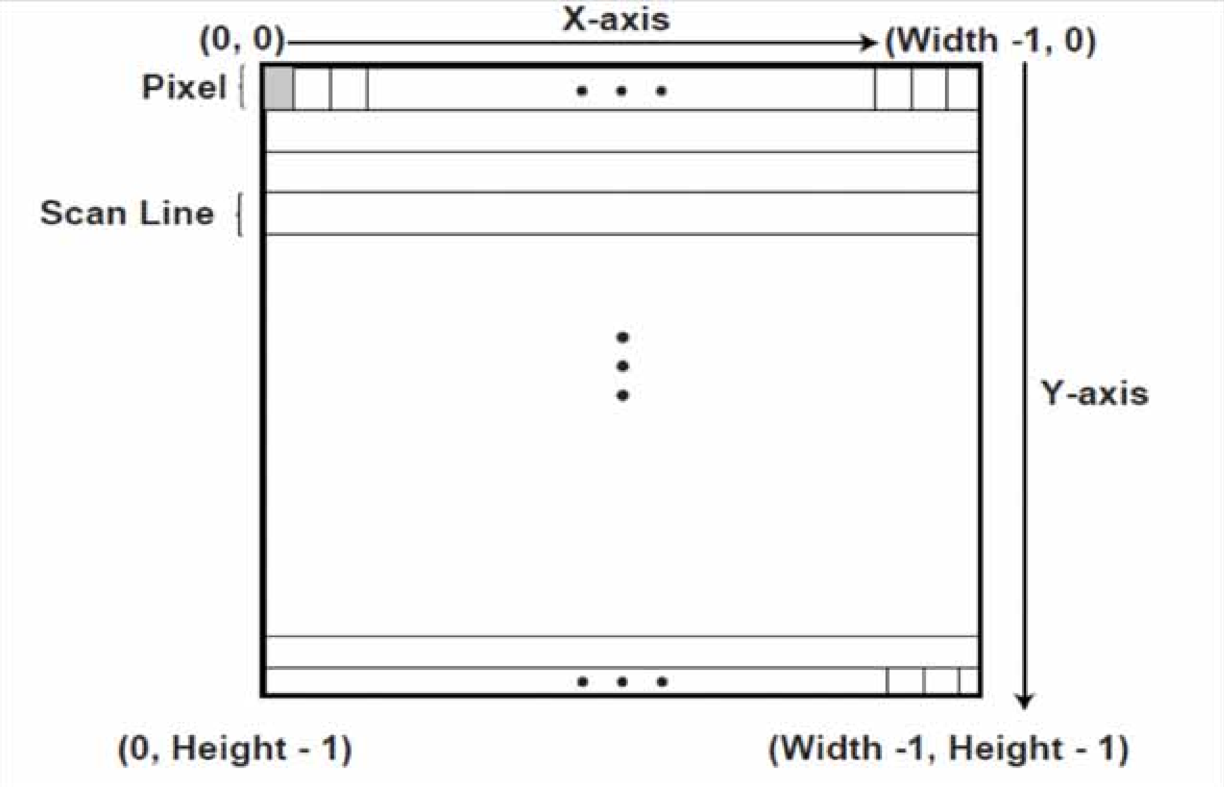
Description

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 86 and "Optional Architectural PPIs" on page 91. 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 227.
- 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 227.

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.

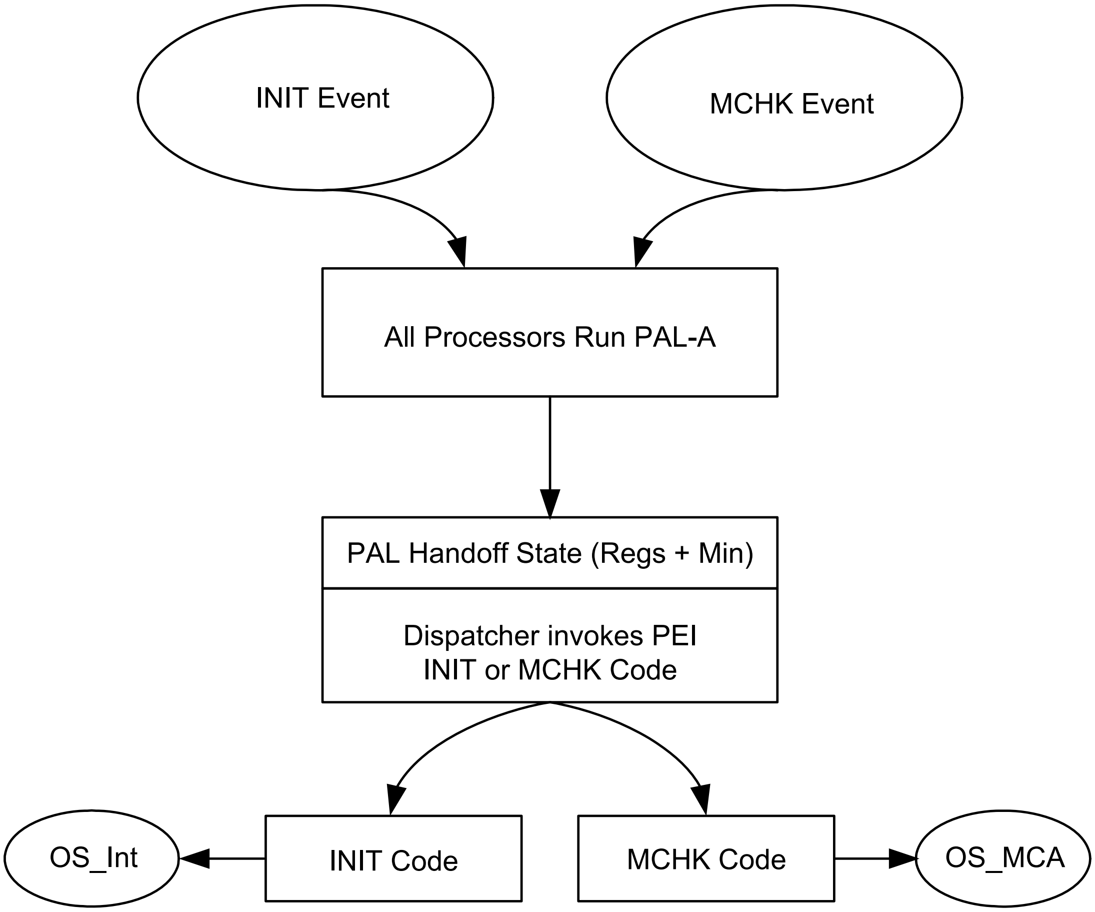


Figure 3. Itanium Processor Boot Path (INIT and MCHK)

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**.

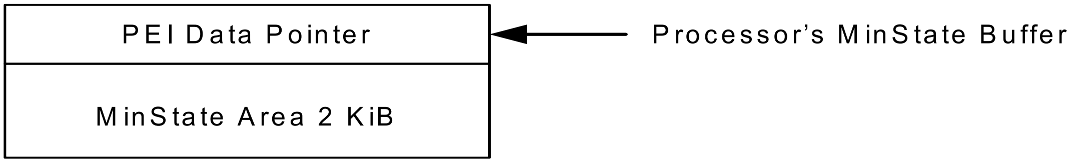


Figure 4. Min-State Buffer Organization

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 215 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.

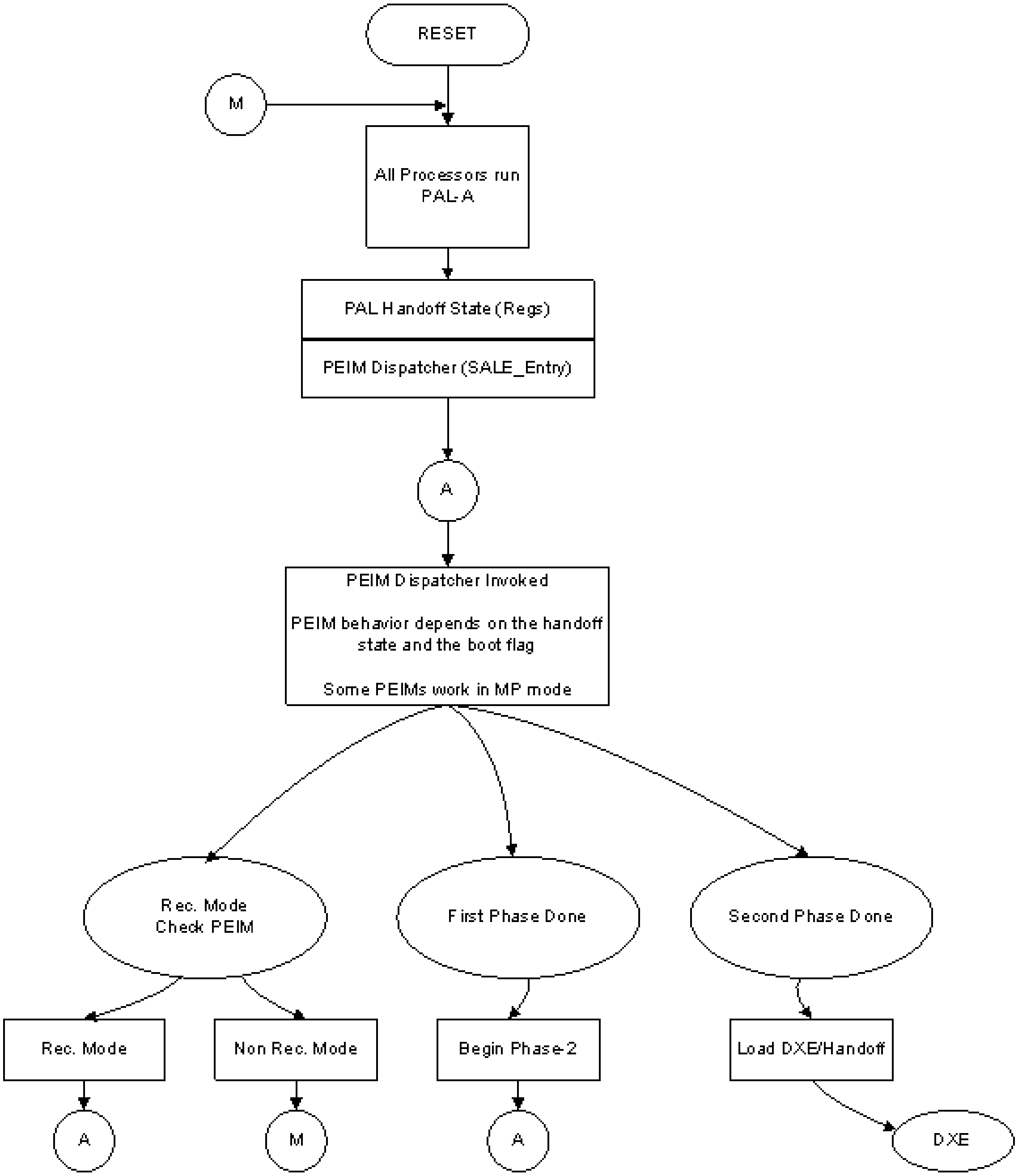


Figure 5. Boot Path in Itanium Processors

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 239.

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. The location of the BFV will be superseded by **EFI_PEI_CORE_FV_LOCATION_PPI** if that exists.

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.

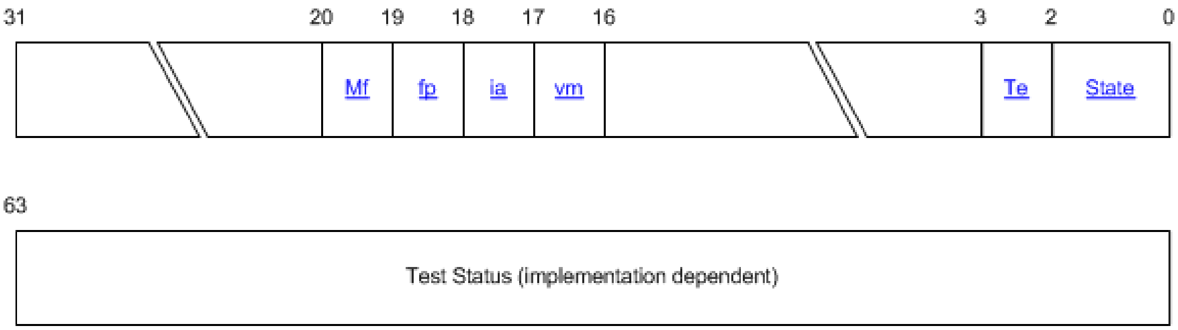


Figure 6. Health Flag Bit Format

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 238.
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.

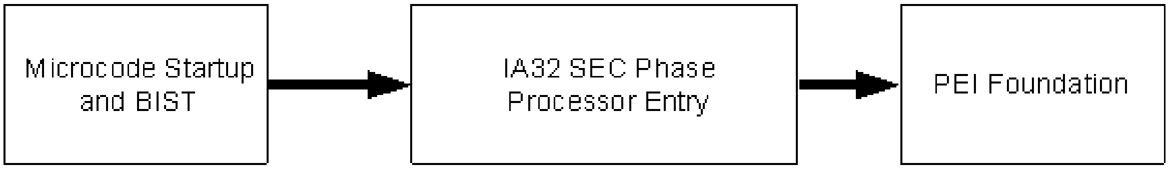


Figure 7. PEI Initialization Steps in IA-32

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.

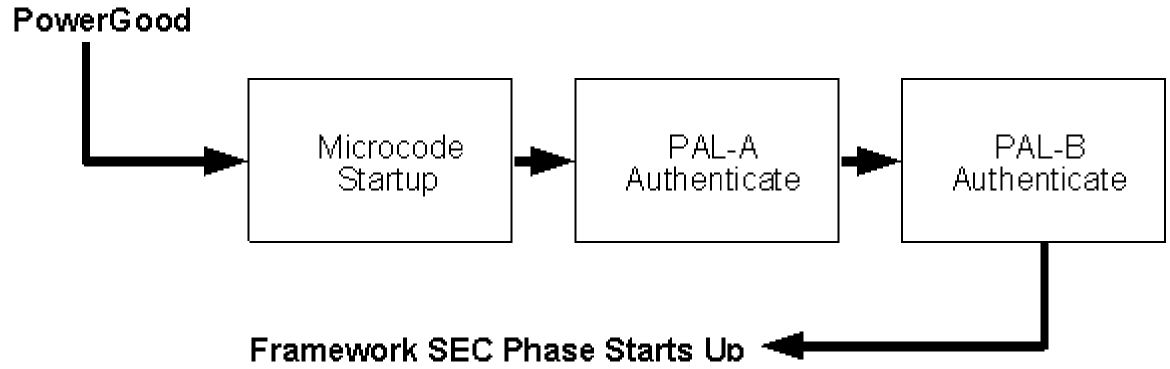


Figure 8. Security (SEC) Phase in the Itanium Processor Family

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.

Platform Initialization (PI) Specification

Volume 2:
Driver Execution Environment Core Interface

Version 1.7

January 2019

Revision History

Revision	Mantis ID / Description	Date
1.7	• 1848 PEI Core PEIM Migration Support Change • 1856 SecCore/PeiCore BFV Requirement Change • 1860 MM MP Protocol issues • 1885 Add extended data for EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR • 1889 New Status Codes • 1891 PEI delayed dispatch • 1892 new pi spec revision • 1955 set (*Attributes) when EFI_PEI_GET_VARIABLE2 returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL • 1967 New architectural PPI for PI PEI Core FV Location • 1969 Incompatible Pci Ignore Option Rom	January 2019
1.6 Errata A	• 1828 Add decorator 'OPTIONAL' for Attributes parameter of EFI_PEI_GET_VARIABLE2. • 1849 Issues in PI Spec Vol. 5 Ch. 18 • 1857 Specifies how notifications are passed from SEC to PEI. • 1884 Variadic API issue in S3 save API • 1907 Clarification of the EFI_MM_COMMUNICATION_PROTOCOL • 1940 Clarify EFI_MM_CONFIGURATION_PROTOCOL_GUID	December 2018
1.6	• 1567 Layered SPI bus • 1648 PI Binding for RISC-V • 1746 Add an FV Extended Header entry that contains the used size of the FV • 1763 MM Handler state notification protocol • 1764 Add additional alignment • 1768 Update the PI Spec to 1.6 • 1777 Update Revision History • 1778 Update front matter	April 2017
1.5 Errata A	• 1587 pre permanent memory page allocation • 1665 Incorrect status code for an AP calling EFI_MP_SERVICES_PROTOCOL.SwitchBSP() • 1734 Outdated EFI spec reference in Save State Write • 1735 Several copy & paste errors in Save State Write • 1747 Clarify that MM_ACCESS_PROTOCOL should cover all MMRAM region used by the platform	April 2017

Revision	Mantis ID / Description	Date
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 • 1593 coalesce language enhancements • 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 • 1607 Update MM guid def'n to match edkII impl • 1626 Add new Status Code for BDS Attempting UEFI BootOrder entries • 1628 Minor feedback for PI 1.5 Vol 4 SMM Draft • 1666 Graphics Device Info Hob	4/26/16
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

Revision	Mantis ID / Description	Date
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.	2/20/15
1.3 Errata A	• 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
1.3	• 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

Revision	Mantis ID / Description	Date
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 • 937SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 • 969Vol 1 errata: TE Header parameters	10/26/12
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
1.2.1	• 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

Revision	Mantis ID / Description	Date
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 • 737 Remove SMM Communication ACPI Table definition . • 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 • 753 SIO PEI and UEFI-Driver Model Architecture • 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

Revision	Mantis ID / Description	Date
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 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	• 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

Revision	Mantis ID / Description	Date
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	• 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

Revision	Mantis ID / Description	Date
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
1.1 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

Revision	Mantis ID / Description	Date
1.1 Errata	• 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	Revises typographical errors and minor omissions--see Errata for details	04/25/08
1.1 correction	Restore (missing) MP protocol	03/12/08
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

Revision	Mantis ID / Description	Date
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
1.0	Initial public release.	8/21/06

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.

Table of Contents

Revision History iii
Table of Contents xiv
List of Figures xix
List of Tables xx

1 Introduction 1
1.1 Overview 1
1.2 Organization of the DXE CIS 1
1.3 Target Audience 2
1.4 Conventions Used in this Document 2
1.4.1 Data Structure Descriptions 3
1.4.2 Protocol Descriptions 3
1.4.3 Procedure Descriptions 4
1.4.4 Instruction Descriptions 4
1.4.5 Pseudo-Code Conventions 4
1.4.6 Typographic Conventions 5
1.5 Requirements 5
1.6 Conventions used in this document 7
1.6.1 Number formats 7
1.6.2 Binary prefixes 7

2 Overview 9
2.1 Driver Execution Environment (DXE) Phase 9
2.2 UEFI System Table 10
2.2.1 Overview 10
2.2.2 UEFI Boot Services Table 11
2.2.3 UEFI Runtime Services Table 11
2.2.4 DXE Services Table 12
2.3 DXE Foundation 12
2.4 DXE Dispatcher 13
2.5 DXE Drivers 13
2.6 DXE Architectural Protocols 13
2.7 Runtime Protocol 14

3 Boot Manager 15
3.1 Boot Manager 15

4 UEFI System Table 16
4.1 DXE Services Table 16
DXE_SERVICES 16
4.2 UEFI Image Entry Point Examples 19
4.2.1 UEFI Application Example 19
4.2.2 Non-UEFI Driver Model Example (Resident in Memory) 21
4.2.3 Non-UEFI Driver Model Example (Nonresident in Memory) 22

4.2.4 UEFI Driver Model Example ... 23
4.2.5 UEFI Driver Model Example (Unloadable) ... 24
4.2.6 UEFI Driver Model Example (Multiple Instances) ... 25

5 Services - Boot Services ... 28
5.1 Extensions to UEFI Boot Service Event Usage ... 28
5.1.1 CreateEvent ... 28
5.1.2 Pre-Defined Event Groups ... 28
5.1.3 Additions to LoadImage() ... 29

6 Runtime Capabilities ... 33
6.1 Additional Runtime Protocol ... 33
6.1.1 Status Code Services ... 33

7 Services - DXE Services ... 34
7.1 Introduction ... 34
7.2 Global Coherency Domain Services ... 34
7.2.1 Global Coherency Domain (GCD) Services Overview ... 34
7.2.2 GCD Memory Resources ... 34
7.2.3 GCD I/O Resources ... 36
7.2.4 Global Coherency Domain Services ... 37
AddMemorySpace() ... 39
AllocateMemorySpace() ... 42
FreeMemorySpace() ... 45
RemoveMemorySpace() ... 47
GetMemorySpaceDescriptor() ... 49
SetMemorySpaceAttributes() ... 52
SetMemorySpaceCapabilities() ... 54
GetMemorySpaceMap() ... 56
AddIoSpace() ... 58
AllocateIoSpace() ... 60
FreeIoSpace() ... 63
RemoveIoSpace() ... 65
GetIoSpaceDescriptor() ... 67
GetIoSpaceMap() ... 69
7.3 Dispatcher Services ... 71
Dispatch() ... 72
Schedule() ... 73
Trust() ... 74
ProcessFirmwareVolume() ... 75

8 Protocols - Device Path Protocol ... 77
8.1 Introduction ... 77
8.2 Firmware Volume Media Device Path ... 77
8.3 Firmware File Media Device Path ... 78

9 DXE Foundation ... 79
9.1 Introduction ... 79
9.2 Hand-Off Block (HOB) List ... 79

9.3 DXE Foundation Data Structures ... 81
9.4 Required DXE Foundation Components ... 82
9.5 Handing Control to DXE Dispatcher ... 84
9.6 DXE Foundation Entry Point ... 85
9.6.1 DXE_ENTRY_POINT ... 85
DXE_ENTRY_POINT ... 85
9.7 Dependencies ... 86
9.7.1 UEFI Boot Services Dependencies ... 86
9.7.2 UEFI Runtime Services Dependencies ... 89
9.7.3 DXE Services Dependencies ... 92
9.8 HOB Translations ... 93
9.8.1 HOB Translations Overview ... 93
9.8.2 PHIT HOB ... 93
9.8.3 CPU HOB ... 94
9.8.4 Resource Descriptor HOBs ... 94
9.8.5 Firmware Volume HOBs ... 95
9.8.6 Memory Allocation HOBs ... 95
9.8.7 GUID Extension HOBs ... 96
10 DXE Dispatcher ... 97
10.1 Introduction ... 97
10.2 Requirements ... 97
10.3 The A Priori File ... 98
EFI_APRIORI_GUID ... 99
10.4 Firmware Volume Image Files ... 99
10.5 Dependency Expressions ... 100
10.6 Dependency Expressions Overview ... 100
10.7 Dependency Expression Instruction Set ... 100
BEFORE ... 102
AFTER ... 103
PUSH ... 104
AND ... 105
OR ... 106
NOT ... 107
TRUE ... 108
FALSE ... 109
END ... 110
SOR ... 111
10.8 Dependency Expression with No Dependencies ... 112
10.9 Empty Dependency Expressions ... 112
10.10 Dependency Expression Reverse Polish Notation (RPN) ... 114
10.11 DXE Dispatcher State Machine ... 114
10.12 Example Orderings ... 116
10.13 Security Considerations ... 119
11 DXE Drivers ... 120
11.1 Introduction ... 120
11.2 Classes of DXE Drivers ... 120

11.2.1 Early DXE Drivers 120
11.2.2 DXE Drivers that Follow the UEFI Driver Model 121
11.2.3 Additional Classifications 121

12 DXE Architectural Protocols 122

12.1 Introduction 122
12.2 Boot Device Selection (BDS) Architectural Protocol 124
EFI_BDS_ARCH_PROTOCOL 124
EFI_BDS_ARCH_PROTOCOL.Entry() 125
12.3 CPU Architectural Protocol 126
EFI_CPU_ARCH_PROTOCOL 126
EFI_CPU_ARCH_PROTOCOL.FlushDataCache() 129
EFI_CPU_ARCH_PROTOCOL.EnableInterrupt() 131
EFI_CPU_ARCH_PROTOCOL.DisableInterrupt() 132
EFI_CPU_ARCH_PROTOCOL.GetInterruptState() 133
EFI_CPU_ARCH_PROTOCOL.Init() 134
EFI_CPU_ARCH_PROTOCOL.RegisterInterruptHandler() 135
EFI_CPU_ARCH_PROTOCOL.GetTimerValue() 137
EFI_CPU_ARCH_PROTOCOL.SetMemoryAttributes() 139
12.4 Metronome Architectural Protocol 141
EFI_METRONOME_ARCH_PROTOCOL 141
EFI_METRONOME_ARCH_PROTOCOL.WaitForTick() 142
12.5 Monotonic Counter Architectural Protocol 143
EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL 143
12.6 Real Time Clock Architectural Protocol 144
EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL 144
12.7 Reset Architectural Protocol 145
EFI_RESET_ARCH_PROTOCOL 145
12.8 Runtime Architectural Protocol 146
EFI_RUNTIME_ARCH_PROTOCOL 146
12.9 Security Architectural Protocols 152
12.9.1 Security Architectural Protocol 152
EFI_SECURITY_ARCH_PROTOCOL 152
EFI_SECURITY_ARCH_PROTOCOL.FileAuthenticationState() 154
12.9.2 Security2 Architectural Protocol 155
EFI_SECURITY2_ARCH_PROTOCOL.FileAuthentication() 157
12.10 Timer Architectural Protocol 158
EFI_TIMER_ARCH_PROTOCOL 158
EFI_TIMER_ARCH_PROTOCOL.RegisterHandler() 160
EFI_TIMER_ARCH_PROTOCOL.SetTimerPeriod() 162
EFI_TIMER_ARCH_PROTOCOL.GetTimerPeriod() 163
EFI_TIMER_ARCH_PROTOCOL.GenerateSoftInterrupt() 164
12.11 Variable Architectural Protocol 165
EFI_VARIABLE_ARCH_PROTOCOL 165
12.12 Variable Write Architectural Protocol 166
EFI_VARIABLE_WRITE_ARCH_PROTOCOL 166
12.13 EFI Capsule Architectural Protocol 166

EFI_CAPSULE_ARCH_PROTOCOL 166
12.14 Watchdog Timer Architectural Protocol 167
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL 167
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.RegisterHandler() 169
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.SetTimerPeriod() 171
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.GetTimerPeriod() 172

13 DXE Boot Services Protocol 173
13.1 Overview 173
13.2 Conventions and Abbreviations 173
13.3 MP Services Protocol Overview 173
13.4 MP Services Protocol 174
EFI_MP_SERVICES_PROTOCOL 174
EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors() 176
EFI_MP_SERVICES_PROTOCOL.GetProcessorInfo() 178
EFI_MP_SERVICES_PROTOCOL.StartupAllAPs() 181
EFI_MP_SERVICES_PROTOCOL.StartupThisAP() 185
EFI_MP_SERVICES_PROTOCOL.SwitchBSP() 188
EFI_MP_SERVICES_PROTOCOL.EnableDisableAP() 190
EFI_MP_SERVICES_PROTOCOL.WhoAmI() 192

14 DXE Runtime Protocols 194
14.1 Introduction 194
14.2 Status Code Runtime Protocol 194
EFI_STATUS_CODE_ PROTOCOL 194
EFI_STATUS_CODE_PROTOCOL.ReportStatusCode() 195

15 Dependency Expression Grammar 199
15.1 Dependency Expression Grammar 199
15.2 Example Dependency Expression BNF Grammar 199
15.3 Sample Dependency Expressions 200
Appendix AError Codes 202
Appendix BGUID Definitions 203

List of Figures

Figure 1. PI Architecture Firmware Phases 10
Figure 2. GCD Memory State Transitions 36
Figure 3. GCD I/O State Transitions 37
Figure 4. HOB List 80
Figure 5. UEFI System Table and Related Components 81
Figure 6. DXE Foundation Components 82
Figure 7. DXE Driver States 115
Figure 8. Sample Firmware Volume 117
Figure 9. DXE Architectural Protocols 123

List of Tables

Table 1. Organization of the DXE CIS 2
Table 2. SI prefixes 7
Table 3. Binary prefixes 8
Table 4. UEFI Boot Services 11
Table 5. UEFI Runtime Services 12
Table 6. DXE Services 12
Table 7. DXE Architectural Protocols 14
Table 8. Status Codes Runtime Protocol 14
Table 9. Supported Subsystem Values 32
Table 10. Status Code Runtime Protocol 33
Table 11. Global Coherency Domain Boot Type Services 38
Table 12. Dispatcher Boot Type Services 71
Table 13. Firmware Volume Media Device Path 77
Table 14. Firmware Volume Device Node Text Representation 77
Table 15. Firmware File Media Device Path 78
Table 16. Firmware Volume File Device Node Text Representation 78
Table 17. Boot Service Dependencies 87
Table 18. Runtime Service Dependencies 90
Table 19. DXE Service Dependencies 92
Table 20. Resource Descriptor HOB to GCD Type Mapping 95
Table 21. Dependency Expression Opcode Summary 101
Table 22. BEFORE Instruction Encoding 102
Table 23. AFTER Instruction Encoding 103
Table 24. PUSH Instruction Encoding 104
Table 25. AND Instruction Encoding 105
Table 26. OR Instruction Encoding 106
Table 27. NOT Instruction Encoding 107
Table 28. TRUE Instruction Encoding 108
Table 29. FALSE Instruction Encoding 109
Table 30. END Instruction Encoding 110
Table 31. SOR Instruction Encoding 111
Table 32. DXE Dispatcher Orderings 118
Table 33. StatusFlag bits 180

1 Introduction

1.1 Overview

This specification defines the core code and services that are required for an implementation of the driver execution environment (DXE) phase of the Unified Extensible Firmware Interface (UEFI) Foundation. This DXE core interface specification (CIS) does the following:

- Describes the basic components of the DXE phase.
- Provides code definitions for services and functions that are architecturally required by the *Unified Extensible Firmware Interface Specification* (UEFI 2.0 specification).
- Presents a set of backward-compatible extensions to the UEFI 2.0 specification.
- Describes the machine preparation that is required for subsequent phases of firmware execution.

See “Organization of the DXE CIS” for more information.

1.2 Organization of the DXE CIS

This DXE core interface specification (CIS) is organized as shown in Table 1. Because the DXE 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 DXE CIS

Book	Description
"Overview" on page 9	Describes the major components of DXE, including the boot manager, firmware core, protocols, and requirements.
"Boot Manager" on page 15	Describes the boot manager, which is used to load UEFI drivers, UEFI applications, and UEFI OS loaders.
"UEFI System Table" on page 16	Describes the DXE Service table.
"Services - Boot Services" on page 28	Describes specific event types for DXE Foundation.
"Runtime Capabilities" on page 33	Contains definitions of a runtime protocol for status code support.
"Services - DXE Services" on page 34	Contains definitions for the fundamental services that are present in a DXE-compliant system before an OS is booted.
"Protocols - Device Path Protocol" on page 77	Defines the device path extensions required by the DXE Foundation.
"DXE Foundation" on page 79	Describes the DXE Foundation that consumes HOBs, Firmware Volumes, and DXE Architectural Protocols to produce an UEFI System Table, UEFI Boot Services, UEFI Runtime Services, and the DXE Services.
"DXE Dispatcher" on page 97	Describes the DXE Dispatcher that is responsible for loading and executing DXE drivers from Firmware Volumes.
"DXE Drivers" on page 120	Describes the different classes of DXE drivers that may be stored in Firmware Volumes.
"DXE Architectural Protocols" on page 122	Describes the Architectural Protocols that are produced by DXE drivers. They are also consumed by the DXE Foundation to produce the UEFI Boot Services, UEFI Runtime Services, and DXE Services.
"DXE Runtime Protocols" on page 194	Lists success, error, and warning codes returned by DXE and UEFI interfaces.
"Dependency Expression Grammar" on page 199	Describes the BNF grammar for a tool that can convert a text file containing a dependency expression into a dependency section of a DXE driver stored in a Firmware Volume.

1.3 Target Audience

This document is intended for the following readers:

- IHVs and OEMs who will be implementing DXE drivers that are stored in firmware volumes.
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in various vendor architecture–based products.

1.4 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.4.1 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.

1.4.2 Protocol Descriptions

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

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

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

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

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

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

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

1.4.3 Procedure Descriptions

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

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

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

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

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

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

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

1.4.4 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.

1.4.5 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).

1.4.6 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.

1.5 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.

1.6 Conventions used in this document

1.6.1 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.

1.6.2 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 Driver Execution Environment (DXE) Phase

The Driver Execution Environment (DXE) phase is where most of the system initialization is performed. Pre-EFI Initialization (PEI), the phase prior to DXE, is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position-independent data structures called *Hand-Off Blocks* (HOBs). HOBs are described in detail in *Volume 3*.

There are several components in the DXE phase:

- "DXE Foundation"
- "DXE Dispatcher"
- A set of "DXE Drivers"

The DXE Foundation produces a set of Boot Services, Runtime Services, and DXE Services. The DXE Dispatcher is responsible for discovering and executing DXE drivers in the correct order. The DXE drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for system services, console devices, and boot devices. These components work together to initialize the platform and provide the services required to boot an operating system. The DXE phase and Boot Device Selection (BDS) phases work together to establish consoles and attempt the booting of operating systems. The DXE phase is terminated when an operating system is successfully booted. The DXE Foundation is composed of boot services code, so no code from the DXE Foundation itself is allowed to persist into the OS runtime environment. Only the runtime data structures allocated by the DXE Foundation and services and data structured produced by runtime DXE drivers are allowed to persist into the OS runtime environment.

Figure 1 shows the phases that a platform with PI Architecture firmware will execute.

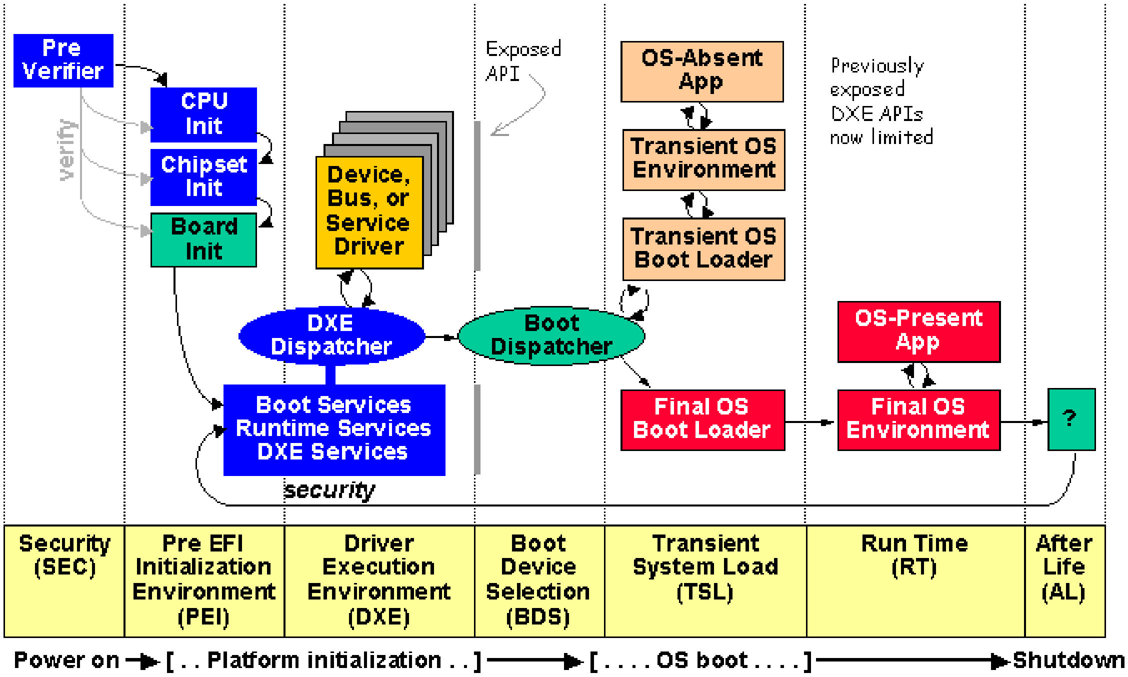

Pre Verifier
verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
DXE Dispatcher
Boot Services Runtime Services DXE Services
security
Exposed API
Boot Dispatcher
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
Previously exposed DXE APIs now limited
OS-Present App
Final OS Environment
?
Security (SEC)
Pre EFI Initialization Environment (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[. . Platform initialization . .]
[. . . . OS boot . . . .]
Shutdown

Figure 1. PI Architecture Firmware Phases

In a PI Architecture firmware implementation, the phase executed prior to DXE is PEI. This specification covers the transition from the PEI to the DXE phase, the DXE phase, and the DXE phase's interaction with the BDS phase. The DXE phase does not require a PEI phase to be executed. The only requirement for the DXE phase to execute is the presence of a valid HOB list. There are many different implementations that can produce a valid HOB list for the DXE phase to execute. The PEI phase in a PI Architecture firmware implementation is just one of many possible implementations.

2.2 UEFI System Table

2.2.1 Overview

The UEFI System Table is passed to every executable component in the DXE phase. The UEFI System Table contains a pointer to the following:

- "UEFI Boot Services Table"
- "UEFI Runtime Services Table"

It also contains pointers to the console devices and their associated I/O protocols. In addition, the UEFI System Table contains a pointer to the UEFI Configuration Table, and this table contains a list of GUID/pointer pairs. The UEFI Configuration Table may include tables such as the "DXE Services Dependencies" on page 92, HOB list, ACPI table, SMBIOS table, and SAL System table.

The UEFI Boot Services Table contains services to access the contents of the handle database. The handle database is where protocol interfaces produced by drivers are registered. Other drivers can use the UEFI Boot Services to look up these services produced by other drivers.

All of the services available in the DXE phase may be accessed through a pointer to the UEFI System Table.

2.2.2 UEFI Boot Services Table

Table 4 provides a summary of the services that are available through the UEFI Boot Services Table. These services are described in detail in the UEFI 2.0 specification. This DXE CIS makes a few minor, backward-compatible extensions to these services.

Table 4. UEFI Boot Services

UEFI Boot Services	Description
Task Priority	Provides services to increase or decrease the current task priority level. This can be used to implement simple locks and to disable the timer interrupt for short periods of time. These services depend on the "CPU Architectural Protocol" on page 126.
Memory	Provides services to allocate and free pages in 4 KiB increments and allocate and free pool on byte boundaries. It also provides a service to retrieve a map of all the current physical memory usage in the platform.
Event and Timer	Provides services to create events, signal events, check the status of events, wait for events, and close events. One class of events is timer events, and that class supports periodic timers with variable frequencies and one-shot timers with variable durations. These services depend on the "CPU Architectural Protocol" on page 126, the "Timer Architectural Protocol" on page 158, the "Metronome Architectural Protocol" on page 141, and the "Watchdog Timer Architectural Protocol" on page 167.
Protocol Handler	Provides services to add and remove handles from the handle database. It also provides services to add and remove protocols from the handles in the handle database. Additional services are available that allow any component to lookup handles in the handle database, and open and close protocols in the handle database.
Image	Provides services to load, start, exit, and unload images using the PE/COFF image format. These services use the services of the "Security Architectural Protocols" on page 152 if it is present.
Driver Support	Provides services to connect and disconnect drivers to devices in the platform. These services are used by the BDS phase to either connect all drivers to all devices, or to connect only the minimum number of drivers to devices required to establish the consoles and boot an operating system. The minimal connect strategy is one possible mechanism to reduce boot time.

2.2.3 UEFI Runtime Services Table

Table 5 provides a summary of the services that are available through the UEFI Runtime Services Table. These services are described in detail in the UEFI 2.0 specification. One additional runtime service, Status Code Services, is described in this specification.

Table 5. UEFI Runtime Services

UEFI Runtime Services	Description
Variable	Provides services to look up, add, and remove environment variables from nonvolatile storage. These services depend on the Variable Architectural Protocol and the Variable Write Architectural Protocol.
Real Time Clock	Provides services to get and set the current time and date. It also provides services to get and set the time and date of an optional wake-up timer. These services depend on the Real Time Clock Architectural Protocol.
Reset	Provides services to shut down or reset the platform. These services depend on the Reset Architectural Protocol.
Virtual Memory	Provides services that allow the runtime DXE components to be converted from a physical memory map to a virtual memory map. These services can only be called once in physical mode. Once the physical to virtual conversion has been performed, these services cannot be called again. These services depend on the Runtime Architectural Protocol.

2.2.4 DXE Services Table

Table 6 provides a summary of the services that are available through the DXE Services Table. These are new services that are available in boot service time and are required only by the DXE Foundation and DXE drivers.

Table 6. DXE Services

DXE Services	Description
Global Coherency Domain	Provides services to manage I/O resources, memory-mapped I/O resources, and system memory resources in the platform. These services are used to dynamically add and remove these resources from the processor’s global coherency domain.
Dispatcher	Provides services to manage DXE drivers that are being dispatched by the DXE Dispatcher.

2.3 DXE Foundation

The DXE Foundation is a boot service image that is responsible for producing the following:

- UEFI Boot Services
- UEFI Runtime Services
- DXE Services

The DXE Foundation consumes a HOB list and the services of the DXE Architectural Protocols to produce the full complement of UEFI Boot Services, UEFI Runtime Services, and DXE Services. The HOB list is described in detail in the *Volume 3*.

The DXE Foundation is an implementation of UEFI. The DXE Foundation defined in this specification is backward compatible with the UEFI 2.0 specification. As a result, both the DXE Foundation and DXE drivers share many of the attributes of UEFI images. Because this specification makes extensions to the standard UEFI interfaces, DXE images will not be functional on UEFI systems that are not compliant with this DXE CIS. However, UEFI images must be functional on all UEFI-compliant systems including those that are compliant with the DXE CIS.

2.4 DXE Dispatcher

The DXE Dispatcher is one component of the DXE Foundation. This component is required to discover DXE drivers stored in firmware volumes and execute them in the proper order. The proper order is determine by a combination of an a priori file that is optionally stored in the firmware volume and the dependency expressions that are part of the DXE drivers. The dependency expression tells the DXE Dispatcher the set of services that a particular DXE driver requires to be present for the DXE driver to execute. The DXE Dispatcher does not allow a DXE driver to execute until all of the DXE driver's dependencies have been satisfied. After all of the DXE drivers have been loaded and executed by the DXE Dispatcher, control is handed to the BDS Architectural Protocol that is responsible for implementing a boot policy that is compliant with the UEFI Boot Manager described in the UEFI 2.0 specification.

2.5 DXE Drivers

The DXE drivers are required to initialize the processor, chipset, and platform. They are also required to produce the DXE Architectural Protocols and any additional protocol services required to produce I/O abstractions for consoles and boot devices.

2.6 DXE Architectural Protocols

Table 7 provides a summary of the DXE Architectural Protocols. The DXE Foundation is abstracted from the platform through the DXE Architectural Protocols. The DXE Architectural Protocols manifest the platform-specific components of the DXE Foundation. DXE drivers that are loaded and executed by the DXE Dispatcher component of the DXE Foundation must produce these protocols.

Table 7. DXE Architectural Protocols

DXE Architectural Protocols	Description
Security Architectural	Allows the DXE Foundation to authenticate files stored in firmware volumes before they are used.
CPU Architectural	Provides services to manage caches, manage interrupts, retrieve the processor's frequency, and query any processor-based timers.
Metronome Architectural	Provides the services required to perform very short calibrated stalls.
Timer Architectural	Provides the services required to install and enable the heartbeat timer interrupt required by the timer services in the DXE Foundation.
BDS Architectural	Provides an entry point that the DXE Foundation calls once after all of the DXE drivers have been dispatched from all of the firmware volumes. This entry point is the transition from the DXE phase to the Boot Device Selection (BDS) phase, and it is responsible for establishing consoles and enabling the boot devices required to boot an OS.
Watchdog Timer Architectural	Provides the services required to enable and disable a watchdog timer in the platform.
Runtime Architectural	Provides the services required to convert all runtime services and runtime drivers from physical mappings to virtual mappings.
Variable Architectural	Provides the services to retrieve environment variables and set volatile environment variables.
Variable Write Architectural Protocol	Provides the services to set nonvolatile environment variables.
Monotonic Counter Architectural	Provides the services required by the DXE Foundation to manage a 64-bit monotonic counter.
Reset Architectural	Provides the services required to reset or shutdown the platform.
Real Time Clock Architectural	Provides the services to retrieve and set the current time and date as well as the time and date of an optional wake-up timer.
Capsule Architectural Protocol	Provides the services to retrieve and set the current time and date as well as the time and date of an optional wake-up timer.

2.7 Runtime Protocol

Table 8 provides a summary of the runtime protocol for status codes.

Table 8. Status Codes Runtime Protocol

Status Code Runtime Protocol:	Provides the services to send status codes from the DXE Foundation or DXE drivers to a log or device.

3 Boot Manager

3.1 Boot Manager

The Boot Manager in DXE executes after all the DXE drivers whose dependencies have been satisfied have been dispatched by the DXE Dispatcher. At that time, control is handed to the Boot Device Selection (BDS) phase of execution. The BDS phase is responsible for implementing the platform boot policy. System firmware that is compliant with this specification must implement the boot policy specified in the Boot Manager chapter of the UEFI 2.0 specification. This boot policy provides flexibility that allows system vendors to customize the user experience during this phase of execution.

The Boot Manager must also support booting from a short-form device path that starts with the first node being a firmware volume device path. The boot manager must use the GUID in the firmware volume device node to match it to a firmware volume in the system. The GUID in the firmware volume device path is compared with the firmware volume name GUID. If a match is made, then the firmware volume device path can be appended to the device path of the matching firmware volume and normal boot behavior can then be used.

The BDS phase is implemented as part of the BDS Architectural Protocol. The DXE Foundation will hand control to the BDS Architectural Protocol after all of the DXE drivers whose dependencies have been satisfied have been loaded and executed by the DXE Dispatcher. The BDS phase is responsible for the following:

- Initializing console devices
- Loading device drivers
- Attempting to load and execute boot selections

If the BDS phase cannot make forward progress, it will reinvoke the DXE Dispatcher to see if the dependencies of any additional DXE drivers have been satisfied since the last time the DXE Dispatcher was invoked.

4 UEFI System Table

4.1 DXE Services Table

DXE_SERVICES

Summary

Contains a table header and pointers to all of the DXE-specific services.

Related Definitions

```
#define DXE_SERVICES_SIGNATURE  0x565245535f455844
#define DXE_SPECIFICATION_MAJOR_REVISION  1
#define DXE_SPECIFICATION_MINOR_REVISION  60
#define DXE_SERVICES_REVISION
((DXE_SPECIFICATION_MAJOR_REVISION<<16) |
(DXE_SPECIFICATION_MINOR_REVISION)

typedef struct {
  EFI_TABLE_HEADER                  Hdr;

  //
  // Global Coherency Domain Services
  //
  EFI_ADD_MEMORY_SPACE              AddMemorySpace;
  EFI_ALLOCATE_MEMORY_SPACE         AllocateMemorySpace;
  EFI_FREE_MEMORY_SPACE             FreeMemorySpace;
  EFI_REMOVE_MEMORY_SPACE           RemoveMemorySpace;
  EFI_GET_MEMORY_SPACE_DESCRIPTOR   GetMemorySpaceDescriptor;
  EFI_SET_MEMORY_SPACE_ATTRIBUTES   SetMemorySpaceAttributes;
  EFI_GET_MEMORY_SPACE_MAP          GetMemorySpaceMap;
  EFI_ADD_IO_SPACE                  AddIoSpace;
  EFI_ALLOCATE_IO_SPACE             AllocateIoSpace;
  EFI_FREE_IO_SPACE                 FreeIoSpace;
  EFI_REMOVE_IO_SPACE               RemoveIoSpace;
  EFI_GET_IO_SPACE_DESCRIPTOR       GetIoSpaceDescriptor;
  EFI_GET_IO_SPACE_MAP              GetIoSpaceMap;

  //
  // Dispatcher Services
  //
  EFI_DISPATCH                      Dispatch;
  EFI_SCHEDULE                      Schedule;
  EFI_TRUST                         Trust;
  //
  // Service to process a single firmware volume found in
  // a capsule
  //
  EFI_PROCESS_FIRMWARE_VOLUME       ProcessFirmwareVolume;
  //
  // Extensions to Global Coherency Domain Services
  //
  EFI_SET_MEMORY_SPACE_CAPABILITIES SetMemorySpaceCapabilities;
} DXE_SERVICES;
```

Parameters

Hdr

The table header for the DXE Services Table. This header contains the **DXE_SERVICES_SIGNATURE** and **DXE_SERVICES_REVISION** values along with the size of the **DXE_SERVICES_TABLE** structure and a 32-bit CRC to verify that the contents of the DXE Services Table are valid.

AddMemorySpace

Adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor. See the **AddMemorySpace()** function description in this document.

AllocateMemorySpace

Allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **AllocateMemorySpace()** function description in this document.

FreeMemorySpace

Frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **FreeMemorySpace()** function description in this document.

RemoveMemorySpace

Removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor. See the **RemoveMemorySpace()** function description in this document.

GetMemorySpaceDescriptor

Retrieves the descriptor for a memory region containing a specified address. See the **GetMemorySpaceDescriptor()** function description in this document.

SetMemorySpaceAttributes

Modifies the attributes for a memory region in the global coherency domain of the processor. See the **SetMemorySpaceAttributes()** function description in this document.

GetMemorySpaceMap

Returns a map of the memory resources in the global coherency domain of the processor. See the **GetMemorySpaceMap()** function description in this document.

AddIoSpace

Adds reserved I/O or I/O resources to the global coherency domain of the processor. See the **AddIoSpace()** function description in this document.

AllocateIoSpace

Allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor. See the **AllocateIoSpace()** function description in this document.

FreeIoSpace

Frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor. See the **FreeIoSpace()** function description in this document.

RemoveIoSpace

Removes reserved I/O or I/O resources from the global coherency domain of the processor. See the **RemoveIoSpace()** function description in this document.

GetIoSpaceDescriptor

Retrieves the descriptor for an I/O region containing a specified address. See the **GetIoSpaceDescriptor()** function description in this document.

GetIoSpaceMap

Returns a map of the I/O resources in the global coherency domain of the processor. See the **GetIoSpaceMap()** function description in this document.

Dispatch

Loads and executed DXE drivers from firmware volumes. See the **Dispatch()** function description in this document.

Schedule

Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume. See the **Schedule()** function description in this document.

Trust

Promotes a file stored in a firmware volume from the untrusted to the trusted state. See the **Trust()** function description in this document.

ProcessFirmwareVolume

Creates a firmware volume handle for a firmware volume that is present in system memory. See the **ProcessFirmwareVolume()** function description in this document.

SetMemorySpaceCapabilities

Modifies the capabilities for a memory region in the global coherency domain of the processor. See the **SetMemorySpaceCapabilities()**function description in this document.

Description

The UEFI DXE Services Table contains a table header and pointers to all of the DXE-specific services. Except for the table header, all elements in the DXE Services Tables are prototypes of function pointers to functions as defined in “Services - DXE Services” on page 34.

4.2 UEFI Image Entry Point Examples

4.2.1 UEFI Application Example

The following example shows the UEFI image entry point for an UEFI application. This application makes use of the UEFI System Table, UEFI Boot Services Table, UEFI Runtime Services Table, and DXE Services Table.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiApplicationEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN                             Index;
  BOOLEAN                           Result;
  EFI_STATUS                        Status;
  EFI_TIME                          *Time;
  UINTN                             NumberOfDescriptors;
  EFI_GCD_MEMORY_SPACE_DESCRIPTOR  MemorySpaceDescriptor;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_NOT_FOUND;
```

```
}

//
// Use UEFI System Table to print “Hello World” to the active console
// output device.
//
Status = gST->ConOut->OutputString (gST->ConOut, L”Hello World\n\r”);
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use UEFI Boot Services Table to allocate a buffer to store the
// current time and date.
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (EFI_TIME),
                (VOID **)&Time
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use the UEFI Runtime Services Table to get the current
// time and date.
//
Status = gRT->GetTime (&Time, NULL)
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use UEFI Boot Services to free the buffer that was used to store
// the current time and date.
//
Status = gBS->FreePool (Time);
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Use the DXE Services Table to get the current GCD Memory Space Map
//
Status = gDS->GetMemorySpaceMap (
                &NumberOfDescriptors,
                &MemorySpaceMap
                );
if (EFI_ERROR (Status)) {
  return Status;
```

```
    }

    //
    // Use UEFI Boot Services to free the buffer that was used to store
    // the GCD Memory Space Map.
    //
    Status = gBS->FreePool (MemorySpaceMap);
    if (EFI_ERROR (Status)) {
      return Status;
    }

    return Status;
  }
```

4.2.2 Non-UEFI Driver Model Example (Resident in Memory)

The following example shows the UEFI image entry point for an UEFI driver that does not follow the *UEFI Driver Model*. Because this driver returns **EFI_SUCCESS**, it will stay resident in memory after it exits.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN    Index;
  BOOLEAN  Result;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_REQUEST_UNLOAD_IMAGE;
  }

  //
  // Implement driver initialization here.
  //

  return EFI_SUCCESS;
}
```

4.2.3 Non-UEFI Driver Model Example (Nonresident in Memory)

The following example shows the UEFI image entry point for an UEFI driver that also does not follow the *UEFI Driver Model*. Because this driver returns the error code **EFI_REQUEST_UNLOAD_IMAGE**, it will not stay resident in memory after it exits.

```
EFI_GUID  gEfiDxeServicesTableGuid = DXE_SERVICES_TABLE_GUID;

EFI_SYSTEM_TABLE            *gST;
EFI_BOOT_SERVICES           *gBS;
EFI_RUNTIME_SERVICES        *gRT;
DXE_SERVICES                *gDS;

EfiDriverEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  UINTN    Index;
  BOOLEAN  Result;

  gST = SystemTable;
  gBS = gST->BootServices;
  gRT = gST->RuntimeServices;

  gDS = NULL;
  for (Index = 0; Index < gST->NumberOfTableEntries; Index++) {
    Result = EfiCompareGuid (
               &gEfiDxeServicesTableGuid,
               &(gST->ConfigurationTable[Index].VendorGuid)
               );
    if (Result) {
      gDS = gST->ConfigurationTable[Index].VendorTable;
    }
  }
  if (gDS == NULL) {
    return EFI_REQUEST_UNLOAD_IMAGE;
  }

  //
  // Implement driver initialization here.
  //

  return EFI_REQUEST_UNLOAD_IMAGE;
}
```

4.2.4 UEFI Driver Model Example

The following is an *UEFI Driver Model* example that shows the driver initialization routine for the ABC device controller that is on the XYZ bus. The **EFI_DRIVER_BINDING_PROTOCOL** is defined in Chapter 9 of the UEFI 2.0 specification. The function prototypes for the **AbcSupported()**, **AbcStart()**, and **AbcStop()** functions are defined in Section 9.1 of the UEFI 2.0 specification. This function saves the driver's image handle and a pointer to the UEFI Boot Services Table in global variables, so that the other functions in the same driver can have access to these values. It then creates an instance of the **EFI_DRIVER_BINDING_PROTOCOL** and installs it onto the driver's image handle.

```
extern EFI_GUID                      gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                    *gBS;
static EFI_DRIVER_BINDING_PROTOCOL   mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  0x10,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  mAbcDriverBinding->ImageHandle         = ImageHandle;
  mAbcDriverBinding->DriverBindingHandle = ImageHandle;

  Status = gBS->InstallMultipleProtocolInterfaces(
                &mAbcDriverBinding->DriverBindingHandle,
                &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                NULL
                );
  return Status;
}
```

4.2.5 UEFI Driver Model Example (Unloadable)

The following is the same *UEFI Driver Model* example as in the UEFI Driver Model Example, except that it also includes the code required to allow the driver to be unloaded through the boot service **Unload()**. Any protocols installed or memory allocated in **AbcEntryPoint()** must be uninstalled or freed in the **AbcUnload()**. The **AbcUnload()** function first checks to see how many controllers this driver is currently managing. If the number of controllers is greater than zero, then this driver cannot be unloaded at this time, so an error is returned.

```
extern EFI_GUID                     gEfiLoadedImageProtocolGuid;
extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
EFI_BOOT_SERVICES                   *gBS;
static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBinding = {
  AbcSupported,
  AbcStart,
  AbcStop,
  1,
  NULL,
  NULL
};

EFI_STATUS
AbcUnload (
  IN EFI_HANDLE  ImageHandle
  );

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS                 Status;
  EFI_LOADED_IMAGE_PROTOCOL  *LoadedImage;

  gBS = SystemTable->BootServices;

  Status = gBS->OpenProtocol (
                  ImageHandle,
                  &gEfiLoadedImageProtocolGuid,
                  &LoadedImage,
                  ImageHandle,
                  NULL,
                  EFI_OPEN_PROTOCOL_GET_PROTOCOL
                  );
  if (EFI_ERROR (Status)) {
    return Status;
  }
```

```
    LoadedImage->Unload = AbcUnload;

    mAbcDriverBinding->ImageHandle         = ImageHandle;
    mAbcDriverBinding->DriverBindingHandle = ImageHandle;

    Status = gBS->InstallMultipleProtocolInterfaces(
                    &mAbcDriverBinding->DriverBindingHandle,
                    &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                    NULL
                    );

    return Status;
  }

  EFI_STATUS
  AbcUnload (
    IN EFI_HANDLE  ImageHandle
    )

  {
    EFI_STATUS  Status;
    UINTN       Count;

    Status = LibGetManagedControllerHandles (ImageHandle, &Count, NULL);
    if (EFI_ERROR (Status)) {
      return Status;
    }

    if (Count > 0) {
      return EFI_ACCESS_DENIED;
    }

    Status = gBS->UninstallMultipleProtocolInterfaces (
                    ImageHandle,
                    &gEfiDriverBindingProtocolGuid, &mAbcDriverBinding,
                    NULL
                    );
    return Status;
  }
```

4.2.6 UEFI Driver Model Example (Multiple Instances)

The following is the same as the first UEFI Driver Model example, except that it produces three **EFI_DRIVER_BINDING_PROTOCOL** instances. The first one is installed onto the driver's image handle. The other two are installed onto newly created handles.

```
  extern EFI_GUID                     gEfiDriverBindingProtocolGuid;
  EFI_BOOT_SERVICES                   *gBS;
```

```
static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingA = {
  AbcSupportedA,
  AbcStartA,
  AbcStopA,
  1,
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingB = {
  AbcSupportedB,
  AbcStartB,
  AbcStopB,
  1,
  NULL,
  NULL
};

static EFI_DRIVER_BINDING_PROTOCOL  mAbcDriverBindingC = {
  AbcSupportedC,
  AbcStartC,
  AbcStopC,
  1,
  NULL,
  NULL
};

AbcEntryPoint(
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )

{
  EFI_STATUS  Status;

  gBS = SystemTable->BootServices;

  //
  // Install mAbcDriverBindingA onto ImageHandle
  //
  mAbcDriverBindingA->ImageHandle         = ImageHandle;
  mAbcDriverBindingA->DriverBindingHandle = ImageHandle;



  Status = gBS->InstallMultipleProtocolInterfaces(
                &mAbcDriverBindingA->DriverBindingHandle,
                &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingA,
                NULL
                );
```

```
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Install mAbcDriverBindingB onto a newly created handle
    //
    mAbcDriverBindingB->ImageHandle         = ImageHandle;
    mAbcDriverBindingB->DriverBindingHandle = NULL;

    Status = gBS->InstallMultipleProtocolInterfaces(
                    &mAbcDriverBindingB->DriverBindingHandle,
                    &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingB,
                    NULL
                    );
    if (EFI_ERROR (Status)) {
      return Status;
    }

    //
    // Install mAbcDriverBindingC onto a newly created handle
    //
    mAbcDriverBindingC->ImageHandle         = ImageHandle;
    mAbcDriverBindingC->DriverBindingHandle = NULL;

    Status = gBS->InstallMultipleProtocolInterfaces(
                    &mAbcDriverBindingC->DriverBindingHandle,
                    &gEfiDriverBindingProtocolGuid, &mAbcDriverBindingC,
                    NULL
                    );

    return Status;
  }
```

5 Services - Boot Services

5.1 Extensions to UEFI Boot Service Event Usage

5.1.1 CreateEvent

CreateEventEx() in UEFI 2.0 allows for registration of events named by GUID's. The DXE foundation defines the following:

```
#define EFI_EVENT_LEGACY_BOOT_GUID
 {0x2a571201, 0x4966, 0x47f6, 0x8b, 0x86, 0xf3, 0x1e,
 0x41, 0xf3, 0x2f, 0x10}
```

This event is to be used with **CreateEventEx()** in order to be notified when the UEFI boot manager is about to boot a legacy boot option. Notification of events of this type is sent just before Int19h is invoked.

5.1.2 Pre-Defined Event Groups

This section describes the pre-defined event groups used by this specification.

EFI_EVENT_GROUP_DXE_DISPATCH_GUID

This event group is notified by the system when the DXE dispatcher finished one round of driver dispatch. This allows the SMM dispatcher get chance to dispatch SMM driver which will depend on UEFI protocols.

Related Definitions

```
#define EFI_EVENT_GROUP_DXE_DISPATCH_GUID \
  { 0x7081e22f, 0xcac6, 0x4053, { 0x94, 0x68, 0x67, 0x57, \
  0x82, 0xcf, 0x88, 0xe5 } \ }
```

5.1.2.1 End of DXE Event

Prior to invoking any UEFI drivers, or applications that are not from the platform manufacturer, or connecting consoles, the platform should signals the event **EFI_END_OF_DXE_EVENT_GUID** End of DXE Event and immediately after that the platform installs DXE SMM Ready to Lock Protocol (defined in volume 4)..

```
#define EFI_END_OF_DXE_EVENT_GROUP_GUID \
 { 0x2ce967a, 0xdd7e, 0x4ffc, { 0x9e, 0xe7, 0x81, 0xc, \
  0xf0, 0x47, 0x8, 0x80 } }
```

From SEC through the signaling of this event, all of the components should be under the authority of the platform manufacturer and not have to worry about interaction or corruption by 3rd party extensible modules such as UEFI drivers and UEFI applications.

Platform may choose to lock certain resources or disable certain interfaces prior to executing third party extensible modules. Transition from the environment where all of the components are under the authority of the platform manufacturer to the environment where third party modules are executed is a two-step process:

1. End of DXE Event is signaled. This event presents the last opportunity to use resources or interfaces that are going to be locked or disabled in anticipation of the invocation of 3rd party extensible modules.
2. DXE SMM Ready to Lock Protocol is installed. PI modules that need to lock or protect their resources in anticipation of the invocation of 3rd party extensible modules should register for notification on installation of this protocol and effect the appropriate protections in their notification handlers

5.1.3 Additions to LoadImage()

Summary

Loads an UEFI image into memory. This function has been extended from the **LoadImage()** Boot Service defined in the UEFI 2.0 specification. The DXE foundation extends this to support an additional image type, allowing UEFI images to be loaded from files stored in firmware volumes. It also validates the image using the services of the Security Architectural Protocol.

Prototype

```
EFI_STATUS
LoadImage (
  IN BOOLEAN          BootPolicy,
  IN EFI_HANDLE       ParentImageHandle,
  IN EFI_DEVICE_PATH  *FilePath,
  IN VOID             *SourceBuffer  OPTIONAL ,
  IN UINTN            SourceSize,
  OUT EFI_HANDLE      *ImageHandle
  );
```

Parameters

BootPolicy

If **TRUE**, indicates that the request originates from the boot manager, and that the boot manager is attempting to load *FilePath* as a boot selection. Ignored if *SourceBuffer* is not **NULL**.

ParentImageHandle

The caller's image handle. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description in the UEFI 2.0 specification. This field is used to initialize the *ParentHandle* field of the **LOADED_IMAGE** protocol for the image that is being loaded.

FilePath

The specific file path from which the image is loaded. Type **EFI_DEVICE_PATH** is defined in the **LocateDevicePath()** function description in the UEFI 2.0 specification.

SourceBuffer

If not **NULL**, a pointer to the memory location containing a copy of the image to be loaded.

SourceSize

The size in bytes of *SourceBuffer*. Ignored if *SourceBuffer* is **NULL**.

ImageHandle

Pointer to the returned image handle that is created when the image is successfully loaded. Type **EFI_HANDLE** is defined in the **InstallProtocolInterface()** function description in the UEFI 2.0 specification.

Description

The **LoadImage()** function loads an UEFI image into memory and returns a handle to the image. The supported subsystem values in the PE image header are listed in "Related Definitions" below. The image is loaded in one of two ways. If *SourceBuffer* is not **NULL**, the function is a memory-to-memory load in which *SourceBuffer* points to the image to be loaded and *SourceSize* indicates the image's size in bytes. *FilePath* specifies where the image specified by *SourceBuffer* and *SourceSize* was loaded. In this case, the caller has copied the image into *SourceBuffer* and can free the buffer once loading is complete.

If *SourceBuffer* is **NULL**, the function is a file copy operation that uses the **EFI_FIRMWARE_VOLUME2_PROTOCOL**, followed by the **SIMPLE_FILE_SYSTEM_PROTOCOL** and then the **LOAD_FILE_PROTOCOL** to access the file referred to by *FilePath*. In this case, the *BootPolicy* flag is passed to the **LOAD_FILE.LoadFile()** function and is used to load the default image responsible for booting when the *FilePath* only indicates the device. For more information see the discussion of the Load File Protocol in Chapter 11 of the UEFI 2.0 specification.

Regardless of the type of load (memory-to-memory or file copy), the function relocates the code in the image while loading it.

The image is also validated using the **FileAuthenticationState()** service of the Security Architectural Protocol (SAP). If the SAP returns the status **EFI_SUCCESS**, then the load operation is completed normally. If the SAP returns the status **EFI_SECURITY_VIOLATION**, then the load operation is completed normally, and the **EFI_SECURITY_VIOLATION** status is returned. In this case, the caller is not allowed to start the image until some platform specific policy is executed to protect the system while executing untrusted code. If the SAP returns the status **EFI_ACCESS_DENIED**, then the image should never be trusted. In this case, the image is unloaded from memory, and **EFI_ACCESS_DENIED** is returned.

Once the image is loaded, firmware creates and returns an **EFI_HANDLE** that identifies the image and supports the **LOADED_IMAGE_PROTOCOL**. The caller may fill in the image's "load options" data, or add additional protocol support to the handle before passing control to the newly loaded image by calling **StartImage()**. Also, once the image is loaded, the caller either starts it by calling **StartImage()** or unloads it by calling **UnloadImage()**.

Related Definitions

```
//*******************************************************************
// Supported subsystem values
//*******************************************************************

#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION          10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER  11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER       12
#define EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER       13
```

Table 9 describes the fields in the above definition.

Table 9. Supported Subsystem Values

Supported Subsystem Values	Description
EFI_IMAGE_SUBSYSTEM_EFI_ APPLICATION	The image is loaded into memory of type **EfiLoaderCode**, and the memory is freed when the application exits.
EFI_IMAGE_SUBSYSTEM_EFI_ BOOT_SERVICE_DRIVER	The image is loaded into memory of type **EfiBootServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed.
EFI_IMAGE_SUBSYSTEM_EFI_ RUNTIME_DRIVER	The image is loaded into memory of type **EfiRuntimeServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed. Images of this type are automatically converted from physical addresses to virtual address when the Runtime Service **SetVirtualAddressMap()** is called.
EFI_IMAGE_SUBSYSTEM_SAL_ RUNTIME_DRIVER	The image is loaded into memory of type **EfiRuntimeServicesCode**. If the image exits with an error code, then the memory for the image is free. If the image exits with **EFI_SUCCESS**, then the memory for the image is not freed. Images of this type are not converted from physical to virtual addresses when the Runtime Service **SetVirtualAddressMap()** is called.

Status Codes Returned

EFI_SUCCESS	The image was loaded into memory.
EFI_SECURITY_VIOLATION	The image was loaded into memory, but the current security policy dictates that the image should not be executed at this time.
EFI_ACCESS_DENIED	The image was not loaded into memory because the current security policy dictates that the image should never be executed.
EFI_NOT_FOUND	The *FilePath* was not found.
EFI_INVALID_PARAMETER	One of the parameters has an invalid value.
EFI_UNSUPPORTED	The image type is not supported, or the device path cannot be parsed to locate the proper protocol for loading the file.
EFI_OUT_OF_RESOURCES	Image was not loaded due to insufficient resources.
EFI_LOAD_ERROR	Image was not loaded because the image format was corrupt or not understood.
EFI_DEVICE_ERROR	Image was not loaded because the device returned a read error.

6 Runtime Capabilities

6.1 Additional Runtime Protocol

6.1.1 Status Code Services

Table 10 lists the runtime protocol that are used to report status codes. This protocol provides a runtime protocol that can be bound by other runtime drivers for reporting status information.

Table 10. Status Code Runtime Protocol

Name	Type	Description
ReportStatusCode	Runtime	Reports status codes at boot services time and runtime.

7 Services - DXE Services

7.1 Introduction

This chapter describes the services in the DXE Services Table. These services include the following:

- Global Coherency Domain (GCD) Services
- Dispatcher Services

The GCD Services are used to manage the system memory, memory-mapped I/O, and I/O resources present in a platform. The Dispatcher Services are used to invoke the DXE Dispatcher and modify the state of a DXE driver that is being tracked by the DXE Dispatcher.

7.2 Global Coherency Domain Services

7.2.1 Global Coherency Domain (GCD) Services Overview

The Global Coherency Domain (GCD) Services are used to manage the memory and I/O resources visible to the boot processor. These resources are managed in two different maps:

- GCD memory space map
- GCD I/O space map

If memory or I/O resources are added, removed, allocated, or freed, then the GCD memory space map and GCD I/O space map are updated. GCD Services are also provided to retrieve the contents of these two resource maps.

The GCD Services can be broken up into two groups. The first manages the memory resources visible to the boot processor, and the second manages the I/O resources visible to the boot processor. Not all processor types support I/O resources, so the management of I/O resources may not be required. However, since system memory resources and memory-mapped I/O resources are required to execute the DXE environment, the management of memory resources is always required.

7.2.2 GCD Memory Resources

The Global Coherency Domain (GCD) Services used to manage memory resources include the following:

- **`AddMemorySpace()`**
- **`AllocateMemorySpace()`**
- **`FreeMemorySpace()`**
- **`RemoveMemorySpace()`**
- **`SetMemorySpaceAttributes()`**
- **`SetMemorySpaceCapabilities()`**

The GCD Services used to retrieve the GCD memory space map include the following:

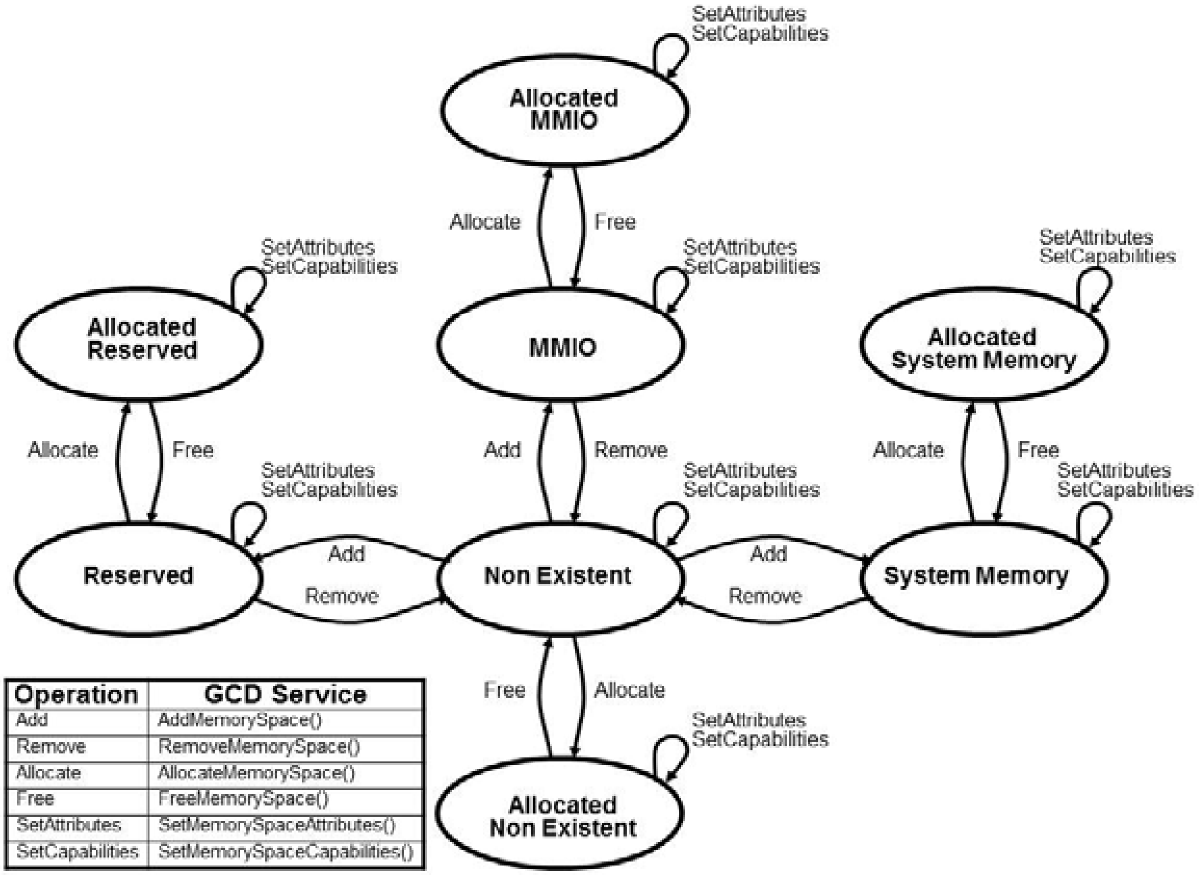
- **GetMemorySpaceDescriptor()**
- **GetMemorySpaceMap()**

The GCD memory space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access memory resources. This information is used to initialize the state of the GCD memory space map. Any memory regions outside this initial region are not available to any of the GCD Services that are used to manage memory resources. The GCD memory space map is designed to describe the memory address space with as many as 64 address lines. Each region in the GCD memory space map can begin and end on a byte boundary. There are additional HOB types that describe the location of system memory, the location memory mapped I/O, the location of firmware devices, the location of firmware volumes, the location of reserved regions, and the location of system memory regions that were allocated prior to the execution of the DXE Foundation. The DXE Foundation must parse the contents of the HOB list to guarantee that memory regions reserved prior to the execution of the DXE Foundation are honored. As a result, the GCD memory space map must reflect the memory regions described in the HOB list. The GCD memory space map provides the DXE Foundation with the information required to initialize the memory services such as **AllocatePages()**, **FreePages()**, **AllocatePool()**, **FreePool()**, and **GetMemoryMap()**. See the UEFI 2.0 specification for definitions of these services.

A memory region described by the GCD memory space map can be in one of several different states:

- Nonexistent memory
- System memory
- Memory-mapped I/O
- Reserved memory

These memory regions can be allocated and freed by DXE drivers executing in the DXE environment. In addition, a DXE driver can attempt to adjust the caching attributes of a memory region. Figure 2 shows the possible state transitions for each byte of memory in the GCD memory space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD services are required to merge similar memory regions that are adjacent to each other into a single memory descriptor, which reduces the number of entries in the GCD memory space map.



Operation	GCD Service
Add	AddMemorySpace()
Remove	RemoveMemorySpace()
Allocate	AllocateMemorySpace()
Free	FreeMemorySpace()
SetAttributes	SetMemorySpaceAttributes()
SetCapabilities	SetMemorySpaceCapabilities()

Figure 2. GCD Memory State Transitions

7.2.3 GCD I/O Resources

The Global Coherency Domain (GCD) Services used to manage I/O resources include the following:

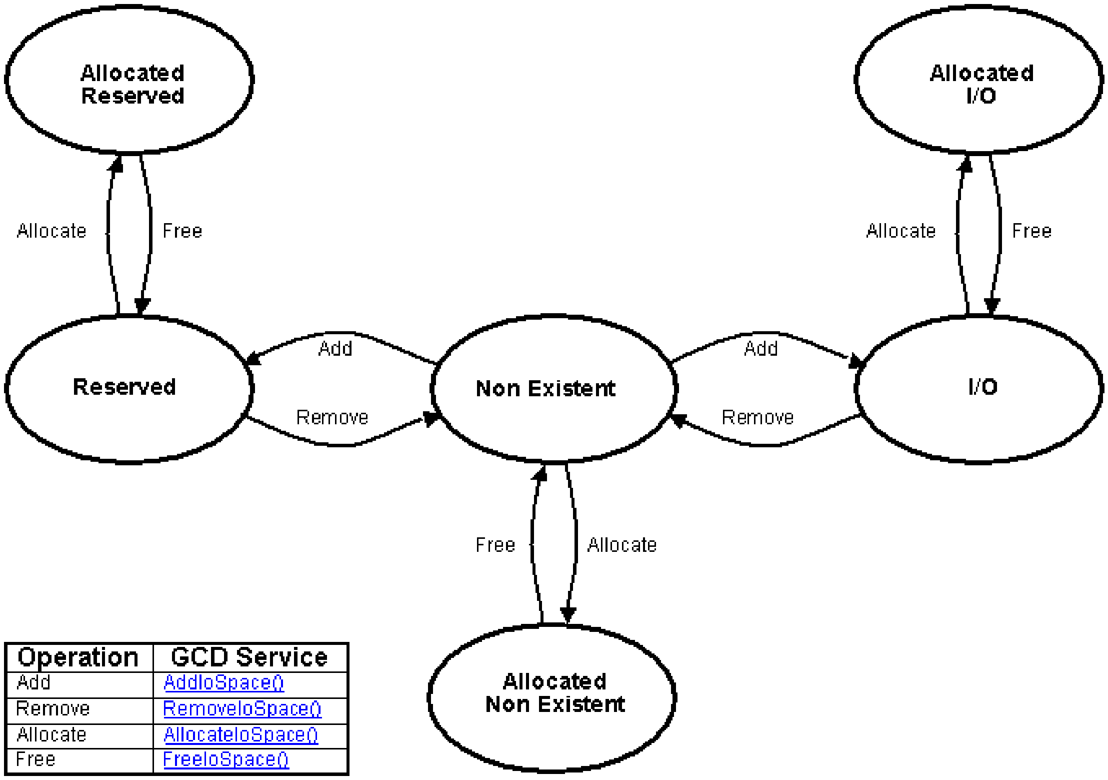
- **`AddIoSpace()`**
- **`AllocateIoSpace()`**
- **`FreeIoSpace()`**
- **`RemoveIoSpace()`**

The GCD Services used to retrieve the GCD I/O space map include the following:

- **`GetIoSpaceDescriptor()`**
- **`GetIoSpaceMap()`**

The GCD I/O space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access I/O resources. This information is used to initialize the state of the GCD I/O space map. Any I/O regions outside this initial region are not available to any of the GCD Services that are used to manage I/O resources. The GCD I/O space map is designed to describe the I/O address space with as many as 64 address lines. Each region in the GCD I/O space map can being and end on a byte boundary.

An I/O region described by the GCD I/O space map can be in several different states. These include nonexistent I/O, I/O, and reserved I/O. These I/O regions can be allocated and freed by DXE drivers executing in the DXE environment. Figure 3 shows the possible state transitions for each byte of I/O in the GCD I/O space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD Services are required to merge similar I/O regions that are adjacent to each other into a single I/O descriptor, which reduces the number of entries in the GCD I/O space map.



Operation	GCD Service
Add	AddIoSpace()
Remove	RemoveIoSpace()
Allocate	AllocateIoSpace()
Free	FreeIoSpace()

Figure 3. GCD I/O State Transitions

7.2.4 Global Coherency Domain Services

The functions that make up Global Coherency Domain (GCD) Services are used during preboot to add, remove, allocate, free, and provide maps of the system memory, memory-mapped I/O, and I/O resources in a platform. These services, used in conjunction with the Memory Allocation Services, provide the ability to manage all the memory and I/O resources in a platform. Table 11 lists the Global Coherency Domain Services.

Table 11. Global Coherency Domain Boot Type Services

Name	Description
AddMemorySpace	This service adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.
AllocateMemorySpace	This service allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
FreeMemorySpace	This service frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
RemoveMemorySpace	This service removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
GetMemorySpaceDescriptor	This service retrieves the descriptor for a memory region containing a specified address.
SetMemorySpaceAttributes	This service modifies the attributes for a memory region in the global coherency domain of the processor.
SetMemorySpaceCapabilities	This service modifies the capabilities for a memory region in the global coherency domain of the processor.
GetMemorySpaceMap	Returns a map of the memory resources in the global coherency domain of the processor.
AddIoSpace	This service adds reserved I/O, or I/O resources to the global coherency domain of the processor.
AllocateIoSpace	This service allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
FreeIoSpace	This service frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
RemoveIoSpace	This service removes reserved I/O, or I/O resources from the global coherency domain of the processor.
GetIoSpaceDescriptor	This service retrieves the descriptor for an I/O region containing a specified address.
GetIoSpaceMap	Returns a map of the I/O resources in the global coherency domain of the processor.

AddMemorySpace()

Summary

This service adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ADD_MEMORY_SPACE) (
  IN EFI_GCD_MEMORY_TYPE     GcdMemoryType,
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length,
  IN UINT64                  Capabilities
  );
```

Parameters

GcdMemoryType

The type of memory resource being added. Type **EFI_GCD_MEMORY_TYPE** is defined in "Related Definitions" below. The only types allowed are **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, **EfiGcdMemoryTypePersistent, EfiGcdMemoryTypeMoreReliable,** and **EfiGcdMemoryTypeMemoryMappedIo**.

BaseAddress

The physical address that is the start address of the memory resource being added. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size, in bytes, of the memory resource that is being added.

Capabilities

The bit mask of attributes that the memory resource region supports. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI 2.0 specification.

Description

The **AddMemorySpace()** function converts unallocated non-existent memory ranges to a range of reserved memory, a range of system memory, or a range of memory mapped I/O. *BaseAddress* and *Length* specify the memory range, and *GcdMemoryType* specifies the memory type. The bit mask of all supported attributes for the memory range being added is specified by *Capabilities*. If the memory range is successfully added, then **EFI_SUCCESS** is returned.

If the memory range specified by *BaseAddress* and *Length* is of type **EfiGcdMemoryTypeSystemMemory** or **EfiGcdMemoryTypeMoreReliable**, then the memory range may be automatically allocated for use by the UEFI memory services. If the addition of the memory range specified by *BaseAddress* and *Length* results in a GCD memory space map containing one or more 4 KiB regions of unallocated **EfiGcdMemoryTypeSystemMemory** or **EfiGcdMemoryTypeMoreReliable** aligned on 4 KiB boundaries, then those regions will always be converted to ranges of allocated **EfiGcdMemoryTypeSystemMemory** or **EfiGcdMemoryTypeMoreReliable** respectively. This extra conversion will never be performed for fragments of memory that do not meet the above criteria.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *GcdMemoryType* is not **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, **EfiGcdMemoryTypeMemoryMappedIo**, **EfiGcdMemoryPersistent** or **EfiGcdMemoryTypeMoreReliable** then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If any portion of the memory range specified by *BaseAddress* and *Length* is not of type **EfiGcdMemoryTypeNonExistent**, then **EFI_ACCESS_DENIED** is returned.

If any portion of the memory range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateMemorySpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to add the memory resource to the global coherency domain of the processor, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_MEMORY_TYPE
//*******************************************************
typedef enum {
  EfiGcdMemoryTypeNonExistent,
  EfiGcdMemoryTypeReserved,
  EfiGcdMemoryTypeSystemMemory,
  EfiGcdMemoryTypeMemoryMappedIo,
  EfiGcdMemoryTypePersistent,
  EfiGcdMemoryTypeMoreReliable,
  EfiGcdMemoryTypeMaximum
} EFI_GCD_MEMORY_TYPE;
```

EfiGcdMemoryTypeNonExistent

A memory region that is visible to the boot processor. However, there are no system components that are currently decoding this memory region.

EfiGcdMemoryTypeReserved

A memory region that is visible to the boot processor. This memory region is being decoded by a system component, but the memory region is not considered to be either system memory or memory-mapped I/O.

EfiGcdMemoryTypeSystemMemory

A memory region that is visible to the boot processor. A memory controller is currently decoding this memory region and the memory controller is producing a tested system memory region that is available to the memory services.

EfiGcdMemoryTypeMemoryMappedIo

A memory region that is visible to the boot processor. This memory region is currently being decoded by a component as memory-mapped I/O that can be used to access I/O devices in the platform.

EfiGcdMemoryTypePersistent

A memory region that is visible to the boot processor. This memory supports byte-addressable non-volatility.

EfiGcdMemoryTypeMoreReliable

A memory region that provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.

Status Codes Returned

EFI_SUCCESS	The memory resource was added to the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdMemoryType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_OUT_OF_RESOURCES	There are not enough system resources to add the memory resource to the global coherency domain of the processor.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* conflicts with a memory resource range that was previously added to the global coherency domain of the processor.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateMemorySpace()**.

AllocateMemorySpace()

Summary

This service allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_MEMORY_SPACE) (
  IN     EFI_GCD_ALLOCATE_TYPE   GcdAllocateType,
  IN     EFI_GCD_MEMORY_TYPE     GcdMemoryType,
  IN     UINTN                   Alignment,
  IN     UINT64                  Length,
  IN OUT EFI_PHYSICAL_ADDRESS    *BaseAddress,
  IN     EFI_HANDLE              ImageHandle,
  IN     EFI_HANDLE              DeviceHandle   OPTIONAL
  );
```

Parameters

GcdAllocateType

The type of allocation to perform. Type **EFI_GCD_ALLOCATE_TYPE** is defined in "Related Definitions" below.

GcdMemoryType

The type of memory resource being allocated. Type **EFI_GCD_MEMORY_TYPE** is defined in **AddMemorySpace()**. The only types allowed are **EfiGcdMemoryTypeNonExistent**, **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystemMemory**, **EfiGcdMemoryTypePersistent,** **EfiGcdMemoryTypeMoreReliable** and **EfiGcdMemoryTypeMemoryMappedIo**.

Alignment

The log base 2 of the boundary that *BaseAddress* must be aligned on output. For example, a value of 0 means that *BaseAddress* can be aligned on any byte boundary, and a value of 12 means that *BaseAddress* must be aligned on a 4 KiB boundary.

Length

The size in bytes of the memory resource range that is being allocated.

BaseAddress

A pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See "Description" below for more information. On output the address is set to the base of the memory resource range that was allocated. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

ImageHandle

The image handle of the agent that is allocating the memory resource. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the memory resource is being allocated. If the memory resource is not being allocated for a device that has an associated device handle, then this parameter is optional and may be **NULL**. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **AllocateMemorySpace()** function searches for a memory range of type *GcdMemoryType* and converts the discovered memory range from the unallocated state to the allocated state. The parameters *GcdAllocateType*, *Alignment*, *Length*, and *BaseAddress* specify the manner in which the GCD memory space map is searched. If a memory range is found that meets the search criteria, then the base address of the memory range is returned in *BaseAddress*, and **EFI_SUCCESS** is returned. *ImageHandle* and *DeviceHandle* are used to convert the memory range from the unallocated state to the allocated state. *ImageHandle* identifies the image that is calling **AllocateMemorySpace()**, and *DeviceHandle* identifies the device that *ImageHandle* is managing that requires the memory range. *DeviceHandle* is optional, because the device that *ImageHandle* is managing might not have an associated device handle. If a memory range meeting the search criteria cannot be found, then **EFI_NOT_FOUND** is returned.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchBottomUp**, then the GCD memory space map is searched from the lowest address up to the highest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchTopDown**, then the GCD memory space map is searched from the highest address down to the lowest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchBottomUp**, then the GCD memory space map is searched from the lowest address up to *BaseAddress* looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchTopDown**, then the GCD memory space map is searched from *BaseAddress* down to the lowest address looking for unallocated memory ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdMemoryType*.

If *GcdAllocateType* is **EfiGcdAllocateAddress**, then the GCD memory space map is checked to see if the memory range starting at *BaseAddress* for *Length* bytes is of type *GcdMemoryType*, unallocated, and begins on a the boundary specified by *Alignment*.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *BaseAddress* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *ImageHandle* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdMemoryType* is not **EfiGcdMemoryTypeNonExistent**, **EfiGcdMemoryTypeReserved**, **EfiGcdMemoryTypeSystem Memory**, **EfiGcdMemoryTypePersistent**, **EfiGcdMemoryTypeMemoryMappedIo**, **EfiGcdMemoryTypeMoreReliable**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdAlocateType* is less than zero, or *GcdAllocateType* is greater than or equal to *EfiGcdMaxAllocateType* then **EFI_INVALID_PARAMETER** is returned.

If there are not enough system resources available to allocate the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_ALLOCATE_TYPE
//*******************************************************
typedef enum {
  EfiGcdAllocateAnySearchBottomUp,
  EfiGcdAllocateMaxAddressSearchBottomUp,
  EfiGcdAllocateAddress,
  EfiGcdAllocateAnySearchTopDown,
  EfiGcdAllocateMaxAddressSearchTopDown,
  EfiGcdMaxAllocateType
} EFI_GCD_ALLOCATE_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The memory resource was allocated from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdAllocateType* is invalid.
EFI_INVALID_PARAMETER	*GcdMemoryType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_INVALID_PARAMETER	*BaseAddress* is **NULL**.
EFI_INVALID_PARAMETER	*ImageHandle* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to allocate the memory resource from the global coherency domain of the processor.
EFI_NOT_FOUND	The memory resource request could not be satisfied.

FreeMemorySpace()

Summary

This service frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FREE_MEMORY_SPACE) (
  IN EFI_PHYSICAL_ADDRESS    BaseAddress,
  IN UINT64                  Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the memory resource being freed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory resource range that is being freed.

Description

The **FreeMemorySpace()** function converts the memory range specified by *BaseAddress* and *Length* from the allocated state to the unallocated state. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were not allocated on previous calls to **AllocateMemorySpace()**, then **EFI_NOT_FOUND** is returned.

If there are not enough system resources available to free the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory resource was freed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.

EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	The memory resource range specified by *BaseAddress* and *Length* was not allocated with previous calls to **AllocateMemorySpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to free the memory resource from the global coherency domain of the processor.

RemoveMemorySpace()

Summary

This service removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_REMOVE_MEMORY_SPACE) (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the memory resource being removed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory resource that is being removed.

Description

The **RemoveMemorySpace()** function converts the memory range specified by *BaseAddress* and *Length* to the memory type **EfiGcdMemoryTypeNonExistent**. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were not added to the GCD memory space map with previous calls to **AddMemorySpace()**, then **EFI_NOT_FOUND** is returned.

If one or more bytes of the memory range specified by *BaseAddress* and *Length* were allocated from the GCD memory space map with previous calls to **AllocateMemorySpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to remove the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory resource was removed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* was not added with previous calls to **AddMemorySpace()**.
EFI_ACCESS_DENIED	One or more bytes of the memory resource range specified by *BaseAddress* and *Length* has been allocated with **AllocateMemorySpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to remove the memory resource from the global coherency domain of the processor.

GetMemorySpaceDescriptor()

Summary

This service retrieves the descriptor for a memory region containing a specified address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MEMORY_SPACE_DESCRIPTOR) (
  IN  EFI_PHYSICAL_ADDRESS             BaseAddress,
  OUT EFI_GCD_MEMORY_SPACE_DESCRIPTOR  *Descriptor
  );
```

Parameters

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Descriptor

A pointer to a caller allocated descriptor. On return, the descriptor describes the memory region containing *BaseAddress*. Type **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is defined in "Related Definitions" below.

Description

The **GetMemorySpaceDescriptor()** function retrieves the descriptor for the memory region that contains the address specified by *BaseAddress*. If a memory region containing *BaseAddress* is found, then the descriptor for that memory region is returned in the caller allocated structure *Descriptor*, and **EFI_SUCCESS** is returned.

If *Descriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If a memory region containing *BaseAddress* is not present in the GCD memory space map, then **EFI_NOT_FOUND** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_MEMORY_SPACE_DESCRIPTOR
//*******************************************************
typedef struct {
  EFI_PHYSICAL_ADDRESS   BaseAddress;
  UINT64                 Length;
  UINT64                 Capabilities;
  UINT64                 Attributes;
  EFI_GCD_MEMORY_TYPE    GcdMemoryType;
  EFI_HANDLE             ImageHandle;
  EFI_HANDLE             DeviceHandle;
} EFI_GCD_MEMORY_SPACE_DESCRIPTOR;
```

Parameters

BaseAddress

The physical address of the first byte in the memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The number of bytes in the memory region.

Capabilities

The bit mask of attributes that the memory region is capable of supporting. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI 2.0 specification.

Attributes

The bit mask of attributes that the memory region is currently using. The bit mask of available attributes is defined in **GetMemoryMap()**.

GcdMemoryType

Type of the memory region. Type **EFI_GCD_MEMORY_TYPE** is defined in the **AddMemorySpace()** function description.

ImageHandle

The image handle of the agent that allocated the memory resource described by *PhysicalStart* and *NumberOfBytes*. If this field is **NULL**, then the memory resource is not currently allocated. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the memory resource has been allocated. If *ImageHandle* is **NULL**, then the memory resource is not currently allocated. If this field is **NULL**, then the memory resource is not associated with a device that is described by a device handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The descriptor for the memory resource region containing *BaseAddress* was returned in *Descriptor*.
EFI_INVALID_PARAMETER	*Descriptor* is **NULL**.
EFI_NOT_FOUND	A memory resource range containing *BaseAddress* was not found.
EFI_NOT_AVAILABLE_YET	The attributes cannot be set because CPU architectural protocol is not available yet.

SetMemorySpaceAttributes()

Summary

This service modifies the attributes for a memory region in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_MEMORY_SPACE_ATTRIBUTES) (
  IN EFI_PHYSICAL_ADDRESS         BaseAddress,
  IN UINT64                       Length,
  IN UINT64                       Attributes
  );
```

Parameters

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory region.

Attributes

The bit mask of attributes to set for the memory region. The bit mask of available attributes is defined in the **GetMemoryMap()**function description in the UEFI 2.0 specification.

Description

The **SetMemorySpaceAttributes()** function modifies the attributes for the memory region specified by *BaseAddress* and *Length* from their current attributes to the attributes specified by *Attributes*. If this modification of attributes succeeds, then **EFI_SUCCESS** is returned.

If the GCD memory space map contains adjacent memory regions that only differ in their base address and length fields, then those adjacent memory regions must be merged into a single memory descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes specified by *Attributes* are not supported for the memory region specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned. The *Attributes* bit mask must be a proper subset of the capabilities bit mask for the specified memory region. The capabilities bit mask is specified when a memory region is added with **AddMemorySpace()** and can be retrieved with **GetMemorySpaceDescriptor()** or **GetMemorySpaceMap()**.

If the attributes for one or more bytes of the memory range specified by *BaseAddress* and *Length* cannot be modified because the current system policy does not allow them to be modified, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to modify the attributes of the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The attributes were set for the memory region.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_UNSUPPORTED	The bit mask of attributes is not support for the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	The attributes for the memory resource range specified by *BaseAddress* and *Length* cannot be modified.
EFI_OUT_OF_RESOURCES	There are not enough system resources to modify the attributes of the memory resource range.

SetMemorySpaceCapabilities()

Summary

This service modifies the capabilities for a memory region in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_MEMORY_SPACE_CAPABILITIES) (
     IN EFI_PHYSICAL_ADDRESS    BaseAddress,
     IN UINT64                  Length,
     IN UINT64                  Capabilities
     );
```

Parameters

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI Specification.

Length

The size in bytes of the memory region.

Capabilities

The bit mask of capabilities that the memory region supports. The bit mask of available attributes is defined in the **GetMemoryMap()** function description in the UEFI specification.

Description

The **SetMemorySpaceCapabilities()** function modifies the capabilities for the memory region specified by *BaseAddress* and *Length* from their current capabilities to the capabilities specified by *Capabilities*. If this modification of capabilities succeeds, then **EFI_SUCCESS** is returned.

If the value for *Capabilities* does not include the current operating memory region attributes (having previously been set by calling **SetMemorySpaceAttributes**) then **EFI_UNSUPPORTED** is returned.

If Length is zero, then **EFI_INVALID_PARAMETER** is returned.

If the capabilities for one or more bytes of the memory range specified by *BaseAddress* and *Length* cannot be modified because the current system policy does not allow them to be modified, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to modify the capabilities of the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The capabilities were set for the memory region.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The capabilities specified by *Capabilities* do not include the memory region attributes currently in use.
EFI_ACCESS_DENIED	The capabilities for the memory resource range specified by *BaseAddress* and *Length* cannot be modified.
EFI_OUT_OF_RESOURCES	There are not enough system resources to modify the capabilities of the memory resource range.

GetMemorySpaceMap()

Summary

Returns a map of the memory resources in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_MEMORY_SPACE_MAP) (
  OUT UINTN                            *NumberOfDescriptors,
  OUT EFI_GCD_MEMORY_SPACE_DESCRIPTOR  **MemorySpaceMap
  );
```

Parameters

NumberOfDescriptors

A pointer to number of descriptors returned in the *MemorySpaceMap* buffer. This parameter is ignored on input, and is set to the number of descriptors in the *MemorySpaceMap* buffer on output.

MemorySpaceMap

A pointer to the array of **EFI_GCD_MEMORY_SPACE_DESCRIPTOR**s. Type **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is defined in **GetMemorySpaceDescriptor()**. This buffer is allocated with **AllocatePool()**, so it is the caller's responsibility to free this buffer with a call to **FreePool()**. The number of descriptors in *MemorySpaceMap* is returned in *NumberOfDescriptors*. See the UEFI 2.0 specification for definitions of **AllocatePool()** and **FreePool()**.

Description

The **GetMemorySpaceMap()** function retrieves the entire GCD memory space map. If there are no errors retrieving the GCD memory space map, then the number of descriptors in the GCD memory space map is returned in *NumberOfDescriptors*, the array of descriptors from the GCD memory space map is allocated with **AllocatePool()**, the descriptors are transferred into *MemorySpaceMap*, and **EFI_SUCCESS** is returned.

If *NumberOfDescriptors* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *MemorySpaceMap* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources to allocate *MemorySpaceMap*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The memory space map was returned in the *MemorySpaceMap* buffer, and the number of descriptors in *MemorySpaceMap* was returned in *NumberOfDescriptors*.
EFI_INVALID_PARAMETER	*NumberOfDescriptors* is **NULL**.

EFI_INVALID_PARAMETER	*MemorySpaceMap* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough resources to allocate *MemorySpaceMap*.

AddIoSpace()

Summary

This service adds reserved I/O, or I/O resources to the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ADD_IO_SPACE) (
  IN EFI_GCD_IO_TYPE          GcdIoType,
  IN EFI_PHYSICAL_ADDRESS     BaseAddress,
  IN UINT64                   Length
  );
```

Parameters

GcdIoType

The type of I/O resource being added. Type **EFI_GCD_IO_TYPE** is defined in "Related Definitions" below. The only types allowed are **EfiGcdIoTypeReserved** and **EfiGcdIoTypeIo**.

BaseAddress

The physical address that is the start address of the I/O resource being added. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource that is being added.

Description

The **AddIoSpace()** function converts unallocated non-existent I/O ranges to a range of reserved I/O, or a range of I/O. *BaseAddress* and *Length* specify the I/O range, and *GcdIoType* specifies the I/O type. If the I/O range is successfully added, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *GcdIoType* is not **EfiGcdIoTypeReserved** or **EfiGcdIoTypeIo**, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If any portion of the I/O range specified by *BaseAddress* and *Length* is not of type **EfiGcdIoTypeNonExistent**, then **EFI_ACCESS_DENIED** is returned.

If any portion of the I/O range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateIoSpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to add the I/O resource to the global coherency domain of the processor, then **EFI_OUT_OF_RESOURCES** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_IO_TYPE
//*******************************************************
typedef enum {
  EfiGcdIoTypeNonExistent,
  EfiGcdIoTypeReserved,
  EfiGcdIoTypeIo,
  EfiGcdIoTypeMaximum
} EFI_GCD_IO_TYPE;
```

EfiGcdIoTypeNonExistent

An I/O region that is visible to the boot processor. However, there are no system components that are currently decoding this I/O region.

EfiGcdIoTypeReserved

An I/O region that is visible to the boot processor. This I/O region is currently being decoded by a system component, but the I/O region cannot be used to access I/O devices.

EfiGcdIoTypeIo

An I/O region that is visible to the boot processor. This I/O region is currently being decoded by a system component that is producing I/O ports that can be used to access I/O devices.

Status Codes Returned

EFI_SUCCESS	The I/O resource was added to the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdIoType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_OUT_OF_RESOURCES	There are not enough system resources to add the I/O resource to the global coherency domain of the processor.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* conflicts with an I/O resource range that was previously added to the global coherency domain of the processor.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* was allocated in a prior call to **AllocateIoSpace()**.

AllocateIoSpace()

Summary

This service allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ALLOCATE_IO_SPACE) (
  IN     EFI_GCD_ALLOCATE_TYPE   AllocateType,
  IN     EFI_GCD_IO_TYPE         GcdIoType,
  IN     UINTN                   Alignment,
  IN     UINT64                  Length,
  IN OUT EFI_PHYSICAL_ADDRESS    *BaseAddress,
  IN     EFI_HANDLE              ImageHandle,
  IN     EFI_HANDLE              DeviceHandle   OPTIONAL
  );
```

Parameters

GcdAllocateType

The type of allocation to perform. Type **EFI_GCD_ALLOCATE_TYPE** is defined in **AllocateMemorySpace()**.

GcdIoType

The type of I/O resource being allocated. Type **EFI_GCD_IO_TYPE** is defined in **AddIoSpace()**. The only types allowed are **EfiGcdIoTypeNonExistent**, **EfiGcdIoTypeReserved**, and **EfiGcdIoTypeIo**.

Alignment

The log base 2 of the boundary that *BaseAddress* must be aligned on output. For example, a value of 0 means that *BaseAddress* can be aligned on any byte boundary, and a value of 12 means that *BaseAddress* must be aligned on a 4 KiB boundary.

Length

The size in bytes of the I/O resource range that is being allocated.

BaseAddress

A pointer to a physical address. On input, the way in which the address is used depends on the value of *Type*. See "Description" below for more information. On output the address is set to the base of the I/O resource range that was allocated. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

ImageHandle

The image handle of the agent that is allocating the I/O resource. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *v*.

DeviceHandle

The device handle for which the I/O resource is being allocated. If the I/O resource is not being allocated for a device that has an associated device handle, then this parameter is optional and may be **NULL**. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **AllocateIoSpace()** function searches for an I/O range of type *GcdIoType* and converts the discovered I/O range from the unallocated state to the allocated state. The parameters *GcdAllocateType*, *Alignment*, *Length*, and *BaseAddress* specify the manner in which the GCD I/O space map is searched. If an I/O range is found that meets the search criteria, then the base address of the I/O range is returned in *BaseAddress*, and **EFI_SUCCESS** is returned. *ImageHandle* and *DeviceHandle* are used to convert the I/O range from the unallocated state to the allocated state. *ImageHandle* identifies the image that is calling **AllocateIoSpace()**, and *DeviceHandle* identifies the device that *ImageHandle* is managing that requires the I/O range. *DeviceHandle* is optional, because the device that *ImageHandle* is managing might not have an associated device handle. If an I/O range meeting the search criteria cannot be found, then **EFI_NOT_FOUND** is returned.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchBottomUp**, then the GCD I/O space map is searched from the lowest address up to the highest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateAnySearchTopDown**, then the GCD I/O space map is searched from the highest address down to the lowest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchBottomUp**, then the GCD I/O space map is searched from the lowest address up to *BaseAddress* looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateMaxAddressSearchTopDown**, then the GCD I/O space map is searched from *BaseAddress* down to the lowest address looking for unallocated I/O ranges of *Length* bytes beginning on a boundary specified by *Alignment* that matches *GcdIoType*.

If *GcdAllocateType* is **EfiGcdAllocateAddress**, then the GCD I/O space map is checked to see if the I/O range starting at *BaseAddress* for *Length* bytes is of type *GcdIoType*, unallocated, and begins on a the boundary specified by *Alignment*.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If *BaseAddress* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *ImageHandle* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdIoType* is not **EfiGcdIoTypeNonExistent**, **EfiGcdIoTypeReserved**, or **EfiGcdIoTypeIo**, then **EFI_INVALID_PARAMETER** is returned.

If *GcdAlocateType* is less than zero, or *GcdAllocateType* is greater than or equal to *EfiGcdMaxAllocateType* then **EFI_INVALID_PARAMETER** is returned.

If there are not enough system resources available to allocate the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was allocated from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*GcdAllocateType* is invalid.
EFI_INVALID_PARAMETER	*GcdIoType* is invalid.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_INVALID_PARAMETER	*BaseAddress* is **NULL**.
EFI_INVALID_PARAMETER	*ImageHandle* is **NULL**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to allocate the I/O resource from the global coherency domain of the processor.
EFI_NOT_FOUND	The I/O resource request could not be satisfied.

FreeIoSpace()

Summary

This service frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FREE_IO_SPACE) (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length
  );
```

Parameters

BaseAddress

The physical address that is the start address of the I/O resource being freed. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource range that is being freed.

Description

The **FreeIoSpace()** function converts the I/O range specified by *BaseAddress* and *Length* from the allocated state to the unallocated state. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were not allocated on previous calls to **AllocateIoSpace()**, then **EFI_NOT_FOUND** is returned.

If there are not enough system resources available to free the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was freed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.

EFI_NOT_FOUND	The I/O resource range specified by *BaseAddress* and *Length* was not allocated with previous calls to **AllocateIoSpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to free the I/O resource from the global coherency domain of the processor.

RemoveIoSpace()

Summary

This service removes reserved I/O, or I/O resources from the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_REMOVE_IO_SPACE) (
  IN EFI_PHYSICAL_ADDRESS   BaseAddress,
  IN UINT64                 Length
  );
```

Parameters

BaseAddress

A pointer to a physical address that is the start address of the I/O resource being removed. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Length

The size in bytes of the I/O resource that is being removed.

Description

The **RemoveIoSpace()** function converts the I/O range specified by *BaseAddress* and *Length* to the I/O type **EfiGcdIoTypeNonExistent**. If this conversion is successful, then **EFI_SUCCESS** is returned.

If the GCD I/O space map contains adjacent I/O regions that only differ in their base address and length fields, then those adjacent I/O regions must be merged into a single I/O descriptor.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the I/O range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were not added to the GCD I/O space map with previous calls to **AddIoSpace()**, then **EFI_NOT_FOUND** is returned.

If one or more bytes of the I/O range specified by *BaseAddress* and *Length* were allocated from the GCD I/O space map with previous calls to **AllocateIoSpace()**, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to remove the I/O range, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O resource was removed from the global coherency domain of the processor.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the I/O resource range specified by *BaseAddress* and *Length*.
EFI_NOT_FOUND	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* was not added with previous calls to **AddIoSpace()**.
EFI_ACCESS_DENIED	One or more bytes of the I/O resource range specified by *BaseAddress* and *Length* has been allocated with **AllocateIoSpace()**.
EFI_OUT_OF_RESOURCES	There are not enough system resources to remove the I/O resource from the global coherency domain of the processor.

GetIoSpaceDescriptor()

Summary

This service retrieves the descriptor for an I/O region containing a specified address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_IO_SPACE_DESCRIPTOR) (
  IN  EFI_PHYSICAL_ADDRESS         BaseAddress,
  OUT EFI_GCD_IO_SPACE_DESCRIPTOR  *Descriptor
  );
```

Parameters

BaseAddress

The physical address that is the start address of an I/O region. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Descriptor

A pointer to a caller allocated descriptor. On return, the descriptor describes the I/O region containing *BaseAddress*. Type **EFI_GCD_IO_SPACE_DESCRIPTOR** is defined in "Related Definitions" below.

Description

The **GetIoSpaceDescriptor()** function retrieves the descriptor for the I/O region that contains the address specified by *BaseAddress*. If an I/O region containing *BaseAddress* is found, then the descriptor for that I/O region is returned in the caller allocated structure *Descriptor*, and **EFI_SUCCESS** is returned.

If *Descriptor* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If an I/O region containing *BaseAddress* is not present in the GCD I/O space map, then **EFI_NOT_FOUND** is returned.

Related Definitions

```
//*******************************************************
// EFI_GCD_IO_SPACE_DESCRIPTOR
//*******************************************************
typedef struct {
  EFI_PHYSICAL_ADDRESS  BaseAddress;
  UINT64                Length;
  EFI_GCD_IO_TYPE       GcdIoType;
  EFI_HANDLE            ImageHandle;
  EFI_HANDLE            DeviceHandle;
} EFI_GCD_IO_SPACE_DESCRIPTOR;
```

Parameters

BaseAddress

Physical address of the first byte in the I/O region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

Number of bytes in the I/O region.

GcdIoType

Type of the I/O region. Type **EFI_GCD_IO_TYPE** is defined in the **AddIoSpace()** function description.

ImageHandle

The image handle of the agent that allocated the I/O resource described by *PhysicalStart* and *NumberOfBytes*. If this field is **NULL**, then the I/O resource is not currently allocated. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DeviceHandle

The device handle for which the I/O resource has been allocated. If *ImageHandle* is **NULL**, then the I/O resource is not currently allocated. If this field is **NULL**, then the I/O resource is not associated with a device that is described by a device handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The descriptor for the I/O resource region containing *BaseAddress* was returned in *Descriptor*.
EFI_INVALID_PARAMETER	*Descriptor* is **NULL**.
EFI_NOT_FOUND	An I/O resource range containing *BaseAddress* was not found.

GetIoSpaceMap()

Summary

Returns a map of the I/O resources in the global coherency domain of the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_IO_SPACE_MAP) (
  OUT UINTN                          *NumberOfDescriptors,
  OUT EFI_GCD_IO_SPACE_DESCRIPTOR    **IoSpaceMap
  );
```

Parameters

NumberOfDescriptors

A pointer to number of descriptors returned in the *IoSpaceMap* buffer. This parameter is ignored on input, and is set to the number of descriptors in the *IoSpaceMap* buffer on output.

IoSpaceMap

A pointer to the array of **EFI_GCD_IO_SPACE_DESCRIPTOR**s. Type **EFI_GCD_IO_SPACE_DESCRIPTOR** is defined in **GetIoSpaceDescriptor()**. This buffer is allocated with **AllocatePool()**, so it is the caller's responsibility to free this buffer with a call to **FreePool()**. The number of descriptors in *IoSpaceMap* is returned in *NumberOfDescriptors*.

Description

The **GetIoSpaceMap()** function retrieves the entire GCD I/O space map. If there are no errors retrieving the GCD I/O space map, then the number of descriptors in the GCD I/O space map is returned in *NumberOfDescriptors*, the array of descriptors from the GCD I/O space map is allocated with **AllocatePool()**, the descriptors are transferred into *IoSpaceMap*, and **EFI_SUCCESS** is returned.

If *NumberOfDescriptors* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *IoSpaceMap* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If there are not enough resources to allocate *IoSpaceMap*, then **EFI_OUT_OF_RESOURCES** is returned.

Status Codes Returned

EFI_SUCCESS	The I/O space map was returned in the *IoSpaceMap* buffer, and the number of descriptors in *IoSpaceMap* was returned in *NumberOfDescriptors*.
EFI_INVALID_PARAMETER	*NumberOfDescriptors* is **NULL**.
EFI_INVALID_PARAMETER	*IoSpaceMap* is **NULL**.

EFI_OUT_OF_RESOURCES	There are not enough resources to allocate `IoSpaceMap`.

7.3 Dispatcher Services

The functions that make up the Dispatcher Services are used during preboot to schedule drivers for execution. A driver may optionally have the Schedule On Request (SOR) flag set in the driver's dependency expression. Drivers with this bit set will not be loaded and invoked until they are explicitly requested to do so. Files loaded from firmware volumes may be placed in the untrusted state by the Security Architectural Protocol. The services in this section provide this ability to clear the SOR flag in a DXE driver's dependency expression and the ability to promote a file from a firmware volume from the untrusted to the trusted state. Table 12 lists the Dispatcher Services.

Table 12. Dispatcher Boot Type Services

Name	Description
Dispatch	Loads and executed DXE drivers from firmware volumes.
Schedule	Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.
Trust	Changes the state of a file stored in a firmware volume from the untrusted state to the trusted state.
ProcessFirmwareVolume	Creates a firmware volume handle for a firmware volume that is present in system memory.

Dispatch()

Summary

Loads and executes DXE drivers from firmware volumes.

Prototype

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

Description

The **Dispatch()** function searches for DXE drivers in firmware volumes that have been installed since the last time the **Dispatch()** service was called. It then evaluates the dependency expressions of all the DXE drivers and loads and executes those DXE drivers whose dependency expression evaluate to **TRUE**. This service must interact with the Security Architectural Protocol to authenticate DXE drivers before they are executed. This process is continued until no more DXE drivers can be executed. If one or more DXE drivers are executed, then **EFI_SUCCESS** is returned. If no DXE drivers are executed, **EFI_NOT_FOUND** is returned.

If an attempt is made to invoke the DXE Dispatcher recursively, then no action is performed by the **Dispatch()** service, and **EFI_ALREADY_STARTED** is returned. In this case, because the DXE Dispatcher is already running, it is not necessary to invoke it again. All the DXE drivers that can be dispatched will be dispatched.

Status Codes Returned

EFI_SUCCESS	One or more DXE driver were dispatched.
EFI_NOT_FOUND	No DXE drivers were dispatched.
EFI_ALREADY_STARTED	An attempt is being made to start the DXE Dispatcher recursively. Thus no action was taken.

Schedule()

Summary

Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SCHEDULE) (
  IN EFI_HANDLE          FirmwareVolumeHandle,
  IN CONST EFI_GUID      *FileName
  );
```

Parameters

FirmwareVolumeHandle

The handle of the firmware volume that contains the file specified by *FileName*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

FileName

A pointer to the name of the file in a firmware volume. This is the file that should have its SOR bit cleared. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **Schedule()** function searches the dispatcher queues for the driver specified by *FirmwareVolumeHandle* and *FileName*. If this driver cannot be found, then **EFI_NOT_FOUND** is returned. If the driver is found, and its Schedule On Request (SOR) flag is not set in its dependency expression, then **EFI_NOT_FOUND** is returned. If the driver is found, and its SOR bit is set in its dependency expression, then the SOR flag is cleared, and **EFI_SUCCESS** is returned. After the SOR flag is cleared, the driver will be dispatched if the remaining portions of its dependency expression are satisfied. This service does not automatically invoke the DXE Dispatcher. Instead, the **Dispatch()** service must be used to invoke the DXE Dispatcher.

Status Codes Returned

EFI_SUCCESS	The DXE driver was found and its SOR bit was cleared.
EFI_NOT_FOUND	The DXE driver does not exist, or the DXE driver exists and its SOR bit is not set.

Trust()

Summary

Promotes a file stored in a firmware volume from the untrusted to the trusted state. Only the Security Architectural Protocol can place a file in the untrusted state. A platform specific component may choose to use this service to promote a previously untrusted file to the trusted state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TRUST) (
  IN EFI_HANDLE             FirmwareVolumeHandle,
  IN CONST EFI_GUID         *FileName
  );
```

Parameters

FirmwareVolumeHandle

The handle of the firmware volume that contains the file specified by *FileName*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

FileName

A pointer to the name of the file in a firmware volume. This is the file that should be promoted from the untrusted state to the trusted state. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **Trust()** function promotes the file specified by *FirmwareVolumeHandle* and *FileName* from the untrusted state to the trusted state. If this file is not found in the queue of untrusted files, then **EFI_NOT_FOUND** is returned. If the driver is found, and its state is changed to trusted and **EFI_SUCCESS** is returned. This service does not automatically invoke the DXE Dispatcher. Instead, the **Dispatch()** service must be used to invoke the DXE Dispatcher.

Status Codes Returned

EFI_SUCCESS	The file was found in the untrusted state, and it was promoted to the trusted state.
EFI_NOT_FOUND	The file was not found in the untrusted state.

ProcessFirmwareVolume()

Summary

Creates a firmware volume handle for a firmware volume that is present in system memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PROCESS_FIRMWARE_VOLUME) (
  IN  CONST VOID  *FirmwareVolumeHeader,
  IN  UINTN       Size,
  OUT EFI_HANDLE  *FirmwareVolumeHandle
  );
```

Parameters

FirmwareVolumeHeader

A pointer to the header of the firmware volume.

Size

The size, in bytes, of the firmware volume.

FirmwareVolumeHandle

On output, a pointer to the created handle. This service will install the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** for the of the firmware volume that is described by *FirmwareVolumeHeader* and *Size*. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The **ProcessFirmwareVolume()** function examines the contents of the buffer specified by *FirmwareVolumeHeader* and *Size*. If the buffer contains a valid firmware volume, then a new handle is created, and the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and a memory-mapped **EFI_DEVICE_PATH_PROTOCOL** are installed onto the new handle. The new handle is returned in *FirmwareVolumeHandle*.

Status Codes Returned

EFI_SUCCESS	The **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** were installed onto *FirmwareVolumeHandle* for the firmware volume described by *FirmwareVolumeHeader* and *Size*.
EFI_VOLUME_CORRUPTED	The firmware volume described by *FirmwareVolumeHeader* and *Size* is corrupted.

EFI_OUT_OF_RESOURCES	There are not enough system resources available to produce the **EFI_FIRMWARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** for the firmware volume described by *FirmwareVolumeHeader* and *Size*.

8 Protocols - Device Path Protocol

8.1 Introduction

This section adds two device path node types that describe files stored in firmware volumes:

- Firmware File Media Device Path
- Firmware Volume Media Device Path

These device path nodes are used by a DXE-aware updated UEFI Boot Service **LoadImage()** to load UEFI images from firmware volumes. This new capability is used by the DXE Dispatcher to load DXE drivers from firmware volumes.

8.2 Firmware Volume Media Device Path

This type is used by systems implementing the PI architecture specifications to describe a firmware volume.

Table 13. Firmware Volume Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path
Sub-Type	1	1	Sub Type 7 – Firmware Volume Media Device Path
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Firmware Volume Name	4	16	Firmware volume name. Type EFI_GUID.

Table 14. Firmware Volume Device Node Text Representation

Device Node Type/Subtype/Other	Description
Type: 4 (Media Device Path) Sub-Type: 7 (Firmware Volume)	**Fv(fv-guid)** The **fv-guid** is a GUID.

8.3 Firmware File Media Device Path

This type is used by systems implementing the PI architecture specifications to describe a firmware file in a firmware volume.

Table 15. Firmware File Media Device Path

Mnemonic	Byte Offset	Byte Length	Description
Type	0	1	Type 4 – Media Device Path
Sub-Type	1	1	Sub Type 6 – Firmware File Media Device Path
Length	2	2	Length of this structure in bytes. Length is 20 bytes.
Firmware File Name	4	16	Firmware file name. Type EFI_GUID.

Table 16. Firmware Volume File Device Node Text Representation

Device Node Type/Subtype/Other	Description
Type: 4 (Media Device Path) Sub-Type: 6 (Firmware File)	**`FvFile(fvfile-guid)`** The **`fvfile-guid`** is a GUID.

9 DXE Foundation

9.1 Introduction

The DXE Foundation is designed to be completely portable with no processor, chipset, or platform dependencies. This lack of dependencies is accomplished by designing in several features:

- The DXE Foundation depends only upon a HOB list for its initial state.

  This means that the DXE Foundation does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE Foundation.
- The DXE Foundation does not contain any hard-coded addresses.

  This means that the DXE Foundation can be loaded anywhere in physical memory, and it can function correctly no matter where physical memory or where Firmware Volumes (FVs) are located in the processor's physical address space.
- The DXE Foundation does not contain any processor-specific, chipset-specific, or platform-specific information.

  Instead, the DXE Foundation is abstracted from the system hardware through a set of DXE Architectural Protocol interfaces. These architectural protocol interfaces are produced by a set of DXE drivers that are invoked by the DXE Dispatcher.

The DXE Foundation must produce the UEFI System Table and its associated set of UEFI Boot Services and UEFI Runtime Services. The DXE Foundation also contains the DXE Dispatcher whose main purpose is to discover and execute DXE drivers stored in FVs. The execution order of DXE drivers is determined by a combination of the optional a priori file and the set of dependency expressions that are associated with the DXE drivers. The FV file format allows dependency expressions to be packaged with the executable DXE driver image. DXE drivers utilize a PE/COFF image format, so the DXE Dispatcher must also contain a PE/COFF loader to load and execute DXE drivers.

The **`GetMemoryMap()`** implementation must include all GCD map entries of types EfiGcdMemoryTypeReserved and EfiGcdMemoryTypeMemoryMappedIo into the UEFI memory map.

9.2 Hand-Off Block (HOB) List

The Hand-Off Block (HOB) list contains all the information that the DXE Foundation requires to produce its memory-based services. The HOB list contains the following:

- Information on the boot mode and the memory that was allocated in the previous phase.
- A description of the system memory that was initialized by the previous phase along with information about the firmware devices that were discovered in the previous phase.

The firmware device information includes the system memory locations of the firmware devices and system memory locations of the firmware volumes that are contained within those firmware devices. The firmware volumes may contain DXE drivers, and the DXE Dispatcher is responsible for loading and executing the DXE drivers that are discovered in those firmware volumes.

The I/O resources and memory-mapped I/O resources that were discovered in the previous phase.

The HOB list must be treated as a read-only data structure. It conveys the state of the system at the time the DXE Foundation is started. The DXE Foundation and DXE drivers should never modify the contents of the HOB list.

Figure 4 shows an example HOB list. The first HOB list entry is always the Phase Handoff Information Table (PHIT) HOB that contains the boot mode and a description of the memory regions used by the previous phase. The rest of the HOB list entries can appear in any order. This example shows the various HOB types that are supported. The most important ones to the DXE Foundation are the HOBs that describe system memory and the firmware volumes. A HOB list is terminated by an end of list HOB. There is one additional HOB type that is not shown. This is a GUIDed HOB that allows a module from the previous phase to pass private data to a DXE driver. Only the DXE driver that recognizes the GUID value in the GUIDed HOB will be able to understand the data in the GUIDed HOB. The DXE Foundation does not consume any GUIDed HOBs. The HOB entries are all designed to be position independent. This allows the DXE Foundation to relocate the HOB list to a different location if the DXE Foundation does not like where the previous phase placed the HOB list in memory.

See "HOB Translations" on page 93 for more information on HOB types.

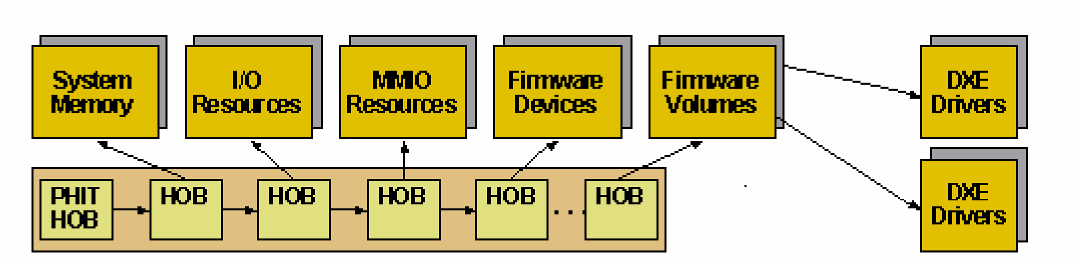


Figure 4. HOB List

9.3 DXE Foundation Data Structures

The DXE Foundation produces the UEFI System Table, and the UEFI System Table is consumed by every DXE driver and executable image invoked by the DXE Dispatcher and BDS. It contains all the information required for these components to utilize the services provided by the DXE Foundation and the services provided by any previously loaded DXE driver. Figure 5 shows the various components that are available through the UEFI System Table.

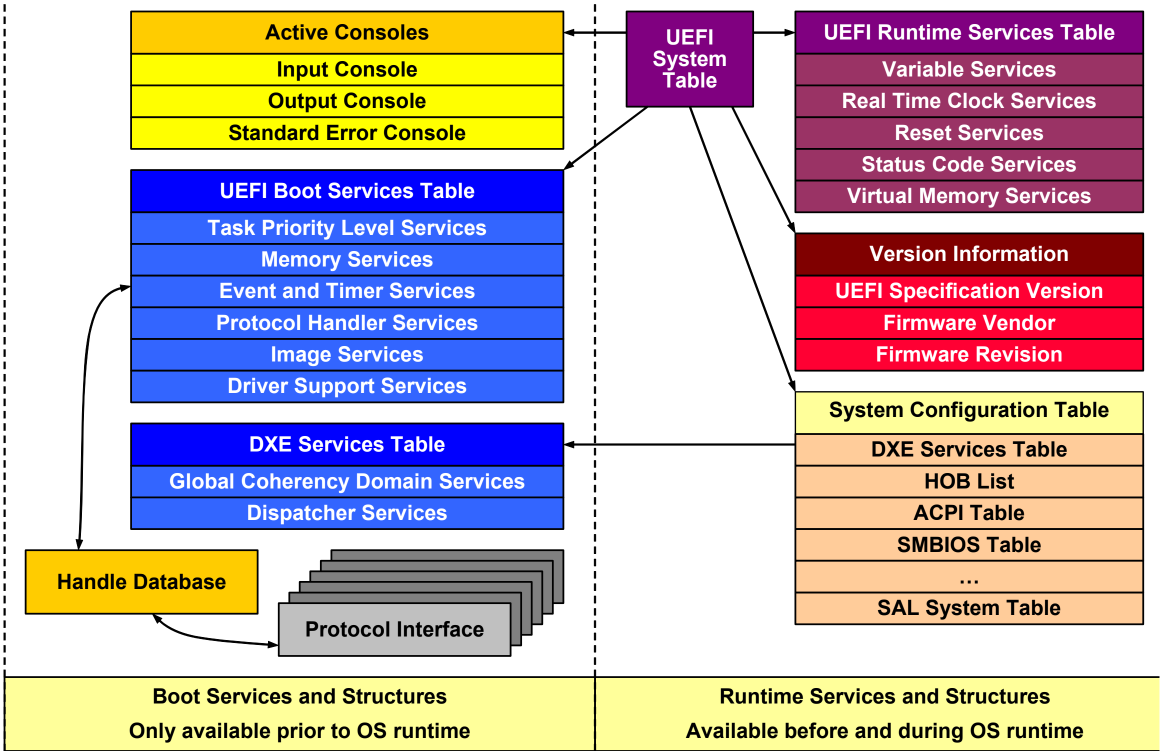


Figure 5. UEFI System Table and Related Components

The DXE Foundation produces the UEFI Boot Services, UEFI Runtime Services, and DXE Services with the aide of the DXE Architectural Protocols. The UEFI System Table also provides access to all the active console devices in the platform and the set of UEFI Configuration Tables. The UEFI Configuration Tables are an extensible list of tables that describe the configuration of the platform. Today, this includes pointers to tables such as DXE Services, the HOB list, ACPI table, SMBIOS table, and the SAL System Table. This list may be expanded in the future as new table types are defined. Also, through the use of the Protocol Handle Services in the UEFI Boot Services Table, any executable image can access the handle database and any of the protocol interfaces that have been registered by DXE drivers.

When the transition to the OS runtime is performed, the handle database, active consoles, UEFI Boot Services, DXE Services, and services provided by boot service DXE drivers are terminated. This frees up memory for use by the OS. This only leaves the UEFI System Table, UEFI Runtime

Services Table, and the UEFI Configuration Tables available in the OS runtime environment. There is also the option of converting all of the UEFI Runtime Services from a physical address space to an OS-specific virtual address space. This address space conversion may be performed only once.

9.4 Required DXE Foundation Components

Figure 6 shows the components that a DXE Foundation must contain. A detailed description of these component follows.

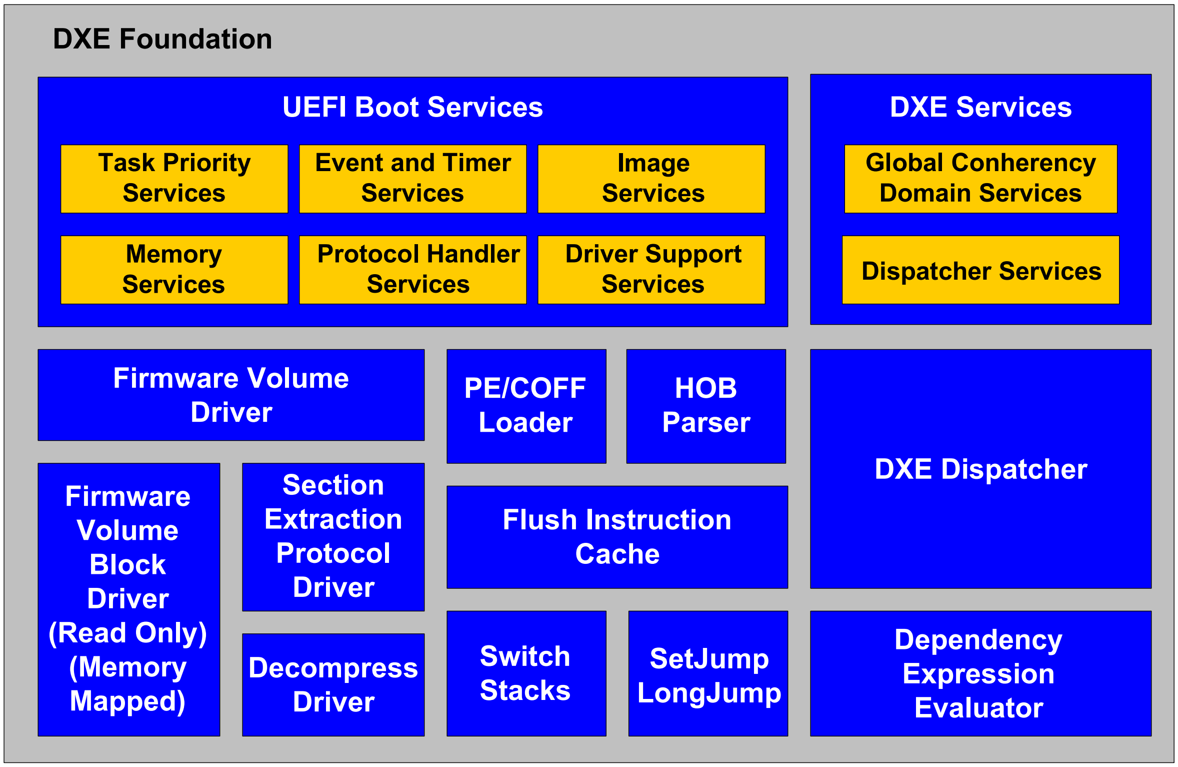


Figure 6. DXE Foundation Components

A DXE Foundation must have the following components:

- An implementation of the UEFI Boot Services. UEFI Boot Services Dependencies describes which services can be made available based on the HOB list alone and which services depend on the presence of architectural protocols.
- An implementation of the DXE Services. DXE Services Dependencies describes which services can be made available based on the HOB list alone and which services depend on the presence of architectural protocols.
- A HOB Parser that consumes the HOB list specified by `HobStart` and initializes the UEFI memory map, GCD memory space map, and GCD I/O space map. See section if for details on the translation from HOBs to the maps maintained by the DXE Foundation

- An implementation if a DXE Dispatcher that includes a dependency expression evaluator. See "DXE Dispatcher" on page 97 for a detailed description of this component.
- A Firmware Volume driver that produces the **EFI_FIRMWARE_VOLUME2_PROTOCOL** for every firmware volume described in the HOB list. This component is used by the DXE Dispatcher to search for *a priori* files and DXE drivers in firmware volumes. See the *Platform Initialization Specification*, Volume 3, for the definition of the Firmware Volume Protocol.
- An instance of the **EFI_DECOMPRESS_PROTOCOL**. See the UEFI 2.0 specification for the detailed requirements for this component. This component is required by the DXE Dispatcher to read compressed sections from DXE drivers stored in firmware volumes. It is expected that most DXE drivers will utilize compressed sections to reduce the size of firmware volumes.
- The DXE Dispatcher uses the Boot Service **StartImage()** to invoke a DXE driver. The Boot Services **StartImage()** and **Exit()** work together to hand control to a DXE driver and return control to the DXE Foundation. Since the Boot Service **Exit()** can be called for anywhere inside a DXE driver, the Boot Service **Exit()** is required to rebalance the stack, so it is in the same state it was in when the Boot Service **Start()** was called. This is typically implemented using the processor-specific functions called **SetJump()** and **LongJump()**. Since the DXE Foundation must use the Boot Services **StartImage()** and **Exit()** to invoke DXE drivers, the routines **SetJump()** and **LongJump()** are required by the DXE Foundation.
- A PE/COFF loader that supports PE32+ image types. This PE/COFF loader is used to implement the UEFI Boot Service **LoadImage()**. The DXE Dispatcher uses the Boot Service **LoadImage()** to load DXE drivers into system memory. If the processor that the DXE Foundation is compiled for requires an instruction cache when an image is loaded into system memory, then an instruction cache flush routine is also required in the DXE Foundation.
- The phase that executed prior to DXE will initialize a stack for the DXE Foundation to use. This stack is described in the HOB list. If the size of this stack does not meet the DXE Foundation's minimum stack size requirement or the stack is not located in memory region that is suitable to the DXE Foundation, then the DXE Foundation will have to allocate a new stack that does meet the minimum size and location requirements. As a result, the DXE Foundation must contain a stack switching routine for the processor type that the DXE Foundation is compiled.

9.5 Handing Control to DXE Dispatcher

The DXE Foundation must complete the following tasks before handing control to the DXE Dispatcher. The order that these tasks are performed is implementation dependent.

- Use the HOB list to initialize the GCD memory space map, the GCD I/O space map, and UEFI memory map.
- Allocate the UEFI Boot Services Table from **EFI_BOOT_SERVICES_MEMORY** and initialize the services that only require system memory to function correctly. The remaining UEFI Boot Services must be filled in with a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the DXE Services Table from **EFI_BOOT_SERVICES_MEMORY** and initialize the services that only require system memory to function correctly. The remaining DXE Services must be filled in with a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the UEFI Runtime Services Table from **EFI_RUNTIME_SERVICES_MEMORY** and initialize all the services to a service that returns **EFI_NOT_AVAILABLE_YET**.
- Allocate the UEFI System Table from **EFI_RUNTIME_SERVICES_MEMORY** and initialize all the fields.
- Build an image handle and **EFI_LOADED_IMAGE_PROTOCOL** instance for the DXE Foundation itself and add it to the handle database.
- If the HOB list is not in a suitable location in memory, then relocate the HOB list to a more suitable location.
- Add the DXE Services Table to the UEFI Configuration Table.
- Add the HOB list to the UEFI Configuration Table.
- Create a notification event for each of the DXE Architectural Protocols. These events will be signaled when a DXE driver installs a DXE Architectural Protocol in the handle database. The DXE Foundation must have a notification function associated with each of these events, so the full complement of UEFI Boot Services, UEFI Runtime Services, and DXE Services can be produced. Each of the notification functions should compute the 32-bit CRC of the UEFI Boot Services Table, UEFI Runtime Services Table, and the DXE Services Table if the **CalculateCrc32()** Boot Services is available.
- Initialize the Decompress Protocol driver that must be available before the DXE Dispatcher can process compressed sections.
- Produce firmware volume handles for the one or more firmware volumes that are described in the HOB list.

Once these tasks have been completed, the DXE Foundation is ready to load and execute DXE drivers stored in firmware volumes. This execution is done by handing control to the DXE Dispatcher. Once the DXE Dispatcher has finished dispatching all the DXE drivers that it can, control is then passed to the BDS Architectural Protocol. If for some reason, any of the DXE Architectural Protocols have not been produced by the DXE drivers, then the system is in an unusable state and the DXE Foundation must halt. Otherwise, control is handed to the BDS Architectural Protocol. The BDS Architectural Protocol is responsible for transferring control to an operating system or system utility.

9.6 DXE Foundation Entry Point

9.6.1 DXE_ENTRY_POINT

The only parameter passed to the DXE Foundation is a pointer to the HOB list. The DXE Foundation and all the DXE drivers must treat the HOB list as read-only data.

The function **DXE_ENTRY_POINT** is the main entry point to the DXE Foundation.

DXE_ENTRY_POINT

Summary

This function is the main entry point to the DXE Foundation.

Prototype

```
typedef
VOID
(EFIAPI *DXE_ENTRY_POINT) (
  IN  CONST VOID   *HobStart
  );
```

Parameters

HobStart

A pointer to the HOB list.

Description

This function is the entry point to the DXE Foundation. The PEI phase, which executes just before DXE, is responsible for loading and invoking the DXE Foundation in system memory. The only parameter that is passed to the DXE Foundation is *HobStart*. This parameter is a pointer to the HOB list that describes the system state at the hand-off to the DXE Foundation. At a minimum, this system state must include the following:

- PHIT HOB
- CPU HOB
- Description of system memory
- Description of one or more firmware volumes

The DXE Foundation is also guaranteed that only one processor is running and that the processor is running with interrupts disabled. The implementation of the DXE Foundation must not make any assumptions about where the DXE Foundation will be loaded or where the stack is located. In general, the DXE Foundation should make as few assumptions about the state of the system as possible. This lack of assumptions will allow the DXE Foundation to be portable to the widest variety of system architectures.

9.7 Dependencies

9.7.1 UEFI Boot Services Dependencies

Table 17 lists all the UEFI Boot Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 17. Boot Service Dependencies

Name	Dependency
CreateEvent	HOB list
CloseEvent	HOB list
SignalEvent	HOB list
WaitForEvent	HOB list
CheckEvent	HOB list
SetTimer	Timer Architectural Protocol
RaiseTPL	CPU Architectural Protocol
RestoreTPL	CPU Architectural Protocol
AllocatePages	HOB list
FreePages	HOB list
GetMemoryMap	HOB list and GetMemorySpaceMap
AllocatePool	HOB list
FreePool	HOB list
InstallProtocolInterface	HOB list
UninstallProtocolInterface	HOB list
ReinstallProtocolInterface	HOB list
RegisterProtocolNotify	HOB list
LocateHandle	HOB list
HandleProtocol	HOB list
LocateDevicePath	HOB list
OpenProtocol	HOB list
CloseProtocol	HOB list
OpenProtocolInformation	HOB list
ConnectController	HOB list
DisconnectController	HOB list
ProtocolsPerHandle	HOB list
LocateHandleBuffer	HOB list
LocateProtocol	HOB list
InstallMultipleProtocolInterfaces	HOB list
UninstallMultipleProtocolInterfaces	HOB list
LoadImage	HOB list
StartImage	HOB list
UnloadImage	HOB list
EFI_IMAGE_ENTRY_POINT	HOB list
Exit	HOB list
ExitBootServices	HOB list
SetWatchDogTimer	Watchdog Architectural Protocol

Name	Dependency
Stall	Metronome Architectural Protocol Timer Architectural Protocol
CopyMem	HOB list
SetMem	HOB list
GetMemory Map	HOB list
GetMemorySpaceMap	GCD Service
GetNextMonotonicCount	Monotonic Counter Architectural Protocol
InstallConfigurationTable	HOB list
CalculateCrc32	Runtime Architectural Protocol

9.7.1.1 SetTimer()

When the DXE Foundation is notified that the **EFI_TIMER_ARCH_PROTOCOL** has been installed, then the Boot Service **SetTimer()** can be made available. The DXE Foundation can use the services of the **EFI_TIMER_ARCH_PROTOCOL** to initialize and hook a heartbeat timer interrupt for the DXE Foundation. The DXE Foundation can use this heartbeat timer interrupt to determine when to signal on-shot and periodic timer events. This service may be called before the **EFI_TIMER_ARCH_PROTOCOL** is installed. However, since a heartbeat timer is not running yet, time is essentially frozen at zero. This means that no periodic or one-shot timer events will fire until the **EFI_TIMER_ARCH_PROTOCOL** is installed.

9.7.1.2 RaiseTPL()

The DXE Foundation must produce the Boot Service **RaiseTPL()** when the memory-based services are initialized. The DXE Foundation is guaranteed to be handed control of the platform with interrupts disabled. Until the DXE Foundation installs a heartbeat timer interrupt and turns on interrupts, this Boot Service can be a very simple function that always succeeds. When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the full version of the Boot Service **RaiseTPL()** can be made available. When an attempt is made to raise the TPL level to **EFI_TPL_HIGH_LEVEL** or higher, then the DXE Foundation should use the services of the **EFI_CPU_ARCH_PROTOCOL** to disable interrupts.

9.7.1.3 RestoreTPL()

The DXE Foundation must produce the Boot Service **RestoreTPL()** when the memory-based services are initialized. The DXE Foundation is guaranteed to be handed control of the platform with interrupts disabled. Until the DXE Foundation installs a heartbeat timer interrupt and turns on interrupts, this Boot Service can be a very simple function that always succeeds. When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the full version of the Boot Service **RestoreTPL()** can be made available. When an attempt is made to restore the TPL level to level below **EFI_TPL_HIGH_LEVEL**, then the DXE Foundation should use the services of the **EFI_CPU_ARCH_PROTOCOL** to enable interrupts.

9.7.1.4 SetWatchdogTimer()

When the DXE Foundation is notified that the **EFI_WATCHDOG_ARCH_PROTOCOL** has been installed, then the Boot Service **SetWatchdogTimer()** can be made available. The DXE

Foundation can use the services of the **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** to set the amount of time before the system's watchdog timer will expire.

9.7.1.5 Stall()

When the DXE Foundation is notified that the **EFI_METRONOME_ARCH_PROTOCOL** has been installed, the DXE Foundation can produce a very simple version of the Boot Service **Stall()**. The granularity of the Boot Service **Stall()** will be based on the period of the **EFI_METRONOME_ARCH_PROTOCOL**.

When the DXE Foundation is notified that the **EFI_TIMER_ARCH_PROTOCOL** has been installed, the DXE Foundation can possibly produce a more accurate version of the Boot Service **Stall()**. This all depends on the periods of the **EFI_METRONOME_ARCH_PROTOCOL** and the period of the **EFI_TIMER_ARCH_PROTOCOL**. The DXE Foundation should produce the Boot Service **Stall()** using the most accurate time base available.

9.7.1.6 GetNextMonotonicCount()

When the DXE Foundation is notified that the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** has been installed, then the Boot Service **GetNextMonotonicCount()** is available. The DXE driver that produces the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** is responsible for directly updating the *GetNextMonotonicCount* field of the UEFI Boot Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Boot Services Table.

9.7.1.7 CalculateCrc32()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Boot Service **CalculateCrc32()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *CalculateCrc32* field of the UEFI Boot Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Boot Services Table.

9.7.1.8 GetMemoryMap()

The **GetMemoryMap()** implementation must include into the UEFI memory map all GCD map entries of types **EfiGcdMemoryTypeReserved** and **EfiPersistentMemory**, and all GCD map entries of type **EfiGcdMemoryTypeMemoryMappedIo** that have **EFI_MEMORY_RUNTIME** attribute set.

9.7.2 UEFI Runtime Services Dependencies

Table 18 lists all the UEFI Runtime Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 18. Runtime Service Dependencies

Name	Dependency
GetVariable	Variable Architectural Protocol
GetNextVariableName	Variable Architectural Protocol
SetVariable	Variable Architectural Protocol / Variable Write Architectural Protocol
GetTime	Real Time Clock Architectural Protocol
SetTime	Real Time Clock Architectural Protocol
GetWakeupTime	Real Time Clock Architectural Protocol
SetWakeupTime	Real Time Clock Architectural Protocol
SetVirtualAddressMap	Runtime Architectural Protocol
ConvertPointer	Runtime Architectural Protocol
ResetSystem	Reset Architectural Protocol
GetNextHighMonotonicCount	Monotonic Counter Architectural Protocol
UpdateCapsule	Capsule Header Protocol
QueryCapsuleCapabilities	Capsule Header Protocol

9.7.2.1 GetVariable()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetVariable()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *GetVariable* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.2 GetNextVariableName()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetNextVariableName()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *GetNextVariableName* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.3 SetVariable()

When the DXE Foundation is notified that the **EFI_VARIABLE_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetVariable()** is available. The DXE driver that produces the **EFI_VARIABLE_ARCH_PROTOCOL** is responsible for directly updating the *SetVariable* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table. The **EFI_VARIABLE_ARCH_PROTOCOL** is required to provide read-only access to all environment variables and write access to volatile environment variables.

When the DXE Foundation is notified that the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** has been installed, then write access to nonvolatile environment variables will also be available. If an attempt is made to call this function for a nonvolatile environment variable prior to the installation of **EFI_VARIABLE_WRITE_ARCH_PROTOCOL**, then **EFI_NOT_AVAILABLE_YET** must be returned. This allows for flexibility in the design and implementation of the variables services in a platform such that read access to environment variables can be provided very early in the DXE phase and write access to nonvolatile environment variables can be provided later in the DXE phase.

9.7.2.4 GetTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *GetTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.5 SetTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *SetTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.6 GetWakeupTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetWakeupTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *GetWakeupTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.7 SetWakeupTime()

When the DXE Foundation is notified that the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetWakeupTime()** is available. The DXE driver that produces the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL** is responsible for directly updating the *SetWakeupTime* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.8 SetVirtualAddressMap()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Runtime Service **SetVirtualAddressMap()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *SetVirtualAddressMap* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.9 ConvertPointer()

When the DXE Foundation is notified that the **EFI_RUNTIME_ARCH_PROTOCOL** has been installed, then the Runtime Service **ConvertPointer()** is available. The DXE driver that produces the **EFI_RUNTIME_ARCH_PROTOCOL** is responsible for directly updating the *ConvertPointer* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.10 ResetSystem()

When the DXE Foundation is notified that the **EFI_RESET_ARCH_PROTOCOL** has been installed, then the Runtime Service **ResetSystem()** is available. The DXE driver that produces the **EFI_RESET_ARCH_PROTOCOL** is responsible for directly updating the *Reset* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.2.11 GetNextHighMonotonicCount()

When the DXE Foundation is notified that the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** has been installed, then the Runtime Service **GetNextHighMonotonicCount()** is available. The DXE driver that produces the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL** is responsible for directly updating the *GetNextHighMonotonicCount* field of the UEFI Runtime Services Table. The DXE Foundation is only responsible for updating the 32-bit CRC of the UEFI Runtime Services Table.

9.7.3 DXE Services Dependencies

Table 19 lists all the DXE Services and the components upon which each of these services depend. The topics that follow describe what responsibilities the DXE Foundation has in producing the services that depend on the presence of DXE Architectural Protocols.

Table 19. DXE Service Dependencies

Name	Dependency
AddMemorySpace	HOB list
AllocateMemorySpace	HOB list
FreeMemorySpace	HOB list
RemoveMemorySpace	HOB list
GetMemorySpaceDescriptor	CPU Architectural Protocol
SetMemorySpaceAttributes	CPU Architectural Protocol
GetMemorySpaceMap	CPU Architectural Protocol
AddIoSpace	HOB list
AllocateIoSpace	HOB list
FreeIoSpace	HOB list
RemoveIoSpace	HOB list
GetIoSpaceDescriptor	HOB list
GetIoSpaceMap	HOB list
Schedule	HOB list

9.7.3.1 GetMemorySpaceDescriptor()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **GetMemorySpaceDescriptor()** is fully functional. This function is made available when the memory-based services are initialized. However, the *Attributes* field of the **EFI_GCD_MEMORY_SPACE_DESCRIPTOR** is not valid until the **EFI_CPU_ARCH_PROTOCOL** is installed.

9.7.3.2 SetMemorySpaceAttributes()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **SetMemorySpaceAttributes()** can be made available. The DXE Foundation can then use the **SetMemoryAttributes()** service of the **EFI_CPU_ARCH_PROTOCOL** to implement the DXE Service **SetMemorySpaceAttributes()**.

9.7.3.3 GetMemorySpaceMap()

When the DXE Foundation is notified that the **EFI_CPU_ARCH_PROTOCOL** has been installed, then the DXE Service **GetMemorySpaceMap()** is fully functional. This function is made available when the memory-based services are initialized. However, the *Attributes* field of the array of **EFI_GCD_MEMORY_SPACE_DESCRIPTOR**s is not valid until the **EFI_CPU_ARCH_PROTOCOL** is installed.

9.8 HOB Translations

9.8.1 HOB Translations Overview

The following topics describe how the DXE Foundation should interpret the contents of the HOB list to initialize the GCD memory space map, GCD I/O space map, and UEFI memory map. After all of the HOBs have been parsed, the Boot Service **GetMemoryMap()** and the DXE Services **GetMemorySpaceMap()** and **GetIoSpaceMap()** should reflect the memory resources, I/O resources, and logical memory allocations described in the HOB list.

See*Volume 3* for detailed information on HOBs.

9.8.2 PHIT HOB

The Phase Handoff Information Table (PHIT) HOB describes a region of tested system memory. This region of memory contains the following:

- HOB list
- Some amount of free memory
- Potentially some logical memory allocations

The PHIT HOB is used by the DXE Foundation to determine the size of the HOB list so that the DXE Foundation can relocate the HOB list to a new location in system memory. The base address of the HOB list is passed to the DXE Foundation in the parameter *HobStart*, and the PHIT HOB field *EfiFreeMemoryBottom* specifies the end of the HOB list.

Since the PHIT HOB may contain some of amount of free memory, the DXE Foundation may use this free memory region in its early initialization phase until the full complement of UEFI memory services are available.

See *Volume 3* for the definition of this HOB type.

9.8.3 CPU HOB

The CPU HOB contains the field *SizeOfMemorySpaceMap*. This field is used to initialize the GCD memory space map. The *SizeOfMemorySpaceMap* field defines the number of address bits that the processor can use to address memory resources. The DXE Foundation must create the primordial GCD memory space map entry of type **EfiGcdMemoryTypeNonExistent** for the region from 0 to **(1 <<** *SizeOfMemorySpaceMap***)**. All future GCD memory space operations must be performed within this memory region.

The CPU HOB also contains the field *SizeOfIoSpaceMap*. This field is used to initialize the GCD I/O space map. The *SizeOfIoSpaceMap* field defines the number of address bits that the processor can use to address I/O resources. The DXE Foundation must create the primordial GCD I/O space map entry of type **EfiGcdIoTypeNonExistent** for the region from 0 to (**1 <<** *SizeOfIoSpaceMap*). All future GCD I/O space operations must be performed within this I/O region.

See *Volume 3* for the definition of this HOB type.

9.8.4 Resource Descriptor HOBs

The DXE Foundation must traverse the HOB list looking for Resource Descriptor HOBs. These HOBs describe memory and I/O resources that are visible to the processor. All of the resource ranges described in these HOBs must fall in the memory and I/O ranges initialized in the GCD maps based on the contents of the CPU HOB. The DXE Foundation will use the DXE Services **AddMemorySpace()** and **AddIoSpace()** to register these memory and I/O resources in the GCD maps.

The *Owner* field of the Resource Descriptor HOB is ignored by the DXE Foundation. The *ResourceType* field and *ResourceAttribute* fields are used to determine the GCD memory type or GCD I/O type of the resource. The table below shows this mapping. The resource range is specified by the *PhysicalStart* and *ResourceLength* fields of the Resource Descriptor HOB.

The *ResourceAttribute* field also contains the caching capabilities of memory regions. If a memory region is being added to the GCD memory space map, then the *ResourceAttribute* field will be used to initialize the supported caching capabilities. The *ResourceAttribute* field is also be used to further qualify memory regions. For example, a system memory region cannot be added to the UEFI memory map if it is read protected. However, it is legal to add a firmware device memory region that is write-protected if the firmware device is a ROM.

See *Volume 3* for the definition of this HOB type.

Table 20. Resource Descriptor HOB to GCD Type Mapping

Resource Descriptor HOB		GCD Map	
Resource Type	Attributes	Memory Type	I/O Type
System Memory	Present	Reserved	
System Memory	Present AND Initialized	Reserved	
System Memory	Present AND Initialized AND Tested	System Memory	
Memory-Mapped I/O		Memory Mapped I/O	
Firmware Device		Memory Mapped I/O	
Memory-Mapped I/O Port		Reserved	
Memory Reserved		Reserved	
I/O			I/O
I/O Reserved			Reserved

9.8.5 Firmware Volume HOBs

The DXE Foundation must traverse the HOB list for Firmware Volume HOBs. There are two types of firmware volume HOBs:

- **EFI_HOB_FIRMWARE_VOLUME**, which describes PI Firmware Volumes.
- **EFI_HOB_FIRMWARE_VOLUME2** which describes PI Firmware Volumes which came from a firmware file within a firmware volume.

When the DXE Foundation discovers a Firmware Volume HOB, the DXE Dispatcher verifies that the firmware volume has not been previously processed. Then a new handle must be created in the handle database, and the **EFI_FIRMWARE_VOLUME2_PROTOCOL** must be installed on that handle. The *BaseAddress* and *Length* fields of the Firmware Volume HOB specific the memory range that the firmware volume consumes. The DXE Service **AllocateMemorySpace()** is used to allocate the memory regions described in the Firmware Volume HOBs to the DXE Foundation. The UEFI Boot Service **InstallProtocolInterface()** is used to create new handles and install protocol interfaces.

See the *Platform Initialization Specification*, Volume 3, for code definitions concerning Hand-Off Blocks, the Firmware Volume Block Protocol and the Firmware Volume Protocol.

9.8.6 Memory Allocation HOBs

Memory Allocation HOBs describe logical memory allocations that occurred prior to the DXE phase. The DXE Foundation must parse the HOB list for this HOB type. When a HOB of this type is discovered, the GCD memory space map must be updated with a call to the DXE Service **AllocateMemorySpace()**. In addition, the UEFI memory map must be updated with logical allocation described by the *MemoryType*, *MemoryBaseAddress*, and *MemoryLength* fields of the Memory Allocation HOB.

Once the DXE Foundation has parsed all of the Memory Allocation HOBs, all of the unallocated system memory regions in the GCD memory space map must be allocated to the DXE Foundation with the DXE Service **AllocateMemorySpace()**. In addition, those same memory regions must be added to the UEFI memory map so those memory regions can be allocated and freed using the Boot Services **AllocatePages()**, **AllocatePool()**, **FreePages()**, and **FreePool()**.

See *Volume 3* for the definition of this HOB type.

9.8.7 GUID Extension HOBs

The DXE Foundation does not require any GUID Extension HOBs. Implementations of the DXE Foundation may use GUID Extension HOBs but shall not require them in order to function correctly. GUID Extension HOBs contain private or implementation-specific data that is being passed from the previous execution phase to a specific DXE driver. DXE drivers may choose to parse the HOB list for GUID Extension HOBs.

See *Volume 3* for the definition of this HOB type.

10 DXE Dispatcher

10.1 Introduction

After the DXE Foundation is initialized, control is handed to the DXE Dispatcher. The DXE Dispatcher examines every firmware volume that is present in the system. Firmware volumes are either declared by HOBs, or they are declared by DXE drivers. For the DXE Dispatcher to run, at least one firmware volume must be declared by a HOB.

The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes. Some DXE drivers may depend on the services produced by other DXE drivers, so the DXE Dispatcher is also required to execute the DXE drivers in the correct order. The DXE drivers may also be produced by a variety of different vendors, so the DXE drivers must describe the services they depend upon. The DXE dispatcher must evaluate these dependencies to determine a valid order to execute the DXE drivers. Some vendors may wish to specify a fixed execution order for some or all of the DXE drivers in a firmware volume, so the DXE dispatcher must support this requirement.

The DXE Dispatcher will ignore file types that it does not recognize.

In addition, the DXE Dispatcher must support the ability to load “emergency patch” drivers. These drivers would be added to the firmware volume to address an issue that was not know at the time the original firmware was built. These DXE drivers would be loaded just before or just after an existing DXE driver.

Finally, the DXE Dispatcher must be flexible enough to support a variety of platform specific security policies for loading and executing DXE drivers from firmware volumes. Some platforms may choose to run DXE drivers with no security checks, and others may choose to check the validity of a firmware volume before it is used, and other may choose to check the validity of every DXE driver in a firmware volume before it is executed.

10.2 Requirements

The DXE Dispatcher must meet the following requirement:

- **Support fixed execution order of DXE drivers.** This fixed execution order is specified in an *a priori* file in the firmware volume.
- **Determine DXE driver execution order based on each driver’s dependencies.** A DXE driver that is stored in a firmware volume may optionally contain a dependency expression section. This section specifies the protocols that the DXE driver requires to execute.
- **Support “emergency patch” DXE drivers.** The dependency expressions are flexible enough to describe the protocols that a DXE drivers may require. In addition, the dependency expression can declare that the DXE driver is to be loaded and executed immediately before or immediately after a different DXE driver.
- **Support platform specific security policies for DXE driver execution.** The DXE Dispatcher is required to use the services of the Security Architecture Protocol every time a firmware volume is discovered and every time a DXE driver is loaded.

When a new firmware volume is discovered, it is first authenticated with the Security Architectural Protocol. The Security Architectural Protocol provides the platform-specific policy for validating all firmware volumes. Then, a search is made for the *a priori* file. The *a priori* file has a fixed file name, and it contains the list of DXE drivers that should be loaded and executed first. There can be at most one *a priori* file per firmware volume, and it is legal to have zero *a priori* files in a firmware volume. Once the DXE drivers from the *a priori* file have been loaded and executed, the dependency expressions of the remaining DXE drivers in the firmware volumes are evaluated to determine the order that they will be loaded and executed. The *a priori* file provides a strongly ordered list of DXE drivers that are not required to use dependency expressions. The dependency expressions provide a weakly ordered execution of the remaining DXE drivers.

The DXE Dispatcher loads the image using **`LoadImage()`** with the *`FilePath`* parameter pointing ot the firmware volume from which the image is located.

Before each DXE driver is executed, it must be authenticated through the Security Architectural Protocol. The Security Architectural Protocol provides the platform-specific policy for validating all DXE drivers.

Control is transferred from the DXE Dispatcher to the BDS Architectural Protocol after the DXE drivers in the *a priori* file and all the DXE drivers whose dependency expressions evaluate to **`TRUE`** have been loaded and executed. The BDS Architectural Protocol is responsible for establishing the console devices and attempting the boot of operating systems. As the console devices are established and access to boot devices is established, additional firmware volumes may be discovered. If the BDS Architectural Protocol is unable to start a console device or gain access to a boot device, it will reinvoke the DXE Dispatcher. This will allow the DXE Dispatcher to load and execute DXE drivers from firmware volumes that have been discovered since the last time the DXE Dispatcher was invoked. Once the DXE Dispatcher has loaded and executed all the DXE drivers it can, control is once again returned to the BDS Architectural Protocol to continue the OS boot process.

10.3 The A Priori File

The *a priori* file is a special file that may be present in a firmware volume. The *a priori* file format described herein must be supported if the DXE Foundation implementation also supports 3rd party firmware volumes. The rule is that there may be at most one *a priori* file per firmware volume present in a platform. The *a priori* file has a known GUID file name, so the DXE Dispatcher can always find the *a priori* file if it is present. Every time the DXE Dispatcher discovers a firmware volume, it first looks for the *a priori* file. The *a priori* file contains the list of DXE drivers from that firmware volume that should be loaded and executed before any other DXE drivers are discovered. The DXE drivers listed in the *a priori* file are executed in the order that they appear. If any of those DXE drivers have an associated dependency expression, then those dependency expressions are ignored. The *a priori* file provides a deterministic execution order of DXE drivers. DXE drivers that are executed solely based on their dependency expression are weakly ordered. This means that the execution order is not completely deterministic between boots or between platforms. There are cases where a deterministic execution order is required. One example would be to list the DXE drivers required to debug the rest of the DXE phase in the *a priori* file. These DXE drivers that provide debug services may have been loaded much later if only their dependency expressions were considered. By loading them earlier, more of the DXE Foundation and DXE drivers can be debugged. Another example is to use the *a priori* file to eliminate the need for dependency expressions. Some embedded platforms may only require a few DXE drivers with a highly deterministic execution order. The *a priori* file can provide this ordering, and none of the DXE drivers would require dependency expressions. The dependency expressions do have some amount of size overhead, so this method may reduce the size of firmware images. The main purpose of the *a priori* file is to provide a greater degree of flexibility in the firmware design of a platform.

See the next topic for the GUID definition of the *a priori* file, which is the file name that is stored in a firmware volume.

The *a priori* file contains the file names of DXE drivers that are stored in the same firmware volume as the *a priori* file. File names in firmware volumes are GUIDs, so the *a priori* file is simply a list of byte-packed values of type **EFI_GUID**. Type **EFI_GUID** is defined in the UEFI 2.0 specification. The DXE Dispatcher reads the list of **EFI_GUID**s from the *a priori* file. Each **EFI_GUID** is used to load and execute the DXE driver with that GUID file name. If the DXE driver specified by the GUID file name is not found in the firmware volume, then the file is skipped. If the *a priori* file is not en even multiple of **EFI_GUID**s in length, then the DXE driver specified by the last **EFI_GUID** in the *a priori* file is skipped.

After all of the DXE drivers listed in the *a priori* file have been loaded and executed, the DXE Dispatcher searches the firmware volume for any additional DXE drivers and executed them according to their dependency expressions.

EFI_APRIORI_GUID

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

GUID

```
#define EFI_APRIORI_GUID  \
  {0xfc510ee7,0xffdc,0x11d4,0xbd,0x41,0x0,0x80,
  0xc7,0x3c,0x88,0x81}
```

10.4 Firmware Volume Image Files

For DXE, while processing a firmware volume, if a file of type **EFI_FV_FIRMWARE_VOLUME_IMAGE** is found, the DXE Dispatcher will check whether information about this firmware volume image file was already described in an **EFI_FIRMWARE_VOLUME_HOB2**. If it was, then the file is ignored.

Otherwise, the DXE Dispatcher will search the file for a section with the type **EFI_SECTION_DXE_DEPEX**, and if found, evaluate the expression against the presently installed entries in the protocol database.

If the file has both a dependency expression that evaluates to **TRUE** (or no dependency expression section) and the file is not already described by an **EFI_FIRMWARE_VOLUME_HOB2**, then the DXE Dispatcher will search the file for a section with the type **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**, copy its contents into memory, create a handle and install the **EFI_FIRMWMARE_VOLUME2_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on the handle.

10.5 Dependency Expressions

10.6 Dependency Expressions Overview

A DXE driver 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 DXE driver has a dependency expression, then it is stored in a *dependency section.* A DXE driver may contain additional sections for compression and security wrappers. The DXE Dispatcher can identify the DXE drivers by their file type. In addition, the DXE Dispatcher can look up the dependency expression for a DXE driver by looking for a dependency section in a DXE driver 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 be small to conserve space. In addition, they are designed to 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 DXE Dispatcher must implement an interpreter for this instruction set in order to evaluate dependency expressions. The instruction set is defined in the following topics.

See “Dependency Expression Grammar” on page 199 for an example BNF grammar for a dependency expression compiler. There are many possible methods of specifying the dependency expression for a DXE driver. Dependency Expression Grammar demonstrates one possible design for a tool that can be used to help build DXE driver images.

10.7 Dependency Expression Instruction Set

The following topics describe each of the dependency expression 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 the UEFI 2.0 specification. These GUIDs represent protocols that are produced by DXE drivers and the file names of DXE drivers 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, DXE drivers that contain dependency expressions should make an effort to keep their dependency expressions as small as possible to help reduce the size of the DXE driver.

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**.

If an instruction encoding extends beyond the end of the dependency section, then the entire dependency expression evaluates to **FALSE**.

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

Table 21 is a summary of the opcodes that are used to build dependency expressions. The following topics describe each of these instructions in detail.

Table 21. Dependency Expression Opcode Summary

Opcode	Description
0x00	BEFORE <File Name GUID>
0x01	AFTER <File Name GUID>
0x02	PUSH <Protocol GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END
0x09	SOR

BEFORE

Syntax

```
BEFORE <File Name GUID>
```

Description

This opcode tells the DXE Dispatcher that the DXE driver that is associated with this dependency expression must be dispatched just before the DXE driver with the file name specified by **GUID**. This means that as soon as the dependency expression for the DXE driver specified by **GUID** evaluates to **TRUE**, then this DXE driver must be placed in the dispatch queue just before the DXE driver with the file name specified by **GUID**.

Operation

None.

Table 22 defines the **BEFORE** instruction encoding.

Table 22. BEFORE Instruction Encoding

Byte	Description
0	0x00
1..16	A 16-byte GUID that represents the file name of a different DXE driver. The format is the same as type **EFI_GUID**.

Behaviors and Restrictions

If this opcode is present in a dependency expression, it must be the first and only opcode in the expression. If is appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.

AFTER

Syntax

AFTER <File Name GUID>

Description

This opcode tells the DXE Dispatcher that the DXE driver that is associated with this dependency expression must be dispatched just after the DXE driver with the file name specified by **GUID**. This means that as soon as the dependency expression for the DXE driver specified by **GUID** evaluates to **TRUE**, then this DXE driver must be placed in the dispatch queue just after the DXE Driver with the file name specified by **GUID**.

Operation

None.

Table 23 defines the **AFTER** instruction encoding.

Table 23. AFTER Instruction Encoding

Byte	Description
0	0x01
1..16	A 16-byte GUID that represents the file name of a different DXE driver. The format is the same as type **EFI_GUID**.

Behaviors and Restrictions

If this opcode is present in a dependency expression, it must be the first and only opcode in the expression. If is appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.

PUSH

Syntax

```
PUSH <Protocol 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 **LocateProtocol()**.

Operation

```
Status = gBS->LocateProtocol (GUID, NULL, &Interface);
if (EFI_ERROR (Status)) {
  PUSH FALSE;
} Else {
  PUSH TRUE;
}
```

Table 24 defines the **PUSH** instruction encoding.

Table 24. PUSH Instruction Encoding

Byte	Description
0	0x02
1..16	A 16-byte GUID that represents a protocol that is produced by a different DXE driver. 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 25 defines the **AND** instruction encoding.

Table 25. 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 26 defines the **OR** instruction encoding.

Table 26. 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 Operand1
PUSH Result
```

Table 27 defines the **NOT** instruction encoding.

Table 27. 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 28 defines the **TRUE** instruction encoding.

Table 28. 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 29 defines the **FALSE** instruction encoding.

Table 29. 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 30 defines the **END** instruction encoding.

Table 30. END Instruction Encoding

Byte	Description
0	0x08.

Behaviors and Restrictions

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

SOR

Syntax

SOR

Description

Indicates that the DXE driver is to remain on the Schedule on Request (SOR) queue until the DXE Service **Schedule()** is called for this DXE. The dependency expression evaluator treats this operation like a No Operation (**NOP**).

Operation

None.

Table 31 defines the **SOR** instruction encoding.

Table 31. SOR Instruction Encoding

Byte	Description
0	0x09.

Behaviors and Restrictions

- If this instruction is present in a dependency expression, it must be the first instruction in the expression. If it appears in any other location in the dependency expression, then the dependency expression is evaluated to **FALSE**.
- This instruction must be followed by a valid dependency expression. If this instruction is the last instruction or it is followed immediately by an **END** instruction, then the dependency expression is evaluated to **FALSE**.

10.8 Dependency Expression with No Dependencies

A DXE driver that does not have any dependencies must have a dependency expression that evaluates to **TRUE** with no dependencies on any protocol GUIDs or file name GUIDs. The DXE Dispatcher will queue all the DXE drivers of this type immediately after the *a priori* file has been processed.

The following code example shows the dependency expression for a DXE driver that does not have any dependencies using the BNF grammar listed in Dependency Expression Grammar. This is followed by the 2-byte dependency expression that is encoded using the instruction set described in "Dependency Expression Instruction Set" on page 100.

```
//
// Source
//
TRUE
END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                      MNEMONIC
====   =======================
================================================
0x00 : 06                          TRUE
0x01 : 08                          END
```

10.9 Empty Dependency Expressions

If a DXE driver file does not contain a dependency section, then the DXE driver has an empty dependency expression. The DXE Foundation must support DXE driver and UEFI drivers that conform to the UEFI 2.0 specification. These UEFI drivers assume that all the UEFI Boot Services and UEFI Runtime Services are available. If an UEFI driver is added to a firmware volume, then the UEFI driver will have an empty dependency expression, and it should not be loaded and executed by the DXE Dispatcher until all the UEFI Boot Services and UEFI Runtime Services are available. The DXE Foundation cannot guarantee that this condition is true until all of the DXE Architectural Protocols have been installed.

From the DXE Dispatcher's perspective, DXE drivers without dependency expressions cannot be loaded until all of the DXE Architectural Protocols have been installed. This is equivalent to an implied dependency expression of all the GUIDs of the architectural protocols ANDed together. This implied dependency expression is shown below. The use of empty dependency expressions may also save space, because DXE drivers that require all the UEFI Boot Services and UEFI Runtime Services to be present can simply remove the dependency section from the DXE driver file.

The code example below shows the dependency expression that is implied by an empty dependency expression using the BNF grammar listed in "Dependency Expression Grammar" on page 199. It also shows the dependency expression after it has been encoded using the instruction set described in "Dependency Expression Instruction Set" on page 100. This fairly complex dependency expression is encoded into a dependency expression that is 216 bytes long. Typical dependency expressions will contain 2 or 3 terms, so those dependency expressions will typically be less than 60 bytes long.

```
//
// Source
//
EFI_BDS_ARCH_PROTOCOL_GUID                AND
EFI_CPU_ARCH_PROTOCOL_GUID                AND
EFI_METRONOME_ARCH_PROTOCOL_GUID          AND
EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID  AND
EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID    AND
EFI_RESET_ARCH_PROTOCOL_GUID              AND
EFI_RUNTIME_ARCH_PROTOCOL_GUID            AND
EFI_SECURITY_ARCH_PROTOCOL_GUID           AND
EFI_TIMER_ARCH_PROTOCOL_GUID              AND
EFI_VARIABLE_ARCH_PROTOCOL_GUID           AND
EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID     AND
EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID
END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                    MNEMONIC
====   =======================
===============================================
0x00 : 02                        PUSH
0x01 : F6 3F 5E 66 CC 46 d4 11   EFI_BDS_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0x11 : 02                        PUSH
0x12 : B1 CC BA 26 42 6F D4 11   EFI_CPU_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0x22 : 03                        AND
0x23 : 02                        PUSH
0x24 : B2 CC BA 26 42 6F d4 11   EFI_METRONOME_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0x34 : 02                        PUSH
0x35 : 72 70 A9 1D DC BD 30 4B
EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID
       99 F1 72 A0 B5 6F FF 2A
0x45 : 03                        AND
0x46 : 03                        AND
0x47 : 02                        PUSH
0x48 : 87 AC CF 27 CC 46 d4 11   EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D


0x58 : 02                        PUSH
0x59 : 88 AC CF 27 CC 46 d4 11   EFI_RESET_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0x69 : 03                        AND
0x6A : 02                        PUSH
0x6B : 53 82 d0 96 83 84 d4 11   EFI_RUNTIME_ARCH_PROTOCOL_GUID
       BC F1 00 80 C7 3C 88 81
```

```
0x7B : 02                          PUSH
0x7C : E3 23 64 A4 17 46 f1 49     EFI_SECURITY_ARCH_PROTOCOL_GUID
       B9 FF D1 BF A9 11 58 39
       82 CE 5A 89 0C CB 2C 95
0xA0 : 02                          PUSH
0xA1 : B3 CC BA 26 42 6F D4 11     EFI_TIMER_ARCH_PROTOCOL_GUID
       BC E7 00 80 C7 3C 88 81
0xB1 : 03                          AND
0xB2 : 02                          PUSH
0xB3 : E2 68 56 1E 81 84 D4 11     EFI_VARIABLE_ARCH_PROTOCOL_GUID
       BC F1 00 80 C7 3C 88 81
0xC3 : 02                          PUSH
0xC4 : 18 F8 41 64 62 63 44 4E     EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID
       B5 70 7D BA 31 DD 24 53
0xD4 : 03                          AND
0xD5 : 03                          AND
0xD6 : 03                          AND
0xD7 : 02                          PUSH
0xD8 : F5 3F 5E 66 CC 46 d4 11     EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID
       9A 38 00 90 27 3F C1 4D
0xE8 : 03                          AND
0xE9 : 08                          END
```

10.10 Dependency Expression Reverse Polish Notation (RPN)

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

The following is a BNF encoding of this grammar. See "Dependency Expression Instruction Set" on page 100 for definitions of the dependency expressions.

```
<statement> ::= SOR <expression> END |
                BEFORE <guid> END |
                AFTER <guid> END |
                <expression> END

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

10.11 DXE Dispatcher State Machine

The DXE Dispatcher is responsible for tracking the state of a DXE driver from the time that the DXE driver is discovered in a firmware volume until the DXE Foundation is terminated with a call to **ExitBootServices()**. During this time, each DXE driver may be in one of several different states. The state machine that the DXE Dispatcher must use to track a DXE driver is shown in Figure 7.

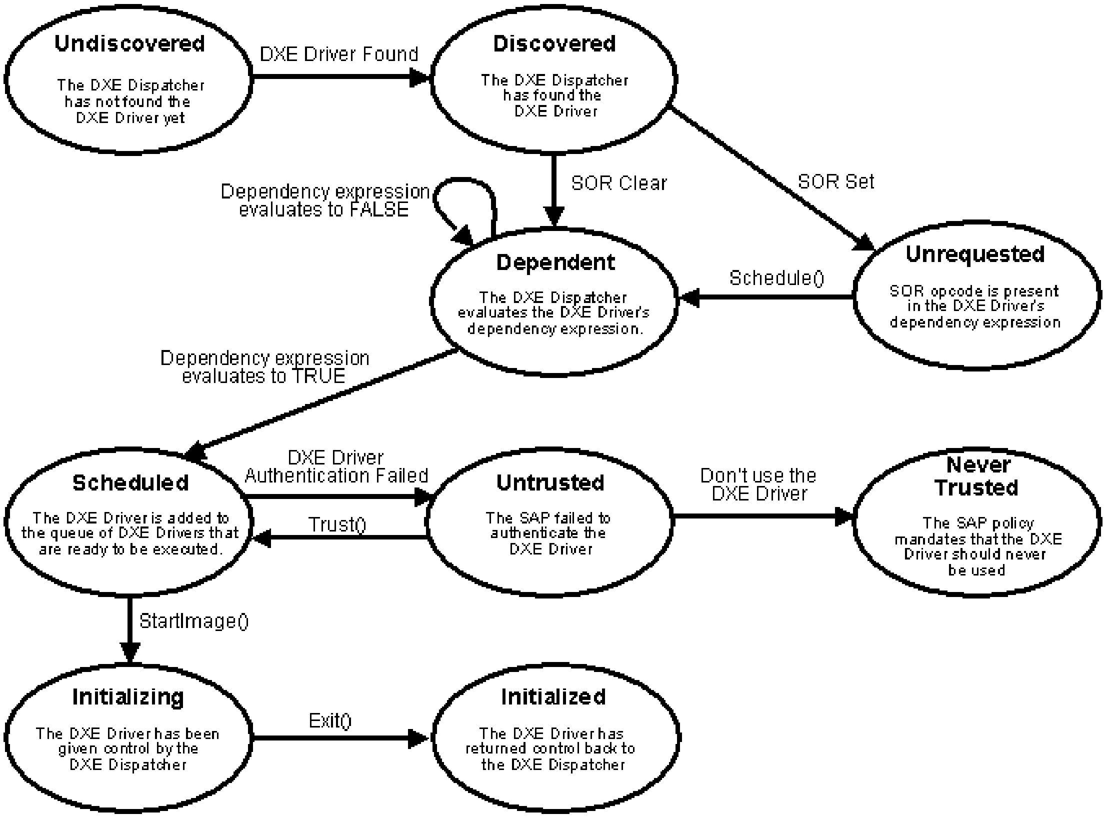

Undiscovered
The DXE Dispatcher has not found the DXE Driver yet
DXE Driver Found
Discovered
The DXE Dispatcher has found the DXE Driver
SOR Clear
SOR Set
Dependency expression evaluates to FALSE
Dependent
The DXE Dispatcher evaluates the DXE Driver's dependency expression.
Schedule()
Unrequested
SOR opcode is present in the DXE Driver's dependency expression
Dependency expression evaluates to TRUE
Scheduled
The DXE Driver is added to the queue of DXE Drivers that are ready to be executed.
DXE Driver Authentication Failed
Trust()
Untrusted
The SAP failed to authenticate the DXE Driver
Don't use the DXE Driver
Never Trusted
The SAP policy mandates that the DXE Driver should never be used
StartImage()
Initializing
The DXE Driver has been given control by the DXE Dispatcher
Exit()
Initialized
The DXE Driver has returned control back to the DXE Dispatcher

Figure 7. DXE Driver States

A DXE driver starts in the “Undiscovered” state, which means that the DXE driver is in a firmware volume that the DXE Dispatcher does not know about yet. When the DXE Dispatcher discovers a new firmware volume, any DXE drivers from that firmware volume listed in the *a priori* file are immediately loaded and executed. DXE drivers listed in the *a priori* file are immediately promoted to the “Scheduled” state. The firmware volume is then searched for DXE drivers that are not listed in the *a priori* file. Any DXE drivers found are promoted from the “Undiscovered” to the “Discovered” state. The dependency expression for each DXE driver is evaluated. If the SOR opcode is present in a DXE driver’s dependency expression, then the DXE driver is placed in the “Unrequested” state. If the SOR opcode is not present in the DXE driver’s dependency expression, then the DXE driver is placed in the “Dependent” state. Once a DXE driver is in the "Unrequested” state, it may only be promoted to the “Dependent” state with a call to the DXE Service `Schedule()`.

Once a DXE Driver is in the “Dependent” state, the DXE Dispatcher will evaluate the DXE driver’s dependency expression. If the DXE driver does not have a dependency expression, then a dependency expression of all the architectural protocols ANDed together is assumed for that DXE driver. If the dependency expression evaluates to **FALSE**, then the DXE driver stays in the

"Dependent" state. If the dependency expression never evaluates to **TRUE**, then it will never leave the "Dependent" state. If the dependency expression evaluates to **TRUE**, then the DXE driver will be promoted to the "Scheduled" state.

A DXE driver that is prompted to the "Scheduled" state is added to the end of the queue of other DXE drivers that have been promoted to the "Scheduled" state. When the DXE driver has reached the head of the queue, the DXE Dispatcher must use the services of the Security Authentication Protocol (SAP) to check the authentication status of the DXE Driver. If the Security Authentication Protocol deems that the DXE Driver violates the security policy of the platform, then the DXE Driver is placed in the "Untrusted" state. The Security Authentication Protocol can also tell the DXE Dispatcher that the DXE driver should never be executed and be placed in the "Never Trusted" state. If a DXE driver is placed in the "Untrusted" state, it can only be promoted back to the "Scheduled" state with a call to the DXE Service **Trust()**.

Once a DXE driver has reached the head of the scheduled queue, and the DXE driver has passed the authentication checks of the Security Authentication Protocol, the DXE driver is loaded into memory with the Boot Service **LoadImage()**. Control is then passed from the DXE Dispatcher to the DXE driver with the Boot Service **StartImage()**. When **StartImage()** is called for a DXE driver, that DXE driver is promoted to the "Initializing" state. The DXE driver returns control to the DXE Dispatcher through the Boot Service **Exit()**. When a DXE driver has returned control to the DXE Dispatcher, the DXE driver is in the terminal state called "Initialized."

The DXE Dispatcher is responsible for draining the queue of DXE drivers in the "Scheduled" state until the queue is empty. Once the queue is empty, then DXE Dispatcher must evaluate all the DXE drivers in the "Dependent" state to see if any of them need to be promoted to the "Scheduled" state. These evaluations need to be performed every time one or more DXE drivers have been promoted to the "Initialized" state, because those DXE drivers may have produced protocol interfaces for which the DXE drivers in the "Dependent" state are waiting.

10.12 Example Orderings

The order that DXE drivers are loaded and executed by the DXE Dispatcher is a mix of strong and weak orderings. The strong orderings are specified through *a priori* files, and the weak orderings are specified by dependency expressions in DXE drivers. Figure 8 shows the contents of a sample firmware volume that contains the following:

- DXE Foundation image
- DXE driver images
- An *a priori* file

The order that these images appear in the firmware volume is arbitrary. The DXE Foundation and the DXE Dispatcher must not make any assumptions about the locations of files in firmware volumes. The *a priori* file contains the GUID file names of the DXE drivers that are to be loaded and executed first. The dependency expressions and the protocols that each DXE driver produces is shown next to each DXE driver image in the firmware volume.

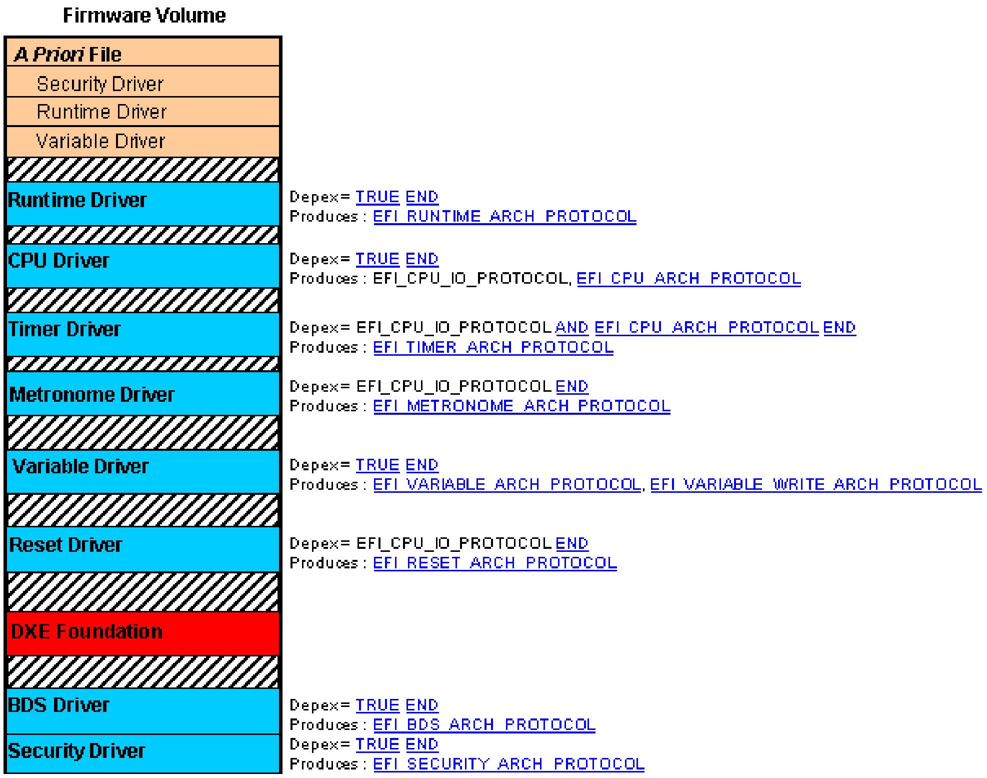

Firmware Volume
A Priori File
Security Driver
Runtime Driver
Variable Driver
Runtime Driver
Depex= TRUE END
Produces : EFI_RUNTIME_ARCH_PROTOCOL
CPU Driver
Depex= TRUE END
Produces : EFI_CPU_IO_PROTOCOL, EFI_CPU_ARCH_PROTOCOL
Timer Driver
Depex= EFI_CPU_IO_PROTOCOL AND EFI_CPU_ARCH_PROTOCOL END
Produces : EFI_TIMER_ARCH_PROTOCOL
Metronome Driver
Depex= EFI_CPU_IO_PROTOCOL END
Produces : EFI_METRONOME_ARCH_PROTOCOL
Variable Driver
Depex= TRUE END
Produces : EFI_VARIABLE_ARCH_PROTOCOL, EFI_VARIABLE_WRITE_ARCH_PROTOCOL
Reset Driver
Depex= EFI_CPU_IO_PROTOCOL END
Produces : EFI_RESET_ARCH_PROTOCOL
DXE Foundation
BDS Driver
Depex= TRUE END
Produces : EFI_BDS_ARCH_PROTOCOL
Security Driver
Depex= TRUE END
Produces : EFI_SECURITY_ARCH_PROTOCOL

Figure 8. Sample Firmware Volume

Based on the contents of the firmware volume in the figure above, the Security Driver, Runtime Driver, and Variable Driver will always be executed first. This is an example of a strongly ordered dispatch due to the *a priori* file. The DXE Dispatcher will then evaluate the dependency expressions of the remaining DXE drivers to determine the order that they will be executed. Based on the dependency expressions and the protocols that each DXE driver produces, there are 30 valid orderings from which the DXE Dispatcher may choose. The BDS Driver and CPU Driver tie for the next drivers to be scheduled, because their dependency expressions are simply **TRUE**. A dependency expression of TRUE means that the DXE driver does not require any other protocol interfaces to be executed. The DXE Dispatcher may choose either one of these drivers to be scheduled first. The Timer Driver, Metronome Driver, and Reset Driver all depend on the protocols produced by the CPU Driver. Once the CPU Driver has been loaded and executed, the Timer Driver, Metronome Driver, and Reset Driver may be scheduled in any order. The table below shows all 30 possible orderings from the sample firmware volume in the figure above. Each ordering is listed from left to right across the table. A reasonable implementation of a DXE Dispatcher would consistently produce the same ordering for a given system configuration. If the configuration of the system is changed in any way (including a order of files stored in a firmware volume), then a different dispatch ordering may be generated, but this new ordering should be consistent until the next system configuration change.

Table 32. DXE Dispatcher Orderings

Dispatch Order								
	1	2	3	4	5	6	7	8
1	Security	Runtime	Variable	BDS	CPU	Timer	Metronome	Reset
2	Security	Runtime	Variable	BDS	CPU	Timer	Reset	Metronome
3	Security	Runtime	Variable	BDS	CPU	Metronome	Timer	Reset
4	Security	Runtime	Variable	BDS	CPU	Metronome	Reset	Timer
5	Security	Runtime	Variable	BDS	CPU	Reset	Timer	Metronome
6	Security	Runtime	Variable	BDS	CPU	Reset	Metronome	Timer
7	Security	Runtime	Variable	CPU	BDS	Timer	Metronome	Reset
8	Security	Runtime	Variable	CPU	BDS	Timer	Reset	Metronome
9	Security	Runtime	Variable	CPU	BDS	Metronome	Timer	Reset
10	Security	Runtime	Variable	CPU	BDS	Metronome	Reset	Timer
11	Security	Runtime	Variable	CPU	BDS	Reset	Timer	Metronome
12	Security	Runtime	Variable	CPU	BDS	Reset	Metronome	Timer
13	Security	Runtime	Variable	CPU	Timer	BDS	Metronome	Reset
14	Security	Runtime	Variable	CPU	Timer	BDS	Reset	Metronome
15	Security	Runtime	Variable	CPU	Timer	Metronome	BDS	Reset
16	Security	Runtime	Variable	CPU	Timer	Metronome	Reset	BDS
17	Security	Runtime	Variable	CPU	Timer	Reset	BDS	Metronome
18	Security	Runtime	Variable	CPU	Timer	Reset	Metronome	BDS
19	Security	Runtime	Variable	CPU	Metronome	Timer	BDS	Reset
20	Security	Runtime	Variable	CPU	Metronome	Timer	Reset	BDS
21	Security	Runtime	Variable	CPU	Metronome	BDS	Timer	Reset
22	Security	Runtime	Variable	CPU	Metronome	BDS	Reset	Timer
23	Security	Runtime	Variable	CPU	Metronome	Reset	Timer	BDS
24	Security	Runtime	Variable	CPU	Metronome	Reset	BDS	Timer
25	Security	Runtime	Variable	CPU	Reset	Timer	Metronome	BDS
26	Security	Runtime	Variable	CPU	Reset	Timer	BDS	Metronome
27	Security	Runtime	Variable	CPU	Reset	Metronome	Timer	BDS
28	Security	Runtime	Variable	CPU	Reset	Metronome	BDS	Timer
29	Security	Runtime	Variable	CPU	Reset	BDS	Timer	Metronome
30	Security	Runtime	Variable	CPU	Reset	BDS	Metronome	Timer

10.13 Security Considerations

The DXE Dispatcher is required to use the services of the Security Architectural Protocol every time a firmware volume is discovered and before each DXE driver is executed. Because the Security Architectural Protocol is produced by a DXE driver, there will be at least one firmware volume discovered, and one or more DXE drivers loaded and executed before the Security Architectural Protocol is installed. The DXE Dispatcher should not attempt to use the services of the Security Architectural Protocol until the Security Architectural Protocol is installed. If a platform requires the Security Architectural Protocol to be present very early in the DXE phase, then the *a priori* file may be used to specify the name of the DXE driver that produces the Security Architectural Protocol.

The Security Architectural Protocol provides a service to evaluate the authentication status of a file. This service can also be used to evaluate the authenticate status of a firmware volume. If the authentication status is good, then no action is taken. If there is a problem with the firmware volume's authentication status, then the Security Architectural Protocol may perform a platform specific action. One option is to force the DXE Dispatcher to ignore the firmware volume so no DXE drivers will be loaded and executed from it. Another is to log the fact that the DXE Dispatcher is going to start dispatching DXE driver from a firmware volume with a questionable authentication status.

The Security Architectural Protocol can also be used to evaluate the authentication status of each DXE driver discovered in a firmware volume. If the authentication status is good, then no action is taken. If there is a problem with the DXE driver's authentication status, then the Security Architectural Protocol may take a platform-specific action. One possibility is to force the DXE driver into the "Untrusted" state, so it will not be considered for dispatch until the Boot Service **`Trust()`** is called for that DXE driver. Another possibility is to have the DXE Dispatcher place the DXE driver in the "Never Trusted" state, so it will never be loaded or executed. Another option is to log the fact that a DXE driver with a questionable authentication status is about to be loaded and executed.

11 DXE Drivers

11.1 Introduction

The DXE architecture provides a rich set of extensible services that provides for a wide variety of different system firmware designs. The DXE Foundation provides the generic services required to locate and execute DXE drivers. The DXE drivers are the components that actually initialize the platform and provide the services required to boot an UEFI-compliant operating system or a set of UEFI-compliant system utilities. There are many possible firmware implementations for any given platform. Because the DXE Foundation has fixed functionality, all the added value and flexibility in a firmware design is embodied in the implementation and organization of DXE drivers.

There are two basic classes of DXE drivers:

- Early DXE Drivers
- DXE Drivers that follow the UEFI Driver Model

Additional classifications of DXE drivers are also possible.

All DXE drivers may consume the UEFI Boot Services, UEFI Runtime Services, and DXE Services to perform their functions. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed. See the following topics for the DXE Architectural Protocols upon which the services depend:

- UEFI Boot Services Dependencies
- UEFI Runtime Services Dependencies
- DXE Services Dependencies

11.2 Classes of DXE Drivers

11.2.1 Early DXE Drivers

The first class of DXE drivers are those that execute very early in the DXE phase. The execution order of these DXE drivers depends on the following:

- The presence and contents of an a priori file
- The evaluation of dependency expressions

These early DXE drivers will typically contain basic services, processor initialization code, chipset initialization code, and platform initialization code. These early drivers will also typically produce the DXE Architectural Protocols that are required for the DXE Foundation to produces its full complement of UEFI Boot Services and UEFI Runtime Services. To support the fastest possible boot time, as much initialization should be deferred to the DXE drivers that follow UEFI Driver Model described in the UEFI 2.0 specification.

The early DXE drivers need to be aware that not all of the UEFI Boot Services, UEFI Runtime Services, and DXE Services may be available when they execute because not all of the DXE Architectural Protocols may be been registered yet.

11.2.2 DXE Drivers that Follow the UEFI Driver Model

The second class of DXE drivers are those that follow the UEFI Driver Model in the UEFI 2.0 specification. These drivers do not touch any hardware resources when they initialize. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the Boot Device Selection (BDS) phase to connect the drivers to the devices that are required to establish consoles and provide access to boot devices. The DXE drivers that follow the UEFI Driver Model ultimately provide software abstractions for console devices and boot devices, but only when they are explicitly asked to do so.

The DXE drivers that follow the UEFI Driver Model do not need to be concerned with dependency expressions. These drivers simply register the Driver Binding Protocol in the handle database when they are executed, and this operation can be performed without the use of any DXE Architectural Protocols. DXE drivers with empty dependency expressions will not be dispatched by the DXE Dispatcher until all of the DXE Architectural Protocols have been installed.

11.2.3 Additional Classifications

DXE drivers can also be classified as the following:

- Boot service drivers
- Runtime drivers

Boot service drivers provide services that are available until the **`ExitBootServices()`** function is called. When **`ExitBootServices()`** is called, all the memory used by boot service drivers is released for use by an operating system.

Runtime drivers provide services that are available before and after **`ExitBootServices()`** is called, including the time that an operating system is running. All of the services in the UEFI Runtime Services Table are produced by runtime drivers.

The DXE Foundation is considered a boot service component, so the DXE Foundation is also released when **`ExitBootServices()`** is called. As a result, runtime drivers may not use any of the UEFI Boot Services, DXE Services, or services produced by boot service drivers after **`ExitBootServices()`** is called.

12 DXE Architectural Protocols

12.1 Introduction

The DXE Foundation is abstracted from the platform hardware through a set of architectural protocols. These protocols function just like other protocols in every way. The only difference is that these architectural protocols are the protocols that the DXE Foundation itself consumes to produce the UEFI Boot Services, UEFI Runtime Services, and DXE Services. DXE drivers that are loaded from firmware volumes produce the DXE Architectural Protocols. This means that the DXE Foundation must have enough services to load and start DXE drivers before even a single DXE driver is executed.

The DXE Foundation is passed a HOB list that must contain a description of some amount of system memory and at least one firmware volume. The system memory descriptors in the HOB list are used to initialize the UEFI services that require only memory to function correctly. The system is also guaranteed to be running on only one processor in flat physical mode with interrupts disabled. The firmware volume is passed to the DXE Dispatcher, and the DXE Dispatcher must contain a read-only firmware file system driver to search for the a priori file and any DXE drivers in the firmware volumes. When a driver is discovered that needs to be loaded and executed, the DXE Dispatcher will use a PE/COFF loader to load and invoke the DXE driver. The early DXE drivers will produce the DXE Architectural Protocols, so the DXE Foundation can produce the full complement of UEFI Boot Services and UEFI Runtime Services.

Figure 9 shows the HOB list being passed to the DXE Foundation.

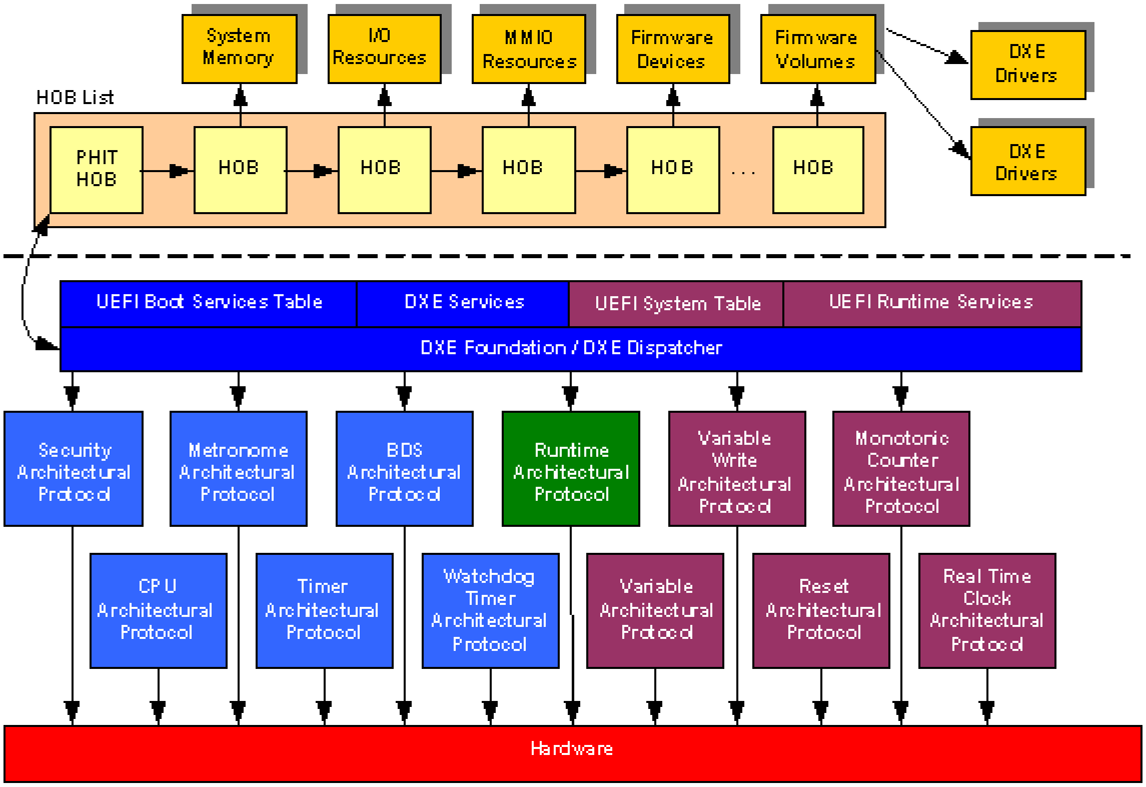

System Memory
I/O Resources
MMIO Resources
Firmware Devices
Firmware Volumes
DXE Drivers
DXE Drivers
HOB List
PHIT HOB
HOB
HOB
HOB
HOB
. . .
HOB
UEFI Boot Services Table
DXE Services
UEFI System Table
UEFI Runtime Services
DXE Foundation / DXE Dispatcher
Security Architectural Protocol
Metronome Architectural Protocol
BDS Architectural Protocol
Runtime Architectural Protocol
Variable Write Architectural Protocol
Monotonic Counter Architectural Protocol
CPU Architectural Protocol
Timer Architectural Protocol
Watchdog Timer Architectural Protocol
Variable Architectural Protocol
Reset Architectural Protocol
Real Time Clock Architectural Protocol
Hardware

Figure 9. DXE Architectural Protocols

The DXE Foundation consumes the services of the DXE Architectural Protocols and produces the following:

- UEFI System Table
- UEFI Boot Services Table
- UEFI Runtime Services Table
- DXE Services Table

The UEFI Boot Services Table and DXE Services Table are allocated from UEFI boot services memory, which means that the UEFI Boot Services Table and DXE Services Table are freed when the OS runtime phase is entered. The UEFI System Table and UEFI Runtime Services Table are allocated from UEFI runtime services memory, and they persist into the OS runtime phase.

When executing upon an UEFI-compliant system, UEFI drivers, applications, and UEFI-aware operating systems can discern if the platform is built upon the Foundation by searching for the DXE Services Table GUID in the UEFI System configuration table.

The DXE Architectural Protocols shown on the left of the figure are used to produce the UEFI Boot Services and DXE Services. The DXE Foundation and these protocols will be freed when the system transitions to the OS runtime phase. The DXE Architectural Protocols shown on the right are used to produce the UEFI Runtime Services. These services will persist in the OS runtime phase. The Runtime Architectural Protocol in the middle is unique. This protocol provides the services that are required to transition the runtime services from physical mode to virtual mode under the direction of an OS. Once this transition is complete, the services of the Runtime Architectural Protocol can no longer be used. The following topics describe all of the DXE Architectural Protocols in detail.

12.2 Boot Device Selection (BDS) Architectural Protocol

EFI_BDS_ARCH_PROTOCOL

Summary

Transfers control from the DXE phase to an operating system or system utility. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_BDS_ARCH_PROTOCOL_GUID  \
  {0x665E3FF6,0x46CC,0x11d4,
  0x9A,0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Protocol Interface Structure

```
typedef struct {
  EFI_BDS_ENTRY  Entry;
} EFI_BDS_ARCH_PROTOCOL;
```

Parameters

Entry

The entry point to BDS. See the **Entry()** function description. This call does not take any parameters, and the return value can be ignored. If it returns, then the dispatcher must be invoked again, if it never returns, then an operating system or a system utility have been invoked.

Description

The **EFI_BDS_ARCH_PROTOCOL** transfers control from DXE to an operating system or a system utility. If there are not enough drivers initialized when this protocol is used to access the required boot device(s), then this protocol should add drivers to the dispatch queue and return control back to the dispatcher. Once the required boot devices are available, then the boot device can be used to load and invoke an OS or a system utility.

EFI_BDS_ARCH_PROTOCOL.Entry()

Summary

Performs Boot Device Selection (BDS) and transfers control from the DXE Foundation to the selected boot device. The implementation of the boot policy must follow the rules outlined in the Boot Manager chapter of the UEFI 2.0 specification. This boot policy allows for flexibility, so the platform vendor will typically customize the implementation of this service.

Prototype

```
typedef
VOID
(EFIAPI *EFI_BDS_ENTRY) (
  IN CONST EFI_BDS_ARCH_PROTOCOL  *This
  );
```

Parameters

This

The **EFI_BDS_ARCH_PROTOCOL** instance.

Description

This function uses policy data from the platform to determine what operating system or system utility should be loaded and invoked. This function call also optionally uses the user's input to determine the operating system or system utility to be loaded and invoked. When the DXE Foundation has dispatched all the drivers on the dispatch queue, this function is called. This function will attempt to connect the boot devices required to load and invoke the selected operating system or system utility. During this process, additional firmware volumes may be discovered that may contain addition DXE drivers that can be dispatched by the DXE Foundation. If a boot device cannot be fully connected, this function calls the DXE Service **Dispatch() t**o allow the DXE drivers from any newly discovered firmware volumes to be dispatched. Then the boot device connection can be attempted again. If the same boot device connection operation fails twice in a row, then that boot device has failed, and should be skipped. This function should never return.

12.3 CPU Architectural Protocol

EFI_CPU_ARCH_PROTOCOL

Summary

Abstracts the processor services that are required to implement some of the DXE services. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce architectural protocols.

GUID

```
#define EFI_CPU_ARCH_PROTOCOL_GUID  \
  {0x26baccb1,0x6f42,0x11d4,0xbc,\
  0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_CPU_ARCH_PROTOCOL {
  EFI_CPU_FLUSH_DATA_CACHE           FlushDataCache;
  EFI_CPU_ENABLE_INTERRUPT           EnableInterrupt;
  EFI_CPU_DISABLE_INTERRUPT          DisableInterrupt;
  EFI_CPU_GET_INTERRUPT_STATE        GetInterruptState;
  EFI_CPU_INIT                       Init;
  EFI_CPU_REGISTER_INTERRUPT_HANDLER RegisterInterruptHandler;
  EFI_CPU_GET_TIMER_VALUE            GetTimerValue;
  EFI_CPU_SET_ATTRIBUTES             SetMemoryAttributes;
  UINT32                             NumberOfTimers;
  UINT32                             DmaBufferAlignment;
} EFI_CPU_ARCH_PROTOCOL;
```

Parameters

FlushDataCache

Flushes a range of the processor's data cache. See the **FlushDataCache()** function description. If the processor does not contain a data cache, or the data cache is fully coherent, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of addresses from the data cache, then the entire data cache must be flushed. This function is used by the root bridge I/O abstractions to flush data caches for DMA operations.

EnableInterrupt

Enables interrupt processing by the processor. See the **EnableInterrupt()** function description. This function is used by the Boot Service **RaiseTPL()** and **RestoreTPL()**.

DisableInterrupt

Disables interrupt processing by the processor. See the **DisableInterrupt()** function description. This function is used by the Boot Service **RaiseTPL()** and **RestoreTPL()**.

GetInterruptState

Retrieves the processor's current interrupt state. See the **GetInterruptState()** function description.

Init

Generates an INIT on the processor. See the **Init()** function description. This function may be used by the **EFI_RESET** Protocol depending upon a specified boot path. If a processor cannot programmatically generate an INIT without help from external hardware, then this function returns **EFI_UNSUPPORTED**.

RegisterInterruptHandler

Associates an interrupt service routine with one of the processor's interrupt vectors. See the **RegisterInterruptHandler()** function description. This function is typically used by the **EFI_TIMER_ARCH_PROTOCOL** to hook the timer interrupt in a system. It can also be used by the debugger to hook exception vectors.

GetTimerValue

Returns the value of one of the processor's internal timers. See the **GetTimerValue()** function description.

SetMemoryAttributes

Change a memory region to support specified memory attributes. See the **SetMemoryAttributes()** function description.

NumberOfTimers

The number of timers that are available in a processor. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.

DmaBufferAlignment

The size, in bytes, of the alignment required for DMA buffer allocations. This is typically the size of the largest data cache line in the platform. This value can be determined by looking at the data cache line sizes of all the caches present in the platform, and returning the largest. This is used by the root bridge I/O abstraction protocols to guarantee that no two DMA buffers ever share the same cache line. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.

Description

The **EFI_CPU_ARCH_PROTOCOL** is used to abstract processor-specific functions from the DXE Foundation. This includes flushing caches, enabling and disabling interrupts, hooking interrupt vectors and exception vectors, reading internal processor timers, resetting the processor, and determining the processor frequency.

The GCD memory space map is initialized by the DXE Foundation based on the contents of the HOB list. The HOB list contains the capabilities of the different memory regions, but it does not contain their current attributes. The DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** is responsible for maintaining the current attributes of the memory regions visible to the processor. This means that the DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** must seed the

GCD memory space map with the initial state of the attributes for all the memory regions visible to the processor. The DXE Service **SetMemorySpaceAttributes()** allows the attributes of a memory range to be modified. The **SetMemorySpaceAttributes()** DXE Service is implemented using the **SetMemoryAttributes()** service of the **EFI_CPU_ARCH_PROTOCOL**.

To initialize the state of the attributes in the GCD memory space map, the DXE driver that produces the **EFI_CPU_ARCH_PROTOCOL** must call the DXE Service **SetMemorySpaceAttributes()** for all the different memory regions visible to the processor passing in the current attributes. If the CPU does not support certain memory region attributes in the UEFI Specification, then these should always be reported as disabled or not present. If the CPU supports additional memory region attributes, then the reported attributes should be those which most closely match but not exceed those described in the specification. This, in turn, will call back to the **SetMemoryAttributes()** service of the **EFI_CPU_ARCH_PROTOCOL**, and all of these calls must return **EFI_SUCCESS**, since the DXE Foundation is only requesting that the attributes of the memory region be set to their current settings. This will force the current attributes in the GCD memory space map to be set to these current settings. After this initialization is complete, the next call to the DXE Service **GetMemorySpaceMap()** will correctly show the current attributes of all the memory regions. In addition, any future calls to the DXE Service **SetMemorySpaceAttributes()** will in turn call the **EFI_CPU_ARCH_PROTOCOL** to see if those attributes can be modified, and if they can, the GCD memory space map will be updated accordingly.

EFI_CPU_ARCH_PROTOCOL.FlushDataCache()

Summary

Flushes a range of the processor's data cache. If the processor does not contain a data cache, or the data cache is fully coherent, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of addresses from the data cache, then the entire data cache must be flushed. This function is used by the root bridge I/O abstractions to flush caches for DMA operations.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_FLUSH_DATA_CACHE) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN EFI_PHYSICAL_ADDRESS         Start,
  IN UINT64                       Length,
  IN EFI_CPU_FLUSH_TYPE           FlushType
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Start

The beginning physical address to flush from the processor's data cache.

Length

The number of bytes to flush from the processor's data cache. This function may flush more bytes than *Length* specifies depending upon the granularity of the flush operation that the processor supports.

FlushType

Specifies the type of flush operation to perform. Type **EFI_CPU_FLUSH_TYPE** is defined in "Related Definitions" below.

Description

This function flushes the range of addresses from *Start* to *Start+Length* from the processor's data cache. If *Start* is not aligned to a cache line boundary, then the bytes before *Start* to the preceding cache line boundary are also flushed. If *Start+Length* is not aligned to a cache line boundary, then the bytes past *Start+Length* to the end of the next cache line boundary are also flushed. If the address range is flushed, then **EFI_SUCCESS** is returned. If the address range cannot be flushed, then **EFI_DEVICE_ERROR** is returned. If the processor does not support the flush type specified by *FlushType*, then **EFI_UNSUPPORTED** is returned. The *FlushType* of *EfiCpuFlushTypeWriteBackInvalidate* must be supported. If the data cache is fully coherent with all DMA operations, then this function can just return **EFI_SUCCESS**. If the processor does not support flushing a range of the data cache, then the entire data cache can be flushed.

Related Definitions

```
typedef enum {
  EfiCpuFlushTypeWriteBackInvalidate,
  EfiCpuFlushTypeWriteBack,
  EfiCpuFlushTypeInvalidate,
  EfiCpuMaxFlushType
} EFI_CPU_FLUSH_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The address range from *Start* to *Start+Length* was flushed from the processor’s data cache.
EFI_UNSUPPORTED	The processor does not support the cache flush type specified by *FlushType*.
EFI_DEVICE_ERROR	The address range from *Start* to *Start+Length* could not be flushed from the processor's data cache.

EFI_CPU_ARCH_PROTOCOL.EnableInterrupt()

Summary

Enables interrupt processing by the processor. This function is used to implement the Boot Services **RaiseTPL()** and **RestoreTPL()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_ENABLE_INTERRUPT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Description

This function enables interrupt processing by the processor. If interrupts are enabled, then **EFI_SUCCESS** is returned. Otherwise, **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	Interrupts are enabled on the processor.
EFI_DEVICE_ERROR	Interrupts could not be enabled on the processor.

EFI_CPU_ARCH_PROTOCOL.DisableInterrupt()

Summary

Disables interrupt processing by the processor. This function is used to implement the Boot Services **RaiseTPL()** and **RestoreTPL()**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_DISABLE_INTERRUPT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL   *This
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

Description

This function disables interrupt processing by the processor. If interrupts are disabled, then **EFI_SUCCESS** is returned. Otherwise, **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	Interrupts are disabled on the processor.
EFI_DEVICE_ERROR	Interrupts could not be disabled on the processor.

EFI_CPU_ARCH_PROTOCOL.GetInterruptState()

Summary

Retrieves the processor’s current interrupt state.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_GET_INTERRUPT_STATE) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL   *This,
  OUT BOOLEAN                       *State
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

State

A pointer to the processor’s current interrupt state. Set to **TRUE** if interrupts are enabled and **FALSE** if interrupts are disabled.

Description

This function retrieves the processor’s current interrupt state a returns it in *State*. If interrupts are currently enabled, then **TRUE** is returned. If interrupts are currently disabled, then **FALSE** is returned. If *State* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. Otherwise, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The processor’s current interrupt state was returned in *State*.
EFI_INVALID_PARAMETER	*State* is **NULL**.

EFI_CPU_ARCH_PROTOCOL.Init()

Summary

Generates an INIT on the processor.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_INIT) (
  IN CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN EFI_CPU_INIT_TYPE            InitType
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

InitType

The type of processor INIT to perform. Type **EFI_CPU_INIT_TYPE** is defined in "Related Definitions" below.

Description

This function generates an INIT on the processor. If this function succeeds, then the processor will be reset, and control will not be returned to the caller. If *InitType* is not supported by this processor, or the processor cannot programmatically generate an INIT without help from external hardware, then **EFI_UNSUPPORTED** is returned. If an error occurs attempting to generate an INIT, then **EFI_DEVICE_ERROR** is returned.

Related Definitions

```
typedef enum {
  EfiCpuInit,
  EfiCpuMaxInitType
} EFI_CPU_INIT_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The processor INIT was performed. This return code should never be seen.
EFI_UNSUPPORTED	The processor INIT operation specified by *InitType* is not supported by this processor.
EFI_DEVICE_ERROR	The processor INIT failed.

EFI_CPU_ARCH_PROTOCOL.RegisterInterruptHandler()

Summary

Registers a function to be called from the processor interrupt handler.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_REGISTER_INTERRUPT_HANDLER) (
  IN CONST EFI_CPU_ARCH_PROTOCOL      *This,
  IN EFI_EXCEPTION_TYPE               InterruptType,
  IN EFI_CPU_INTERRUPT_HANDLER        InterruptHandler
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

InterruptType

Defines which interrupt or exception to hook. Type **EFI_EXCEPTION_TYPE** and the valid values for this parameter are defined in **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

InterruptHandler

A pointer to a function of type **EFI_CPU_INTERRUPT_HANDLER** that is called when a processor interrupt occurs. If this parameter is **NULL**, then the handler will be uninstalled. Type **EFI_CPU_INTERRUPT_HANDLER** is defined in “Related Definitions” below.

Description

The **RegisterInterruptHandler()** function registers and enables the handler specified by *InterruptHandler* for a processor interrupt or exception type specified by *InterruptType*. If *InterruptHandler* is **NULL**, then the handler for the processor interrupt or exception type specified by *InterruptType* is uninstalled. The installed handler is called once for each processor interrupt or exception.

If the interrupt handler is successfully installed or uninstalled, then **EFI_SUCCESS** is returned.

If *InterruptHandler* is not **NULL**, and a handler for *InterruptType* has already been installed, then **EFI_ALREADY_STARTED** is returned.

If *InterruptHandler* is **NULL**, and a handler for *InterruptType* has not been installed, then **EFI_INVALID_PARAMETER** is returned.

If *InterruptType* is not supported, then **EFI_UNSUPPORTED** is returned.

The **EFI_CPU_ARCH_PROTOCOL** implementation of this function must handle saving and restoring system context to the system context record around calls to the interrupt handler. It must also perform the necessary steps to return to the context that was interrupted by the interrupt. No chaining of interrupt handlers is allowed.

Related Definitions

```
typedef
VOID
(*EFI_CPU_INTERRUPT_HANDLER) (
  IN EFI_EXCEPTION_TYPE  InterruptType,
  IN EFI_SYSTEM_CONTEXT  SystemContext
  );
```

InterruptType

Defines the type of interrupt or exception that occurred on the processor. This parameter is processor architecture specific. The type **EFI_EXCEPTION_TYPE** and the valid values for this parameter are defined in **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

SystemContext

A pointer to the processor context when the interrupt occurred on the processor. Type **EFI_SYSTEM_CONTEXT** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** of the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The handler for the processor interrupt was successfully installed or uninstalled.
EFI_ALREADY_STARTED	*InterruptHandler* is not **NULL**, and a handler for *InterruptType* was previously installed.
EFI_INVALID_PARAMETER	*InterruptHandler* is **NULL**, and a handler for *InterruptType* was not previously installed.
EFI_UNSUPPORTED	The interrupt specified by *InterruptType* is not supported.

EFI_CPU_ARCH_PROTOCOL.GetTimerValue()

Summary

Returns a timer value from one of the processor's internal timers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_GET_TIMER_VALUE) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL  *This,
  IN  UINT32                       TimerIndex,
  OUT UINT64                       *TimerValue,
  OUT UINT64                       *TimerPeriod  OPTIONAL
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

TimerIndex

Specifies which processor timer is to be returned in *TimerValue*. This parameter must be between 0 and *NumberOfTimers***-1**.

TimerValue

Pointer to the returned timer value.

TimerPeriod

A pointer to the amount of time that passes in femtoseconds (10^{-15}) for each increment of *TimerValue*. If *TimerValue* does not increment at a predictable rate, then 0 is returned. The amount of time that has passed between two calls to **GetTimerValue()** can be calculated with the formula (*TimerValue2 – TimerValue1*) ***** *TimerPeriod*. This parameter is optional and may be **NULL**.

Description

This function reads the processor timer specified by *TimerIndex* and returns it in *TimerValue*. If *TimerValue* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If *TimerPeriod* is not **NULL**, then the amount of time that passes in femtoseconds (10^{-15}) for each increment if *TimerValue* is returned in *TimerPeriod*. If the timer does not run at a predictable rate, then a *TimerPeriod* of 0 is returned. If *TimerIndex* does not specify a valid timer in this processor, then **EFI_INVALID_PARAMETER** is returned. The valid range for *TimerIndex* is 0..*NumberOfTimers*-1. If the processor does not contain any readable timers, then this function returns **EFI_UNSUPPORTED**. If an error occurs attempting to read one of the processor's timers, then **EFI_DEVICE_ERROR** is returned.

Status Codes Returned

EFI_SUCCESS	The processor timer value specified by *TimerIndex* was returned in *TimerValue*.
EFI_INVALID_PARAMETER	*TimerValue* is **NULL**.
EFI_INVALID_PARAMETER	*TimerIndex* is not valid.
EFI_UNSUPPORTED	The processor does not have any readable timers.
EFI_DEVICE_ERROR	An error occurred attempting to read one of the processor's timers.

EFI_CPU_ARCH_PROTOCOL.SetMemoryAttributes()

Summary

Change a memory region to support specified memory attributes.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CPU_SET_MEMORY_ATTRIBUTES) (
  IN  CONST EFI_CPU_ARCH_PROTOCOL   *This,
  IN  EFI_PHYSICAL_ADDRESS          BaseAddress,
  IN  UINT64                        Length,
  IN  UINT64                        Attributes
  );
```

Parameters

This

The **EFI_CPU_ARCH_PROTOCOL** instance.

BaseAddress

The physical address that is the start address of a memory region. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** function description in the UEFI 2.0 specification.

Length

The size in bytes of the memory region.

Attributes

A bit mask that specifies the memory region attributes. See the UEFI Boot Service **GetMemoryMap()** for the set of legal attribute bits.

Description

This function changes the attributes for the memory region specified by *BaseAddress* and *Length* to support those specified by *Attributes*. If the memory region attributes are changed so that they do not conflict with those specified by *Attributes*, then **EFI_SUCCESS** is returned.

This function modifies the attributes for the memory region specified by *BaseAddress* and *Length* from their current attributes to the attributes specified by *Attributes*. If this modification of attributes succeeds, then **EFI_SUCCESS** is returned.

If *Length* is zero, then **EFI_INVALID_PARAMETER** is returned.

If the processor does not support one or more bytes of the memory range specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes specified by *Attributes* are not supported for the memory region specified by *BaseAddress* and *Length*, then **EFI_UNSUPPORTED** is returned.

If the attributes for one or more bytes of the memory range specified by *BaseAddress* and *Length* cannot be modified because the current system policy does not allow them to be modified, then **EFI_ACCESS_DENIED** is returned.

If there are not enough system resources available to modify the attributes of the memory range, then **EFI_OUT_OF_RESOURCES** is returned.

If *Attributes* specifies a combination of memory attributes that cannot be set together, then **EFI_INVALID_PARAMETER** is returned. For example, if both **EFI_MEMORY_UC** and **EFI_MEMORY_WT** are set.

Status Codes Returned

EFI_SUCCESS	The attributes were set for the memory region.
EFI_INVALID_PARAMETER	*Length* is zero.
EFI_INVALID_PARAMETER	*Attributes* specified an illegal combination of attributes that cannot be set together.
EFI_UNSUPPORTED	The processor does not support one or more bytes of the memory resource range specified by *BaseAddress* and *Length*.
EFI_UNSUPPORTED	The bit mask of attributes is not support for the memory resource range specified by *BaseAddress* and *Length*.
EFI_ACCESS_DENIED	The attributes for the memory resource range specified by *BaseAddress* and *Length* cannot be modified.
EFI_OUT_OF_RESOURCES	There are not enough system resources to modify the attributes of the memory resource range.

12.4 Metronome Architectural Protocol

EFI_METRONOME_ARCH_PROTOCOL

Summary

Used to wait for ticks from a known time source in a platform. This protocol may be used to implement a simple version of the **Stall()** Boot Service. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce DXE Architectural Protocols.

GUID

```
#define EFI_METRONOME_ARCH_PROTOCOL_GUID  \
  {0x26baccb2,0x6f42,0x11d4,0xbc,\
  0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_METRONOME_ARCH_PROTOCOL {
  EFI_METRONOME_WAIT_FOR_TICK  WaitForTick;
  UINT32                       TickPeriod;
} EFI_METRONOME_ARCH_PROTOCOL;
```

Parameters

WaitForTick

Waits for a specified number of ticks from a known time source in the platform. See the **WaitForTick()** function description. The actual time passed between entry of this function and the first tick is between 0 and *TickPeriod* 100 ns units. To guarantee that at least *TickPeriod* time has elapsed, wait for two ticks.

TickPeriod

The period of platform's known time source in 100 ns units. This value on any platform must not exceed 200 µs. The value in this field is a constant that must not be modified after the Metronome architectural protocol is installed. All consumers must treat this as a read-only field.

Description

This protocol provides access to a known time source in the platform to the DXE Foundation. The DXE Foundation uses this known time source to produce DXE Foundation services that require calibrated delays.

EFI_METRONOME_ARCH_PROTOCOL.WaitForTick()

Summary

Waits for a specified number of ticks from a known time source in a platform.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_METRONOME_WAIT_FOR_TICK) (
  IN CONST EFI_METRONOME_ARCH_PROTOCOL   *This,
  IN UINT32                              TickNumber
  );
```

Parameters

This

The **EFI_METRONOME_ARCH_PROTOCOL** instance.

TickNumber

Number of ticks to wait.

Description

The **WaitForTick()** function waits for the number of ticks specified by *TickNumber* from a known time source in the platform. If *TickNumber* of ticks are detected, then **EFI_SUCCESS** is returned. The actual time passed between entry of this function and the first tick is between 0 and *TickPeriod* 100 ns units. If you want to guarantee that at least *TickPeriod* time has elapsed, wait for two ticks. This function waits for a hardware event to determine when a tick occurs. It is possible for interrupt processing, or exception processing to interrupt the execution of the **WaitForTick()** function. Depending on the hardware source for the ticks, it is possible for a tick to be missed. This function cannot guarantee that ticks will not be missed. If a timeout occurs waiting for the specified number of ticks, then **EFI_TIMEOUT** is returned.

Status Codes Returned

EFI_SUCCESS	The wait for the number of ticks specified by *TickNumber* succeeded.
EFI_TIMEOUT	A timeout occurred waiting for the specified number of ticks.

12.5 Monotonic Counter Architectural Protocol

EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL

Summary

Provides the services required to access the system's monotonic counter. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce DXE Architectural Protocols.

GUID

```
#define EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID  \
  {0x1da97072,0xbddc,0x4b30,0x99,\
  0xf1,0x72,0xa0,0xb5,0x6f,0xff,0x2a}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **GetNextHighMonotonicCount()** field of the UEFI Runtime Services Table and the **GetNextMonotonicCount()** field of the UEFI Boot Services Table. See Services - Runtime Services and Services - Boot Services for details on these services. After the field of the UEFI Runtime Services Table and the field of the UEFI Boot Services Table have been initialized, the driver must install the **EFI_MONOTONIC_COUNTER_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the monotonic counter services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table and the 32-bit CRC of the UEFI Boot Services Table.

12.6 Real Time Clock Architectural Protocol

EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL

Summary

Provides the services required to access a system's real time clock hardware. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID  \
  {0x27CFAC87,0x46CC,0x11d4,0x9A,\
  0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **GetTime()**, **SetTime(), GetWakeupTime()**, and **SetWakeupTime()** fields of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 33 for details on these services. After the four fields of the UEFI Runtime Services Table have been initialized, the driver must install the **EFI_REAL_TIME_CLOCK_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the real time clock–related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.7 Reset Architectural Protocol

EFI_RESET_ARCH_PROTOCOL

Summary

Provides the service required to reset a platform. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_RESET_ARCH_PROTOCOL_GUID  \
  {0x27CFAC88,0x46CC,0x11d4,0x9A,\
  0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **ResetSystem()** field of the UEFI Runtime Services Table. See “Runtime Capabilities” on page 33 for details on this service. After this field of the UEFI Runtime Services Table has been initialized, the driver must install the **EFI_RESET_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the reset system service is now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.8 Runtime Architectural Protocol

The following topics provide a detailed description of the **EFI_RUNTIME_ARCH_PROTOCOL**. The DXE Foundation contains no runtime code, so all runtime code is contained in DXE Architectural Protocols. This is due to the fact that runtime code must be callable in physical or virtual mode. The Runtime Architectural Protocol contains the UEFI runtime services that are callable only in physical mode. The Runtime Architectural Protocol can be thought of as the runtime portion of the DXE Foundation.

The Runtime Architectural Protocol contains support for transition of runtime drivers from physical mode calling to virtual mode calling.

EFI_RUNTIME_ARCH_PROTOCOL

Summary

Allows the runtime functionality of the DXE Foundation to be contained in a separate driver. It also provides hooks for the DXE Foundation to export information that is needed at runtime. As such, this protocol allows services to the DXE Foundation to manage runtime drivers and events. This protocol also implies that the runtime services required to transition to virtual mode, **SetVirtualAddressMap()** and **ConvertPointer()**, have been registered into the UEFI Runtime Table in the UEFI System Table. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

GUID

```
#define EFI_RUNTIME_ARCH_PROTOCOL_GUID  \
  {0xb7dfb4e1,0x52f,0x449f,0x87,\
  0xbe,0x98,0x18,0xfc,0x91,0xb7,0x33}
```

Protocol Interface Structure

```
typedef struct _EFI_RUNTIME_ARCH_PROTOCOL {
EFI_LIST_ENTRY          ImageHead;
EFI_LIST_ENTRY          EventHead;
UINTN                   MemoryDescriptorSize;
UINT32                  MemoryDesciptorVersion;
UINTN                   MemoryMapSize;
EFI_MEMORY_DESCRIPTOR   *MemoryMapPhysical;
EFI_MEMORY_DESCRIPTOR   *MemoryMapVirtual;
BOOLEAN                 VirtualMode;
BOOLEAN                 AtRuntime;
} EFI_RUNTIME_ARCH_PROTOCOL;
```

Parameters

ImageHead

A list of type **EFI_RUNTIME_IMAGE_ENTRY** where the DXE Foundation inserts items into the list and the Runtime AP consumes the data to implement the **SetVirtualAddressMap()** call.

EventHead

A list of type **EFI_RUNTIME_EVENT_ENTRY** where the DXE Foundation inserts items into the list and the Runtime AP consumes the data to implement the **SetVirtualAddressMap()** call.

MemoryDescriptorSize

Size of a memory descriptor that is returned by **GetMemoryMap()**. This value is updated by the DXE Foundation.

MemoryDescriptorVersion

Version of a memory descriptor that is return by **GetMemoryMap()**. This value is updated by the DXE Foundation.

MemoryMapSize

Size of the memory map in bytes contained in **MemoryMapPhysical** and **MemoryMapVirtual**. This value is updated by the DXE Foundation when memory for *MemoryMapPhysical* gets allocated.

MemoryMapPhysical

Pointer to a runtime buffer that contains a copy of the memory map returned via **GetMemoryMap()**. The memory must be allocated by the DXE Foundation so that it is accounted for in the memory map.

MemoryMapVirtual

Pointer to *MemoryMapPhysical* that is updated to virtual mode after **SetVirtualAddressMap()**. The DXE Foundation updates this value when it updates *MemoryMapPhysical* with the same physical address. The Runtime AP is responsible for converting *MemoryMapVirtual* to a virtual pointer.

VirtualMode

Boolean that is **TRUE** if **SetVirtualAddressMap()** has been called. This field is set by the Runtime AP. When *VirtualMode* is **TRUE** *MemoryMapVirtual* pointer contains the virtual address of the *MemoryMapPhysical*.

AtRuntime

Boolean that is **TRUE** if **ExitBootServices()** has been called. This field is set by the Runtime AP.

Related Definitions

```
//*******************************************************************
// EFI_LIST_ENTRY
//*******************************************************************
struct _EFI_LIST_ENTRY {
  struct _EFI_LIST_ENTRY   *ForwardLink;
  struct _EFI_LIST_ENTRY   *BackLink;
} EFI_LIST_ENTRY;
```

ForwardLink

A pointer next node in the doubly linked list.

BackLink

A pointer previous node in the doubly linked list.

```
//*******************************************************************
// EFI_RUNTIME_IMAGE_ENTRY
//*******************************************************************
typedef struct {
   VOID                *ImageBase;
   UINT64              ImageSize;
   VOID                *RelocationData;
   EFI_HANDLE          Handle;
   EFI_LIST_ENTRY      Link;
} EFI_RUNTIME_IMAGE_ENTRY;
```

ImageBase

Start of image that has been loaded in memory. It is a pointer to either the DOS header or PE header of the image. Type **EFI_PHYSICAL_ADDRESS** is defined in the **AllocatePages()** UEFI 2.0 specification.

ImageSize

Size in bytes of the image represented by *ImageBase*.

RelocationData

Information about the fix-ups that were performed on *ImageBase* when it was loaded into memory. This information is needed when the virtual mode fix-ups are reapplied so that data that has been programmatically updated will not be fixed up. If code updates a global variable the code is responsible for fixing up the variable for virtual mode.

Handle

The *ImageHandle* passed into *ImageBase* when it was loaded. See **EFI_IMAGE_ENTRY_POINT** for the definition of *ImageHandle*.

Link

Entry for this node in the **EFI_RUNTIME_ARCHITECTURE_PROTOCOL.ImageHead** list.

```
//*******************************************************************
// EFI_RUNTIME_EVENT_ENTRY
//*******************************************************************
typedef struct {
  UINT32                    Type;
  EFI_TPL                   NotifyTpl;
  EFI_EVENT_NOTIFY          NotifyFunction;
  VOID                      *NotifyContext;
  EFI_EVENT                 *Event;
  EFI_LIST_ENTRY            Link;
}  EFI_RUNTIME_EVENT_ENTRY;
```

Parameters

Type

The same as *Type* passed into **CreateEvent()**.

NotifyTpl

The same as *NotifyTpl* passed into **CreateEvent()**. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

NotifyFunction

The same as *NotifyFunction* passed into **CreateEvent()**. Type **EFI_EVENT_NOTIFY** is defined in the **CreateEvent()** function description.

NotifyContext

The same as *NotifyContext* passed into **CreateEvent()**.

Event

The **EFI_EVENT** returned by **CreateEvent()**. Event must be in runtime memory. Type **EFI_EVENT** is defined in the **CreateEvent()** function description.

Link

Entry for this node in the **EFI_RUNTIME_ARCHITECTURE_PROTOCOL.EventHead** list.

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **SetVirtualAddressMap()** and **ConvertPointer()** fields of the UEFI Runtime Services Table and the **CalculateCrc32()** field of the UEFI Boot Services Table. See "Runtime Capabilities" on page 33 and "Services - Boot Services" on page 28 for details on these services. After the two fields of the UEFI Runtime Services Table and the one field of the UEFI Boot Services Table have been initialized, the driver must install the **EFI_RUNTIME_ARCH_PROTOCOL_GUID** on a new handle with an **EFI_RUNTIME_ARCH_PROTOCOL** interface pointer. The installation of this protocol informs the DXE Foundation that the virtual memory services and the 32-bit CRC services are now available, and the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table and the 32-bit CRC of the UEFI Boot Services Table.

All runtime DXE Foundation services are provided by the **EFI_RUNTIME_ARCH_PROTOCOL**. This includes the support for registering runtime images that must be fixed up again when a transition is made from physical mode to virtual mode. This protocol also supports all events that are defined to fire at runtime. This protocol also contains a CRC-32 function that will be used by the DXE Foundation as a boot service. The **EFI_RUNTIME_ARCH_PROTOCOL** needs the CRC-32 function when a transition is made from physical mode to virtual mode and the UEFI System Table and UEFI Runtime Table are fixed up with virtual pointers.

12.9 Security Architectural Protocols

The **EFI_SECURITY_ARCH_PROTOCOL** and **EFI_SECURITY2_ARCH_PROTOCOL** abstract policy actions on image invocation and other security controls from the DXE core to a security driver.

12.9.1 Security Architectural Protocol

EFI_SECURITY_ARCH_PROTOCOL

Summary

Abstracts security-specific functions from the DXE Foundation for purposes of handling GUIDed section encapsulations. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and any other DXE drivers that need to validate the authentication of files.

See also Security2 Architectural Protocol section below.

GUID

```
#define EFI_SECURITY_ARCH_PROTOCOL_GUID  \
  {0xA46423E3,0x4617,0x49f1,0xB9,\
  0xFF,0xD1,0xBF,0xA9,0x11,0x58,0x39}
```

Protocol Interface Structure

```
typedef struct _EFI_SECURITY_ARCH_PROTOCOL {
  EFI_SECURITY_FILE_AUTHENTICATION_STATE
                                 FileAuthenticationState;
} EFI_SECURITY_ARCH_PROTOCOL;
```

Parameters

FileAuthenticationState

This service is called upon fault with respect to the authentication of a section of a file. See the **FileAuthenticationState()** function description.

Description

The **EFI_SECURITY_ARCH_PROTOCOL** is used to abstract platform-specific policy from the DXE Foundation. This includes locking flash upon failure to authenticate, attestation logging, and other exception operations.

The driver that produces the **EFI_SECURITY_ARCH_PROTOCOL** may also optionally install the **EFI_SECURITY_POLICY_PROTOCOL_GUID** onto a new handle with a **NULL** interface. The existence of this GUID in the protocol database means that the GUIDed Section Extraction Protocol should authenticate the contents of an Authentication Section. The expectation is that the GUIDed Section Extraction protocol will look for the existence of the **EFI_SECURITY_POLICY_PROTOCOL_GUID** in the protocol database. If it exists, then the publication thereof is taken as an injunction to attempt an authentication of any section wrapped in an Authentication Section. See the *Platform Initialization Specification*, Volume 3, for details on the GUIDed Section Extraction Protocol and Authentication Sections.

Additional GUID Definitions

```
#define EFI_SECURITY_POLICY_PROTOCOL_GUID  \

{0x78E4D245,0xCD4D,0x4a05,0xA2,0xBA,0x47,0x43,0xE8,0x6C,0xFC,0xA
B}
```

EFI_SECURITY_ARCH_PROTOCOL.FileAuthenticationState()

Summary

The DXE Foundation uses this service to check the authentication status of a file. This allows the system to execute a platform-specific policy in response the different authentication status values.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SECURITY_FILE_AUTHENTICATION_STATE) (
  IN  CONST EFI_SECURITY_ARCH_PROTOCOL   *This,
  IN  UINT32                              AuthenticationStatus,
  IN  CONST EFI_DEVICE_PATH_PROTOCOL     *File
  );
```

Parameters

This

The **EFI_SECURITY_ARCH_PROTOCOL** instance.

AuthenticationStatus

The authentication type returned from the Section Extraction Protocol. See the *Platform Initialization Specification*, Volume 3, for details on this type.

File

A pointer to the device path of the file that is being dispatched. This will optionally be used for logging. Type **EFI_DEVICE_PATH_PROTOCOL** is defined Chapter 8 of the UEFI 2.0 specification.

Description

The **EFI_SECURITY_ARCH_PROTOCOL** (SAP) is used to abstract platform-specific policy from the DXE Foundation response to an attempt to use a file that returns a given status for the authentication check from the section extraction protocol.

The possible responses in a given SAP implementation may include locking flash upon failure to authenticate, attestation logging for all signed drivers, and other exception operations. The *File* parameter allows for possible logging within the SAP of the driver.

If *File* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is safe for the DXE Foundation to use, then **EFI_SUCCESS** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is not safe for the DXE Foundation to use under any circumstances, then **EFI_ACCESS_DENIED** is returned.

If the file specified by *File* with an authentication status specified by *AuthenticationStatus* is not safe for the DXE Foundation to use right now, but it might be possible to use it at a future time, then **EFI_SECURITY_VIOLATION** is returned.

Status Codes Returned

EFI_SUCCESS	The file specified by *File* did authenticate, and the platform policy dictates that the DXE Foundation may use *File*.
EFI_INVALID_PARAMETER	*File* is **NULL**.
EFI_SECURITY_VIOLATION	The file specified by *File* did not authenticate, and the platform policy dictates that *File* should be placed in the untrusted state. A file may be promoted from the untrusted to the trusted state at a future time with a call to the **Trust()** DXE Service.
EFI_ACCESS_DENIED	The file specified by *File* did not authenticate, and the platform policy dictates that *File* should not be used for any purpose.

12.9.2 Security2 Architectural Protocol

Summary

Abstracts security-specific functions from the DXE Foundation of UEFI Image Verification, Trusted Computing Group (TCG) measured boot, and User Identity policy for image loading and consoles. This protocol must be produced by a boot service or runtime DXE driver.

This protocol is optional and must be published prior to the **EFI_SECURITY_ARCH_PROTOCOL**. As a result, the same driver must publish both of these interfaces.

When both Security and Security2 Architectural Protocols are published, *LoadImage* must use them in accordance with the following rules:

- The Security2 protocol must be used on every image being loaded.
- The Security protocol must be used after the Securiy2 protocol and only on images that have been read using Firmware Volume protocol.

When only Security architectural protocol is published, *LoadImage* must use it on every image being loaded.

GUID

```
#define EFI_SECURITY2_ARCH_PROTOCOL_GUID \
    {0x94ab2f58, 0x1438, 0x4ef1, 0x91, \
    0x52, 0x18, 0x94, 0x1a, 0x3a, 0xe, 0x68}
```

Protocol Interface Structure

```
typedef struct _EFI_SECURITY2_ARCH_PROTOCOL {
 EFI_SECURITY2_FILE_AUTHENTICATION FileAuthentication;
} EFI_SECURITY2_ARCH_PROTOCOL;
```

Parameters

FileAuthentication

This service is called by DXE Foundation from the *LoadImage* service to verify and/or measure the image and from the *ConnectController* service to probe whether a specific device path can be connected.

Description

The **EFI_SECURITY2_ARCH_PROTOCOL** is used to abstract platform-specific policy from the DXE Foundation. This includes measuring the PE/COFF image prior to invoking, comparing the image against a policy (whether a white-list/black-list of public image verification keys or registered hashes).

EFI_SECURITY2_ARCH_PROTOCOL.FileAuthentication()

Summary

The DXE Foundation uses this service to measure and/or verify a UEFI image.

Prototype

```
typedef EFI_STATUS (EFIAPI
*EFI_SECURITY_FILE_AUTHENTICATION_STATE) (
   IN CONST EFI_SECURITY2_ARCH_PROTOCOL *This,
   IN CONST EFI_DEVICE_PATH_PROTOCOL    *DevicePath,
   IN VOID                              *FileBuffer,
   IN UINTN                             FileSize,
   IN BOOLEAN                           BootPolicy
);
```

Parameters

This

The **EFI_SECURITY2_ARCH_PROTOCOL** instance.

DevicePath

A pointer to the device path of the file that is being dispatched or the location that is being connected. This will optionally be used for logging. Type **EFI_DEVICE_PATH_PROTOCOL** is defined Chapter 9 of the UEFI Specification.

FileBuffer

A pointer to the buffer with the UEFI file image

FileSize

The size of the file.

BootPolicy

A boot policy that was used to call **LoadImage()** UEFI service. If **FileAuthentication()** is invoked not from the **LoadImage()**, *BootPolicy* must be set to **FALSE**.

Description

This service abstracts the invocation of Trusted Computing Group (TCG) measured boot, UEFI Secure boot, and UEFI User Identity infrastructure. For the former two, the DXE Foundation invokes the **FileAuthentication()** with a *DevicePath* and corresponding image in FileBuffer memory. The TCG measurement code will record the *FileBuffer* contents into the appropriate PCR. The image verification logic will confirm the integrity and provenance of the image in *FileBuffer* of length *FileSize*. The origin of the image will be *DevicePath* in these cases.

If *DevicePath* is NULL, the origin of the image is unknown. Implementation of this service must apply to such image security policy that is applied to the image with the least trusted origin.

If the `FileBuffer` is NULL, the interface will determine if the `DevicePath` can be connected in order to support the User Identification policy.

Status Codes Returned

EFI_SUCCESS	The file specified by `DevicePath` and non-NULL `FileBuffer` did authenticate, and the platform policy dictates that the DXE Foundation may use the file
EFI_SUCCESS	The device path specified by NULL device path `DevicePath` and non-NULL `FileBuffer` did authenticate, and the platform policy dictates that the DXE Foundation may execute the image in `FileBuffer`.
EFI_SUCCESS	`FileBuffer` is NULL and current user has permission to start UEFI device drivers on the device path specified by `DevicePath`.
EFI_SECURITY_VIOLATION	The file specified by `DevicePath` and `FileBuffer` did not authenticate, and the platform policy dictates that the file should be placed in the untrusted state. The image has been added tothe file execution table.
EFI_ACCESS_DENIED	The file specified by `File` and `FileBuffer` did not authenticate, and the platform policy dictates that the DXE Foundation many not use `File.`
EFI_SECURITY_VIOLATION	`FileBuffer FileBuffer` is NULL and the user has no permission to start UEFI device drivers on the device path specified by `DevicePath`.
EFI_SECURITY_VIOLATION	`FileBuffer` is not NULL and the user has no permission to load drivers from the device path specified by `DevicePath`. The image has been added into the list of the deferred images.

12.10 Timer Architectural Protocol

EFI_TIMER_ARCH_PROTOCOL

Summary

Used to set up a periodic timer interrupt using a platform specific timer, and a processor-specific interrupt vector. This protocol enables the use of the **`SetTimer()`** Boot Service. This protocol must be produce by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols.

GUID

```
#define EFI_TIMER_ARCH_PROTOCOL_GUID  \
  {0x26baccb3,0x6f42,0x11d4,0xbc,\
  0xe7,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Protocol Interface Structure

```
typedef struct _EFI_TIMER_ARCH_PROTOCOL {
  EFI_TIMER_REGISTER_HANDLER         RegisterHandler;
  EFI_TIMER_SET_TIMER_PERIOD         SetTimerPeriod;
  EFI_TIMER_GET_TIMER_PERIOD         GetTimerPeriod;
  EFI_TIMER_GENERATE_SOFT_INTERRUPT  GenerateSoftInterrupt;
} EFI_TIMER_ARCH_PROTOCOL;
```

Parameters

RegisterHandler

Registers a handler that will be called each time the timer interrupt fires. See the **RegisterHandler()** function description. *TimerPeriod* defines the minimum time between timer interrupts, so *TimerPeriod* will also be the minimum time between calls to the registered handler.

SetTimerPeriod

Sets the period of the timer interrupt in 100 ns units. See the **SetTimerPeriod()** function description. This function is optional and may return **EFI_UNSUPPORTED**. If this function is supported, then the timer period will be rounded up to the nearest supported timer period.

GetTimerPeriod

Retrieves the period of the timer interrupt in 100 ns units. See the **GetTimerPeriod()** function description.

GenerateSoftInterrupt

Generates a soft timer interrupt that simulates the firing of the timer interrupt. This service can be used to invoke the registered handler if the timer interrupt has been masked for a period of time. See the **GenerateSoftInterrupt()** function description.

Description

This protocol provides the services to initialize a periodic timer interrupt and to register a handler that is called each time the timer interrupt fires. It may also provide a service to adjust the rate of the periodic timer interrupt. When a timer interrupt occurs, the handler is passed the amount of time that has passed since the previous timer interrupt.

EFI_TIMER_ARCH_PROTOCOL.RegisterHandler()

Summary

Registers a handler that is called each timer the timer interrupt fires.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_REGISTER_HANDLER) (
  IN CONST EFI_TIMER_ARCH_PROTOCOL   *This,
  IN EFI_TIMER_NOTIFY                NotifyFunction
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

NotifyFunction

The function to call when a timer interrupt fires. This function executes at **EFI_TPL_HIGH_LEVEL**. The DXE Foundation will register a handler for the timer interrupt, so it can know how much time has passed. This information is used to signal timer based events. **NULL** will unregister the handler. Type **EFI_TIMER_NOTIFY** is defined in "Related Definitions" below.

Description

This function registers the handler *NotifyFunction* so it is called every time the timer interrupt fires. It also passes the amount of time since the last handler call to the *NotifyFunction*. If *NotifyFunction* is **NULL**, then the handler is unregistered. If the handler is registered, then **EFI_SUCCESS** is returned. If the processor does not support registering a timer interrupt handler, then **EFI_UNSUPPORTED** is returned. If an attempt is made to register a handler when a handler is already registered, then **EFI_ALREADY_STARTED** is returned. If an attempt is made to unregister a handler when a handler is not registered, then **EFI_INVALID_PARAMETER** is returned. If an error occurs attempting to register the *NotifyFunction* with the timer interrupt, then **EFI_DEVICE_ERROR** is returned.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_TIMER_NOTIFY) (
  IN UINT64  Time
  );
```

Paramters

Time

Time since the last timer interrupt in 100 ns units. This will typically be *TimerPeriod*, but if a timer interrupt is missed, and the **EFI_TIMER_ARCH_PROTOCOL** driver can detect missed interrupts, then *Time* will contain the actual amount of time since the last interrupt.

Status Codes Returned

EFI_SUCCESS	The timer handler was registered.
EFI_UNSUPPORTED	The platform does not support timer interrupts.
EFI_ALREADY_STARTED	*NotifyFunction* is not **NULL**, and a handler is already registered.
EFI_INVALID_PARAMETER	*NotifyFunction* is **NULL**, and a handler was not previously registered.
EFI_DEVICE_ERROR	The timer handler could not be registered.

EFI_TIMER_ARCH_PROTOCOL.SetTimerPeriod()

Summary

Sets the rate of the periodic timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_SET_TIMER_PERIOD) (
  IN   CONST EFI_TIMER_ARCH_PROTOCOL   *This,
  IN   UINT64                          TimerPeriod
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

The rate to program the timer interrupt in 100 ns units. If the timer hardware is not programmable, then **EFI_UNSUPPORTED** is returned. If the timer is programmable, then the timer period will be rounded up to the nearest timer period that is supported by the timer hardware. If *TimerPeriod* is set to 0, then the timer interrupts will be disabled.

Description

This function adjusts the period of timer interrupts to the value specified by *TimerPeriod*. If the timer period is updated, then **EFI_SUCCESS** is returned. If the timer hardware is not programmable, then **EFI_UNSUPPORTED** is returned. If an error occurs while attempting to update the timer period, then the timer hardware will be put back in its state prior to this call, and **EFI_DEVICE_ERROR** is returned. If *TimerPeriod* is 0, then the timer interrupt is disabled. This is not the same as disabling the processor's interrupts. Instead, it must either turn off the timer hardware, or it must adjust the interrupt controller so that a processor interrupt is not generated when the timer interrupt fires.

Status Codes Returned

EFI_SUCCESS	The timer period was changed.
EFI_UNSUPPORTED	The platform cannot change the period of the timer interrupt.
EFI_DEVICE_ERROR	The timer period could not be changed due to a device error.

EFI_TIMER_ARCH_PROTOCOL.GetTimerPeriod()

Summary

Retrieves the rate of the periodic timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_GET_TIMER_PERIOD) (
  IN  CONST EFI_TIMER_ARCH_PROTOCOL   *This,
  OUT UINT64                          *TimerPeriod
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

A pointer to the timer period to retrieve in 100 ns units. If 0 is returned, then the timer is currently disabled.

Description

This function retrieves the period of timer interrupts in 100 ns units, returns that value in *TimerPeriod*, and returns **EFI_SUCCESS**. If *TimerPeriod* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If a *TimerPeriod* of 0 is returned, then the timer is currently disabled.

Status Codes Returned

EFI_SUCCESS	The timer period was returned in *TimerPeriod*.
EFI_INVALID_PARAMETER	*TimerPeriod* is **NULL**.

EFI_TIMER_ARCH_PROTOCOL.GenerateSoftInterrupt()

Summary

Generates a soft timer interrupt.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TIMER_GENERATE_SOFT_INTERRUPT) (
  IN  CONST EFI_TIMER_ARCH_PROTOCOL    *This
  );
```

Parameters

This

The **EFI_TIMER_ARCH_PROTOCOL** instance.

Description

This function generates a soft timer interrupt. If the platform does not support soft timer interrupts, then **EFI_UNSUPPORTED** is returned. Otherwise, **EFI_SUCCESS** is returned. If a handler has been registered through the **EFI_TIMER_ARCH_PROTOCOL.RegisterHandler()** service, then a soft timer interrupt will be generated. If the timer interrupt is enabled when this service is called, then the registered handler will be invoked. The registered handler should not be able to distinguish a hardware-generated timer interrupt from a software-generated timer interrupt.

Status Codes Returned

EFI_SUCCESS	The soft timer interrupt was generated.
EFI_UNSUPPORTED	The platform does not support the generation of soft timer interrupts.

12.11 Variable Architectural Protocol

EFI_VARIABLE_ARCH_PROTOCOL

Summary

Provides the services required to get and set environment variables. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation.

GUID

```
#define EFI_VARIABLE_ARCH_PROTOCOL_GUID  \
  {0x1e5668e2,0x8481,0x11d4,0xbc,\
  0xf1,0x0,0x80,0xc7,0x3c,0x88,0x81}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the **GetVariable()**, **GetNextVariableName()**, **SetVariable()** and **QueryVariableInfo()** fields of the UEFI Runtime Services Table. See “Runtime Capabilities” on page 33 for details on these services. After the three fields of the UEFI Runtime Services Table have been initialized, the driver must install the **EFI_VARIABLE_ARCH_PROTOCOL_GUID** on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the read-only and the volatile environment variable related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table. The full complement of environment variable services are not available until both this protocol and **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** are installed. DXE drivers that require read-only access or read/write access to volatile environment variables must have this architectural protocol in their dependency expressions. DXE drivers that require write access to nonvolatile environment variables must have the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** in their dependency expressions.

12.12 Variable Write Architectural Protocol

EFI_VARIABLE_WRITE_ARCH_PROTOCOL

Summary

Provides the services required to set nonvolatile environment variables. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation.

GUID

```
#define EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID  \
  {0x6441f818,0x6362,0x4e44,0xb5,\
  0x70,0x7d,0xba,0x31,0xdd,0x24,0x53}
```

Description

The DXE driver that produces this protocol must be a runtime driver. This driver may update the **SetVariable()** field of the UEFI Runtime Services Table. See "Runtime Capabilities" on page 33 for details on this service. After the UEFI Runtime Services Table has been initialized, the driver must install the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL_GUID** on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the write services for nonvolatile environment variables are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table. The full complement of environment variable services are not available until both this protocol and **EFI_VARIABLE_ARCH_PROTOCOL** are installed. DXE drivers that require read-only access or read/write access to volatile environment variables must have the **EFI_VARIABLE_WRITE_ARCH_PROTOCOL** in their dependency expressions. DXE drivers that require write access to nonvolatile environment variables must have this architectural protocol in their dependency expressions.

12.13 EFI Capsule Architectural Protocol

EFI_CAPSULE_ARCH_PROTOCOL

Summary

Provides the services for capsule update.

GUID

```
#define EFI_CAPSULE_ARCH_PROTOCOL_GUID \
  { 0x5053697e, 0x2cbc, 0x4819, 0x90, \
  0xd9, 0x5, 0x80, 0xde, 0xee, 0x57, 0x54 }
```

Description

The DXE Driver that produces this protocol must be a runtime driver. The driver is responsible for initializing the **CapsuleUpdate()** and **QueryCapsuleCapabilities()** fields of the UEFI Runtime Services Table. After the two fields of the UEFI Runtime Services Table have been initialized, the driver must install the **EFI_CAPSULE_ARCH_PROTOCOL_GUID** on a new handle with a **NULL** interface pointer. The installation of this protocol informs the DXE Foundation that the Capsule related services are now available and that the DXE Foundation must update the 32-bit CRC of the UEFI Runtime Services Table.

12.14 Watchdog Timer Architectural Protocol

The following topics provide a detailed description of the **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL**. This protocol is used to implement the Boot Service **SetWatchdogTimer()**. The watchdog timer may be implemented in software using Boot Services, or it may be implemented with specialized hardware. The protocol provides a service to register a handler when the watchdog timer fires and a service to set the amount of time to wait before the watchdog timer is fired.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL

Summary

Used to program the watchdog timer and optionally register a handler when the watchdog timer fires. This protocol must be produced by a boot service or runtime DXE driver and may be consumed only by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols. If a platform wishes to perform a platform-specific action when the watchdog timer expires, then the DXE driver that contains the implementation of the **EFI_BDS_ARCH_PROTOCOL** should use this protocol's **RegisterHandler()** service.

GUID

```
#define EFI_WATCHDOG_TIMER_ARCH_PROTOCOL_GUID  \
  {0x665E3FF5,0x46CC,0x11d4,0x9A,\
  0x38,0x00,0x90,0x27,0x3F,0xC1,0x4D}
```

Protocol Interface Structure

```
typedef struct _EFI_WATCHDOG_TIMER_ARCH_PROTOCOL {
  EFI_WATCHDOG_TIMER_REGISTER_HANDLER  RegisterHandler;
  EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD  SetTimerPeriod;
  EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD  GetTimerPeriod;
} EFI_WATCHDOG_TIMER_ARCH_PROTOCOL;
```

Parameters

RegisterHandler

Registers a handler that is invoked when the watchdog timer fires. See the **RegisterHandler()** function description.

SetTimerPeriod

Sets the amount of time in 100 ns units to wait before the watchdog timer is fired. See the **SetTimerPeriod()** function description. If this function is supported, then the watchdog timer period will be rounded up to the nearest supported watchdog timer period.

GetTimerPeriod

Retrieves the amount of time in 100 ns units that the system will wait before the watchdog timer is fired. See the **GetTimerPeriod()** function description.

Description

This protocol provides the services required to implement the Boot Service **SetWatchdogTimer()**. It provides a service to set the amount of time to wait before firing the watchdog timer, and it also provides a service to register a handler that is invoked when the watchdog timer fires. This protocol can implement the watchdog timer by using the event and timer Boot Services, or it can make use of custom hardware. When the watchdog timer fires, control will be passed to a handler if one has been registered. If no handler has been registered, or the registered handler returns, then the system will be reset by calling the Runtime Service **ResetSystem()**.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.RegisterHandler()

Summary

Registers a handler that is to be invoked when the watchdog timer fires.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_REGISTER_HANDLER) (
  IN CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL   *This,
  IN EFI_WATCHDOG_TIMER_NOTIFY                NotifyFunction
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

NotifyFunction

The function to call when the watchdog timer fires. If this is **NULL**, then the handler will be unregistered. Type **EFI_WATCHDOG_TIMER_NOTIFY** is defined in "Related Definitions" below.

Description

This function registers a handler that is to be invoked when the watchdog timer fires. By default, **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** will call the Runtime Service **ResetSystem()** when the watchdog timer fires. If a *NotifyFunction* is registered, then *NotifyFunction* will be called before the Runtime Service **ResetSystem()** is called. If *NotifyFunction* is **NULL**, then the watchdog handler is unregistered. If a watchdog handler is registered, then **EFI_SUCCESS** is returned. If an attempt is made to register a handler when a handler is already registered, then **EFI_ALREADY_STARTED** is returned. If an attempt is made to uninstall a handler when a handler is not installed, then return **EFI_INVALID_PARAMETER**.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_WATCHDOG_TIMER_NOTIFY) (
  IN UINT64   Time
  );
```

Time

The time in 100 ns units that has passed since the watchdog timer was armed. For the notify function to be called, this must be greater than *TimerPeriod*.

Status Codes Returned

EFI_SUCCESS	The watchdog timer handler was registered or unregistered.

EFI_ALREADY_STARTED	*NotifyFunction* is not **NULL**, and a handler is already registered.
EFI_INVALID_PARAMETER	*NotifyFunction* is **NULL**, and a handler was not previously registered.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.SetTimerPeriod()

Summary

Sets the amount of time in the future to fire the watchdog timer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD) (
  IN CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL   *This,
  IN UINT64                                   TimerPeriod
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

The amount of time in 100 ns units to wait before the watchdog timer is fired. If *TimerPeriod* is zero, then the watchdog timer is disabled.

Description

This function sets the amount of time to wait before firing the watchdog timer to *TimerPeriod* 100 ns units. If *TimerPeriod* is zero, then the watchdog timer is disabled.

Status Codes Returned

EFI_SUCCESS	The watchdog timer has been programmed to fire in *Time* 100 ns units.
EFI_DEVICE_ERROR	A watchdog timer could not be programmed due to a device error.

EFI_WATCHDOG_TIMER_ARCH_PROTOCOL.GetTimerPeriod()

Summary

Retrieves the amount of time in 100 ns units that the system will wait before firing the watchdog timer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD) (
  IN  CONST EFI_WATCHDOG_TIMER_ARCH_PROTOCOL   *This,
  OUT UINT64                                   *TimerPeriod
  );
```

Parameters

This

The **EFI_WATCHDOG_TIMER_ARCH_PROTOCOL** instance.

TimerPeriod

A pointer to the amount of time in 100 ns units that the system will wait before the watchdog timer is fired. If *TimerPeriod* of zero is returned, then the watchdog timer is disabled.

Description

This function retrieves the amount of time the system will wait before firing the watchdog timer. This period is returned in *TimerPeriod*, and **EFI_SUCCESS** is returned. If *TimerPeriod* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The amount of time that the system will wait before firing the watchdog timer was returned in *TimerPeriod*.
EFI_INVALID_PARAMETER	*TimerPeriod* is **NULL**.

13 DXE Boot Services Protocol

13.1 Overview

This chapter defines the services required for the Multiprocessor (MP) Services Protocol of Platform Initialization Specification.

This specification does the following:

- Describes the basic components of the MP Services Protocol
- Provides code definitions for the MP Services Protocol and the MP-related type definitions.

13.2 Conventions and Abbreviations

The following terms are used throughout this specification.

AP

Application processor. All other processors in a computer system other than the boot-strap processor are called application processors.

BSP

Boot-strap processor. A processor in an MP platform that is chosen to execute the modules that are necessary for booting the system. It is not necessary that the same processor that was selected earlier as a BSP shall remain a BSP throughout an entire boot session.

DXE

Driver Execute Environment. Environment to support running modular code in the form of EFI drivers; common to all platforms; typically in C language.

EFI

Extensible Firmware Interface – the specification containing interface definitions for firmware. This includes both interfaces used by the operating system for booting as well as interfaces that are used for internal construction of firmware.

MP

Multiprocessor.

13.3 MP Services Protocol Overview

The MP Services Protocol provides a generalized way of performing following tasks:

- Retrieving information of multi-processor environment and MP-related status of specific processors.
- Dispatching user-provided function to APs.
- Maintain MP-related processor status.

The MP Services Protocol must be produced on any system with more than one logical processor.

The Protocol is available only during boot time.

MP Services Protocol is hardware-independent. Most of the logic of this protocol is architecturally neutral. It abstracts the multi-processor environment and status of processors, and provides interfaces to retrieve information, maintain, and dispatch.

MP Services Protocol may be consumed by ACPI module. The ACPI module may use this protocol to retrieve data that are needed for an MP platform and report them to OS.

MP Services Protocol may also be used to program and configure processors, such as MTRR synchronization for memory space attributes setting in DXE Services.

MP Services Protocol may be used by non-CPU DXE drivers to speed up platform boot by taking advantage of the processing capabilities of the APs, for example, using APs to help test system memory in parallel with other device initialization.

Diagnostics applications may also use this protocol for multi-processor.

13.4 MP Services Protocol

This section contains the basic definitions of the MP Services Protocol.

EFI_MP_SERVICES_PROTOCOL

Summary

When installed, the MP Services Protocol produces a collection of services that are needed for MP management.

GUID

```
#define EFI_MP_SERVICES_PROTOCOL_GUID \
  {0x3fdda605,0xa76e,0x4f46,{0xad,0x29,0x12,0xf4,\
  0x53,0x1b,0x3d,0x08}}
```

Protocol Interface Structure

```
typedef struct _EFI_MP_SERVICES_PROTOCOL {
 EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS GetNumberOfProcessors;
 EFI_MP_SERVICES_GET_PROCESSOR_INFO       GetProcessorInfo;
 EFI_MP_SERVICES_STARTUP_ALL_APS          StartupAllAPs;
 EFI_MP_SERVICES_STARTUP_THIS_AP          StartupThisAP;
 EFI_MP_SERVICES_SWITCH_BSP               SwitchBSP;
 EFI_MP_SERVICES_ENABLEDISABLEAP          EnableDisableAP;
 EFI_MP_SERVICES_WHOAMI                   WhoAmI;
} EFI_MP_SERVICES_PROTOCOL;
```

Parameters

GetNumberOfProcessors

Gets the number of logical processors and the number of enabled logical processors in the system.

GetProcessorInfo

Gets detailed information on the requested processor at the instant this call is made.

StartupAllAPs

Starts up all the enabled APs in the system to run the function provided by the caller.

StartupThisAP

Starts up the requested AP to run the function provided by the caller.

SwitchBSP

Switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes.

EnableDisableAP

Enables and disables the given AP from that point onward.

WhoAmI

Gets the handle number of the caller processor.

Description

The MP Services Protocol must be produced on any system with more than one logical processor. Before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** 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 is implementation dependent before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled.

After the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled, all the APs must be placed in the OS compatible CPU state as defined by the UEFI Specification. Implementations of this protocol may use the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** to force APs into the OS compatible state as defined by the UEFI Specification. Modules that use this protocol must guarantee that all non-blocking mode requests on all APs have been completed before the UEFI event **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** is signaled. Since the order that event notification functions in the same event group are executed is not deterministic, an event of type **EFI_EVENT_LEGACY_BOOT_GUID** or **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES** can not be used to guarantee that APs have completed their non-blocking mode requests.

EFI_MP_SERVICES_PROTOCOL.GetNumberOfProcessors()

Summary

This service retrieves the number of logical processor in the platform and the number of those logical processors that are currently enabled. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_NUMBER_OF_PROCESSORS) (
  IN  EFI_MP_SERVICES_PROTOCOL  *This,
  OUT UINTN                     *NumberOfProcessors,
  OUT UINTN                     *NumberOfEnabledProcessors
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

NumberOfProcessors

Pointer to the total number of logical processors in the system, including the BSP and all enabled and disabled APs.

NumberOfEnabledProcessors

Pointer to the number of logical processors in the platform including the BSP that are currently enabled.

Description

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.

Since MP Service Protocol provides services to enable and disable processors dynamically, the number of enabled logical processors may vary during the course of a boot session.

This service may only be called from the BSP.

If this service is called from an AP, then **EFI_DEVICE_ERROR** is returned. If *NumberOfProcessors* or *NumberOfEnabledProcessors* is **NULL**, then **EFI_INVALID_PARAMETER i**s 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_PROTOCOL.GetProcessorInfo()

Summary

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.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_GET_PROCESSOR_INFO) (
  IN  EFI_MP_SERVICES_PROTOCOL    *This,
  IN  UINTN                        ProcessorNumber,
  OUT EFI_PROCESSOR_INFORMATION   *ProcessorInfoBuffer
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of processor. 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_PROTOCOL.GetNumberOfProcessors()**.

ProcessorInfoBuffer

A pointer to the buffer where information for the requested processor is deposited. The buffer is allocated by the caller. Type **EFI_PROCESSOR_INFORMATION** is defined in "Related Definitions" below.

Description

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 data of 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.

This service may only be called from the BSP.

Related Definitions

```
//*******************************************************
// EFI_PROCESSOR_INFORMATION
//*******************************************************
typedef struct {
  UINT64                     ProcessorId;
  UINT32                    StatusFlag;
  EFI_CPU_PHYSICAL_LOCATION Location;
} EFI_PROCESSOR_INFORMATION;
```

ProcessorId

The unique processor ID determined by system hardware.

For IPF, the lower 16 bits contains id/eid, and higher bits are reserved.

StatusFlag

Flags indicating if the processor is BSP or AP, if the processor is enabled or disabled, and if the processor is healthy. The bit format is defined below.

Location

The physical location of the processor, including the physical package number that identifies the cartridge, the physical core number within package, and logical thread number within core. Type **EFI_PHYSICAL_LOCATION** is defined below.

```
//*******************************************************
// StatusFlag Bits Definition
//*******************************************************
#define PROCESSOR_AS_BSP_BIT         0x00000001
#define PROCESSOR_ENABLED_BIT        0x00000002
#define PROCESSOR_HEALTH_STATUS_BIT  0x00000004
```

PROCESSOR_AS_BSP_BIT

This bit indicates whether the processor is playing the role of BSP. If the bit is 1, then the processor is BSP. Otherwise, it is AP.

PROCESSOR_ENABLED_BIT

This bit indicates whether the processor is enabled. If the bit is 1, then the processor is enabled. Otherwise, it is disabled.

PROCESSOR_HEALTH_STATUS_BIT

This bit indicates whether the processor is healthy. If the bit is 1, then the processor is healthy. Otherwise, some fault has been detected for the processor.

Bits 3..31 are reserved and must be 0. The following table shows all the possible combinations of the *StatusFlag* bits:

Table 33. StatusFlag bits

BSP	ENABLED	HEALTH	Description
0	0	0	Unhealthy Disabled AP.
0	0	1	Healthy Disabled AP.
0	1	0	Unhealthy Enabled AP.
0	1	1	Healthy Enabled AP.
1	0	0	Invalid. The BSP can never be in the disabled state.
1	0	1	Invalid. The BSP can never be in the disabled state.
1	1	0	Unhealthy Enabled BSP.
1	1	1	Healthy Enabled BSP.

```
//*********************************************************
// EFI_CPU_PHYSICAL_LOCATION
//*********************************************************
typedef struct {
  UINT32  Package;
  UINT32  Core;
  UINT32  Thread;
} EFI_CPU_PHYSICAL_LOCATION;
```

Package

Zero-based physical package number that identifies the cartridge of the processor.

Core

Zero-based physical core number within package of the processor.

Thread

Zero-based logical thread number within core of the processor.

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_PROTOCOL.StartupAllAPs()

Summary

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 and non-blocking requests. The non-blocking requests use EFI events so the BSP can detect when the APs have finished. See "Non-blocking Execution Support" below for details. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_STARTUP_ALL_APS) (
  IN  EFI_MP_SERVICES_PROTOCOL *This,
  IN  EFI_AP_PROCEDURE          Procedure,
  IN  BOOLEAN                   SingleThread,
  IN  EFI_EVENT                 WaitEvent                OPTIONAL,
  IN  UINTN                     TimeoutInMicroSeconds,
  IN  VOID                      *ProcedureArgument       OPTIONAL,
  OUT UINTN                     **FailedCpuList          OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

Procedure

A pointer to the function to be run on enabled APs of the system. Type **EFI_AP_PROCEDURE** is defined in the “Related Definitions” of this function, below.

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.

WaitEvent

The event created by the caller with **CreateEvent()** service.

If it is **NULL**, then execute in blocking mode. BSP waits until all APs finish or *TimeoutInMicroSeconds* expires.

If it’s not **NULL**, then execute in non-blocking mode. BSP requests the function specified by *Procedure* to be started on all the enabled APs, and go on executing immediately. If all return from *Procedure* or *TimeoutInMicroSeconds* expires, this event is signaled. The BSP can use the **CheckEvent()** or **WaitForEvent()** services to check the state of event.

Type **EFI_EVENT** is defined in **CreateEvent()** in the *Unified Extensible Firmware Interface Specification (Version 2.0).*

TimeoutInMicroseconds

Indicates the time limit in microseconds for APs to return from *Procedure*, either for blocking or non-blocking mode. 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_PROTOCOL.StartupAllAPs()** or **EFI_MP_SERVICES_PROTOCOL.StartupThisAP().**

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

If the timeout expires in non-blocking mode, *WaitEvent* is signaled with **SignalEvent()**.

ProcedureArgument

The parameter passed into *Procedure* for all APs.

FailedCpuList

If **NULL**, this parameter is ignored.

Otherwise, if all APs finish successfully, then its content is set to **NULL**. If not all APs finish before timeout expires, then its content is set to address of the buffer holding handle numbers of the failed APs. The buffer is allocated by MP Service Protocol, and it's the caller's responsibility to free the buffer with **FreePool()** service.

In blocking mode, it is ready for consumption when the call returns. In non-blocking mode, it is ready when *WaitEvent* is signaled.

The list of failed CPU is terminated by **END_OF_CPU_LIST**. It is defined in "Related Definitions" below.

Description

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 *WaitEvent* is **NULL**, execution is in blocking mode. The BSP waits until all APs finish or *TimeoutInMicroSecs* expires. Otherwise, execution is in non-blocking mode, and the BSP returns from this service without waiting for APs. If a non-blocking mode is requested after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled, then **EFI_UNSUPPORTED** must be returned.

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_PROTOCOL.StartupAllAPs()**and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP().** If *FailedCpuList* is not **NULL**, its content points to the list of processor handle numbers in which Procedure was terminated.

This service may only be called from the BSP.

Note: *It is the responsibility of the consumer of the* **EFI_MP_SERVICES_PROTOCOL.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 Protocol 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. EFI services and protocols may not be called by APs unless otherwise specified.*

Related Definitions

```
#define END_OF_CPU_LIST    0xffffffff

typedef
VOID
(EFIAPI *EFI_AP_PROCEDURE) (
  IN VOID  *ProcedureArgument
);
```

ProcedureArgument

Pointer to the procedure's argument

Non-Blocking Execution Support

The following usage guidelines must be followed for non-blocking execution support.

In blocking execution mode, BSP waits until all APs finish or *TimeoutInMicroSeconds* expires.

In non-blocking execution mode, BSP is freed to return to the caller and then proceed to the next task without having to wait for APs. The following sequence needs to occur in a non-blocking execution mode:

1. The caller that intends to use this MP Services Protocol in non-blocking mode creates *WaitEvent* by calling the EFI **CreateEvent()** service.

The caller invokes **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()**. If the parameter *WaitEvent* is not **NULL**, then **StartupAllAPs()** executes in non-blocking mode. It requests the function specified by *Procedure* to be started on all the enabled APs, and releases the BSP to continue with other tasks.

2. The caller can use the **CheckEvent()** and **WaitForEvent()** services to check the state of the *WaitEvent* created in step 1.
3. When the APs complete their task or *TimeoutInMicroSecondss* expires, the MP Service signals *WaitEvent* by calling the EFI **SignalEvent()** function. If *FailedCpuList* is not **NULL**, its content is available when *WaitEvent* is signaled. If all APs returned from *Procedure* prior to the timeout, then *FailedCpuList* is set to **NULL**. If not all APs return from *Procedure* before the timeout, then *FailedCpuList* is filled in with the list of the failed APs. The buffer is allocated by MP Service Protocol using **AllocatePool()**. It is the caller's responsibility to free the buffer with **FreePool()** service.
4. This invocation of **SignalEvent()** function informs the caller that invoked **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** that either all the APs completed

the specified task or a timeout occurred. The contents of *FailedCpuList* can be examined to determine which APs did not complete the specified task prior to the timeout.

Status Codes Returned

EFI_SUCCESS	In blocking mode, all APs have finished before the timeout expired.
EFI_SUCCESS	In non-blocking mode, function has been dispatched to all enabled APs.
EFI_UNSUPPORTED	A non-blocking mode request was made after the UEFI event **EFI_EVENT_GROUP_READY_TO_BOOT** was signaled.
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_PROTOCOL.StartupThisAP()

Summary

This service lets the caller get one enabled AP to execute a caller-provided function. The caller can request the BSP to either wait for the completion of the AP or just proceed with the next task by using the EFI event mechanism. See the "Non-blocking Execution Support" section in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** for more details. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_STARTUP_THIS_AP) (
  IN   EFI_MP_SERVICES_PROTOCOL*This,
  IN   EFI_AP_PROCEDURE        Procedure,
  IN   UINTN                   ProcessorNumber,
  IN   EFI_EVENT               WaitEvent               OPTIONAL,
  IN   UINTN                   TimeoutInMicroseconds,
  IN   VOID                    *ProcedureArgument      OPTIONAL,
  OUT  BOOLEAN                 *Finished               OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

Procedure

A pointer to the function to be run on the designated AP. Type **EFI_AP_PROCEDURE** is defined in **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()**.

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_PROTOCOL.GetNumberOfProcessors()**.

WaitEvent

The event created by the caller with **CreateEvent()** service.

If it is **NULL**, then execute in blocking mode. BSP waits until this AP finishes or *TimeoutInMicroSeconds* expires.

If it's not **NULL**, then execute in non-blocking mode. BSP requests the function specified by *Procedure* to be started on the AP, and go on executing immediately. If this AP finishes or *TimeoutInMicroSeconds* expires, this event is signaled. BSP can use the **CheckEvent()** and **WaitForEvent()** services to check the state of event.

Type **EFI_EVENT** is defined in **CreateEvent()** in the *Unified Extensible Firmware Interface Specification (Version 2.0)*

TimeoutInMicrosecsond

Indicates the time limit in microseconds for this AP to finish the function, either for blocking or non-blocking mode. Zero means infinity.

If the timeout expires before this AP returns from Procedure, then Procedure on the AP is terminated. The AP is available for subsequent calls to **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP()**.

If the timeout expires in blocking mode, BSP returns **EFI_TIMEOUT**.

If the timeout expires in non-blocking mode, *WaitEvent* is signaled with **SignalEvent()**.

ProcedureArgument

The parameter passed into *Procedure* on the specified AP.

Finished

If **NULL**, this parameter is ignored.

In blocking mode, this parameter is ignored.

In non-blocking mode, if AP returns from *Procedure* before the timeout expires, its content is set to **TRUE.** Otherwise, the value is set to **FALSE**. The caller can determine if the AP returned from *Procedure* by evaluating this value.

Description

This function is used to dispatch one enabled AP to the function specified by *Procedure* passing in the argument specified by *ProcedureArgument*.

If *WaitEvent* is **NULL**, execution is in blocking mode. The BSP waits until the AP finishes or *TimeoutInMicroSecondss* expires. Otherwise, execution is in non-blocking mode. BSP proceeds to the next task without waiting for the AP. If a non-blocking mode is requested after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled, then **EFI_UNSUPPORTED** must be returned.

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_PROTOCOL.StartupAllAPs()** and **EFI_MP_SERVICES_PROTOCOL.StartupThisAP()**.

This service may only be called from the BSP.

Status Codes Returned

EFI_SUCCESS	In blocking mode, specified AP finished before the timeout expires.
EFI_SUCCESS	In non-blocking mode, the function has been dispatched to specified AP.

EFI_UNSUPPORTED	A non-blocking mode request was made after the UEFI event **EFI_EVENT_GROUP_READY_TO_BOOT** was signaled.
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_READY	The specified AP is busy.
EFI_NOT_FOUND	The processor with the handle specified by *ProcessorNumber* does not exist.
EFI_INVALID_PARAMETER	*ProcessorNumber* specifies the BSP or disabled AP.
EFI_INVALID_PARAMETER	*Procedure* is **NULL**.

EFI_MP_SERVICES_PROTOCOL.SwitchBSP()

Summary

This service switches the requested AP to be the BSP from that point onward. This service changes the BSP for all purposes. This service may only be called from the current BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_SWITCH_BSP) (
  IN EFI_MP_SERVICES_PROTOCOL    *This,
  IN UINTN                       ProcessorNumber,
  IN BOOLEAN                     EnableOldBSP
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** 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_PROTOCOL.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. The new BSP can take over the execution of the old BSP and continue seamlessly from where the old one left off. This service may not be supported after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled.

If the BSP cannot be switched prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

This call can only be performed by the current BSP.

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_DEVICE_ERROR	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.

EFI_MP_SERVICES_PROTOCOL.EnableDisableAP()

Summary

This service lets the caller enable or disable an AP from this point onward. This service may only be called from the BSP.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_ENABLEDISABLEAP) (
  IN EFI_MP_SERVICES_PROTOCOL*This,
  IN UINTN                    ProcessorNumber,
  IN BOOLEAN                  EnableAP,
  IN UINT32                   *HealthFlag  OPTIONAL
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

The handle number of 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_PROTOCOL.GetNumberOfProcessors()**.

EnableAP

Specifies the new state for the processor specified by *ProcessorNumber*. **TRUE** 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_PROTOCOL.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 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. This service may not be supported after the UEFI Event **EFI_EVENT_GROUP_READY_TO_BOOT** is signaled.

If the enable or disable AP operation cannot be completed prior to the return from this service, then **EFI_UNSUPPORTED** must be returned.

This service may only be called from the BSP.

Status Codes Returned

EFI_SUCCESS	The specified AP successfully enabled or disabled.
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_PROTOCOL.WhoAmI()

Summary

This return the handle number for the calling processor. This service may be called from the BSP and APs.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MP_SERVICES_WHOAMI) (
  IN  EFI_MP_SERVICES_PROTOCOL  *This,
  OUT UINTN                     *ProcessorNumber
  );
```

Parameters

This

A pointer to the **EFI_MP_SERVICES_PROTOCOL** instance.

ProcessorNumber

Pointer to the handle number of 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_PROTOCOL.GetNumberOfProcessors()**.

Description

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_PROTOCOL.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**.

14 DXE Runtime Protocols

14.1 Introduction

In addition to the architectural protocols listed earlier, there is also a runtime protocol. Specifically, the ability to report status codes is runtime-callable service that allows for emitting status and progress information. It was formerly part of the 0.9 DXE-CIS runtime table, but in consideration of UEFI 2.0 compatibility, this capability has become a separate runtime protocol.

14.2 Status Code Runtime Protocol

EFI_STATUS_CODE_ PROTOCOL

Summary

Provides the service required to report a status code to the platform firmware. This protocol must be produced by a runtime DXE driver.

GUID

```
#define EFI_STATUS_CODE_RUNTIME_PROTOCOL_GUID  \
  { 0xd2b2b828, 0x826, 0x48a7, 0xb3, 0xdf, 0x98, 0x3c, \
  0x0, 0x60, 0x24, 0xf0}
```

Protocol Interface Structure

```
typedef struct _EFI_STATUS_CODE_PROTOCOL {
  EFI_REPORT_STATUS_CODE         ReportStatusCode;
} EFI_STATUS_CODE_PROTOCOL;
```

Parameters

ReportStatusCode

Emit a status code.

Description

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for providing the **ReportStatusCode()** service with the **EFI_STATUS_CODE_PROTOCOL**.

EFI_STATUS_CODE_PROTOCOL.ReportStatusCode()

Summary

Provides an interface that a software module can call to report a status code.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_REPORT_STATUS_CODE) (
  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

Type

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

Value

Describes the current status of a hardware or software entity. This included 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. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Data

This optional parameter may be used to pass additional data. Type **EFI_STATUS_CODE_DATA** is defined in volume 3 of this specification. The contents of this data type may have additional GUID-specific data.

Description

Various software modules including drivers can call this function to report a status code. No disposition of the status code is guaranteed. The **ReportStatusCode()** function may choose to log the status code, but this action is not required.

It is possible that this function may get called at **EFI_TPL_LEVEL_HIGH**. Therefore, this function cannot call any protocol interface functions or services (including memory allocation) that are not guaranteed to work at **EFI_TPL_LEVEL_HIGH**. It should be noted that **SignalEvent()** could be called by this function because it works at any TPL including **EFI_TPL_L**

EVEL_HIGH. It is possible for an implementation to use events to log the status codes when the TPL level is reduced.

ReportStatusCode() function can perform other implementation specific work, but that is not specified in the architecture document.

In case of an error, the caller can specify the severity. In most cases, the entity that reports the error may not have a platform wide view and may not be able to accurately assess the impact of the error condition. The DXE driver that produces the Status Code Protocol, **EFI_STATUS_CODE_PROTOCOL**, is responsible for assessing the true severity level based on the reported severity and other information. This DXE driver may perform platform specific actions based on the type and severity of the status code being reported.

If *Data* is present, the Status Code Protocol driver treats it as read only data. The Status Code Protocol driver must copy *Data* to a local buffer in an atomic operation before performing any other actions. This is necessary to make this function re-entrant. The size of the local buffer may be limited. As a result, some of the *Data* can be lost. The size of the local buffer should at least be 256 bytes in size. Larger buffers will reduce the probability of losing part of the *Data*. Note than multiple status codes may be reported at elevated TPL levels before the TPL level is reduced. Allocating multiple local buffers may reduce the probability losing status codes at elevated TPL levels. If all of the local buffers are consumed, then this service may not be able to perform the platform specific action required by the status code being reported. As a result, if all the local buffers are consumed, the behavior of this service is undefined.

If the *CallerId* parameter is not **NULL**, then it is required to point to a constant GUID. In other words, the caller may not reuse or release the buffer pointed to by *CallerId*.

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
```

Parameters

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_DEVICE_ERROR	The function should not be completed due to a device error.

15 Dependency Expression Grammar

15.1 Dependency Expression Grammar

This topic contains an example BNF grammar for a DXE driver dependency expression compiler that converts a dependency expression source file into a dependency section of a DXE driver stored in a firmware volume.

15.2 Example Dependency Expression BNF Grammar

```
<depex>    ::= BEFORE <guid>
             | AFTER <guid>
             | SOR <bool>
             | <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]
```

15.3 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_CPU_IO_PROTOCOL_GUID AND EFI_CPU_ARCH_PROTOCOL_GUID END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                            MNEMONIC
====   =======================
===============================================
0x00 : 02                                PUSH
0x01 : 26 25 73 b0 c8 38 40 4b    EFI_CPU_IO_PROTOCOL_GUID
       88 77 61 c7 b0 6a ac 45
0x11 : 02                                PUSH
0x12 : b1 cc ba 26 42 6f d4 11    EFI_CPU_ARCH_PROTOCOL_GUID
       bc e7 00 80 c7 3c 88 81
0x22 : 03                                AND
0x23 : 08                                END

//
// Source
//
AFTER (EFI_CPU_DRIVER_FILE_NAME_GUID) END

//
// Opcodes, Operands, and Binary Encoding
//
ADDR   BINARY                            MNEMONIC
====   =======================
===============================================
0x00 : 01                                AFTER
0x01 : 93 e5 7b 98 43 16 0b 45    EFI_CPU_DRIVER_FILE_NAME_GUID
       be 4f 8f 07 66 6e 36 56
0x11 : 08                                END

//
// Source
//
SOR EFI_CPU_IO_PROTOCOL_GUID END

//
// Opcodes, Operands and Binary Encoding
//
ADDR   BINARY                            MNEMONIC
```

```
====   =======================
===============================================
0x00 : 09                         SOR
0x01 : 02                         PUSH
0x02 : b1 cc ba 26 42 6f d4 11    EFI_CPU_IO_PROTOCOL_GUID
       bc e7 00 80 c7 3c 88 81
0x12 : 03                         END
```

Appendix A Error Codes

```
#define DXE_ERROR(a)                    (MAX_BIT|MAX_BIT >> 2 | (a))
```

EFI_REQUEST_UNLOAD_IMAGE	DXE_ERROR (1)	If this value is returned by an EFI image, then the image should be unloaded.
EFI_NOT_AVAILABLE_YET	DXE_ERROR (2)	If this value is returned by an API, it means the capability is not yet installed/available/ready to use.

Appendix B GUID Definitions

B.1 DXE Services Table GUID

```
#define DXE_SERVICES_TABLE_GUID  \
  {0x5ad34ba,0x6f02,0x4214,0x95,0x2e,0x4d,0xa0,
  0x39,0x8e,0x2b,0xb9}
```

B.2 HOB List GUID

```
#define HOB_LIST_GUID  \
  {0x7739f24c,0x93d7,0x11d4,0x9a,0x3a,0x0,0x90,\
  0x27,0x3f,0xc1,0x4d}
```

Platform Initialization (PI) Specification

Volume 3:
Shared Architectural Elements

Version 1.7
January 2019

Revision History

Revision	Mantis ID / Description	Date
1.7	• 1848 PEI Core PEIM Migration Support Change • 1856 SecCore/PeiCore BFV Requirement Change • 1860 MM MP Protocol issues • 1885 Add extended data for EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR • 1889 New Status Codes • 1891 PEI delayed dispatch • 1892 new pi spec revision • 1955 set (*Attributes) when EFI_PEI_GET_VARIABLE2 returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL • 1967 New architectural PPI for PI PEI Core FV Location • 1969 Incompatible Pci Ignore Option Rom	January 2019
1.6 Errata A	• 1828 Add decorator 'OPTIONAL' for Attributes parameter of EFI_PEI_GET_VARIABLE2. • 1849 Issues in PI Spec Vol. 5 Ch. 18 • 1857 Specifies how notifications are passed from SEC to PEI. • 1884 Variadic API issue in S3 save API • 1907 Clarification of the EFI_MM_COMMUNICATION_PROTOCOL • 1940 Clarify EFI_MM_CONFIGURATION_PROTOCOL_GUID	December 2018
1.6	• 1567 Layered SPI bus • 1648 PI Binding for RISC-V • 1746 Add a FV Extended Header entry the contains the used size of the FV • 1763 MM Handler state notification protocol • 1764 Add additional alignment • 1768 Update the PI Spec to 1.6 • 1777 Update Revision History for PI Spec 1.6 • 1778 Update front matter for PI spec	April 2017
1.5 Errata A	• 1587 pre permanent memory page allocation • 1665 Incorrect status code for an AP calling EFI_MP_SERVICES_PROTOCOL.SwitchBSP() • 1734 outdated EFI spec reference • 1735 fix copy & paste errors • 1747 Clarify that MM_ACCESS_PROTOCOL should cover all MMRAM region used by the platform	April 2017

Revision	Mantis ID / Description	Date
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 • 1593 coalesce language enhancements • 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 • 1607 Update MM guid def'n to match edkII impl • 1626 Add new Status Code for BDS Attempting UEFI BootOrder entries • 1628 Minor feedback for PI 1.5 Vol 4 SMM Draft • 1666 Graphics Device Info Hob	4/26/16
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

Revision	Mantis ID / Description	Date
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.	2/20/15
1.3 Errata A	• 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
1.3	• 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

Revision	Mantis ID / Description	Date
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 • 937SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 • 969Vol 1 errata: TE Header parameters	10/26/12
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
1.2.1	• 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

Revision	Mantis ID / Description	Date
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 • 737 Remove SMM Communication ACPI Table definition . • 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 • 753 SIO PEI and UEFI-Driver Model Architecture • 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

Revision	Mantis ID / Description	Date
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 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	• 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

Revision	Mantis ID / Description	Date
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	• 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

Revision	Mantis ID / Description	Date
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
1.1 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

Revision	Mantis ID / Description	Date
1.1 Errata	• 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	Revises typographical errors and minor omissions--see Errata for details	04/25/08
1.1 correction	Restore (missing) MP protocol	03/12/08
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

Revision	Mantis ID / Description	Date
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
1.0	Initial public release.	8/21/06

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.

Table of Contents

Revision History.. iii
Table of Contents... xiii
List of Tables... xxv
List of Figures xxvii

1 Shared Architectural Elements 1
1.1 Overview 1
1.2 Target Audience 1
1.3 Conventions Used in this Document 1
1.3.1 Data Structure Descriptions 1
1.3.2 Pseudo-Code Conventions 2
1.3.3 Typographic Conventions 2
1.4 Conventions used in this document 4
1.4.1 Number formats 4
1.4.2 Binary prefixes 5

2 Firmware Storage Design Discussion 6
2.1 Firmware Storage Introduction 6
2.1.1 Firmware Devices 6
2.1.2 Firmware Volumes 6
2.1.3 Firmware File System 7
2.1.4 Firmware Files 7
2.1.5 Firmware File Sections 14
2.2 PI Architecture Firmware File System Format 16
2.2.1 Firmware Volume Format 17
2.2.2 Firmware File System Format 17
2.2.3 Firmware File Format 18
2.2.4 Firmware File Section Format 20
2.2.5 File System Initialization 20
2.2.6 Traversal and Access to Files 25
2.2.7 File Integrity and State 26
2.2.8 File State Transitions 27

3 Firmware Storage Code Definitions.. 32
3.1 Firmware Storage Code Definitions Introduction 32
3.2 Firmware Storage Formats 32
3.2.1 Firmware Volume 32
EFI_FIRMWARE_VOLUME_HEADER 32
3.2.2 Firmware File System 39
EFI_FIRMWARE_FILE_SYSTEM2_GUID 39
EFI_FIRMWARE_FILE_SYSTEM3_GUID 39
EFI_FFS_VOLUME_TOP_FILE_GUID 40
3.2.3 Firmware File 41
EFI_FFS_FILE_HEADER 41
3.2.4 Firmware File Section 47
EFI_COMMON_SECTION_HEADER 47
3.2.5 Firmware File Section Types 49
EFI_SECTION_COMPATIBILITY16 49
EFI_SECTION_COMPRESSION 50
EFI_SECTION_DISPOSABLE 52
EFI_SECTION_DXE_DEPEX 53
EFI_SECTION_FIRMWARE_VOLUME_IMAGE 54
EFI_SECTION_FREEFORM_SUBTYPE_GUID 55
EFI_SECTION_GUID_DEFINED 56
EFI Signed Sections 58
EFI_SECTION_PE32 60
EFI_SECTION_PEI_DEPEX 61
EFI_SECTION_PIC 62
EFI_SECTION_RAW 63
EFI_SECTION_MM_DEPEX 64
EFI_SECTION_TE 65
EFI_SECTION_USER_INTERFACE 66
EFI_SECTION_VERSION 67
3.3 PEI 68
EFI_PEI_FIRMWARE_VOLUME_INFO_PPI 68
EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI 69
3.3.1 PEI Firmware Volume PPI 70
EFI_PEI_FIRMWARE_VOLUME_PPI 70
EFI_PEI_FIRMWARE_VOLUME_PPI.ProcessVolume() 72
EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByType() 73
EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByName() 74
EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo() 75
EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo2() 76
EFI_PEI_FIRMWARE_VOLUME_PPI.GetVolumeInfo() 77
EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType() 78
EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType2() 79
3.3.2 PEI Load File PPI 80
EFI_PEI_LOAD_FILE_PPI 80
EFI_PEI_LOAD_FILE_PPI.LoadFile() 81
3.3.3 PEI Guided Section Extraction PPI 82

EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI 82
EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection() 84
3.3.4 PEI Decompress PPI 86
EFI_PEI_DECOMPRESS_PPI.. 86
EFI_PEI_DECOMPRESS_PPI.Decompress() 88
3.4 DXE.. 89
3.4.1 Firmware Volume2 Protocol.. 89
EFI_FIRMWARE_VOLUME2_PROTOCOL... 89
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes() 91
EFI_FIRMWARE_VOLUME2_PROTOCOL.SetVolumeAttributes().................... 95
EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadFile() 97
EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadSection().............................. 102
EFI_FIRMWARE_VOLUME2_PROTOCOL.WriteFile() 104
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetNextFile() 107
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetInfo()....................................... 109
EFI_FIRMWARE_VOLUME2_PROTOCOL.SetInfo() 111
3.4.2 Firmware Volume Block2 Protocol 112
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL....................................... 112
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetAttributes() 114
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.SetAttributes()................ 115
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetPhysicalAddress().... 116
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetBlockSize().............. 117
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Read()........................... 118
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Write()........................... 120
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.EraseBlocks() 122
3.4.3 Guided Section Extraction Protocol 123
EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL 123
EFI_GUIDED_SECTION_EXTRACTION_PROCOCOL.ExtractSection() 124
3.5 SMM.. 125
3.5.1 SMM Firmware Volume Protocol 125
EFI_SMM_FIRMWARE_VOLUME_PROTOCOL 125
3.5.2 SMM Firmware Volume Block Protocol... 126
EFI_SMM_FIRMWARE_VOLUME_BLOCK_PROTOCOL 126

4 HOB Design Discussion 127
4.1 Explanation of HOB Terms 127
4.2 HOB Overview 127
4.3 Example HOB Producer Phase Memory Map and Usage 128
4.4 HOB List........ 128
4.5 Constructing the HOB List 129
4.5.1 Constructing the Initial HOB List 129
4.5.2 HOB Construction Rules 129
4.5.3 Adding to the HOB List........ 130

5 HOB Code Definitions 131
5.1 HOB Introduction 131
5.2 HOB Generic Header 132
EFI_HOB_GENERIC_HEADER 132
5.3 PHIT HOB 134
EFI_HOB_HANDOFF_INFO_TABLE (PHIT HOB) 134
5.4 Memory Allocation HOB 136
5.4.1 Memory Allocation HOB 136
EFI_HOB_MEMORY_ALLOCATION 136
5.4.2 Boot-Strap Processor (BSP) Stack Memory Allocation HOB 139
EFI_HOB_MEMORY_ALLOCATION_STACK 139
5.4.3 Boot-Strap Processor (BSP) BSPSTORE Memory Allocation HOB 141
EFI_HOB_MEMORY_ALLOCATION_BSP_STORE 141
5.4.4 Memory Allocation Module HOB 142
EFI_HOB_MEMORY_ALLOCATION_MODULE 142
5.5 Resource Descriptor HOB 143
EFI_HOB_RESOURCE_DESCRIPTOR 143
5.6 GUID Extension HOB 150
EFI_HOB_GUID_TYPE 150
5.7 Firmware Volume HOB 151
EFI_HOB_FIRMWARE_VOLUME 151
EFI_HOB_FIRMWARE_VOLUME2 152
EFI_HOB_FIRMWARE_VOLUME3 153
5.8 CPU HOB 155
EFI_HOB_CPU 155
5.9 Memory Pool HOB 156
EFI_HOB_MEMORY_POOL 156
5.10 UEFI Capsule HOB 156
EFI_HOB_UEFI_ CAPSULE 156
5.11 Unused HOB 158
EFI_HOB_TYPE_UNUSED 158
5.12 End of HOB List HOB 159
EFI_HOB_TYPE_END_OF_HOB_LIST 159
5.13 SMRAM Memory Hob 159
EFI_SMRAM_HOB_DESCRIPTOR_BLOCK 159

6 Status Codes..161
6.1 Status Codes Overview..161
6.1.1 Organization of the Status Codes Specification..161
6.2 Terms..161
6.3 Types of Status Codes..162
6.3.1 Status Code Classes..164
6.3.2 Instance Number..164
6.4 Hardware Classes..165
6.4.1 Computing Unit Class..165
6.4.2 User-Accessible Peripheral Class..176
6.4.3 I/O Bus Class..186
6.5 Software Classes..197
6.5.1 Host Software Class..197
6.5.2 Instance Number..197
6.5.3 Progress Code Operations..198
6.5.4 Error Code Operations..198
6.5.5 Subclasses..200
6.5.6 Runtime (RT) Subclass..211
6.6 Code Definitions..221
6.6.1 Data Structures..221
6.6.2 Extended Data Header..221
EFI_STATUS_CODE_DATA..221
EFI_STATUS_CODE_DATA_TYPE_STRING_GUID..223
EFI_STATUS_CODE_SPECIFIC_DATA_GUID..226
6.6.3 Enumeration Schemes..226
6.6.4 Common Extended Data Formats..228
EFI_DEVICE_PATH_EXTENDED_DATA..228
EFI_DEVICE_HANDLE_EXTENDED_DATA..230
EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA..231
6.7 Class Definitions..232
6.7.1 Computing Unit Class..233
EFI_COMPUTING_UNIT_VOLTAGE_ERROR_DATA..240
EFI_COMPUTING_UNIT_MICROCODE_UPDATE_ERROR_DATA..242
EFI_COMPUTING_UNIT_TIMER_EXPIRED_ERROR_DATA..243
EFI_HOST_PROCESSOR_MISMATCH_ERROR_DATA..244
EFI_COMPUTING_UNIT_THERMAL_ERROR_DATA..246
EFI_CACHE_INIT_DATA..247
EFI_COMPUTING_UNIT_CPU_DISABLED_ERROR_DATA..248
EFI_MEMORY_EXTENDED_ERROR_DATA..249
EFI_STATUS_CODE_DIMM_NUMBER..252
EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA..254
EFI_MEMORY_RANGE_EXTENDED_DATA..255
6.7.2 User-Accessible Peripherals Class..255
6.7.3 I/O Bus Class..264
6.7.4 Software Classes..271
EFI_DEBUG_ASSERT_DATA..292
EFI_STATUS_CODE_EXCEP_EXTENDED_DATA..293

EFI_STATUS_CODE_START_EXTENDED_DATA 295
EFI_LEGACY_OPROM_EXTENDED_DATA 296
EFI_RETURN_STATUS_EXTENDED_DATA 297

7 Report Status Code Routers 298
7.1 Overview 298
7.2 Code Definitions 298
7.2.1 Report Status Code Handler Protocol 298
EFI_RSC_HANDLER_PROTOCOL 298
EFI_RSC_HANDLER_PROTOCOL.Register() 300
EFI_RSC_HANDLER_PROTOCOL.Unregister() 301
7.2.2 Report Status Code Handler PPI 301
EFI_PEI_RSC_HANDLER_PPI 301
EFI_PEI_RSC_HANDLER_PPI.Register() 303
EFI_PEI_RSC_HANDLER_PPI.Unregister() 304
7.2.3 SMM Report Status Code Handler Protocol 304
EFI_SMM_RSC_HANDLER_PROTOCOL 304
EFI_SMM_RSC_HANDLER_PROTOCOL.Register() 306
EFI_SMM_RSC_HANDLER_PROTOCOL.Unregister() 307

8 PCD 308
8.1 PCD Protocol Definitions 308
8.1.1 PCD Protocol 308
EFI_PCD_PROTOCOL 308
EFI_PCD_PROTOCOL.SetSku () 312
EFI_PCD_PROTOCOL.Get8 () 313
EFI_PCD_PROTOCOL.Get16 () 314
EFI_PCD_PROTOCOL.Get32 () 315
EFI_PCD_PROTOCOL.Get64 () 316
EFI_PCD_PROTOCOL.GetPtr () 317
EFI_PCD_PROTOCOL.GetBool () 318
EFI_PCD_PROTOCOL.GetSize () 319
EFI_PCD_PROTOCOL.Set8 () 320
EFI_PCD_PROTOCOL.Set16 () 321
EFI_PCD_PROTOCOL.Set32 () 322
EFI_PCD_PROTOCOL.Set64 () 323
EFI_PCD_PROTOCOL.SetPtr () 324
EFI_PCD_PROTOCOL.SetBool () 325
EFI_PCD_PROTOCOL.CallbackOnSet () 326
EFI_PCD_PROTOCOL.CancelCallback () 327
EFI_PCD_PROTOCOL.GetNextToken () 328
EFI_PCD_PROTOCOL.GetNextTokenSpace () 329
8.1.2 Get PCD Information Protocol 329
EFI_GET_PCD_INFO_PROTOCOL 329
EFI_GET_PCD_INFO_PROTOCOL.GetInfo () 331
EFI_GET_PCD_INFO_PROTOCOL.GetSku () 333
8.2 PCD PPI Definitions 333
8.2.1 PCD PPI 333
EFI_PEI_PCD_PPI 333
EFI_PEI_PCD_PPI.SetSku () 336
EFI_PEI_PCD_PPI.Get8 () 337
EFI_PEI_PCD_PPI.Get16 () 338
EFI_PEI_PCD_PPI.Get32 () 339
EFI_PEI_PCD_PPI.Get64 () 340
EFI_PEI_PCD_PPI.GetPtr () 341
EFI_PEI_PCD_PPI.GetBool () 342
EFI_PEI_PCD_PPI.GetSize () 343
EFI_PEI_PCD_PPI.Set8 () 344
EFI_PEI_PCD_PPI.Set16 () 345
EFI_PEI_PCD_PPI.Set32 () 346
EFI_PEI_PCD_PPI.Set64 () 347
EFI_PEI_PCD_PPI.SetPtr () 348
EFI_PEI_PCD_PPI.SetBool() 350
EFI_PEI_PCD_PPI.CallbackOnSet () 351
EFI_PEI_PCD_PPI.CancelCallback () 352
EFI_PEI_PCD_PPI.GetNextToken () 353
EFI_PEI_PCD_PPI.GetNextTokenSpace () 354

8.2.2 Get PCD Information PPI 354
EFI_GET_PCD_INFO_PPI 354
EFI_GET_PCD_INFO_PPI.GetInfo () 356
EFI_GET_PCD_INFO_PPI.GetSku () 357

List of Tables

SI prefixes 5
Binary prefixes 5
Defined File Types 9
Architectural Section Types 16
Descriptions of **EFI_FVB_ATTRIBUTES_2** 36
Bit Allocation Definitions 45
Supported FFS Alignments 46
Description of Fields for *CompressionType* 51
Descriptions of Fields for *GuidedSectionHeader.Attributes* 58
AuthenticationStatus Bit Definitions 86
Descriptions of Fields for **EFI_FV_ATTRIBUTES** 94
Supported Alignments for **EFI_FV_FILE_ATTRIB_ALIGNMENT** 100
Description of fields for EFI_FV_WRITE_POLICY 105
Translation of HOB Specification Terminology 127
EFI_RESOURCE_ATTRIBUTE_TYPE fields 147
HOB Producer Phase Resource Types 149
Organization of This Specification 161
Class Definitions 164
Progress Code Operations: Computing Unit Class 165
Error Code Operations: Computing Unit Class 166
Computing Unit Class: Subclasses 167
Progress and Error Code Operations: Computing Unit Unspecified Subclass 167
Progress and Error Code Operations: Host Processor Subclass 169
Progress and Error Code Operations: Service Processor Subclass 171
Progress and Error Code Operations: Cache Subclass 172
Progress and Error Code Operations: Memory Subclass 173
Progress and Error Code Operations: Chipset Subclass 175
Progress Code Operations: User-Accessible Peripheral Class 176
Error Code Operations: User-Accessible Peripheral Class 177
Defined Subclasses: User-Accessible Peripheral Class 178
Progress and Error Code Operations: Peripheral Unspecified Subclass 179
Progress and Error Code Operations: Keyboard Subclass 180
Progress and Error Code Operations: Mouse Subclass 181
Progress and Error Code Operations: Local Console Subclass 182
Progress and Error Code Operations: Remote Console Subclass 182
Progress and Error Code Operations: Serial Port Subclass 183
Progress and Error Code Operations: Parallel Port Subclass 183
Progress and Error Code Operations: Fixed Media Subclass 184
Progress and Error Code Operations: Removable Media Subclass 184
Progress and Error Code Operations: Audio Input Subclass 185
Progress and Error Code Operations: Audio Output Subclass 185
Progress and Error Code Operations: LCD Device Subclass 186
Progress and Error Code Operations: Network Device Subclass 186

Progress Code Operations: I/O Bus Class 188
Error Code Operations: I/O Bus Class 189
Defined Subclasses: I/O Bus Class 190
Progress and Error Code Operations: I/O Bus Unspecified Subclass 191
Progress and Error Code Operations: PCI Subclass 192
Progress and Error Code Operations: USB Subclass 193
Progress and Error Code Operations: IBA Subclass 193
Progress and Error Code Operations: AGP Subclass 193
Progress and Error Code Operations: PC Card Subclass 194
Progress and Error Code Operations: LPC Subclass 194
Progress and Error Code Operations: SCSI Subclass 195
Progress and Error Code Operations: ATA/ATAPI/SATA Subclass 195
Progress and Error Code Operations: FC Subclass 196
Progress and Error Code Operations: IP Network Subclass 196
Progress and Error Code Operations: SMBus Subclass 197
Progress and Error Code Operations: I2C Subclass 197
Progress Code Operations: Host Software Class 198
Error Code Operations: Host Software Class 199
Defined Subclasses: Host Software Class 201
Progress and Error Code Operations: Host Software Unspecified Subclass 202
Progress and Error Code Operations: SEC Subclass 203
Progress and Error Code Operations: PEI Foundation Subclass 204
Progress and Error Code Operations: PEI Module Subclass 205
Progress and Error Code Operations: DXE Foundation Subclass 207
Progress and Error Code Operations: DXE Boot Service Driver Subclass 208
Progress and Error Code Operations: DXE Runtime Service Driver Subclass 210
Progress and Error Code Operations: SMM Driver Subclass 210
Progress and Error Code Operations: UEFI Application Subclass 210
Progress and Error Code Operations: OS Loader Subclass 211
Progress and Error Code Operations: Runtime Subclass 211
Progress and Error Code Operations: PEI Subclass 213
Progress and Error Code Operations: Boot Services Subclass 215
Progress and Error Code Operations: Runtime Services Subclass 218
Progress and Error Code Operations: DXE Services Subclass 220
Progress Code Enumeration Scheme 227
Debug Code Enumeration Scheme 228
Class Definitions 233
Defined Subclasses: Computing Unit Class 233
Decription of EFI_CPU_STATE_CHANGE_CAUSE fields 249
Definitions to describe Group Operations 252
Defined Subclasses: User-Accessible Peripheral Class 256
Defined Subclasses: I/O Bus Class 264
Defined Subclasses: Host Software Class 271

List of Figures

Figure 1. Example File Image (Graphical and Tree Representations).................................. 15
Figure 2. The Firmware Volume Format 17
Figure 3. Typical FFS File Layout 19
Figure 4. File Header 2 layout for files larger than 16Mb 19
Figure 5. Format of a section (below 16Mb) 20
Figure 6. Format of a section using the ExtendedLength field.................................. 20
Figure 7. Creating a File 28
Figure 8. Updating a File.. 30
Figure 9. Bit Allocation of FFS Attributes.. 44
Figure 10. EFI_FV_FILE_ATTRIBUTES fields 99
Figure 11. Example HOB Producer Phase Memory Map and Usage 128
Figure 12. Hierarchy of Status Code Operations 163
Figure 13. Status Code Services 298

1 Shared Architectural Elements

1.1 Overview

This volume describes the basic concepts behind Platform Initialization (PI) firmware storage and Hand-Off Blocks implementation.

The basic Platform Initialization (PI) firmware storage concepts include:

- Firmware Volumes
- Firmware File Systems
- Firmware Files
- Standard Binary Layout
- Pre-EFI Initialization (PEI) PEIM-to-PEIM Interfaces (PPIs)
- Driver Execution Environment (DXE) Protocols

The core code that is required for an implementation of Hand-Off Blocks (HOBs) in the Platform Initialization (PI) Architecture specifications are also shown. A HOB is a binary data structure that passes system state information from the HOB producer phase to the HOB consumer phase in the PI Architecture. This HOB specification does the following:

- Describes the basic components of HOBs and the rules for constructing them
- Provides code definitions for the HOB data types and structures that are architecturally required by the PI Architecture specifications

1.2 Target Audience

This document is intended for the following readers:

- Independent hardware vendors (IHVs) and original equipment manufacturers (OEMs) who will be implementing firmware components that are stored in firmware volumes
- Firmware developers who create firmware products or those who modify these products for use in platforms

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 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).

1.3.3 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 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.

1.4 Conventions used in this document

1.4.1 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.

1.4.2 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 1. 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 2. 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 Firmware Storage Design Discussion

2.1 Firmware Storage Introduction

This specification describes how files should be stored and accessed within non-volatile storage. Firmware implementations must support the standard PI Firmware Volume and Firmware File System format (described below), but may support additional storage formats.

2.1.1 Firmware Devices

A *firmware device* is a persistent physical repository that contains firmware code and/or data. It is typically a flash component but may be some other type of persistent storage. A single physical firmware device may be divided into smaller pieces to form multiple logical firmware devices. Similarly, multiple physical firmware devices may be aggregated into one larger logical firmware device.

This section describes the characteristics of typical physical firmware devices.

2.1.1.1 Flash

Flash devices are the most common non-volatile repository for firmware volumes. Flash devices are often divided into sectors (or blocks) of possibly differing sizes, each with different run-time characteristics. Flash devices have several unique qualities that are reflected in the design of the firmware file system:

- Flash devices can be erased on a sector-by-sector basis. After an erasure, all bits within a sector return to their *erase value*, either all 0 or all 1.
- Flash devices can be written on a bit-by-bit basis if the change is from its erase value to the non-erase value. For example, if the erase value is 1, then a bit with the value 1 can be changed to 0.
- Flash devices can only change from a non-erase value to an erase value by performing an erase operation on an entire flash sector.
- Some flash devices can enable or disable reads and writes to the entire flash device or to individual flash sectors.
- Some flash devices can lock the current enable or disable state of reads and writes until the next reset.
- Flash writes and erases are often longer operations than reads.
- Flash devices often place restrictions on the operations that can be performed while a write or erase is occurring.

2.1.2 Firmware Volumes

A Firmware Volume (FV) is a logical firmware device. In this specification, the basic storage repository for data and/or code is the firmware volume. Each firmware volume is organized into a file system. As such, the file is the base unit of storage for firmware.

Each firmware volume has the following attributes:

- **Name.** Each volume has a name consisting of an UEFI Globally Unique Identifier (GUID).
- **Size.** Each volume has a size, which describes the total size of all volume data, including any header, files, and free space.
- **Format.** Each volume has a format, which describes the Firmware File System used in the body of the volume.
- **Memory Mapped?** Some volumes may be memory-mapped, which indicates that the entire contents of the volume appear at once in the memory address space of the processor.
- **Sticky Write?** Some volumes may require special erase cycles in order to change bits from a non-erase value to an erase value.
- **Erase Polarity.** If a volume supports "Sticky Write," then all bits within the volume will return to this value (0 or 1) after an erase cycle.
- **Alignment.** The first byte of a volume is required to be aligned on some power-of-two boundary. At a minimum, this must be greater than or equal to the highest file alignment value. If **EFI_FVB2_WEAK_ALIGNMENT** is set in the volume header then the first byte of the volume can be aligned on any power-of-two boundary. A weakly aligned volume can not be moved from its initial linked location and maintain its alignment.
- **Read Enable/Disable Capable/Status.** Volumes may have the ability to change from readable to hidden.
- **Write Enable/Disable Capable/Status.** Volumes may have the ability to change from writable to write protected.
- **Lock Capable/Status.** Volumes may be able to have their capabilities locked.
- **Read-Lock Capable/Status.** Volumes may have the ability to lock their read status.
- **Write-Lock Capable/Status.** Volumes may have the ability to lock their write status.

Firmware volumes may also contain additional information describing the mapping between OEM file types and a GUID.

2.1.3 Firmware File System

A firmware file system (FFS) describes the organization of files and (optionally) free space within the firmware volume. Each firmware file system has a unique GUID, which is used by the firmware to associate a driver with a newly exposed firmware volume.

The PI Firmware File System is described in "Firmware File System Format" on page 17.

2.1.4 Firmware Files

Firmware files are code and/or data stored in firmware volumes.

Each of the files has the following attributes:

- **Name.** Each file has a name consisting of an UEFI GUID. File names must be unique within a firmware volume. Some file names have special significance.
- **Type.** Each file has a type. There are four ranges of file types: Normal (0x00-0xBF), OEM (0xC0-0xDF), Debug (0xE0-0xEF) and Firmware Volume Specific (0xF0-0xFF). For more information on types, see "Firmware File Types" on page 8.

- **Alignment.** Each file's data can be aligned on some power-of-two boundary. The specific boundaries that are supported depend on the alignment and format of the firmware volume. If **EFI_FVB2_WEAK_ALIGNMENT** is set in the volume header then file alignment does not depend on volume alignment.
- **Size.** Each file's data is zero or more bytes.

Specific firmware volume formats may support additional attributes, such as integrity verification and staged file creation. The file data of certain file types is sub-divided in a standardized fashion into "Firmware File Sections" on page 14.

Non-standard file types are supported through the use of the OEM file types. See "Firmware File Types" on page 8 for more information.

In the PEI phase, file-related services are provided through the PEI Services Table, using **FfsFindNextFile**, **FfsFindFileByName** and **FfsGetFileInfo**. In the DXE phase, file-related services are provided through the **EFI_FIRMWARE_VOLUME2_PROTOCOL** services attached to a volume's handle (**ReadFile**, **ReadSection**, **WriteFile** and **GetNextFile**).

2.1.4.1 Firmware File Types

Consider an application file named FOO.EXE. The format of the contents of FOO.EXE is implied by the ".EXE" in the file name. Depending on the operating environment, this extension typically indicates that the contents of FOO.EXE are a PE/COFF image and follow the PE/COFF image format.

Similarly, the PI Firmware File System defines the contents of a file that is returned by the firmware volume interface.

The PI Firmware File System defines an enumeration of file types. For example, the type **EFI_FV_FILETYPE_DRIVER** indicates that the file is a DXE driver and is interesting to the DXE Dispatcher. In the same way, files with the type **EFI_FV_FILETYPE_PEIM** are interesting to the PEI Dispatcher.

Table 3. Defined File Types

Name	Value	Description
EFI_FV_FILETYPE_RAW	0x01	Binary data
EFI_FV_FILETYPE_FREEFORM	0x02	Sectioned data
EFI_FV_FILETYPE_SECURITY_CORE	0x03	Platform core code used during the SEC phase
EFI_FV_FILETYPE_PEI_CORE	0x04	PEI Foundation
EFI_FV_FILETYPE_DXE_CORE	0x05	DXE Foundation
EFI_FV_FILETYPE_PEIM	0x06	PEI module (PEIM)
EFI_FV_FILETYPE_DRIVER	0x07	DXE driver
EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER	0x08	Combined PEIM/DXE driver
EFI_FV_FILETYPE_APPLICATION	0x09	Application
EFI_FV_FILETYPE_MM	0x0A	Contains a PE32+ image that will be loaded into MMRAM in MM Traditional Mode.
EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE	0x0B	Firmware volume image
EFI_FV_FILETYPE_COMBINED_MM_DXE	0x0C	Contains PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into MMRAM in MM Tradition Mode.
EFI_FV_FILETYPE_MM_CORE	0x0D	MM Foundation that support MM Traditional Mode.
EFI_FV_FILETYPE_MM_STANDALONE	0x0E	Contains a PE32+ image that will be loaded into MMRAM in MM Standalone Mode.
EFI_FV_FILETYPE_MM_CORE_STANDALONE	0x0F	MM Foundation that support MM Tradition Mode and MM Standalone Mode.
EFI_FV_FILETYPE_OEM_MIN… EFI_FV_FILETYPE_OEM_MAX	0xC0- 0xDF	OEM File Types
EFI_FV_FILETYPE_DEBUG_MIN… EFI_FV_FILETYPE_DEBUG_MAX	0xE0- 0xEF	Debug/Test File Types
EFI_FV_FILETYPE_FFS_MIN… EFI_FV_FILETYPE_FFS_MAX	0xF0- 0xFF	Firmware File System Specific File Types
EFI_FV_FILETYPE_FFS_PAD	0xF0	Pad File For FFS

2.1.4.1.1 EFI_FV_FILETYPE_APPLICATION

The file type **EFI_FV_FILETYPE_APPLICATION** denotes a file that contains a PE32 image that can be loaded using the UEFI Boot Service **LoadImage()**. Files of type **EFI_FV_FILETYPE_APPLICATION** are not dispatched by the DXE Dispatcher.

This file type is a sectioned file that must be constructed in accordance with the following rule:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.

There are no restrictions on the encapsulation of the leaf section.

In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the PE32 for the application in question is defined by the **LoadImage()** boot service. See the *Platform Initialization Driver Execution Environment Core Interface Specification* for details.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.2 EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER

The file type **EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER** denotes a file that contains code suitable for dispatch by the PEI Dispatcher, as well as a PE32 image that can be dispatched by the DXE Dispatcher. It has two uses:

- Enables sharing code between PEI and DXE to reduce firmware storage requirements.
- Enables bundling coupled PEIM/driver pairs in the same file.

This file type is a sectioned file and must follow the intersection of all rules defined for both **EFI_FV_FILETYPE_PEIM** and **EFI_FV_FILETYPE_DRIVER** files. This intersection is listed below:

- The file must contain one and only one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section; however, care must be taken to ensure any execute-in-place requirements are satisfied.
- The file must not contain more than one **EFI_SECTION_DXE_DEPEX** section.
- The file must not contain more than one **EFI_SECTION_PEI_DEPEX** section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.3 EFI_FV_FILETYPE_COMBINED_SMM_DXE

The file type **EFI_FV_FILETYPE_COMBINED_MM_DXE** denotes a file that contains a PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into MMRAM in MM Traditional Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_DXE_DEPEX** section. This section is ignored when the file is loaded into SMRAM.
- The file must contain no more than one **EFI_SECTION_MM_DEPEX** section. This section is ignored when the file is dispatched by the DXE Dispatcher.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.4 EFI_FV_FILETYPE_DRIVER

The file type **EFI_FV_FILETYPE_DRIVER** denotes a file that contains a PE32 image that can be dispatched by the DXE Dispatcher.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_DXE_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections.

In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Driver Execution Environment Core Interface Specification* for details.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.5 EFI_FV_FILETYPE_DXE_CORE

The file type **EFI_FV_FILETYPE_DXE_CORE** denotes the DXE Foundation file. This image is the one entered upon completion of the PEI phase of a UEFI boot cycle.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one and only one executable section, which must have a type of **EFI_SECTION_PE32**.
- The file must contain no more than one **EFI_SECTION_VERSION** section.

The sections that are described in the rules above may be optionally encapsulated in compression and/or additional GUIDed sections as required by the platform design.

As long as the above rules are followed, the file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.6 EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE

The file type **EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE** denotes a file that contains one or more firmware volume images.

This file type is a sectioned file that must be constructed in accordance with the following rule:

- The file must contain at least one section of type **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**. There are no restrictions on encapsulation of this section.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.7 EFI_FV_FILETYPE_FREEFORM

The file type **EFI_FV_FILETYPE_FREEFORM** denotes a sectioned file that may contain any combination of encapsulation and leaf sections. While the section layout can be parsed, the consumer of this type of file must have *a priori* knowledge of how it is to be used.

Standard firmware file system services will not return the handle of any pad files, nor will they permit explicit creation of such files. The *Name* field of the **EFI_FFS_FILE_HEADER** and **EFI_FFS_FILE_HEADER2** structures is considered invalid for pad files and will not be used in any operation that requires name comparisons.

A single **EFI_SECTION_FREEFORM_SUBTYPE_GUID** section may be included in a file of type **EFI_FV_FILETYPE_FREEFORM** to provide additional file type differentiation. While it is permissible to omit the **EFI_SECTION_FREEFORM_SUBTYPE_GUID** section entirely, there must never be more than one instance of it.

2.1.4.1.8 EFI_FV_FILETYPE_FFS_PAD

A pad file is an FFS-defined file type that is used to pad the location of the file that follows it in the storage file. The normal state of any valid (not deleted or invalidated) file is that both its header and data are valid. This status is indicated using the *State* bits with *State* **= 00000111b**. Pad files differ from all other types of files in that any pad file in this state must *not* have any data written into the data space. It is essentially a file filled with free space.

Standard firmware file system services will not return the handle of any pad files, nor will they permit explicit creation of such files. The *Name* field of the **EFI_FFS_FILE_HEADER** structure is considered invalid for pad files and will not be used in any operation that requires name comparisons.

2.1.4.1.9 EFI_FV_FILETYPE_PEIM

The file type **EFI_FV_FILETYPE_PEIM** denotes a file that is a PEI module (PEIM). A PEI module is dispatched by the PEI Foundation based on its dependencies during execution of the PEI phase. See the *Platform Initialization Pre-EFI Initialization Core Interface Specification* for details on PEI operation.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain one and only one executable section. This section must have one of the following types:
  - **EFI_SECTION_PE32**
  - **EFI_SECTION_PIC**
  - **EFI_SECTION_TE**
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_PEI_DEPEX** section.

As long as the above rules are followed, the file may contain other leaf and encapsulation sections as required or enabled by the platform design. Care must be taken to ensure that additional encapsulations do not render the file inaccessible due to execute-in-place requirements.

2.1.4.1.10 EFI_FV_FILETYPE_PEI_CORE

The file type **EFI_FV_FILETYPE_PEI_CORE** denotes the PEI Foundation file. This image is entered upon completion of the SEC phase of a PI Architecture-compliant boot cycle.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain one and only one executable section. This section must have one of the following types:
  - **EFI_SECTION_PE32**
  - **EFI_SECTION_PIC**
  - **EFI_SECTION_TE**
- The file must contain no more than one **EFI_SECTION_VERSION** section.

As long as the above rules are followed, the file may contain other leaf and encapsulations as required/enabled by the platform design.

2.1.4.1.11 EFI_FV_FILETYPE_RAW

The file type **EFI_FV_FILETYPE_RAW** denotes a file that does not contain sections and is treated as a raw data file. The consumer of this type of file must have *a priori* knowledge of its format and content. Because there are no sections, there are no construction rules.

2.1.4.1.12 EFI_FV_FILETYPE_SECURITY_CODE

The file type **EFI_FV_FILETYPE_SECURITY_CORE** denotes code and data that comprise the first part of PI Architecture firmware to execute. Its format is undefined with respect to the PI Architecture, as differing platform architectures may have varied requirements.

2.1.4.1.13 EFI_FV_FILETYPE_MM

The file type **EFI_FV_FILETYPE_MM** denotes a file that contains a PE32+ image that will be loaded into MMRAM in MM Tradition Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

• The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.

• The file must contain no more than one **EFI_SECTION_VERSION** section.

• The file must contain no more than one **EFI_SECTION_MM_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.14 EFI_FV_FILETYPE_MM_CORE

The file type **EFI_FV_FILETYPE_MM_CORE** denotes the MM Foundation file that only supports MM Traditional Mode. This image will be loaded by MM IPL into MMRAM.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one and only one executable section, which must have a type of **EFI_SECTION_PE32**.
- The file must contain no more than one **EFI_SECTION_VERSION** section.

The sections that are described in the rules above may be optionally encapsulated in compression and/or additional GUIDed sections as required by the platform design.

As long as the above rules are followed, the file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.15 EFI_FV_FILETYPE_MM_STANDALONE

The file type **EFI_FV_FILETYPE_MM_STANDALONE** denotes a file that contains a PE32+ image that will be loaded into SMRAM in SMM Standalone Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_MM_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the MM driver that will be dispatched is defined by MM Foundation Dispatcher. See the *Platform Initialization Specification, Volume 4* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

2.1.4.1.16 EFI_FV_FILETYPE_MM_CORE_STANDALONE

The file type **EFI_FV_FILETYPE_SMM_CORE_STANDALONE** denotes the MM Foundation file that support MM Traditional Mode and MM Standalone Mode. This image will be loaded by standalone MM IPL into MMRAM.

2.1.5 Firmware File Sections

Firmware file sections are separate discrete “parts” within certain file types. Each section has the following attributes:

- **Type.** Each section has a type. For more information on section types, see “Firmware File Section Types” on page 16.
- **Size.** Each section has a size.

While there are many types of sections, they fall into the following two broad categories:

- Encapsulation sections
- Leaf sections

Encapsulation sections are essentially containers that hold other sections. The sections contained within an encapsulation section are known as *child* sections, and the encapsulation section is known as the *parent* section. Encapsulation sections may have many children. An encapsulation section’s children may be leaves and/or more encapsulation sections and are called *peers* relative to each other. An encapsulation section does not contain data directly; instead it is just a vessel that ultimately terminates in leaf sections.

Files that are built with sections can be thought of as a tree, with encapsulation sections as nodes and leaf sections as the leaves. The file image itself can be thought of as the root and may contain an arbitrary number of sections. Sections that exist in the root have no parent section but are still considered peers.

Unlike encapsulation sections, leaf sections directly contain data and do not contain other sections. The format of the data contained within a leaf section is defined by the type of the section.

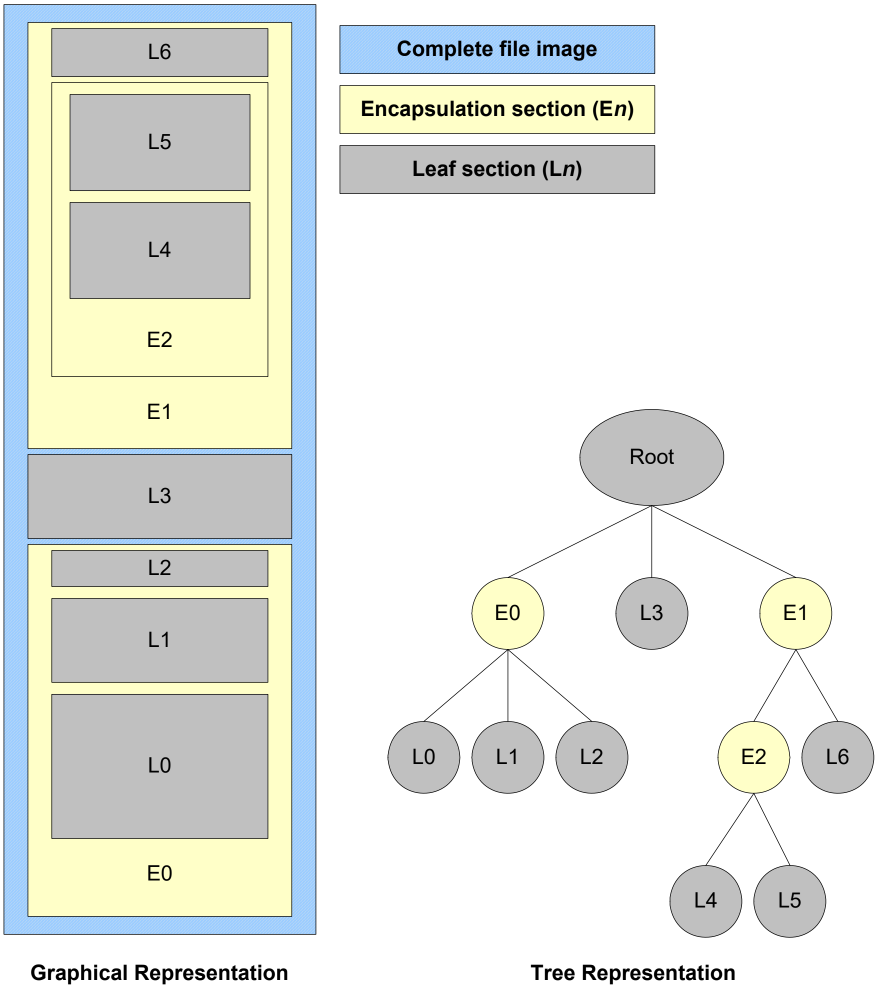


Figure 1. Example File Image (Graphical and Tree Representations)

In the example shown in Figure 1, the file image root contains two encapsulation sections (E0 and E1) and one leaf section (L3). The first encapsulation section (E0) contains children, all of which are leaves (L0, L1, and L2). The second encapsulation section (E1) contains two children, one that is an encapsulation (E2) and the other that is a leaf (L6). The last encapsulation section (E2) has two children that are both leaves (L4 and L5)

In the PEI phase, section-related services are provided through the PEI Service Table, using **FfsFindSectionData**. In the DXE phase, section-related services are provided through the **EFI_FIRMWARE_VOLUME2_PROTOCOL** services attached to a volume's handle (**ReadSection**).

2.1.5.1 Firmware File Section Types

Table 4 lists the defined architectural section types.

Table 4. Architectural Section Types

Name	Value	Description
EFI_SECTION_COMPRESSION	0x01	Encapsulation section where other sections are compressed.
EFI_SECTION_GUID_DEFINED	0x02	Encapsulation section where other sections have format defined by a GUID.
EFI_SECTION_DISPOSABLE	0x03	Encapsulation section used during the build process but not required for execution.
EFI_SECTION_PE32	0x10	PE32+ Executable image.
EFI_SECTION_PIC	0x11	Position-Independent Code.
EFI_SECTION_TE	0x12	Terse Executable image.
EFI_SECTION_DXE_DEPEX	0x13	DXE Dependency Expression.
EFI_SECTION_VERSION	0x14	Version, Text and Numeric.
EFI_SECTION_USER_INTERFACE	0x15	User-Friendly name of the driver.
EFI_SECTION_COMPATIBILITY16	0x16	DOS-style 16-bit EXE.
EFI_SECTION_FIRMWARE_VOLUME_IMAGE	0x17	PI Firmware Volume image.
EFI_SECTION_FREEFORM_SUBTYPE_GUID	0x18	Raw data with GUID in header to define format.
EFI_SECTION_RAW	0x19	Raw data.
EFI_SECTION_PEI_DEPEX	0x1b	PEI Dependency Expression.
EFI_SECTION_MM_DEPEX	0x1c	Leaf section type for determining the dispatch order for an MM Traditional driver in MM Traditional Mode or MM Standaline driver in MM Standalone Mode.

2.2 PI Architecture Firmware File System Format

This section describes the standard binary encoding for PI Firmware Files, PI Firmware Volumes, and the PI Firmware File System. Implementations that allow the non-vendor firmware files or firmware volumes to be introduced into the system must support the standard formats. This section also describes how features of the standard format map into the standard PEI and DXE interfaces.

The standard firmware file and volume format also introduces additional attributes and capabilities that are used to guarantee the integrity of the firmware volume.

The standard format is broken into three levels: the firmware volume format, the firmware file system format, and the firmware file format.

The standard firmware volume format (Figure 2) consists of two parts: the firmware volume header and the firmware volume data. The firmware volume header describes all of the attributes specified in “Firmware Volumes” on page 6. The header also contains a GUID which describes the format of the firmware file system used to organize the firmware volume data. The firmware volume header can support other firmware file systems other than the PI Firmware File System.

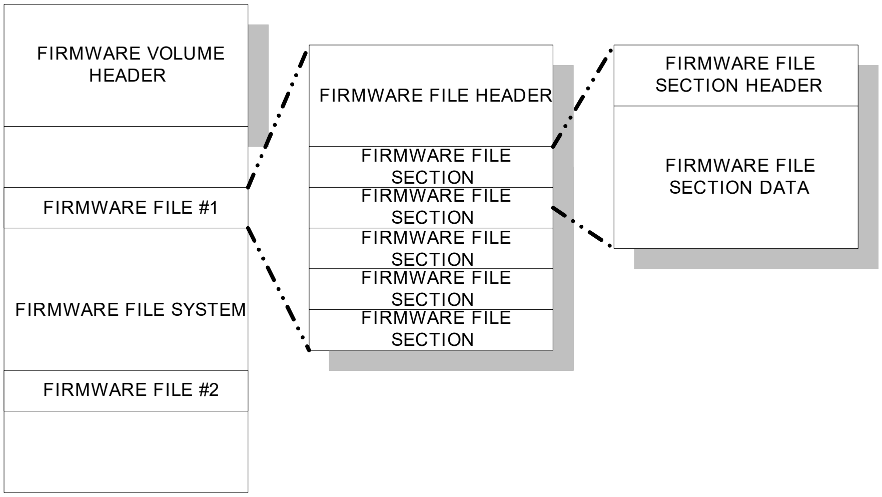


Figure 2. The Firmware Volume Format

The PI Firmware File System format describes how firmware files and free space are organized within the firmware volume.

The PI Firmware File format describes how files are organized. The firmware file format consists of two parts: the firmware file header and the firmware file data.

2.2.1 Firmware Volume Format

The PI Architecture Firmware Volume format describes the binary layout of a firmware volume. The firmware volume format consists of a header followed by the firmware volume data. The firmware volume header is described by **EFI_FIRMWARE_VOLUME_HEADER**.

The format of the firmware volume data is described by a GUID. Valid files system GUID values are **EFI_FIRMWARE_FILE_SYSTEM2_GUID** and **EFI_FIRMWARE_FILE_SYSTEM3_GUID**.

2.2.2 Firmware File System Format

The PI Architecture Firmware File System is a binary layout of file storage within firmware volumes. It is a flat file system in that there is no provision for any directory hierarchy; all files reside in the root directly. Files are stored end to end without any directory entry to describe which files are present. Parsing the contents of a firmware volume to obtain a listing of files present requires walking the firmware volume from beginning to end.

All files stored with the FFS must follow the "PI Architecture Firmware File System Format" on page 16. The standard file header provides for several levels of integrity checking to help detect file corruption, should it occur for some reason.

This section describes:

- PI Architecture's Firmware File System GUID (s)
- Volume Top File (VTF)

2.2.2.1 Firmware File System GUID

The PI Architecture firmware volume header contains a data field for the file system GUID. See **EFI_FIRMWARE_VOLUME_HEADER** on page 32 for more information on the firmware volume header. There are two valid FFS file system, the GUID is defined as **EFI_FIRMWARE_FILE_SYSTEM2_GUID** on page 39 and **EFI_FIRMWARE_FILE_SYSTEM3_GUID**.

If the FFS file system is backward compatible with **EFI_FIRMWARE_FILE_SYSTEM2_GUID** and supports files larger than 16 MB then **EFI_FIRMWARE_FILE_SYSTEM3_GUID** is used.

2.2.2.2 Volume Top File

A Volume Top File (VTF) is a file that must be located such that the last byte of the file is also the last byte of the firmware volume. Regardless of the file type, a VTF must have the file name GUID of **EFI_FFS_VOLUME_TOP_FILE_GUID** on page 40.

Firmware file system driver code must be aware of this GUID and insert a pad file as necessary to guarantee the VTF is located correctly at the top of the firmware volume on write and update operations. File length and alignment requirements must be consistent with the top of volume. Otherwise, a write error occurs and the firmware volume is not modified.

2.2.3 Firmware File Format

All FFS files begin with a header that is aligned on an 8-byteboundry with respect to the beginning of the firmware volume. FFS files can contain the following parts:

- Header
- Data

It is possible to create a file that has only a header and no data, which consumes 24 bytes of space. This type of file is known as a *zero-length file.*

If the file contains data, the data immediately follows the header. The format of the data within a file is defined by the *Type* field in the header, either **EFI_FFS_FILE_HEADER** or **EFI_FFS_FILE_HEADER2** in section 3.2.3.

Figure 3 illustrates the layout of a (typical) PI Architecture Firmware File *smaller* than 16 Mb:

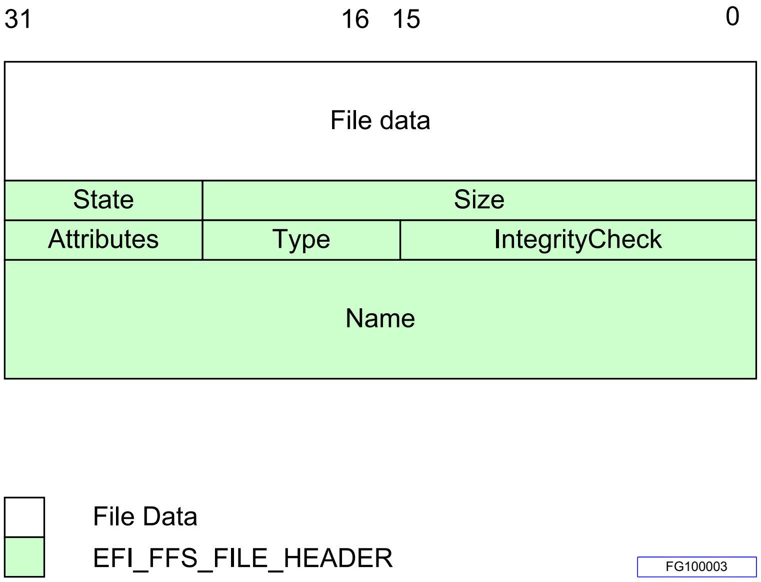


Figure 3. Typical FFS File Layout

Figure 4 illustrates the layout of a PI Architecture Firmware File **larger** than 16 Mb:

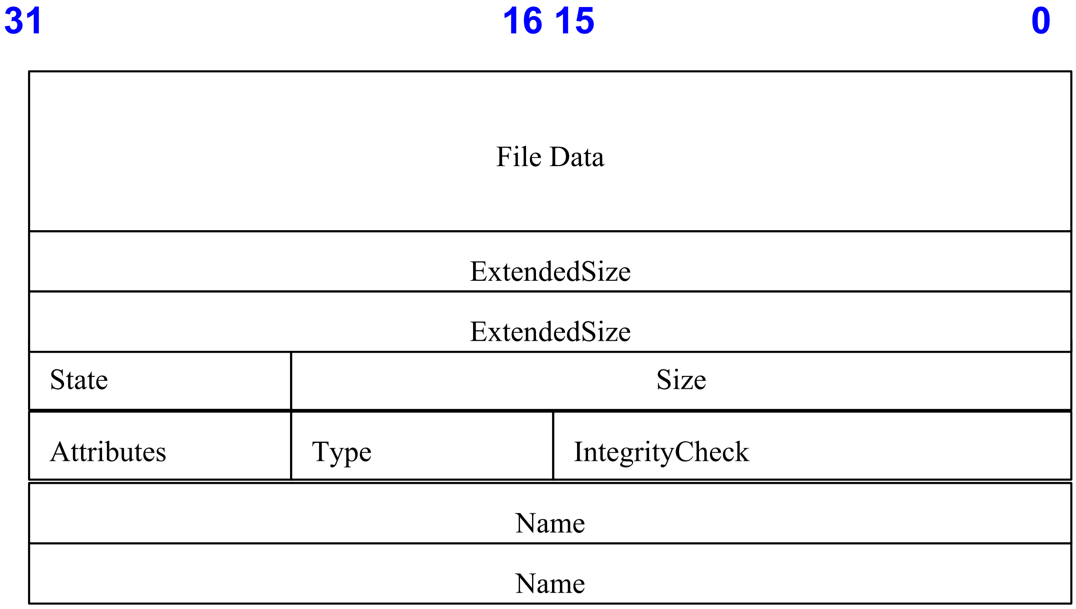


Figure 4. File Header 2 layout for files larger than 16Mb

2.2.4 Firmware File Section Format

This section describes the standard firmware file section layout.

Each section begins with a section header, followed by data defined by the section type.

The section headers aligned on 4 byte boundaries relative to the start of the file's image. If padding is required between the end of one section and the beginning of the next to achieve the 4-byte alignment requirement, all padding bytes must be initialized to zero.

Many section types are variable in length and are more accurately described as data streams rather than data structures.

Regardless of section type, all section headers begin with a 24-bit integer indicating the section size, followed by an 8-bit section type. The format of the remainder of the section header and the section data is defined by the section type. If the section size is 0xFFFFFF then the size is defined by a 32-bit integer that follows the 32-bit section header. Figures 5 and 6 shows the general format of a section.

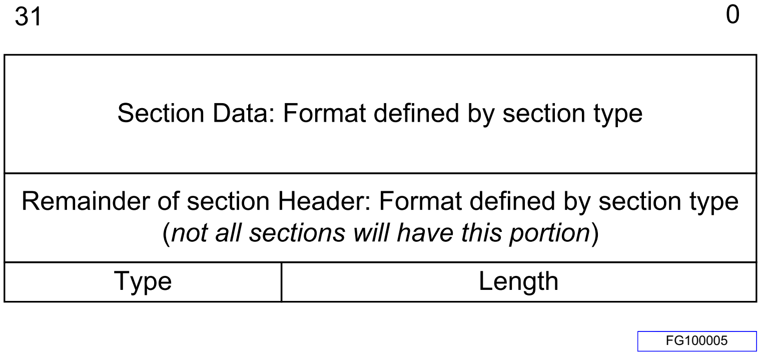

31
0
Section Data: Format defined by section type
Remainder of section Header: Format defined by section type
(*not all sections will have this portion*)
Type
Length
FG100005

Figure 5. Format of a section (below 16Mb)

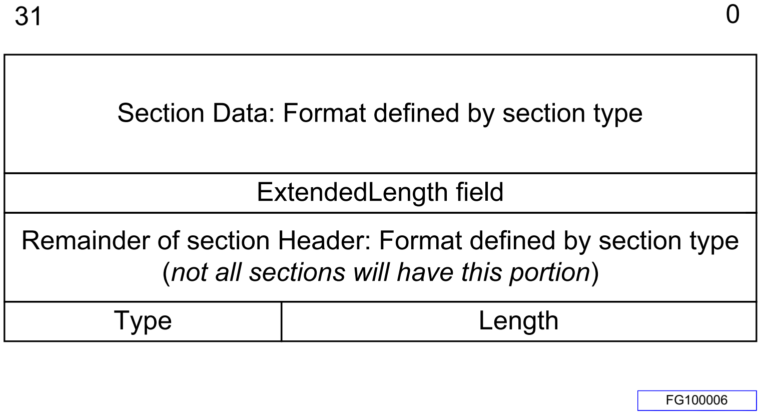

31
0
Section Data: Format defined by section type
ExtendedLength field
Remainder of section Header: Format defined by section type
(*not all sections will have this portion*)
Type
Length
FG100006

Figure 6. Format of a section using the ExtendedLength field

2.2.5 File System Initialization

The algorithm below describes a method of FFS initialization that ensures FFS file corruption can be detected regardless of the cause.

The *State* byte of each file must be correctly managed to ensure the integrity of the file system is not compromised in the event of a power failure during any FFS operation. It is expected that an FFS driver will produce an instance of the Firmware Volume Protocol and that all normal file operations will take place in that context. All file operations must follow all the creation, update, and deletion rules described in this specification to avoid file system corruption.

The following **FvCheck()** pseudo code must be executed during FFS initialization to avoid file system corruption. If at any point a failure condition is reached, then the firmware volume is corrupted and a crisis recovery is initiated.All FFS files, including files of type **EFI_FV_FILETYPE_FFS_PAD** must be evaluated during file system initialization. It is legal for multiple pad files with this file type to have the same Name field in the file header. No checks for duplicate files should be performed on pad files.

```
// Firmware volume initialization entry point – returns TRUE
// if FFS driver can use this firmware volume.
BOOLEAN FvCheck(Fv)
{
  // first check out firmware volume header
  if (FvHeaderCheck(Fv) == FALSE) {
    FAILURE();// corrupted firmware volume header
  }
  if (!((Fv->FvFileSystemId == EFI_FIRMWARE_FILE_SYSTEM2_GUID) || \
  (Fv->FvFileSystemId == EFI_FIRMWARE_FILE_SYSTEM3_GUID))){
    return (FALSE);  // This firmware volume is not
                     // formatted with FFS
  }
  // next walk files and verify the FFS is in good shape
  for (FilePtr = FirstFile; Exists(Fv, FilePtr);
          FilePtr = NextFile(Fv, FilePtr)) {
    if (FileCheck (Fv, FilePtr) != 0) {
      FAILURE(); // inconsistent file system
    }
  }
  if (CheckFreeSpace (Fv, FilePtr) != 0) {
    FAILURE();
  }
  return (TRUE);     // this firmware volume can be used by the FFS
                     // driver and the file system is OK
}
// FvHeaderCheck – returns TRUE if FvHeader checksum is OK.
BOOLEAN FvHeaderCheck (Fv)
{
  return (Checksum (Fv.FvHeader) == 0);
}
// Exists – returns TRUE if any bits are set in the file header
BOOLEAN Exists(Fv, FilePtr)
{
  return (BufferErased (Fv.ErasePolarity,
                FilePtr, sizeof (EFI_FIRMWARE_VOLUME_HEADER) == FALSE);
}
// BufferErased – returns TRUE if no bits are set in buffer
BOOLEAN BufferErased (ErasePolarity, BufferPtr, BufferSize)
{
  UINTN  Count;
  if (Fv.ErasePolarity == 1) {
    ErasedByte = 0xff;
  } else {
    ErasedByte = 0;
  }
  for (Count = 0; Count < BufferSize; Count++) {
    if (BufferPtr[Count] != ErasedByte) {
      return FALSE;
    }
  }
  return TRUE;
}
// GetFileState – returns high bit set of state field.
  UINT8 GetFileState (Fv, FilePtr) {
  UINT8 FileState;
  UINT8 HighBit;
  FileState = FilePtr->State;
  if (Fv.ErasePolarity != 0) {
    FileState = ~FileState;
  }
  HighBit = 0x80;
  while (HighBit != 0 && (HighBit & FileState) == 0) {
    HighBit = HighBit >> 1;
  }
  return HighBit;
}
// FileCheck – returns TRUE if the file is OK
BOOLEAN FileCheck (Fv, FilePtr) {
  switch (GetFileState (Fv, FilePtr)) {
    case EFI_FILE_HEADER_CONSTRUCTION:
      SetHeaderBit (Fv, FilePtr, EFI_FILE_HEADER_INVALID);
      break;
    case EFI_FILE_HEADER_VALID:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
      Break;
    case EFI_FILE_DATA_VALID:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (DuplicateFileExists (Fv, FilePtr,
                     EFI_FILE_DATA_VALID) != NULL) {
        return (FALSE);
      }
      break;
    case EFI_FILE_MARKED_FOR_UPDATE:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (FilePtr->State & EFI_FILE_DATA_VALID) == 0) {
        return (FALSE);
      }
      if (FilePtr->Type == EFI_FV_FILETYPE_FFS_PAD) {
        SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
      }
      else {
        if (DuplicateFileExists (Fv, FilePtr, EFI_FILE_DATA_VALID)) {
          SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
        }
        else {
          if (Fv->Attributes & EFI_FVB_STICKY_WRITE) {
            CopyFile (Fv, FilePtr);
            SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
          }
          else {
            ClearHeaderBit (Fv, FilePtr, EFI_FILE_MARKED_FOR_UPDATE);
          }
        }
      }
      break;
    case EFI_FILE_DELETED:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      break;
    case EFI_FILE_HEADER_INVALID:
      break;
  }
  return (TRUE);
}
// FFS_FILE_PTR * DuplicateFileExists (Fv, FilePtr, StateBit)
//    This function searches the firmware volume for another occurrence
//    of the file described by FilePtr, in which the duplicate files
// high state bit that is set is defined by the parameter StateBit.
// It returns a pointer to a duplicate file if it exists and NULL
// if it does not.  If the file type is EFI_FV_FILETYPE_FFS_PAD
// then NULL must be returned.

// CopyFile (Fv, FilePtr)
//    The purpose of this function is to clear the
//    EFI_FILE_MARKED_FOR_UPDATE bit from FilePtr->State
//    in firmware volumes that have EFI_FVB_STICKY_WRITE == TRUE.
//    The file is copied exactly header and all, except that the
//    EFI_FILE_MARKED_FOR_UPDATE bit in the file header of the
//    new file is clear.
// VerifyHeaderChecksum (FilePtr)
//    The purpose of this function is to verify the file header
//    sums to zero. See IntegrityCheck.Checksum.Header definition
//    for details.
// VerifyFileChecksum (FilePtr)
//    The purpose of this function is to verify the file integrity
//    check. See IntegrityCheck.Checksum.File definition for details.
```

2.2.6 Traversal and Access to Files

The Security (SEC), PEI, and early DXE code must be able to traverse the FFS and read and execute files before a write-enabled DXE FFS driver is initialized. Because the FFS may have inconsistencies due to a previous power failure or other system failure, it is necessary to follow a set of rules to verify the validity of files prior to using them. It is not incumbent on SEC, PEI, or the early read-only DXE FFS services to make any attempt to recover or modify the file system. If any situation exists where execution cannot continue due to file system inconsistencies, a recovery boot is initiated.

There is one inconsistency that the SEC, PEI, and early DXE code can deal with without initiating a recovery boot. This condition is created by a power failure or other system failure that occurs during a file update on a previous boot. Such a failure will cause two files with the same file name GUID to exist within the firmware volume. One of them will have the **EFI_FILE_MARKED_FOR_UPDATE** bit set in its *State* field but will be otherwise a completely valid file. The other one may be in any state of construction up to and including **EFI_FILE_DATA_VALID**. All files used prior to the initialization of the write-enabled DXE FFS driver *must* be screened with this test prior to their use. If this condition is discovered, it is permissible to initiate a recovery boot and allow the recovery DXE to complete the update.

The following pseudo code describes the method for determining which of these two files to use. The inconsistency is corrected during the write-enabled initialization of the DXE FFS driver.

```
// Screen files to ensure we get the right one in case
// of an inconsistency.
FFS_FILE_PTR EarlyFfsUpdateCheck(FFS_FILE_PTR * FilePtr) {
  FFS_FILE_PTR * FilePtr2;
  if (VerifyHeaderChecksum (FilePtr) != TRUE) {
    return (FALSE);
  }
  if (VerifyFileChecksum (FilePtr) != TRUE) {
    return (FALSE);
  }
  switch (GetFileState (Fv, FilePtr)) {
    case EFI_FILE_DATA_VALID:
      return (FilePtr);
      break;
    case EFI_FILE_MARKED_FOR_UPDATE:
      FilePtr2 = DuplicateFileExists (Fv, FilePtr,
                            EFI_FILE_DATA_VALID);
      if (FilePtr2 != NULL) {
        if (VerifyHeaderChecksum (FilePtr) != TRUE) {
          return (FALSE);
        }
        if (VerifyFileChecksum (FilePtr) != TRUE) {
          return (FALSE);
        }
        return (FilePtr2);
      } else {
        return (FilePtr);
      }
      break;
  }
}
```

Note: *There is no check for duplicate files once a file in the* **EFI_FILE_DATA_VALID** *state is located. The condition where two files in a single firmware volume have the same file name GUID and are both in the* **EFI_FILE_DATA_VALID** *state cannot occur if the creation and update rules that are defined in this specification are followed.*

2.2.7 File Integrity and State

File corruption, regardless of the cause, must be detectable so that appropriate file system repair steps may be taken. File corruption can come from several sources but generally falls into three categories:

- General failure
- Erase failure
- Write failure

A *general failure* is defined to be apparently random corruption of the storage media. This corruption can be caused by storage media design problems or storage media degradation, for example. This type of failure can be as subtle as changing a single bit within the contents of a file. With good system design and reliable storage media, general failures should not happen. Even so, the FFS enables detection of this type of failure.

An *erase failure* occurs when a block erase of firmware volume media is not completed due to a power failure or other system failure. While the erase operation is not defined, it is expected that most implementations of FFS that allow file write and delete operations will also implement a mechanism to reclaim deleted files and coalesce free space. If this operation is not completed correctly, the file system can be left in an inconsistent state.

Similarly, a *write failure* occurs when a file system write is in progress and is not completed due to a power failure or other system failure. This type of failure can leave the file system in an inconsistent state.

All of these failures are detectable during FFS initialization, and, depending on the nature of the failure, many recovery strategies are possible. Careful sequencing of the *State* bits during normal file transitions is sufficient to enable subsequent detection of write failures. However, the *State* bits alone are not sufficient to detect all occurrences of general and/or erase failures. These types of failures require additional support, which is enabled with the file header *IntegrityCheck* field.

For sample code that provides a method of FFS initialization that can detect FFS file corruption, regardless of the cause, see “File System Initialization” on page 20.

2.2.8 File State Transitions

2.2.8.1 Overview

There are three basic operations that may be done with the FFS:

- Creating a file
- Deleting a file
- Updating a file

All state transitions must be done carefully at all times to ensure that a power failure never results in a corrupted firmware volume. This transition is managed using the *State* field in the file header.

For the purposes of the examples below, positive decode logic is assumed (**EFI_FVB_ERASE_POLARITY = 0**). In actual use, the **EFI_FVB_ERASE_POLARITY** in the firmware volume header is referenced to determine the truth value of all FFS *State* bits. All *State* bit transitions must be atomic operations. Further, except when specifically noted, only the most significant *State* bit that is **TRUE** has meaning. Lower-order *State* bits are superseded by higher-order *State* bits.

Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** on page 32.

2.2.8.2 Initial State

The initial condition is that of “free space.” All free space in a firmware volume must be initialized such that all bits in the free space contain the value of **EFI_FVB_ERASE_POLARITY**. As such, if the free space is interpreted as an FFS file header, all *State* bits are **FALSE**.

Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** on page 32

2.2.8.3 Creating a File

A new file is created by allocating space from the firmware volume immediately beyond the end of the preceding file (or the firmware volume header if the file is the first one in the firmware volume). Figure 7 illustrates the steps to create a new file, which are detailed below the figure.

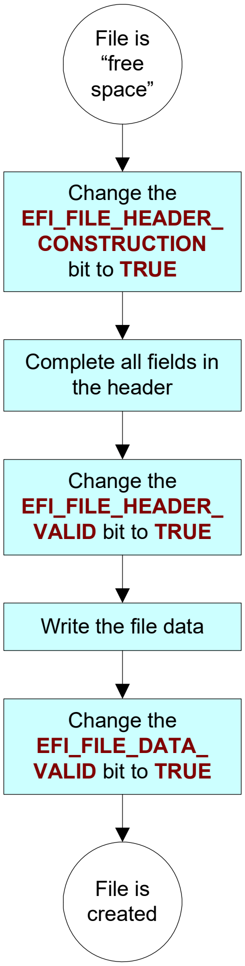

File is "free space"
Change the EFI_FILE_HEADER_ CONSTRUCTION bit to TRUE
Complete all fields in the header
Change the EFI_FILE_HEADER_ VALID bit to TRUE
Write the file data
Change the EFI_FILE_DATA_ VALID bit to TRUE
File is created

Figure 7. Creating a File

As shown in Figure 7, the following steps are required to create a new file:

1. Allocate space in the firmware volume for a new file header, either **EFI_FFS_FILE_HEADER**, or **EFI_FFS_FILE_HEADER2** if the file is 16MB or more in size, and complete all fields of the header (except for the *State* field, which is updated independently from the rest of the header). This allocation is done by interpreting the free space as a file header and changing the **EFI_FILE_HEADER_CONSTRUCTION** bit to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are

made. This transition yields *State* = **00000001b,** which indicates the header construction has begun but has not yet been completed. This value has the effect of "claiming" the FFS header space from the firmware volume free space.

While in this state, the following fields of the FFS header are initialized and written to the firmware volume:

- *Name*
- *IntegrityCheck.Header*
- *Type*
- *Attributes*
- *Size*

If **FFS_ATTRIB_LARGE_FILE** is set in *Attributes* the *Size* field of the FFS header must be zero and *ExtendedSize* must contian the size of the FFS file.The value of *IntegrityCheck.Header* is calculated as described in **EFI_FFS_FILE_HEADER**.

2. Mark the new header as complete and write the file data. To mark the header as complete, the **EFI_FILE_HEADER_VALID** bit is changed to **TRUE**. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* = **00000011b,** which indicates the header construction is complete, but the file data has not yet been written. This value has the effect of "claiming" the full length of the file from the firmware volume free space. Once the **EFI_FILE_HEADER_VALID** bit is set, no further changes to the following fields may be made:

   - *Name*
   - *IntegrityCheck.Header*
   - *Type*
   - *Attributes*
   - *Size*

   While in this state, the file data and *IntegrityCheck.File* are written to the firmware volume. The order in which these are written does not matter. The calculation of the value for *IntegrityCheck.File* is described in **EFI_FFS_FILE_HEADER** on page 41.

3. Mark the data as valid. To mark the data as valid, the **EFI_FILE_DATA_VALID** bit is changed to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* = **00000111b,** which indicates the file data is fully written and is valid.

2.2.8.4 Deleting a File

Any file with **EFI_FILE_HEADER_VALID** set to **TRUE** and **EFI_FILE_HEADER_INVALID** and **EFI_FILE_DELETED** set to **FALSE** is a candidate for deletion.

To delete a file, the **EFI_FILE_DELETED** bit is set to the **TRUE** state. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* = **0001xx11b**, which indicates the file is marked deleted. Its header is still valid, however, in as much as its length field is used in locating the next file in the firmware volume.

Note: *The* **EFI_FILE_HEADER_INVALID** *bit must be left in the* **FALSE** *state.*

2.2.8.5 Updating a File

A file update is a special case of file creation where the file being added already exists in the firmware volume. At all times during a file update, only one of the files, either the new one or the old one, is valid at any given time. This validation is possible by using the **EFI_FILE_MARKED_FOR_UPDATE** bit in the old file.

Figure 8 illustrates the steps to update a file, which are detailed below the figure.

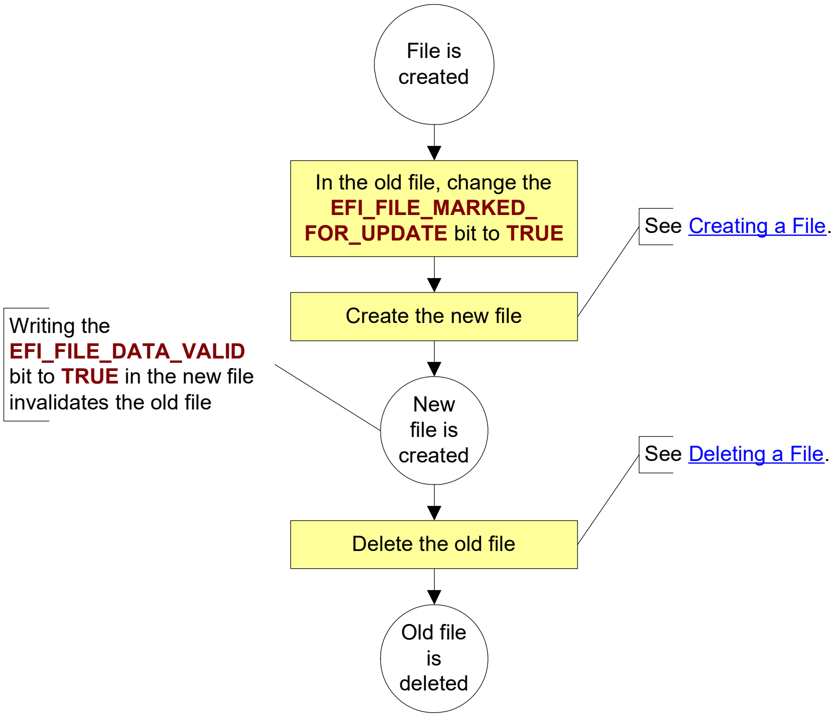

File is created
In the old file, change the EFI_FILE_MARKED_ FOR_UPDATE bit to TRUE
See Creating a File.
Writing the EFI_FILE_DATA_VALID bit to TRUE in the new file invalidates the old file
Create the new file
New file is created
See Deleting a File.
Delete the old file
Old file is deleted

Figure 8. Updating a File

As shown in Figure 8, the following steps are required to update a file:

1. Set the **EFI_FILE_MARKED_FOR_UPDATE** bit to **TRUE** in the old file. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00001111b,** which indicates the file is marked for update. A file in this state remains valid as long as no other file in the firmware volume has the same name and a *State* of **000001xxb.**
2. Create the new file following the steps described in "Creating a File" on page 28. When the new file becomes valid, the old file that was marked for update becomes invalid. That is to say, a file marked for update is valid only as long as there is no file with the same name in the firmware volume that has a *State* of **000001xxb.** In this way, only one of the files, either the new or the old, is valid at any given time. The act of writing the **EFI_FILE_DATA_VALID** bit in the new file's *State* field has the additional effect of invalidating the old file.
3. Delete the old file following the steps described in "Deleting a File" on page 29.

3 Firmware Storage Code Definitions

3.1 Firmware Storage Code Definitions Introduction

This section provides the code definitions for:

- The PI Architecture Firmware Storage binary formats for volumes, file system, files, and file sections.
- The PEI interfaces that support firmware volumes, firmware file systems, firmware files, and firmware file sections.
- The DXE protocols that support firmware volumes, firmware file systems, firmware files, and firmware file sections.

3.2 Firmware Storage Formats

3.2.1 Firmware Volume

EFI_FIRMWARE_VOLUME_HEADER

Summary

Describes the features and layout of the firmware volume.

Prototype

```
typedef struct {
  UINT8                  ZeroVector[16];
  EFI_GUID               FileSystemGuid;
  UINT64                 FvLength;
  UINT32                 Signature;
  EFI_FVB_ATTRIBUTES_2   Attributes;
  UINT16                 HeaderLength;
  UINT16                 Checksum;
  UINT16                 ExtHeaderOffset;
  UINT8                  Reserved[1];
  UINT8                  Revision;
  EFI_FV_BLOCK_MAP       BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

Parameters

ZeroVector

The first 16 bytes are reserved to allow for the reset vector of processors whose reset vector is at address 0.

FileSystemGuid

Declares the file system with which the firmware volume is formatted. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *Unified Extensible Firmware Interface Specification*, version 2.0 (UEFI 2.0 specification).

FvLength

Length in bytes of the complete firmware volume, including the header.

Signature

Set to **{'_','F','V','H'}**.

Attributes

Declares capabilities and power-on defaults for the firmware volume. Current state is determined using the **GetAttributes()** function and is not maintained in the *Attributes* field of the firmware volume header. Type **EFI_FVB_ATTRIBUTES_2** is defined in “Related Definitions” below.

HeaderLength

Length in bytes of the complete firmware volume header.

Checksum

A 16-bit checksum of the firmware volume header. A valid header sums to zero.

ExtHeaderOffset

Offset, relative to the start of the header, of the extended header (**EFI_FIRMWARE_VOLUME_EXT_HEADER**) or zero if there is no extended header. The extended header is followed by zero or more variable length extension entries. Each extension entry is prefixed with the **EFI_FIRMWARE_VOLUME_EXT_ENTRY** structure (see “Related Definitions” below), which defines the type and size of the extension entry. The extended header is always 32-bit aligned relative to the start of the FIRMWARE VOLUME.

If there is an instance of the **EFI_FIRMWARE_VOLUME_EXT_HEADER**, then the firmware shall build an instance of the Firmware Volume Media Device Path (ref Vol2, Section 8.2).

Reserved

In this version of the specification, this field must always be set to zero.

Revision

Set to 2. Future versions of this specification may define new header fields and will increment the *Revision* field accordingly.

FvBlockMap[]

An array of run-length encoded *FvBlockMapEntry* structures. The array is terminated with an entry of **{0,0}**.

FvBlockMapEntry.NumBlocks

The number of blocks in the run.

FvBlockMapEntry.BlockLength

The length of each block in the run.

Description

A firmware volume based on a block device begins with a header that describes the features and layout of the firmware volume. This header includes a description of the capabilities, state, and block map of the device.

The block map is a run-length-encoded array of logical block definitions. This design allows a reasonable mechanism of describing the block layout of typical firmware devices. Each block can be referenced by its logical block address (LBA). The LBA is a zero-based enumeration of all of the blocks—i.e., LBA 0 is the first block, LBA 1 is the second block, and LBA *n* is the (*n*-1) device.

The header is always located at the beginning of LBA 0.

Related Definitions

```
//*************************************************
// EFI_FVB_ATTRIBUTES_2
//*************************************************
typedef UINT32 EFI_FVB_ATTRIBUTES_2

// Attributes bit definitions
#define EFI_FVB2_READ_DISABLED_CAP    0x00000001
#define EFI_FVB2_READ_ENABLED_CAP     0x00000002
#define EFI_FVB2_READ_STATUS          0x00000004

#define EFI_FVB2_WRITE_DISABLED_CAP   0x00000008
#define EFI_FVB2_WRITE_ENABLED_CAP    0x00000010
#define EFI_FVB2_WRITE_STATUS         0x00000020

#define EFI_FVB2_LOCK_CAP             0x00000040
#define EFI_FVB2_LOCK_STATUS          0x00000080

#define EFI_FVB2_STICKY_WRITE         0x00000200
#define EFI_FVB2_MEMORY_MAPPED        0x00000400
#define EFI_FVB2_ERASE_POLARITY       0x00000800

#define EFI_FVB2_READ_LOCK_CAP        0x00001000
#define EFI_FVB2_READ_LOCK_STATUS     0x00002000

#define EFI_FVB2_WRITE_LOCK_CAP       0x00004000
#define EFI_FVB2_WRITE_LOCK_STATUS    0x00008000
```

```
#define EFI_FVB2_ALIGNMENT            0x001F0000
#define EFI_FVB2_WEAK_ALIGNMENT       0x80000000
#define EFI_FVB2_ALIGNMENT_1          0x00000000
#define EFI_FVB2_ALIGNMENT_2          0x00010000
#define EFI_FVB2_ALIGNMENT_4          0x00020000
#define EFI_FVB2_ALIGNMENT_8          0x00030000
#define EFI_FVB2_ALIGNMENT_16         0x00040000
#define EFI_FVB2_ALIGNMENT_32         0x00050000
#define EFI_FVB2_ALIGNMENT_64         0x00060000
#define EFI_FVB2_ALIGNMENT_128        0x00070000
#define EFI_FVB2_ALIGNMENT_256        0x00080000
#define EFI_FVB2_ALIGNMENT_512        0x00090000
#define EFI_FVB2_ALIGNMENT_1K         0x000A0000
#define EFI_FVB2_ALIGNMENT_2K         0x000B0000
#define EFI_FVB2_ALIGNMENT_4K         0x000C0000
#define EFI_FVB2_ALIGNMENT_8K         0x000D0000
#define EFI_FVB2_ALIGNMENT_16K        0x000E0000
#define EFI_FVB2_ALIGNMENT_32K        0x000F0000
#define EFI_FVB2_ALIGNMENT_64K        0x00100000
#define EFI_FVB2_ALIGNMENT_128K       0x00110000
#define EFI_FVB2_ALIGNMENT_256K       0x00120000
#define EFI_FVB2_ALIGNMENT_512K       0x00130000
#define EFI_FVB2_ALIGNMENT_1M         0x00140000
#define EFI_FVB2_ALIGNMENT_2M         0x00150000
#define EFI_FVB2_ALIGNMENT_4M         0x00160000
#define EFI_FVB2_ALIGNMENT_8M         0x00170000
#define EFI_FVB2_ALIGNMENT_16M        0x00180000
#define EFI_FVB2_ALIGNMENT_32M        0x00190000
#define EFI_FVB2_ALIGNMENT_64M        0x001A0000
#define EFI_FVB2_ALIGNMENT_128M       0x001B0000
#define EFI_FVB2_ALIGNMENT_256M       0x001C0000
#define EFI_FVB2_ALIGNMENT_512M       0x001D0000
#define EFI_FVB2_ALIGNMENT_1G         0x001E0000
#define EFI_FVB2_ALIGNMENT_2G         0x001F0000
```

Table 5 describes the fields in the above definition:

Table 5. Descriptions of EFI_FVB_ATTRIBUTES_2

Attribute	Description
EFI_FVB2_READ_DISABLED_CAP	**TRUE** if reads from the firmware volume may be disabled.
EFI_FVB2_READ_ENABLED_CAP	**TRUE** if reads from the firmware volume may be enabled.
EFI_FVB2_READ_STATUS	**TRUE** if reads from the firmware volume are currently enabled.
EFI_FVB2_WRITE_DISABLED_CAP	**TRUE** if writes to the firmware volume may be disabled.
EFI_FVB2_WRITE_ENABLED_CAP	**TRUE** if writes to the firmware volume may be enabled.
EFI_FVB2_WRITE_STATUS	**TRUE** if writes to the firmware volume are currently enabled.
EFI_FVB2_LOCK_CAP	**TRUE** if firmware volume attributes may be locked down.
EFI_FVB2_LOCK_STATUS	**TRUE** if firmware volume attributes are currently locked down.
EFI_FVB2_STICKY_WRITE	**TRUE** if a block erase is required to transition bits from **(NOT)EFI_FVB2_ERASE_POLARITY** to **EFI_FVB2_ERASE_POLARITY**. That is, after erasure, a write may negate a bit in the **EFI_FVB2_ERASE_POLARITY** state, but a write cannot flip it back again. A block erase cycle is required to transition bits from the **(NOT)EFI_FVB2_ERASE_POLARITY** state back to the **EFI_FVB2_ERASE_POLARITY** state. See the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** on page 112.
EFI_FVB2_MEMORY_MAPPED	**TRUE** if firmware volume is memory mapped.
EFI_FVB2_ERASE_POLARITY	Value of all bits after erasure. See the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** on page 112.
EFI_FVB2_READ_LOCK_CAP	**TRUE** if the firmware volume's read-status can be locked.
EFI_FVB2_READ_LOCK_STATUS	**TRUE** If the firmware volume's read-status is locked.
EFI_FVB2_WRITE_LOCK_CAP	**TRUE** if the firmware volume’s write status can be locked.
EFI_FVB2_WRITE_LOCK_STATUS	**TRUE** if the firmware volume’s write-status is locked.
EFI_FVB2_ALIGNMENT	The first byte of the firmware volume must be placed at an address which is an even multiple of 2^(*this field*). For example, a value of 5 in this field would mean a required alignment of 32 bytes.
EFI_FVB2_WEAK_ALIGNMENT	TRUE if the firmware volume can be less than the the highest file alignment value.

All other **EFI_FVB_ATTRIBUTES_2** bits are reserved and must be zero.

```
typedef struct {
  UINT32 NumBlocks;
  UINT32 Length;
} EFI_FV_BLOCK_MAP;
```

NumBlocks

The number of sequential blocks which are of the same size.

Length

The size of the blocks.

```
typedef struct {
  EFI_GUID FvName;
  UINT32   ExtHeaderSize;
} EFI_FIRMWARE_VOLUME_EXT_HEADER;
```

FvName

Firmware volume name.

ExtHeaderSize

Size of the rest of the extension header, including this structure.

After the extension header, there is an array of variable-length extension header entries, each prefixed with the **EFI_FIRMWARE_VOLUME_EXT_ENTRY** structure.

```
typedef struct {
  UINT16                          ExtEntrySize;
  UINT16                          ExtEntryType;
} EFI_FIRMWARE_VOLUME_EXT_ENTRY;
```

ExtEntrySize

Size of this header extension.

ExtEntryType

Type of the header. See **EFI_FV_EXT_TYPE_x**.

```
#define EFI_FV_EXT_TYPE_OEM_TYPE 0x01
typedef struct {
  EFI_FIRMWARE_VOLUME_EXT_ENTRY  Hdr;
  UINT32                          TypeMask;
  //EFI_GUID                      Types[];
} EFI_FIRMWARE_VOLUME_EXT_ENTRY_OEM_TYPE;
```

Hdr

Standard extension entry, with the type **EFI_FV_EXT_TYPE_OEM_TYPE**.

TypeMask

A bit mask, one bit for each file type between 0xC0 (bit 0) and 0xDF (bit 31). If a bit is ‘1’, then the GUID entry exists in *Types*. If a bit is ‘0’ then no GUID entry exists in *Types*. For example, the value 0x01010301 would indicate that there would be five total entries in *Types* for file types 0xC0 (bit 0), 0xC8 (bit 4), 0xC9 (bit 5), 0xD0 (bit 16), and 0xD8 (bit 24).

Types

An array of GUIDs, each GUID representing an OEM file type.

This extension header provides a mapping between a GUID and an OEM file type.

```
#define EFI_FV_EXT_TYPE_GUID_TYPE 0x0002
typedef struct {
  EFI_FIRMWARE_VOLUME_EXT_ENTRY  Hdr;
  EFI_GUID                       FormatType;
  //UINT8                        Data[];
} EFI_FIRMWARE_VOLUME_EXT_ENTRY_GUID_TYPE;
```

Hdr

Standard extension entry, with the type **EFI_FV_EXT_TYPE_OEM_TYPE**.

FormatType

Vendor-specific GUID

Length

Length of the data following this field

Data

An arry of bytes of length Length.

This extension header **EFI_FIRMWARE_VOLUME_EXT_ENTRY_GUID_TYPE** provides a vendor-specific GUID *FormatType* type which includes a length and a successive series of data bytes. Values 0x00, 0x03..0xffff are reserved by the specification.

```
#define EFI_FV_EXT_TYPE_USED_SIZE_TYPE 0x03
    typedef struct {
      EFI_FIRMWARE_VOLUME_EXT_ENTRY Hdr;
      UINT32 UsedSize;
    } EFI_FIRMWARE_VOLUME_EXT_ENTRY_USED_SIZE_TYPE;
```

Hdr

Standard extension entry, with the type **EFI_FV_EXT_TYPE_USED_SIZE_TYPE**.

UsedSize

The number of bytes of the FV that are in uses. The remaining **EFI_FIRMWARE_VOLUME_HEADER** FvLength minus UsedSize bytes in the FV must contain the value implied by **EFI_FVB2_ERASE_POLARITY**.

The **EFI_FIRMWARE_VOLUME_EXT_ENTRY_USED_SIZE_TYPE** can be used to find out how many **EFI_FVB2_ERASE_POLARITY** bytes are at the end of the FV.

3.2.1.1 EFI Signed Firmware Volumes

There may be one or more headers with a *FormatType* of value **EFI_FIRMWARE_CONTENTS_SIGNED_GUID**.

A *signed firmware volume* is a cryptographic signature across the entire volume. To process the contents and verify the integrity of the volume, the **EFI_FIRMWARE_VOLUME_EXT_ENTRY_GUID_TYPE** *Data[]*shall contain an instance of **WIN_CERTIFICATE_UEFI_GUID** where the *CertType* **=** **EFI_CERT_TYPE_PKCS7_GUID**or **EFI_CERT_TYPE_RSA2048_SHA256_GUID**.

3.2.2 Firmware File System

EFI_FIRMWARE_FILE_SYSTEM2_GUID

Summary

The firmware volume header contains a data field for the file system GUID. See the **EFI_FIRMWARE_VOLUME_HEADER** on page 32 for more information on the firmware volume header.

GUID

```
// {8C8CE578-8A3D-4f1c-9935-896185C32DD3}
#define EFI_FIRMWARE_FILE_SYSTEM2_GUID \
  { 0x8c8ce578, 0x8a3d, 0x4f1c, \
  0x99, 0x35, 0x89, 0x61, 0x85, 0xc3, 0x2d, 0xd3 }
```

EFI_FIRMWARE_FILE_SYSTEM3_GUID

Summary

The firmware volume header contains a data field for the file system GUID. See the **EFI_FIRMWARE_VOLUME_HEADER** on page 32 for more information on the firmware volume header.

EFI_FIRMWARE_FILE_SYSTEM3_GUID indicates support for **FFS_ATTRIB_LARGE_SIZE** and thus support for files 16MB or larger. **EFI_FIRMWARE_FILE_SYSTEM2_GUID** volume does not contain large files. Files 16 MB or larger use a **EFI_FFS_FILE_HEADER2** and smaller files use **EFI_FFS_FILE_HEADER. EFI_FIRMWARE_FILE_SYSTEM2_GUID** allows backward compatibility with previous versions of this specification

GUID

```
// {5473C07A-3DCB-4dca-BD6F-1E9689E7349A}
#define EFI_FIRMWARE_FILE_SYSTEM3_GUID \
  { 0x5473c07a, 0x3dcb, 0x4dca, \
  { 0xbd, 0x6f, 0x1e, 0x96, 0x89, 0xe7, 0x34, 0x9a } }
```

EFI_FFS_VOLUME_TOP_FILE_GUID

Summary

A Volume Top File (VTF) is a file that must be located such that the last byte of the file is also the last byte of the firmware volume. Regardless of the file type, a VTF must have the file name GUID of **EFI_FFS_VOLUME_TOP_FILE_GUID** as defined below.

GUID

```
// {1BA0062E-C779-4582-8566-336AE8F78F09}

#define EFI_FFS_VOLUME_TOP_FILE_GUID \
  { 0x1BA0062E, 0xC779, 0x4582, 0x85, 0x66, 0x33, 0x6A, \
  0xE8, 0xF7, 0x8F, 0x9 }
```

3.2.3 Firmware File

EFI_FFS_FILE_HEADER

Summary

Each file begins with a header that describes the state and contents of the file. The header is 8-byte aligned with respect to the beginning of the firmware volume.

Prototype

```
typedef struct {
  EFI_GUID                    Name;
  EFI_FFS_INTEGRITY_CHECK     IntegrityCheck;
  EFI_FV_FILETYPE             Type;
  EFI_FFS_FILE_ATTRIBUTES     Attributes;
  UINT8                       Size[3];
  EFI_FFS_FILE_STATE          State;
} EFI_FFS_FILE_HEADER;

typedef struct {
  EFI_GUID                   Name;
  EFI_FFS_INTEGRITY_CHECK IntegrityCheck;
  EFI_FV_FILETYPE            Type;
  EFI_FFS_FILE_ATTRIBUTES Attributes;
  UINT8                      Size[3];
  EFI_FFS_FILE_STATE         State;
  UINT64                     ExtendedSize;
} EFI_FFS_FILE_HEADER2;
```

Parameters

Name

This GUID is the file name. It is used to uniquely identify the file. There may be only one instance of a file with the file name GUID of *Name* in any given firmware volume, except if the file type is **EFI_FV_FILETYPE_FFS_PAD**.

IntegrityCheck

Used to verify the integrity of the file. Type **EFI_FFS_INTEGRITY_CHECK** is defined in "Related Definitions" below.

Type

Identifies the type of file. Type **EFI_FV_FILETYPE** is defined in "Related Definitions," below. FFS-specific file types are defined in **EFI_FV_FILETYPE_FFS_PAD**.

Attributes

Declares various file attribute bits. Type **EFI_FFS_FILE_ATTRIBUTES** is defined in "Related Definitions" below.

Size

The length of the file in bytes, including the FFS header. The length of the file data is either **(*Size* – sizeof(EFI_FFS_FILE_HEADER))**. This calculation means a zero-length file has a *Size* of 24 bytes, which is **sizeof(EFI_FFS_FILE_HEADER)**.

Size is *not* required to be a multiple of 8 bytes. Given a file *F*, the next file header is located at the next 8-byte aligned firmware volume offset following the last byte of the file *F*.

State

Used to track the state of the file throughout the life of the file from creation to deletion. Type **EFI_FFS_FILE_STATE** is defined in "Related Definitions" below. See "File Integrity and State" on page 26 for an explanation of how these bits are used.

ExtendedSize

If **FFS_ATTRIB_LARGE_FILE** is set in *Attributes* then *ExtendedSize* exists and *Size* must be set to zero.

If **FFS_ATTRIB_LARGE_FILE** is not set then **EFI_FFS_FILE_HEADER** is used.

Description

The file header may use one of two structures to define the file. If the size of the file is larger than 0xFFFFFF the **EFI_FFS_FILE_HEADER2** structure must be used, otherwise the **EFI_FFS_FILE_HEADER** structure must be used. The structure used is determined by the **FFS_ATTRIB_LARGE_FILE** attribute in the *Attributes* member. Note that all of the structure elements other than *ExtendedSize* are the same in the two structures. The *ExtendedSize* member is used instead of the Size member when the **EFI_FFS_FILE_HEADER2** structure is used (**FFS_ATTRIB_LARGE_FILE** is set).

Related Definitions

```
//********************************************************
// EFI_FFS_INTEGRITY_CHECK
//********************************************************
typedef union {
  struct {
    UINT8                Header;
    UINT8                File;
  }                      Checksum;
  UINT16                 Checksum16;
} EFI_FFS_INTEGRITY_CHECK;
```

Header

The *IntegrityCheck.Checksum.Header* field is an 8-bit checksum of the file header. The *State* and *IntegrityCheck.Checksum.File* fields are assumed to be zero and the checksum is calculated such that the entire header sums to zero. The

IntegrityCheck.Checksum.Header field is valid anytime the **EFI_FILE_HEADER_VALID** bit is set in the *State* field. See "File Integrity and State" on page 26 for more details.

If the **FFS_ATTRIB_LARGE_FILE** bit of the *Attributes* field is set the header size is sizeof (**EFI_FFS_FILE_HEADER2**), if it is clear the header size is sizeof (**EFI_FFS_FILE_HEADER**).

File

If the **FFS_ATTRIB_CHECKSUM** (see definition below) bit of the *Attributes* field is set to one, the *IntegrityCheck.Checksum.File* field is an 8-bit checksum of the file data. If the **FFS_ATTRIB_CHECKSUM** bit of the *Attributes* field is cleared to zero, the *IntegrityCheck.Checksum.File* field must be initialized with a value of 0xAA. The *IntegrityCheck.Checksum.File* field is valid any time the **EFI_FILE_DATA_VALID** bit is set in the *State* field. See "File Integrity and State" on page 26 for more details.

Checksum

IntegrityCheck. Checksum16 is the full 16 bits of the *IntegrityCheck* field.

```
//*************************************************************
// EFI_FV_FILETYPE
//*************************************************************
typedef UINT8 EFI_FV_FILETYPE;

//*************************************************************
// EFI_FFS_FILE_ATTRIBUTES
//*************************************************************
typedef UINT8 EFI_FFS_FILE_ATTRIBUTES;

// FFS File Attributes
#define FFS_ATTRIB_LARGE_FILE              0x01
#define FFS_ATTRIB_FIXED                   0x04
#define FFS_ATTRIB_DATA_ALIGNMENT          0x38
#define FFS_ATTRIB_CHECKSUM                0x40
```

Figure 9 depicts the bit allocation of the *Attributes* field in an FFS file's header.

7	6	5	4	3	2	1	0
Reserved. Must be set to 0	FFS_ATTRIB_CHECKSUM	FFS_ATTRIB_DATA_ALIGNMENT			FFS_ATTRIB_FIXED	**FFS_ATTRIB_DATA_ALIGNMENT_2**	FFS_ATTRIB_LARGE_FILE

Figure 9. Bit Allocation of FFS `Attributes`

Table 6 provides descriptions of the fields in the above definition.

Table 6. Bit Allocation Definitions

Value	Definition
FFS_ATTRIB_FIXED	Indicates that the file may not be moved from its present location.
FFS_ATTRIB_LARGE_FILE	Indicates that large files are supported and the **EFI_FFS_FILE_HEADER2** is in use.
FFS_ATTRIB_DATA_ALIGNMENT and FFS_ATTRIB_DATA_ALIGNMENT 2	Indicates that the beginning of the file data (not the file header) must be aligned on a particular boundary relative to the firmware volume base. The three bits in this field are an enumeration of alignment possibilities. The firmware volume interface allows alignments based on powers of two from byte alignment to 16MiB alignment. FFS does not support this full range. The table below maps all FFS supported alignments to **FFS_ATTRIB_DATA_ALIGNMENT** and **FFS_ATTRIB_DATA_ALIGNMENT2** values and firmware volume interface alignment values. No other alignments are supported by FFS. When a file with an alignment requirement is created, a pad file may need to be created before it to ensure proper data alignment. See “EFI_FV_FILETYPE_FFS_PAD” on page 12 for more information regarding pad files.
FFS_ATTRIB_CHECKSUM	Determines the interpretation of *IntegrityCheck.Checksum.File.* See the *IntegrityCheck* definition above for specific usage.

Table 7 maps all FFS-supported alignments to **FFS_ATTRIB_DATA_ALIGNMENT** and **FFS_ATTRIB_DATA_ALIGNMENT2** values and firmware volume interface alignment values.

Table 7. Supported FFS Alignments

Required Alignment (bytes)	Alignment Value in FFS *Attributes* Field	FFS_ATTRIB_DATA_ ALIGNMENT2 in FFS *Attributes* Field	Alignment Value in Firmware Volume Interfaces
1	0	0	0
16	1	0	4
128	2	0	7
512	3	0	9
1KiB	4	0	10
4KiB	5	0	12
32KiB	6	0	15
64KiB	7	0	16
128KiB	0	1	17
256KiB	1	1	18
512KiB	2	1	19
1MiB	3	1	20
2MiB	4	1	21
4MiB	5	1	22
8MiB	6	1	23
16MiB	7	1	24

```
//*************************************************************
// EFI_FFS_FILE_STATE
//*************************************************************
typedef UINT8 EFI_FFS_FILE_STATE;

// FFS File State Bits
#define EFI_FILE_HEADER_CONSTRUCTION      0x01
#define EFI_FILE_HEADER_VALID             0x02
#define EFI_FILE_DATA_VALID               0x04
#define EFI_FILE_MARKED_FOR_UPDATE        0x08
#define EFI_FILE_DELETED                  0x10
#define EFI_FILE_HEADER_INVALID           0x20
```

All other *State* bits are reserved and must be set to **EFI_FVB_ERASE_POLARITY**. See "File Integrity and State" on page 26 for an explanation of how these bits are used. Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** on page 32.

3.2.4 Firmware File Section

EFI_COMMON_SECTION_HEADER

Summary

Defines the common header for all the section types.

Prototype

```
typedef struct {
  UINT8                       Size[3];
  EFI_SECTION_TYPE            Type;
} EFI_COMMON_SECTION_HEADER;

typedef struct {
  UINT8                       Size[3];
  EFI_SECTION_TYPE            Type;
  UINT32                      ExtendedSize;
} EFI_COMMON_SECTION_HEADER2;
```

Parameters

Size

A 24-bit unsigned integer that contains the total size of the section in bytes, including the **EFI_COMMON_SECTION_HEADER**. For example, a zero-length section has a *Size* of 4 bytes.

Type

Declares the section type. Type **EFI_SECTION_TYPE** is defined in "Related Definitions" below.

ExtendedSize

If *Size* is 0xFFFFFF then *ExtendedSize* contains the size of the section. If *Size* is not equal to 0xFFFFFF then this field does not exist.

Description

The type **EFI_COMMON_SECTION_HEADER** defines the common header for all the section types.

If *Size* is 0xFFFFFF the size of the section header is sizeof (**EFI_COMMON_SECTION_HEADER2**). If *Size* is not equal to 0xFFFFFF then the size of the section header is sizeof (**EFI_COMMON_SECTION_HEADER**).

The **EFI_COMMON_SECTION_HEADER2** structure is only used if the section is too large to be described using **EFI_COMMON_SECTION_HEADER**. Large sections using **EFI_COMMON_SECTION_HEADER2** can only exist in a file using **EFI_FFS_FILE_HEADER2**, the **FFS_ATTRIB_LARGE_FILE** attribute in the file header is set.

Related Definitions

```
//************************************************************
// EFI_SECTION_TYPE
//************************************************************
typedef UINT8 EFI_SECTION_TYPE;

//************************************************************
// The section type EFI_SECTION_ALL is a pseudo type. It is
// used as a wild card when retrieving sections. The section
// type EFI_SECTION_ALL matches all section types.
//************************************************************
#define EFI_SECTION_ALL                               0x00

//************************************************************
// Encapsulation section Type values
//************************************************************
#define EFI_SECTION_COMPRESSION                       0x01
#define EFI_SECTION_GUID_DEFINED                      0x02
#define EFI_SECTION_DISPOSABLE                        0x03

//************************************************************
// Leaf section Type values
//************************************************************
#define EFI_SECTION_PE32                              0x10
#define EFI_SECTION_PIC                               0x11
#define EFI_SECTION_TE                                0x12
#define EFI_SECTION_DXE_DEPEX                         0x13
#define EFI_SECTION_VERSION                           0x14
#define EFI_SECTION_USER_INTERFACE                    0x15
#define EFI_SECTION_COMPATIBILITY16                   0x16
#define EFI_SECTION_FIRMWARE_VOLUME_IMAGE             0x17
#define EFI_SECTION_FREEFORM_SUBTYPE_GUID             0x18
#define EFI_SECTION_RAW                               0x19
#define EFI_SECTION_PEI_DEPEX                         0x1B
#define EFI_SECTION_MM_DEPEX                          0x1C
```

All other values are reserved for future use.

3.2.5 Firmware File Section Types

EFI_SECTION_COMPATIBILITY16

Summary

A leaf section type that contains an IA-32 16-bit executable image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER  EFI_COMPATIBILITY16_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_COMPATIBILITY16_SECTION2;
```

Description

A *Compatibility16 image section* is a leaf section that contains an IA-32 16-bit executable image. IA-32 16-bit legacy code that may be included in PI Architecture firmware is stored in a 16-bit executable image. **EFI_COMPATIBILITY16_SECTION2** is used if the section is 16MB or larger.

EFI_SECTION_COMPRESSION

Summary

An encapsulation section type in which the section data is compressed.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  UINT32                    UncompressedLength;
  UINT8                     CompressionType;
} EFI_COMPRESSION_SECTION;

typedef struct {
  EFI_COMMON_SECTION_HEADER2 CommonHeader;
  UINT32                     UncompressedLength;
  UINT8                      CompressionType;
} EFI_COMPRESSION_SECTION2;
```

Parameters

CommonHeader

Usual common section header. *CommonHeader.Type =* **EFI_SECTION_COMPRESSION**.

UncompressedLength

UINT32 that indicates the size of the section data after decompression.

CompressionType

Indicates which compression algorithm is used.

Description

A *compression section* is an encapsulation section in which the section data is compressed. To process the contents and extract the enclosed section stream, the section data must be decompressed using the decompressor indicated by the *CompressionType* parameter. The decompressed image is then interpreted as a section stream. **EFI_COMPRESSION_SECTION2** is used if the section is 16MB or larger.

Related Definitions

```
//************************************************************
// CompressionType values
//************************************************************
#define EFI_NOT_COMPRESSED             0x00
#define EFI_STANDARD_COMPRESSION       0x01
```

Table 8 describes the fields in the above definition.

Table 8. Description of Fields for `CompressionType`

Field	Description
`EFI_NOT_COMPRESSED`	Indicates that the encapsulated section stream is not compressed. This type is useful to grouping sections together without requiring a decompressor.
`EFI_STANDARD_COMPRESSION`	Indicates that the encapsulated section stream is compressed using the compression standard defined by the UEFI 2.0 specification.

EFI_SECTION_DISPOSABLE

Summary

An encapsulation section type in which the section data is disposable.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_DISPOSABLE_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_DISPOSABLE_SECTION2;
```

Parameters

None

Description

A disposable section is an encapsulation section in which the section data may be disposed of during the process of creating or updating a firmware image without significant impact on the usefulness of the file. The *Type* field in the section header is set to **EFI_SECTION_DISPOSABLE**. This allows optional or descriptive data to be included with the firmware file which can be removed in order to conserve space. The contents of this section are implementation specific, but might contain debug data or detailed integration instructions. **EFI_DISPOSABLE_SECTION2** is used if the section is 16MB or larger.

EFI_SECTION_DXE_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for a DXE driver.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_DXE_DEPEX_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_DXE_DEPEX_SECTION2;
```

Description

The *DXE dependency expression section* is a leaf section that contains a dependency expression that is used to determine the dispatch order for a DXE driver. See the *Platform Initialization Driver Execution Environment Core Interface Specification* for details regarding the format of the dependency expression. **EFI_DXE_DEPEX_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_FIRMWARE_VOLUME_IMAGE

Summary

A leaf section type that contains a PI Firmware Volume.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER
EFI_FIRMWARE_VOLUME_IMAGE_SECTION;

typedef EFI_COMMON_SECTION_HEADER2
EFI_FIRMWARE_VOLUME_IMAGE_SECTION2;
```

Description

A *firmware volume image section* is a leaf section that contains a PI Firmware Volume Image. **EFI_FIRMWARE_VOLUME_IMAGE_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_FREEFORM_SUBTYPE_GUID

Summary

A leaf section type that contains a single **EFI_GUID** in the header to describe the raw data.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  EFI_GUID                  SubTypeGuid;
} EFI_FREEFORM_SUBTYPE_GUID_SECTION;

typedef struct {
  EFI_COMMON_SECTION_HEADER2 CommonHeader;
  EFI_GUID                   SubTypeGuid;
} EFI_FREEFORM_SUBTYPE_GUID_SECTION2;
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type =* **EFI_SECTION_FREEFORM_SUBTYPE_GUID**.

SubtypeGuid

This GUID is defined by the creator of the file. It is a vendor-defined file type. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

A *free-form subtype GUID section* is a leaf section that contains a single **EFI_GUID**in the header to describe the raw data. It is typically used in files of type **EFI_FV_FILETYPE_FREEFORM** to provide an extensibility mechanism for file types. See "EFI_FV_FILETYPE_FREEFORM" on page 12 for more details about **EFI_FV_FILETYPE_FREEFORM** files.

EFI_SECTION_GUID_DEFINED

Summary

An encapsulation section type in which the method of encapsulation is defined by an identifying GUID.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  EFI_GUID                  SectionDefinitionGuid;
  UINT16                    DataOffset;
  UINT16                    Attributes;
  //                        GuidSpecificHeaderFields;
} EFI_GUID_DEFINED_SECTION;

typedef struct {
  EFI_COMMON_SECTION_HEADER2 CommonHeader;
  EFI_GUID                  SectionDefinitionGuid;
  UINT16                    DataOffset;
  UINT16                    Attributes;
  //                        GuidSpecificHeaderFields;
} EFI_GUID_DEFINED_SECTION2;
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type =* **EFI_SECTION_GUID_DEFINED.**

SectionDefinitionGuid

GUID that defines the format of the data that follows. It is a vendor-defined section type. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

DataOffset

Contains the offset in bytes from the beginning of the common header to the first byte of the data.

Attributes

Bit field that declares some specific characteristics of the section contents. The bits are defined in “Related Definitions” below.

GuidSpecificHeaderFields

Zero or more bytes of data that are defined by the section’s GUID. An example of this data would be a digital signature and manifest.

Data

Zero or more bytes of arbitrary data. The format of the data is defined by *SectionDefinitionGuid*.

Description

A *GUID-defined section* contains a section-type-specific header that contains an identifying GUID, followed by an arbitrary amount of data. It is an encapsulation section in which the method of encapsulation is defined by the GUID. A matching instance of **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** (DXE) or **EFI_GUIDED_SECTION_EXTRACTION_PPI** (PEI) is required to extract the contents of this encapsulation section.

The GUID-defined section enables custom encapsulation section types for any purpose. One commonly expected use is creating an encapsulation section to enable a cryptographic authentication of the section contents. **EFI_GUID_DEFINED_SECTION2** must be used if the section is 16MB or larger.

Related Definitions

```
//****************************************************
// Bit values for GuidedSectionHeader.Attributes
//****************************************************
#define EFI_GUIDED_SECTION_PROCESSING_REQUIRED     0x01
#define EFI_GUIDED_SECTION_AUTH_STATUS_VALID       0x02
```

Table 9 describes the fields in the above definition.

Table 9. Descriptions of Fields for *GuidedSectionHeader.Attributes*

Field	Description
EFI_GUIDED_SECTION_PROCESSING_REQUIRED	Set to 1 if the section requires processing to obtain meaningful data from the section contents. Processing would be required, for example, if the section contents were encrypted or compressed. If the **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** bit is cleared to zero, it is possible to retrieve the section's contents without processing in the absence of an associated instance of the **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** (DXE) or **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI** (PEI).. In this case, the beginning of the encapsulated section stream is indicated by the value of *DataOffset*.
EFI_GUIDED_SECTION_AUTH_STATUS_VALID	Set to 1 if the section contains authentication data that is reported through the *AuthenticationStatus* parameter returned from the **GUIDed Section Extraction Protocol**. If the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit is clear, the *AuthenticationStatus* parameter is not used.

All other bits are reserved and must be set to zero. Together, the **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** and **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bits provide the necessary data to set the proper bits of the *AuthenticationStatus* output parameter in the event that no **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** is available and the data is still returned.

EFI Signed Sections

For **EFI_GUID_DEFINED_SECTION** and **EFI_GUID_DEFINED_SECTION2** there is a *SectionDefinitionGuid* of type **EFI_FIRMWARE_CONTENTS_SIGNED_GUID.**

The *GuidSpecificHeaderFields* shall include an entry *SignatureInfo* of type **WIN_CERTIFICATE_UEFI_GUID**.

```
#define EFI_FIRMWARE_CONTENTS_SIGNED_GUID \
{ 0xf9d89e8, 0x9259, 0x4f76, \
{ 0xa5, 0xaf, 0xc, 0x89, 0xe3, 0x40, 0x23, 0xdf } }
```

The *signed section* is an encapsulation section in which the section data is cryptographically signed. To process the contents and extract the enclosed section stream, the section data integrity must be accessed by evaluating the enclosed data via the cryptographic information in the *SignatureInfo*. The *CertType* **= EFI_CERT_TYPE_PKCS7_GUID** or **EFI_CERT_TYPE_RSA2048_SHA256_GUID**.

The signed image is then interpreted as a section stream. **EFI_GUID_DEFINED_SECTION2** is used if the section is 16MB or larger.

EFI_SECTION_PE32

Summary

A leaf section type that contains a complete PE32+ image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_PE32_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_PE32_SECTION2;
```

Description

The *PE32+ image section* is a leaf section that contains a complete PE32+ image. Normal UEFI executables are stored within PE32+ images. **EFI_PE32_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_PEI_DEPEX

Summary

A leaf section type that is used to determine dispatch order for a PEIM.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_PEI_DEPEX_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_PEI_DEPEX_SECTION2;
```

Description

The *PEI dependency expression section* is a leaf section that contains a dependency expression that is used to determine dispatch order for a PEIM. See the *Platform Initialization Pre-EFI Initialization Core Interface Specification* for details regarding the format of the dependency expression. **EFI_PEI_DEPEX_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_PIC

Summary

A leaf section type that contains a position-independent-code (PIC) image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_PIC_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_PIC_SECTION2;
```

Description

A *PIC image section* is a leaf section that contains a position-independent-code (PIC) image.

In addition to normal PE32+ images that contain relocation information, PEIM executables may be PIC and are referred to as *PIC images.* A PIC image is the same as a PE32+ image except that all relocation information has been stripped from the image and the image can be moved and will execute correctly without performing any relocation or other fix-ups. **EFI_PIC_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_RAW

Summary

A leaf section type that contains an array of zero or more bytes.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_RAW_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_RAW_SECTION2;
```

Description

A *raw section* is a leaf section that contains an array of zero or more bytes. No particular formatting of these bytes is implied by this section type. **EFI_RAW_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_MM_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for an MM driver.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_MM_DEPEX_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_MM_DEPEX_SECTION2;
```

Description

The *MM dependency expression section* is a leaf section that contains a dependency expression that is used to determine the dispatch order for MM drivers. Before the MMRAM invocation of the MM driver's entry point, this dependency expression must evaluate to TRUE. See the *Platform Initialization Specification, Volume 2* for details regarding the format of the dependency expression.

The dependency expression may refer to protocols installed in either the UEFI or the MM protocol database. **EFI_MM_DEPEX_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_TE

Summary

A leaf section that contains a Terse Executable (TE) image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_TE_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_TE_SECTION2;
```

Description

The *terse executable section* is a leaf section that contains a Terse Executable (TE) image. A TE image is an executable image format specific to the PI Architecture that is used for storing executable images in a smaller amount of space than would be required by a full PE32+ image. Only PEI Foundation and PEIM files may contain a TE section. **EFI_TE_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_USER_INTERFACE

Summary

A leaf section type that contains a Unicode string that contains a human-readable file name.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  CHAR16                    FileNameString[];
} EFI_USER_INTERFACE_SECTION;

typedef struct {
  EFI_COMMON_SECTION_HEADER2 CommonHeader;
  CHAR16                     FileNameString[];
} EFI_USER_INTERFACE_SECTION2;
```

Description

The *user interface file name section* is a leaf section that contains a Unicode string that contains a human-readable file name.

This section is optional and is not required for any file types. There must never be more than one user interface file name section contained within a file. **EFI_USER_INTERFACE_SECTION2** must be used if the section is 16MB or larger.

EFI_SECTION_VERSION

Summary

A leaf section type that contains a numeric build number and an optional Unicode string that represents the file revision.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  UINT16                    BuildNumber;
  CHAR16                    VersionString[];
} EFI_VERSION_SECTION;

typedef struct {
  EFI_COMMON_SECTION_HEADER2 CommonHeader;
  UINT16                     BuildNumber;
  CHAR16                     VersionString[];
} EFI_VERSION_SECTION2;
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_VERSION**.

BuildNumber

A **UINT16** that represents a particular build. Subsequent builds have monotonically increasing build numbers relative to earlier builds.

VersionString

A null-terminated Unicode string that contains a text representation of the version. If there is no text representation of the version, then an empty string must be provided.

Description

A *version section* is a leaf section that contains a numeric build number and an optional Unicode string that represents the file revision.

To facilitate versioning of PEIMs, DXE drivers, and other files, a version section may be included in a file. There must never be more than one version section contained within a file. **EFI_VERSION_SECTION2** must be used if the section is 16MB or larger.

3.3 PEI

EFI_PEI_FIRMWARE_VOLUME_INFO_PPI

Summary

Provides location and format of a firmware volume.

GUID

```
#define EFI_PEI_FIRMWARE_VOLUME_INFO_PPI_GUID \
  { 0x49edb1c1, 0xbf21, 0x4761, \
  0xbb, 0x12, 0xeb, 0x0, 0x31, 0xaa, 0xbb, 0x39 }
```

Prototype

```
typedef struct _EFI_PEI_FIRMWARE_VOLUME_INFO_PPI {
  EFI_GUID FvFormat;
  VOID     *FvInfo;
  UINT32   FvInfoSize;
  EFI_GUID *ParentFvName;
  EFI_GUID *ParentFileName;
} EFI_PEI_FIRMWARE_VOLUME_INFO_PPI ;
```

Parameters

FvFormat

Unique identifier of the format of the memory-mapped firmware volume.

FvInfo

Points to a buffer which allows the **EFI_PEI_FIRMWARE_VOLUME_PPI** to process the volume. The format of this buffer is specific to the *FvFormat*. For memory-mapped firmware volumes, this typically points to the first byte of the firmware volume.

FvInfoSize

Size of the data provided by *FvInfo*. For memory-mapped firmware volumes, this is typically the size of the firmware volume.

ParentFvName, ParentFileName

If the firmware volume originally came from a firmware file, then these point to the parent firmware volume name and firmware volume file. If it did not originally come from a firmware file, these should be **NULL**.

Description

This PPI describes the location and format of a firmware volume. The *FvFormat* can be **EFI_FIRMWARE_FILE_SYSTEM2_GUID** or the GUID for a user-defined format. The **EFI_FIRMWARE_FILE_SYSTEM2_GUID** is the PI Firmware Volume format.

EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI

Summary

Provides location and format of a firmware volume.

GUID

```
#define EFI_PEI_FIRMWARE_VOLUME_INFO_PPI2_GUID \
{ 0xea7ca24b, 0xded5, 0x4dad, \
0xa3, 0x89, 0xbf, 0x82, 0x7e, 0x8f, 0x9b, 0x38 }
```

Prototype

```
typedef struct _EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI {
  EFI_GUID FvFormat;
  VOID          *FvInfo;
  UINT32        FvInfoSize;
  EFI_GUID      *ParentFvName;
  EFI_GUID      *ParentFileName;
  UINT32        AuthenticationStatus;
} EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI ;
```

Parameters

FvFormat

Unique identifier of the format of the memory-mapped firmware volume.

FvInfo

Points to a buffer which allows the **EFI_PEI_FIRMWARE_VOLUME_PPI** to process the volume. The format of this buffer is specific to the *FvFormat*. For memory-mapped firmware volumes, this typically points to the first byte of the firmware volume.

FvInfoSize

Size of the data provided by *FvInfo*. For memory-mapped firmware volumes, this is typically the size of the firmware volume.

ParentFvName, ParentFileName

If the firmware volume originally came from a firmware file, then these point to the parent firmware volume name and firmware volume file. If it did not originally come from a firmware file, these should be NULL.

AuthenticationStatus

Authentication status.

Description

This PPI describes the location, format and authentication status of a firmware volume. The *FvFormat* can be **EFI_FIRMWARE_FILE_SYSTEM2_GUID** or the GUID for a user-defined format. The **EFI_FIRMWARE_FILE_SYSTEM2_GUID** is the PI Firmware Volume format.

3.3.1 PEI Firmware Volume PPI

EFI_PEI_FIRMWARE_VOLUME_PPI

Summary

Provides functions for accessing a memory-mapped firmware volume of a specific format.

GUID

The GUID for this PPI is the same as the firmware volume format GUID.

Prototype

```
typedef struct _EFI_PEI_FIRMWARE_VOLUME_PPI {
  EFI_PEI_FV_PROCESS_FV        ProcessVolume;
  EFI_PEI_FV_FIND_FILE_TYPE    FindFileByType;
  EFI_PEI_FV_FIND_FILE_NAME    FindFileByName;
  EFI_PEI_FV_GET_FILE_INFO     GetFileInfo;
  EFI_PEI_FV_GET_INFO          GetVolumeInfo;
  EFI_PEI_FV_FIND_SECTION      FindSectionByType;
  EFI_PEI_FV_GET_FILE_INFO2    GetFileInfo2;
  EFI_PEI_FV_FIND_SECTION2     FindSectionByType2;
  UINT32                       Signature;
  UINT32                       Revision;
} EFI_PEI_FIRMWARE_VOLUME_PPI;
```

Parameters

ProcessVolume

Process a firmware volume and create a volume handle.

FindFileByType

Find all files of a specific type.

FindFileByName

Find the file with a specific name.

GetFileInfo

Return the information about a specific file

GetVolumeInfo

Return the firmware volume attributes.

FindSectionByType

Find the first section of a specific type.

GetFileInfo2

Return the information with authentication status about a specific file.

FindSectionByType2

Find the section with authentication status of a specific type.

Signature

Signature is used to keep backward-compatibility, set to {'P','F','V','P'}.

Revision

Revision for further extension.

```
# define EFI_PEI_FIRMWARE_VOLUME_PPI_REVISION 0x00010030
```

EFI_PEI_FIRMWARE_VOLUME_PPI.ProcessVolume()

Summary

Process a firmware volume and create a volume handle.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_PROCESS_FV) (
  IN  CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN  VOID                              *Buffer,
  IN  UINTN                             BufferSize,
  OUT EFI_PEI_FV_HANDLE                 *FvHandle
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

Buffer

Points to the start of the buffer.

BufferSize

Size of the buffer.

FvHandle

Points to the returned firmware volume handle. The firmware volume handle must be unique within the system. The type **EFI_PEI_FV_HANDLE** is defined in the PEI Services **FfsFindNextVolume()**.

Description

Create a volume handle from the information in the buffer. For memory-mapped firmware volumes, *Buffer* and *BufferSize* refer to the start of the firmware volume and the firmware volume size. For non memory-mapped firmware volumes, this points to a buffer which contains the necessary information for creating the firmware volume handle. Normally, these values are derived from the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI**.

Status Codes Returned

EFI_SUCCESS	Firmware volume handle created.
EFI_VOLUME_CORRUPTED	Volume was corrupt.

EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByType()

Summary

Finds the next file of the specified type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_FILE_TYPE) (
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN EFI_FV_FILETYPE                   SearchType,
  IN EFI_PEI_FV_HANDLE                 FvHandle,
  IN OUT EFI_PEI_FILE_HANDLE           *FileHandle
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

SearchType

A filter to find only files of this type. Type **EFI_FV_FILETYPE_ALL** causes no filtering to be done.

FvHandle

Handle of firmware volume in which to search.

FileHandle

Points to the current handle from which to begin searching or NULL to start at the beginning of the firmware volume. Updated upon return to reflect the file found.

Description

This service enables PEI modules to discover additional firmware files. The *FileHandle* must be unique within the system.

Status Codes Returned

EFI_SUCCESS	The file was found.
EFI_NOT_FOUND	The file was not found. *FileHandle* contains NULL.

EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByName()

Summary

Find a file within a volume by its name.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_FILE_NAME) (
  IN  CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN  CONST EFI_GUID                    *FileName,
  IN  EFI_PEI_FV_HANDLE                 *FvHandle,
  OUT EFI_PEI_FILE_HANDLE              *FileHandle
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

FileName

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

FvHandle

Upon entry, the pointer to the firmware volume to search or **NULL** if all firmware volumes should be searched. Upon exit, the actual firmware volume in which the file was found.

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 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	*FileHandle* or *FileName* was NULL.

EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo()

Summary

Returns information about a specific file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_GET_FILE_INFO) (
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN EFI_PEI_FILE_HANDLE               FileHandle,
  OUT EFI_FV_FILE_INFO                 *FileInfo
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

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.

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

EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo2()

Summary

Returns information about a specific file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_GET_FILE_INFO2) (
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN EFI_PEI_FILE_HANDLE               FileHandle,
  OUT EFI_FV_FILE_INFO2                *FileInfo
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

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.

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

EFI_PEI_FIRMWARE_VOLUME_PPI.GetVolumeInfo()

Summary

Return information about the firmware volume.

Prototypes

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_GET_INFO)(
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI  *This,
  IN EFI_PEI_FV_HANDLE                  FvHandle,
  OUT EFI_FV_INFO                       *VolumeInfo
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

FvHandle

Handle to the firmware handle.

VolumeInfo

Points to the returned firmware volume information.

Description

This function returns information about the firmware volume.

Status Codes Returned

EFI_SUCCESS	Information returned successfully.
EFI_INVALID_PARAMETER	*FvHandle* does not indicate a valid firmware volume or *VolumeInfo* is **NULL**

EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType()

Summary

Find the next matching section in the firmware file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_SECTION) (
  IN   CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN   EFI_SECTION_TYPE                  SearchType,
  IN   EFI_PEI_FILE_HANDLE               FileHandle,
  OUT VOID                               **SectionData
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

SearchType

A filter to find only sections of this type.

FileHandle

Handle of firmware file in which to search.

SectionData

Updated upon return to point to the section found.

Description

This service enables PEI modules to discover sections of a given type within a valid file.

Status Codes Returns

EFI_SUCCESS	Section was found.
EFI_NOT_FOUND	Section of the specified type was not found. *SectionData* contains **NULL**.

EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType2()

Summary

Find the next matching section in the firmware file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_SECTION2) (
  IN   CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN   EFI_SECTION_TYPE                  SearchType,
  IN UINTN                               SearchInstance,
  IN   EFI_PEI_FILE_HANDLE               FileHandle,
  OUT VOID                               **SectionData
  OUT UINT32                             *AuthenticationStatus
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

SearchType

A filter to find only sections of this type.

SearchInstance

A filter to find the specific instance of sections.

FileHandle

Handle of firmware file in which to search.

SectionData

Updated upon return to point to the section found.

AuthenticationStatus

Updated upon return to point to the authentication status for this section.

Description

This service enables PEI modules to discover sections of a given instance and type within a valid file.

Status Codes Returns

EFI_SUCCESS	Section was found.
EFI_NOT_FOUND	Section of the specified type was not found. *SectionData* contains **NULL**.

3.3.2 PEI Load File PPI

EFI_PEI_LOAD_FILE_PPI

Summary

Installed by a PEIM that supports the Load File PPI.

GUID

```
#define EFI_PEI_LOAD_FILE_PPI_GUID \
  { 0xb9e0abfe, 0x5979, 0x4914, \
  0x97, 0x7f, 0x6d, 0xee, 0x78, 0xc2, 0x78, 0xa6 }
```

Prototype

```
typedef struct _EFI_PEI_LOAD_FILE_PPI {
    EFI_PEI_LOAD_FILE LoadFile;
} EFI_PEI_LOAD_FILE_PPI;
```

Parameters

LoadFile

Loads a PEIM into memory for subsequent execution. See the **LoadFile()** function description.

Description

This PPI is a pointer to the Load File service. This service will be published by a PEIM. The PEI Foundation will use this service to launch the known PEI module images.

EFI_PEI_LOAD_FILE_PPI.LoadFile()

Summary

Loads a PEIM into memory for subsequent execution.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_LOAD_FILE) (
  IN     CONST EFI_PEI_LOAD_FILE_PPI *This,
  IN     EFI_PEI_FILE_HANDLE         FileHandle,
  OUT    EFI_PHYSICAL_ADDRESS        *ImageAddress,
  OUT    UINT64                      *ImageSize,
  OUT    EFI_PHYSICAL_ADDRESS        *EntryPoint,
  OUT    UINT32                      *AuthenticationState
  );
```

Parameters

This

Interface pointer that implements the Load File PPI instance.

FileHandle

File handle of the file to load. Type **EFI_PEI_FILE_HANDLE** is defined in **FfsFindNextFile()**.

ImageAddress

Pointer to the address of the loaded image.

ImageSize

Pointer to the size of the loaded image.

EntryPoint

Pointer to the entry point of the image.

AuthenticationState

On exit, points to the attestation authentication state of the image or 0 if no attestation was performed. The format of *AuthenticationState* is defined in **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()** on page 84

Description

This service is the single member function of **EFI_LOAD_FILE_PPI**. This service separates image loading and relocating from the PEI Foundation. For example, if there are compressed images or images that need to be relocated into memory for performance reasons, this service performs that transformation. This service is very similar to the **EFI_LOAD_FILE_PROTOCOL** in the UEFI 2.0 specification. The abstraction allows for an implementation of the **LoadFile()** service to support different image types in the future. There may be more than one instance of this PPI in the system.

For example, the PEI Foundation might support only XIP images natively, but another PEIM might contain support for relocatable images. There must be an **LoadFile()** instance that at least supports the PE/COFF and Terse Executable (TE) image format.

For sectioned files, this function should use **FfsFindSectionData** in order to find the executable image section.

This service must support loading of XIP images with or without copying them to a permanent memory. If the image within the specified file cannot be loaded because it must be copied into memory (either because the FV is not memory mapped or because the image contains relocations), and the permanent memory is not available, the function will return **EFI_NOT_SUPPORTED**. If permanent memory is available, then the PEIM should be loaded into permanent memory unless the image is not relocatable. If the image cannot be loaded into permanent memory due to insufficient amount of the available permanent memory, the function will return **EFI_WARN_BUFFER_TOO_SMALL** in case of XIP image, and **EFI_OUT_OF_RESOURCES** in case of non-XIP image. When **EFI_WARN_BUFFER_TOO_SMALL** is returned, all the output parameters are valid and the image can be invoked.

Any behavior PEIM which requires to be executed from code permanent memory should include wait for **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI** and **EFI_PEI_LOAD_FILE_PPI** to be installed.

Status Codes Returned

EFI_SUCCESS	The image was loaded successfully.
EFI_OUT_OF_RESOURCES	There was not enough memory.
EFI_LOAD_ERROR	There was no supported image in the file
EFI_INVALID_PARAMETER	*FileHandle* was not a valid firmware file handle.
EFI_INVALID_PARAMETER	*EntryPoint* was NULL.
EFI_UNSUPPORTED	An image requires relocations or is not memory mapped.
EFI_WARN_BUFFER_TOO_SMALL	There is not enough heap to allocate the requested size. This will not prevent the XIP image from being invoked.

3.3.3 PEI Guided Section Extraction PPI

EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI

Summary

If a GUID-defined section is encountered when doing section extraction, the PEI Foundation or the **EFI_PEI_FILE_LOADER_PPI** instance calls the appropriate instance of the GUIDed Section Extraction PPI to extract the section stream contained therein.

GUID

Typically, protocol interface structures are identified by associating them with a GUID. Each instance of a protocol with a given GUID must have the same interface structure. While all instances of the GUIDed Section Extraction PPI must have the same interface structure, they do not all have the same GUID. The GUID that is associated with an instance of the GUIDed Section Extraction Protocol is used to correlate it with the GUIDed section type that it is intended to process.

PPI Structure

```
typedef struct _EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI {
  EFI_PEI_EXTRACT_GUIDED_SECTION     ExtractSection;
} EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI;
```

Parameters

ExtractSection

Takes the GUIDed section as input and produces the section stream data. See the **ExtractSection()** function description.

EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()

Summary

Processes the input section and returns the data contained therein along with the authentication status.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_EXTRACT_GUIDED_SECTION)(
  IN CONST EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI *This,
  IN CONST VOID                                 *InputSection,
  OUT VOID                                      **OutputBuffer,
  OUT UINTN                                     *OutputSize,
  OUT UINT32                                    *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI** instance.

InputSection

Buffer containing the input GUIDed section to be processed.

OutputBuffer

**OutputBuffer* is allocated from PEI permanent memory and contains the new section stream.

OutputSize

A pointer to a caller-allocated **UINTN** in which the size of **OutputBuffer* allocation is stored. If the function returns anything other than **EFI_SUCCESS**, the value of **OutputSize* is undefined.

AuthenticationStatus

A pointer to a caller-allocated **UINT32** that indicates the authentication status of the output buffer. If the input section's *GuidedSectionHeader.Attributes* field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit as clear, **AuthenticationStatus* must return zero. These bits reflect the status of the extraction operation. If the function returns anything other than **EFI_SUCCESS**, the value of **AuthenticationStatus* is undefined.

Description

The **ExtractSection()** function processes the input section and returns a pointer to the section contents. If the section being extracted does not require processing (if the section's *GuidedSectionHeader.Attributes* has the **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** field cleared), then *OutputBuffer* is just updated to point to the start of the section's contents. Otherwise, **Buffer* must be allocated from PEI permanent memory.

If the section being extracted contains authentication information (the section's *GuidedSectionHeader.Attributes* field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit set), the values returned in *AuthenticationStatus* must reflect the results of the authentication operation.

If the section contains other encapsulation sections, their contents do not need to be extracted or decompressed.

Related Definitions

```
//************************************************************
// Bit values for AuthenticationStatus
//************************************************************
#define EFI_AUTH_STATUS_PLATFORM_OVERRIDE  0x01
#define EFI_AUTH_STATUS_IMAGE_SIGNED       0x02
#define EFI_AUTH_STATUS_NOT_TESTED         0x04
#define EFI_AUTH_STATUS_TEST_FAILED        0x08

// all other bits are reserved and must be 0
```

The bit definitions above lead to the evaluations of *AuthenticationStatus*: in Table 10.

Table 10. *AuthenticationStatus* **Bit Definitions**

Bit	Definition
xx00	Image was not signed.
xxx1	Platform security policy override. Assumes same meaning as 0010 (the image was signed, the signature was tested, and the signature passed authentication test).
0010	Image was signed, the signature was tested, and the signature passed authentication test.
0110	Image was signed and the signature was not tested. This can occur if there is no GUIDed Section Extraction Protocol available to process a GUID-defined section, but it was still possible to retrieve the data from the GUID-defined section directly.
1010	Image was signed, the signature was tested, and the signature failed the authentication test.
1110	To generate this code, there must be at least two layers of GUIDed encapsulations. In one layer, the *AuthenticationStatus* was returned as 0110; in another layer, it was returned as 1010. When these two results are OR-ed together, the aggregate result is 1110.

Status Codes Returned

EFI_SUCCESS	The *InputSection* was successfully processed and the section contents were returned.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The GUID in *InputSection* does not match this instance of the GUIDed Section Extraction PPI.

3.3.4 PEI Decompress PPI

EFI_PEI_DECOMPRESS_PPI

Summary

Provides decompression services to the PEI Foundatoin.

GUID

```
#define EFI_PEI_DECOMPRESS_PPI_GUID \
  { 0x1a36e4e7, 0xfab6, 0x476a, \
  { 0x8e, 0x75, 0x69, 0x5a, 0x5, 0x76, 0xfd, 0xd7 } }
```

PPI Structure

```
typedef struct _EFI_PEI_DECOMPRESS_PPI {
  EFI_PEI_DECOMPRESS_DECOMPRESS   Decompress;
} EFI_PEI_DECOMPRESS_PPI;
```

Members

Decompress

Decompress a single compression section in a firmware file. See **Decompress()** for more information.

Description

This PPI’s single member function decompresses a compression encapsulated section. It is used by the PEI Foundation to process sectioned files. Prior to the installation of this PPI, compression sections will be ignored.

EFI_PEI_DECOMPRESS_PPI.Decompress()

Summary

Decompress a single section.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DECOMPRESS_DECOMPRESS)(
  IN  CONST EFI_PEI_DECOMPRESS_PPI      *This,
  IN  CONST EFI_COMPRESSION_SECTION     *InputSection,
  OUT VOID                              **OutputBuffer,
  OUT UINTN                             *OutputSize
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_DECOMPRESS_PEI PPI**.

InputSection

Points to the compressed section.

OutputBuffer

Holds the returned pointer to the decompressed sections.

OutputSize

Holds the returned size of the decompress section streams.

Description

Decompresses the data in a compressed section and returns it as a series of standard PI Firmware File Sections. The required memory is allocated from permanent memory.

Status Codes Returned

EFI_SUCCESS	The section was decompressed successfully. *OutputBuffer* contains the resulting data and *OutputSize* contains the resulting size.
EFI_OUT_OF_RESOURCES	Unable to allocate sufficient memory to hold the decompressed data.
EFI_UNSUPPORTED	The compression type specified in the compression header is unsupported.

3.4 DXE

3.4.1 Firmware Volume2 Protocol

EFI_FIRMWARE_VOLUME2_PROTOCOL

Summary

The Firmware Volume Protocol provides file-level access to the firmware volume. Each firmware volume driver must produce an instance of the Firmware Volume Protocol if the firmware volume is to be visible to the system during the DXE phase. The Firmware Volume Protocol also provides mechanisms for determining and modifying some attributes of the firmware volume.

GUID

```
#define EFI_FIRMWARE_VOLUME2_PROTOCOL_GUID    \
  { 0x220e73b6, 0x6bdb, 0x4413, 0x84, 0x5, 0xb9, 0x74, \
  0xb1, 0x8, 0x61, 0x9a }
```

Protocol Interface Structure

```
typedef struct_EFI_FIRMWARE_VOLUME_PROTOCOL {
  EFI_FV_GET_ATTRIBUTES       GetVolumeAttributes;
  EFI_FV_SET_ATTRIBUTES       SetVolumeAttributes;
  EFI_FV_READ_FILE            ReadFile;
  EFI_FV_READ_SECTION         ReadSection;
  EFI_FV_WRITE_FILE           WriteFile;
  EFI_FV_GET_NEXT_FILE        GetNextFile;
  UINT32                      KeySize;
  EFI_HANDLE                  ParentHandle;
  EFI_FV_GET_INFO             GetInfo;
  EFI_FV_SET_INFO             SetInfo;
} EFI_FIRMWARE_VOLUME2_PROTOCOL;
```

Parameters

GetVolumeAttributes

Retrieves volume capabilities and current settings. See the **GetVolumeAttributes()** function description.

SetVolumeAttributes

Modifies the current settings of the firmware volume. See the **SetVolumeAttributes()** function description.

ReadFile

Reads an entire file from the firmware volume. See the **ReadFile()** function description.

ReadSection

Reads a single section from a file into a buffer. See the **ReadSection()** function description.

WriteFile

Writes an entire file into the firmware volume. See the **WriteFile()** function description.

GetNextFile

Provides service to allow searching the firmware volume. See the **GetNextFile()** function description.

KeySize

Data field that indicates the size in bytes of the *Key* input buffer for the **GetNextFile()** API.

ParentHandle

Handle of the parent firmware volume. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

GetInfo

Gets the requested file or volume information. See the **GetInfo()** function description.

SetInfo

Sets the requested file information. See the **SetInfo()** function description.

Description

The Firmware Volume Protocol contains the file-level abstraction to the firmware volume as well as some firmware volume attribute reporting and configuration services. The Firmware Volume Protocol is the interface used by all parts of DXE that are not directly involved with managing the firmware volume itself. This abstraction allows many varied types of firmware volume implementations. A firmware volume may be a flash device or it may be a file in the UEFI system partition, for example. This level of firmware volume implementation detail is not visible to the consumers of the Firmware Volume Protocol.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_GET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  OUT EFI_FV_ATTRIBUTES                   *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

FvAttributes

Pointer to an **EFI_FV_ATTRIBUTES** in which the attributes and current settings are returned. Type **EFI_FV_ATTRIBUTES** is defined in “Related Definitions” below.

Description

Because of constraints imposed by the underlying firmware storage, an instance of the Firmware Volume Protocol may not be to able to support all possible variations of this architecture. These constraints and the current state of the firmware volume are exposed to the caller using the **GetVolumeAttributes()** function.

GetVolumeAttributes() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **GetVolumeAttributes()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Related Definitions

```
//**************************************************************
// EFI_FV_ATTRIBUTES
//**************************************************************
typedef UINT64 EFI_FV_ATTRIBUTES;

//**************************************************************
// EFI_FV_ATTRIBUTES bit definitions
//**************************************************************
```

```
// EFI_FV_ATTRIBUTES bit semantics
#define EFI_FV2_READ_DISABLE_CAP      0x0000000000000001
#define EFI_FV2_READ_ENABLE_CAP       0x0000000000000002
#define EFI_FV2_READ_STATUS           0x0000000000000004

#define EFI_FV2_WRITE_DISABLE_CAP     0x0000000000000008
#define EFI_FV2_WRITE_ENABLE_CAP      0x0000000000000010
#define EFI_FV2_WRITE_STATUS          0x0000000000000020

#define EFI_FV2_LOCK_CAP              0x0000000000000040
#define EFI_FV2_LOCK_STATUS           0x0000000000000080
#define EFI_FV2_WRITE_POLICY_RELIABLE 0x0000000000000100

#define EFI_FV2_READ_LOCK_CAP         0x0000000000001000
#define EFI_FV2_READ_LOCK_STATUS      0x0000000000002000
#define EFI_FV2_WRITE_LOCK_CAP        0x0000000000004000
#define EFI_FV2_WRITE_LOCK_STATUS     0x0000000000008000
#define EFI_FV2_ALIGNMENT             0x00000000001F0000
```

```
#define EFI_FV2_ALIGNMENT_1           0x0000000000000000
#define EFI_FV2_ALIGNMENT_2           0x0000000000010000
#define EFI_FV2_ALIGNMENT_4           0x0000000000020000
#define EFI_FV2_ALIGNMENT_8           0x0000000000030000
#define EFI_FV2_ALIGNMENT_16          0x0000000000040000
#define EFI_FV2_ALIGNMENT_32          0x0000000000050000
#define EFI_FV2_ALIGNMENT_64          0x0000000000060000
#define EFI_FV2_ALIGNMENT_128         0x0000000000070000
#define EFI_FV2_ALIGNMENT_256         0x0000000000080000
#define EFI_FV2_ALIGNMENT_512         0x0000000000090000
#define EFI_FV2_ALIGNMENT_1K          0x00000000000A0000
#define EFI_FV2_ALIGNMENT_2K          0x00000000000B0000
#define EFI_FV2_ALIGNMENT_4K          0x00000000000C0000
#define EFI_FV2_ALIGNMENT_8K          0x00000000000D0000
#define EFI_FV2_ALIGNMENT_16K         0x00000000000E0000
#define EFI_FV2_ALIGNMENT_32K         0x00000000000F0000
#define EFI_FV2_ALIGNMENT_64K         0x0000000000100000
#define EFI_FV2_ALIGNMENT_128K        0x0000000000110000
#define EFI_FV2_ALIGNMENT_256K        0x0000000000120000
#define EFI_FV2_ALIGNMENT_512K        0x0000000000130000
#define EFI_FV2_ALIGNMENT_1M          0x0000000000140000
#define EFI_FV2_ALIGNMENT_2M          0x0000000000150000
#define EFI_FV2_ALIGNMENT_4M          0x0000000000160000
#define EFI_FV2_ALIGNMENT_8M          0x0000000000170000
#define EFI_FV2_ALIGNMENT_16M         0x0000000000180000
#define EFI_FV2_ALIGNMENT_32M         0x0000000000190000
#define EFI_FV2_ALIGNMENT_64M         0x00000000001A0000
#define EFI_FV2_ALIGNMENT_128M        0x00000000001B0000
#define EFI_FV2_ALIGNMENT_256M        0x00000000001C0000
#define EFI_FV2_ALIGNMENT_512M        0x00000000001D0000
#define EFI_FV2_ALIGNMENT_1G          0x00000000001E0000
#define EFI_FV2_ALIGNMENT_2G          0x00000000001F0000
```

Table 11 describes the fields in the above definition.

Table 11. Descriptions of Fields for EFI_FV_ATTRIBUTES

Field	Description
EFI_FV_READ_DISABLED_CAP	Set to 1 if it is possible to disable reads from the firmware volume.
EFI_FV_READ_ENABLED_CAP	Set to 1 if it is possible to enable reads from the firmware volume.
EFI_FV_READ_STATUS	Indicates the current read state of the firmware volume. Set to 1 if reads from the firmware volume are enabled.
EFI_FV_WRITE_DISABLED_CAP	Set to 1 if it is possible to disable writes to the firmware volume.
EFI_FV_WRITE_ENABLED_CAP	Set to 1 if it is possible to enable writes to the firmware volume.
EFI_FV_WRITE_STATUS	Indicates the current state of the firmware volume. Set to 1 if writes to the firmware volume are enabled.
EFI_FV_LOCK_CAP	Set to 1 if it is possible to lock firmware volume read/write attributes.
EFI_FV_LOCK_STATUS	Set to 1 if firmware volume attributes are locked down.
EFI_FV_WRITE_POLICY_RELIABLE	Set to 1 if the firmware volume supports “reliable” writes..
EFI_FV_READ_LOCK_CAP	Set to 1 if it is possible to lock the read status for the firmware volume.
EFI_FV_READ_LOCK_STATUS	Indicates the current read lock state of the firmware volume. Set to 1 if the read lock is currently enabled.
EFI_FV_WRITE_LOCK_CAP	Set to 1 if it is possible to lock the write status for the firmware volume.
EFI_FV_WRITE_LOCK_STATUS	Indicates the current write lock state of the firmware volume. Set to 1 if the write lock is currently enabled.
EFI_FV_ALIGNMENT	The first byte of the firmware volume must be at an address which is an even multiple of the alignment specified.

All other bits are reserved and are cleared to zero.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME2_PROTOCOL.SetVolumeAttributes()

Summary

Modifies the current settings of the firmware volume according to the input parameter.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_SET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  IN OUT EFI_FV_ATTRIBUTES         *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

FvAttributes

On input, *FvAttributes* is a pointer to an **EFI_FV_ATTRIBUTES** containing the desired firmware volume settings. On successful return, it contains the new settings of the firmware volume. On unsuccessful return, *FvAttributes* is not modified and the firmware volume settings are not changed. Type **EFI_FV_ATTRIBUTES** is defined in **GetVolumeAttributes()**.

Description

The **SetVolumeAttributes()** function is used to set configurable firmware volume attributes. Only **EFI_FV_READ_STATUS**, **EFI_FV_WRITE_STATUS**, and **EFI_FV_LOCK_STATUS** may be modified, and then only in accordance with the declared capabilities. All other bits of **FvAttributes* are ignored on input. On successful return, all bits of **FvAttributes* are valid and it contains the completed **EFI_FV_ATTRIBUTES** for the volume.

To modify an attribute, the corresponding status bit in the **EFI_FV_ATTRIBUTES** is set to the desired value on input. The **EFI_FV_LOCK_STATUS** bit does not affect the ability to read or write the firmware volume. Rather, once the **EFI_FV_LOCK_STATUS** bit is set, it prevents further modification to all the attribute bits.

SetVolumeAttributes() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **SetVolumeAttributes()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The requested firmware volume attributes were set and the resulting **EFI_FV_ATTRIBUTES** is returned in *FvAttributes*.

EFI_INVALID_PARAMETER	*FvAttributes*:**EFI_FV_READ_STATUS** is set to 1 on input, but the device does not support enabling reads (*FvAttributes*:**EFI_FV_READ_ENABLE_CAP** is clear on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	*FvAttributes*:**EFI_FV_READ_STATUS** is cleared to 0 on input, but the device does not support disabling reads (*FvAttributes*:**EFI_FV_READ_DISABLE_CAP** is clear on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	*FvAttributes*:**EFI_FV_WRITE_STATUS** is set to 1 on input, but the device does not support enabling writes (*FvAttributes*:**EFI_FV_WRITE_ENABLE_CAP** is clear on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	*FvAttributes*:**EFI_FV_WRITE_STATUS** is cleared to 0 on input, but the device does not support disabling writes (*FvAttributes*:**EFI_FV_WRITE_DISABLE_CAP** is clear on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	*FvAttributes*:**EFI_FV_LOCK_STATUS** is set on input, but the device does not support locking (*FvAttributes*:**EFI_FV_LOCK_CAP** is clear on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.
EFI_ACCESS_DENIED	Device is locked and does not allow attribute modification (*FvAttributes*:**EFI_FV_LOCK_STATUS** is set on return from **GetVolumeAttributes()**). Actual volume attributes are unchanged.

EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadFile()

Summary

Retrieves a file and/or file information from the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_READ_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL   *This,
  IN CONST EFI_GUID                       *NameGuid,
  IN OUT VOID                             **Buffer,
  IN OUT UINTN                            *BufferSize,
  OUT EFI_FV_FILETYPE                     *FoundType,
  OUT EFI_FV_FILE_ATTRIBUTES              *FileAttributes,
  OUT UINT32                              *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NameGuid

Pointer to an **EFI_GUID**, which is the file name. All firmware file names are **EFI_GUID**s. A single firmware volume must not have two valid files with the same file name **EFI_GUID**. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Buffer

Pointer to a pointer to a buffer in which the file contents are returned, not including the file header. See "Description" below for more details on the use of the *Buffer* parameter.

BufferSize

Pointer to a caller-allocated **UINTN.** It indicates the size of the memory represented by **Buffer*. See "Description" below for more details on the use of the *BufferSize* parameter.

FoundType

Pointer to a caller-allocated **EFI_FV_FILETYPE**. See "Firmware File Types" on page 8 for **EFI_FV_FILETYPE** related definitions.

FileAttributes

Pointer to a caller-allocated **EFI_FV_FILE_ATTRIBUTES**. Type **EFI_FV_FILE_ATTRIBUTES** is defined in "Related Definitions" below.

AuthenticationStatus

Pointer to a caller-allocated **UINT32** in which the authentication status is returned. See "Related Definitions" in **EFI_SECTION_EXTRACTION_PROCOCOL.ExtractSection()** for more information.

Description

ReadFile() is used to retrieve any file from a firmware volume during the DXE phase. The actual binary encoding of the file in the firmware volume media may be in any arbitrary format as long as it does the following:

- It is accessed using the Firmware Volume Protocol.
- The image that is returned follows the image format defined in Code Definitions: PI Firmware File Format.

If the input value of *Buffer==NULL*, it indicates the caller is requesting only that the type, attributes, and size of the file be returned and that there is no output buffer. In this case, the following occurs:

- **BufferSize* is returned with the size that is required to successfully complete the read.
- The output parameters **FoundType* and **FileAttributes* are returned with valid values.
- The returned value of **AuthenticationStatus* is undefined.

If the input value of *Buffer!=NULL*, the output buffer is specified by a double indirection of the *Buffer* parameter. The input value of **Buffer* is used to determine if the output buffer is caller allocated or is dynamically allocated by **ReadFile()**.

If the input value of **Buffer!=NULL*, it indicates the output buffer is caller allocated. In this case, the input value of **BufferSize* indicates the size of the caller-allocated output buffer. If the output buffer is not large enough to contain the entire requested output, it is filled up to the point that the output buffer is exhausted and **EFI_WARN_BUFFER_TOO_SMALL** is returned, and then **BufferSize* is returned with the size required to successfully complete the read. All other output parameters are returned with valid values.

If the input value of **Buffer==NULL*, it indicates the output buffer is to be allocated by **ReadFile()**. In this case, **ReadFile()** will allocate an appropriately sized buffer from boot services pool memory, which will be returned in **Buffer*. The size of the new buffer is returned in **BufferSize* and all other output parameters are returned with valid values.

ReadFile() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **ReadFile()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

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

```
//*************************************************************
// EFI_FV_FILE_ATTRIBUTES
//*************************************************************
typedef UINT32 EFI_FV_FILE_ATTRIBUTES;

#define EFI_FV_FILE_ATTRIB_ALIGNMENT         0x0000001F
#define EFI_FV_FILE_ATTRIB_FIXED             0x00000100
#define EFI_FV_FILE_ATTRIB_MEMORY_MAPPED     0x00000200
```

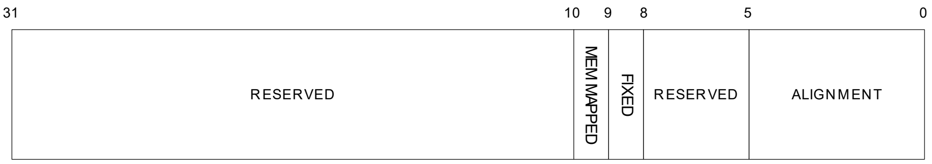


Figure 10. EFI_FV_FILE_ATTRIBUTES fields

This value is returned by **EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadFile()** and the PEI Service **FfsGetFileInfo()**. It is not the same as **EFI_FFS_FILE_ATTRIBUTES**.

The *Reserved* field must be set to zero.

The **EFI_FV_FILE_ATTRIB_ALIGNMENT** field indicates that the beginning of the file data (not the file header) must be aligned on a particular boundary relative to the beginning of the firmware volume. This alignment only makes sense for block-oriented firmware volumes. This field is an enumeration of alignment possibilities. The allowable alignments are powers of two from byte alignment to 2GB alignment. The supported alignments are described in Table 12. All other values are reserved.

Table 12. Supported Alignments for EFI_FV_FILE_ATTRIB_ALIGNMENT

Required Alignment (bytes)	Alignment Value in *Attributes* Field
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1KiB	10
2KiB	11
4KiB	12
8KiB	13
16KiB	14
32KiB	15
64KiB	16
128 KiB	17
256 KiB	18
512 KiB	19
1 MiB	20
2 MiB	21
4 MiB	22
8 MiB	23
16 MiB	24
32 MiB	25
64 MiB	26
128 MiB	27
256 MiB	28
512 MiB	29
1 GiB	30
2 GiB	31

The **EFI_FV_FILE_ATTRIB_FIXED** attribute indicates that the file has a fixed location and should not be moved (1) or may be moved to any address consistent with the alignment specified in **EFI_FV_FILE_ATTRIB_ALIGNMENT**.

The **EFI_FV_FILE_ATTRIB_MEMORY_MAPPED** attribute indicates that the file is memory mapped in the firmware volume and thus its contents may be accessed directly. If this is clear, then

Buffer is invalid. This value can be derived from the **EFI_FV_ATTRIBUTES** value returned by **EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes()** or the PEI Service **FfsGetVolumeInfo()**.

Status Codes Returned

EFI_SUCCESS	The call completed successfully.
EFI_WARN_BUFFER_TOO_SMALL	The buffer is too small to contain the requested output. The buffer is filled and the output is truncated.
EFI_OUT_OF_RESOURCES	An allocation failure occurred.
EFI_NOT_FOUND	*Name* was not found in the firmware volume.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.

EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadSection()

Summary

Locates the requested section within a file and returns it in a buffer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_READ_SECTION) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL   *This,
  IN CONST EFI_GUID                       *NameGuid,
  IN EFI_SECTION_TYPE                     SectionType,
  IN UINTN                                SectionInstance,
  IN OUT VOID                             **Buffer,
  IN OUT UINTN                            *BufferSize,
  OUT UINT32                              *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NameGuid

Pointer to an **EFI_GUID**, which indicates the file name from which the requested section will be read. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the Related Definitions for section 3.2.4.

SectionType

Indicates the section type to return. *SectionType* in conjunction with *SectionInstance* indicates which section to return. Type **EFI_SECTION_TYPE** is defined in **EFI_COMMON_SECTION_HEADER**.

SectionInstance

Indicates which instance of sections with a type of *SectionType* to return. *SectionType* in conjunction with *SectionInstance* indicates which section to return. *SectionInstance* is zero based.

Buffer

Pointer to a pointer to a buffer in which the section contents are returned, not including the section header. See “Description” below for more details on the usage of the *Buffer* parameter.

BufferSize

Pointer to a caller-allocated **UINTN.** It indicates the size of the memory represented by **Buffer*. See “Description” below for more details on the usage of the *BufferSize* parameter.

AuthenticationStatus

Pointer to a caller-allocated **UINT32** in which the authentication status is returned. See **EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()** for more information.

Description

ReadSection() is used to retrieve a specific section from a file within a firmware volume. The section returned is determined using a depth-first, left-to-right search algorithm through all sections found in the specified file. See "Firmware File Sections" on page 14 for more details about sections.

The output buffer is specified by a double indirection of the *Buffer* parameter. The input value of **Buffer* is used to determine if the output buffer is caller allocated or is dynamically allocated by **ReadSection()**.

If the input value of **Buffer!=NULL*, it indicates that the output buffer is caller allocated. In this case, the input value of **BufferSize* indicates the size of the caller-allocated output buffer. If the output buffer is not large enough to contain the entire requested output, it is filled up to the point that the output buffer is exhausted and **EFI_WARN_BUFFER_TOO_SMALL** is returned, and then **BufferSize* is returned with the size that is required to successfully complete the read. All other output parameters are returned with valid values.

If the input value of **Buffer==NULL*, it indicates the output buffer is to be allocated by **ReadSection()**. In this case, **ReadSection()** will allocate an appropriately sized buffer from boot services pool memory, which will be returned in **Buffer*. The size of the new buffer is returned in **BufferSize* and all other output parameters are returned with valid values.

ReadSection() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **ReadSection()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The call completed successfully.
EFI_WARN_BUFFER_TOO_SMALL	The caller-allocated buffer is too small to contain the requested output. The buffer is filled and the output is truncated.
EFI_OUT_OF_RESOURCES	An allocation failure occurred.
EFI_NOT_FOUND	The requested file was not found in the firmware volume.
EFI_NOT_FOUND	The requested section was not found in the specified file.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.
EFI_PROTOCOL_ERROR	The requested section was not found, but the file could not be fully parsed because a required **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** was not found. It is possible the requested section exists within the file and could be successfully extracted once the required **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** is published.

EFI_FIRMWARE_VOLUME2_PROTOCOL.WriteFile()

Summary

Writes one or more files to the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_WRITE_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN UINT32                          NumberOfFiles,
  IN EFI_FV_WRITE_POLICY             WritePolicy,
  IN EFI_FV_WRITE_FILE_DATA          *FileData
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NumberOfFiles

Indicates the number of elements in the array pointed to by *FileData*.

WritePolicy

Indicates the level of reliability for the write in the event of a power failure or other system failure during the write operation. Type **EFI_FV_WRITE_POLICY** is defined in "Related Definitions" below.

FileData

Pointer to an array of **EFI_FV_WRITE_FILE_DATA.** Each element of *FileData[]* represents a file to be written. Type **EFI_FV_WRITE_FILE_DATA** is defined in "Related Definitions" below.

Description

WriteFile() is used to write one or more files to a firmware volume. Each file to be written is described by an **EFI_FV_WRITE_FILE_DATA** structure.

The caller must ensure that any required alignment for all files listed in the *FileData* array is compatible with the firmware volume. Firmware volume capabilities can be determined using the **GetVolumeAttributes()** call.

Similarly, if the *WritePolicy* is set to **EFI_FV_RELIABLE_WRITE**, the caller must check the firmware volume capabilities to ensure **EFI_FV_RELIABLE_WRITE** is supported by the firmware volume. **EFI_FV_UNRELIABLE_WRITE** must always be supported.

Writing a file with a size of zero (*FileData[n].BufferSize == 0*) deletes the file from the firmware volume if it exists. Deleting a file must be done one at a time. Deleting a file as part of a multiple file write is not allowed.

WriteFile() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **WriteFile()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Related Definitions

```
//*******************************************************************
// EFI_FV_WRITE_POLICY
//*******************************************************************
typedef UINT32 EFI_FV_WRITE_POLICY

#define EFI_FV_UNRELIABLE_WRITE     0x00000000
#define EFI_FV_RELIABLE_WRITE       0x00000001
```

All other values of **EFI_FV_WRITE_POLICY** are reserved. Table 13 describes the fields in the above definition.

Table 13. Description of fields for EFI_FV_WRITE_POLICY

Field	Description
EFI_FV_UNRELIABLE_WRITE	This value in the *WritePolicy* parameter indicates that there is no required reliability if a power failure or other system failure occurs during a write operation. Updates may leave a combination of old and new files. Data loss, including complete loss of all files involved, is also permissible. In essence, no guarantees are made regarding what files will be present following a system failure during a write with a *WritePolicy* of **EFI_FV_UNRELIABLE_WRITE**. The advantage of this mode is that it can be implemented to use much less space in the storage media. Space-constrained firmware volumes may be able to support writes where it would be otherwise impossible.
EFI_FV_RELIABLE_WRITE	This value in the *WritePolicy* parameter indicates that, on the next initialization of the firmware volume following a power failure or other system failure during a write, all files listed in the *FileData* array are completely written and are valid, or none is written and the state of the firmware volume is the same as it was before the write operation was attempted.

```
//*******************************************************************
// EFI_FV_WRITE_FILE_DATA
//*******************************************************************
```

```
typedef struct {
  EFI_GUID                  *NameGuid,
  EFI_FV_FILETYPE           Type,
  EFI_FV_FILE_ATTRIBUTES    FileAttributes
  VOID                      *Buffer,
  UINT32                    BufferSize
} EFI_FV_WRITE_FILE_DATA;
```

NameGuid

Pointer to a GUID, which is the file name to be written. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Type

Indicates the type of file to be written. Type **EFI_FV_FILETYPE** is defined in "Related Definitions" of **EFI_FFS_FILE_HEADER** on page 41.

FileAttributes

Indicates the attributes for the file to be written. Type **EFI_FV_FILE_ATTRIBUTES** is defined in **ReadFile()**.

Buffer

Pointer to a buffer containing the file to be written.

BufferSize

Indicates the size of the file image contained in *Buffer*.

Status Codes Returned

EFI_SUCCESS	The write completed successfully.
EFI_OUT_OF_RESOURCES	The firmware volume does not have enough free space to storefile(s).
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_WRITE_PROTECTED	The firmware volume is configured to disallow writes.
EFI_NOT_FOUND	A delete was requested, but the requested file was not found in the firmware volume.
EFI_INVALID_PARAMETER	A delete was requested with a multiple file write.
EFI_INVALID_PARAMETER	An unsupported *WritePolicy* was requested.
EFI_INVALID_PARAMETER	An unknown file type was specifiedspecified or the specified file type is not supported by the firmware file system.
EFI_INVALID_PARAMETER	A file system specific error has occurred.

Other than **EFI_DEVICE_ERROR**, all error codes imply the firmware volume has not been modified. In the case of **EFI_DEVICE_ERROR**, the firmware volume may have been corrupted and appropriate repair steps must be taken.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetNextFile()

Summary

Retrieves information about the next file in the firmware volume store that matches the search criteria.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_GET_NEXT_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  IN OUT VOID                           *Key,
  IN OUT EFI_FV_FILETYPE                *FileType,
  OUT EFI_GUID                          *NameGuid,
  OUT EFI_FV_FILE_ATTRIBUTES            *Attributes,
  OUT UINTN                             *Size
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

Key

Pointer to a caller-allocated buffer that contains implementation-specific data that is used to track where to begin the search for the next file. The size of the buffer must be at least *This->KeySize* bytes long. To re-initialize the search and begin from the beginning of the firmware volume, the entire buffer must be cleared to zero. Other than clearing the buffer to initiate a new search, the caller must not modify the data in the buffer between calls to **GetNextFile()**.

FileType

Pointer to a caller-allocated **EFI_FV_FILETYPE**. The **GetNextFile()** API can filter its search for files based on the value of the **FileType* input. A **FileType* input of **EFI_FV_FILETYPE_ALL** causes **GetNextFile()** to search for files of all types. If a file is found, the file's type is returned in **FileType*. **FileType* is not modified if no file is found. See"Related Definitions" of **EFI_FFS_FILE_HEADER** on page 41.

NameGuid

Pointer to a caller-allocated **EFI_GUID**. If a matching file is found, the file's name is returned in **NameGuid*. If no matching file is found, **NameGuid* is not modified. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Attributes

Pointer to a caller-allocated **EFI_FV_FILE_ATTRIBUTES**. If a matching file is found, the file's attributes are returned in **Attributes*. If no matching file is found, ***Attributes*** is not modified. Type **EFI_FV_FILE_ATTRIBUTES** is defined in **ReadFile()**.

Size

Pointer to a caller-allocated **UINTN**. If a matching file is found, the file's size is returned in ***Size**. If no matching file is found, ***Size*** is not modified.

Description

GetNextFile() is the interface that is used to search a firmware volume for a particular file. It is called successively until the desired file is located or the function returns **EFI_NOT_FOUND**.

To filter uninteresting files from the output, the type of file to search for may be specified in ***FileType***. For example, if ***FileType*** is **EFI_FV_FILETYPE_DRIVER**, only files of this type will be returned in the output. If ***FileType*** is **EFI_FV_FILETYPE_ALL**, no filtering of file types is done.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.

The *Key* parameter is used to indicate a starting point of the search. If the buffer ***Key*** is completely initialized to zero, the search re-initialized and starts at the beginning. Subsequent calls to **GetNextFile()** must maintain the value of ***Key*** returned by the immediately previous call. The actual contents of ***Key*** are implementation specific and no semantic content is implied.

GetNextFile() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **GetNextFile()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	**The output parameters are filled with data obtained from the first matching file that was found.**
EFI_NOT_FOUND	No files of type *FileType* were found.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetInfo()

Summary

Return information about a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FV_GET_INFO) (
  IN      CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN      CONST EFI_GUID                      *InformationType,
  IN OUT  UINTN                               *BufferSize,
  OUT     VOID                                *Buffer
  );
```

Parameters

This

A pointer to the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance that is the file handle the requested information is for.

InformationType

The type identifier for the information being requested. Type **EFI_GUID** is defined in the UEFI 2.0 specification.

BufferSize

On input, the size of Buffer. On output, the amount of data returned in *Buffer*. In both cases, the size is measured in bytes.

Buffer

A pointer to the data buffer to return. The buffer's type is indicated by InformationType.

Description

The **GetInfo()** function returns information of type *InformationType* for the requested firmware volume. If the volume does not support the requested information type, then **EFI_UNSUPPORTED** is returned. If the buffer is not large enough to hold the requested structure, **EFI_BUFFER_TOO_SMALL** is returned and the *BufferSize* is set to the size of buffer that is required to make the request. The information types defined by this specification are required information types that all file systems must support.

Status Codes Returned

EFI_SUCCESS	The information was retrieved.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.

EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small to read the current directory entry. *BufferSize* has been updated with the size needed to complete the request.

EFI_FIRMWARE_VOLUME2_PROTOCOL.SetInfo()

Summary

Sets information about a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FV_SET_INFO) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN CONST EFI_GUID                      *InformationType,
  IN UINTN                               BufferSize,
  IN CONST VOID                          *Buffer
  );
```

Parameters

This

A pointer to the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance that is the file handle the information is for.

InformationType

The type identifier for the information being set. Type **EFI_GUID** is defined in the UEFI 2.0 specification.

BufferSize

The size, in bytes, of *Buffer*.

Buffer

A pointer to the data buffer to write. The buffer's type is indicated by *InformationType*.

Description

The **SetInfo()** function sets information of type *InformationType* on the requested firmware volume.

Status Codes Returned

EFI_SUCCESS	The information was set.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The media is read only.
EFI_VOLUME_FULL	The volume is full.
EFI_BAD_BUFFER_SIZE	*BufferSize* is smaller than the size of the type indicated by *InformationType*.

3.4.2 Firmware Volume Block2 Protocol

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL

Summary

This optional protocol provides control over block-oriented firmware devices.

GUID

```
//{8F644FA9-E850-4db1-9CE2-0B44698E8DA4}
#define EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL_GUID \
  {0x8f644fa9, 0xe850, 0x4db1, 0x9c, 0xe2, 0xb, 0x44, \
  0x69, 0x8e, 0x8d, 0xa4}
```

Protocol Interface Structure

```
typedef struct _EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL {
  EFI_FVB_GET_ATTRIBUTES              GetAttributes;
  EFI_FVB_SET_ATTRIBUTES              SetAttributes;
  EFI_FVB_GET_PHYSICAL_ADDRESS        GetPhysicalAddress;
  EFI_FVB_GET_BLOCK_SIZE              GetBlockSize;
  EFI_FVB_READ                        Read;
  EFI_FVB_WRITE                       Write;
  EFI_FVB_ERASE_BLOCKS                EraseBlocks;
  EFI_HANDLE                          ParentHandle;
} EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL;
```

Parameters

GetAttributes

Retrieves the current volume attributes. See the **GetAttributes()** function description.

SetAttributes

Sets the current volume attributes. See the **SetAttributes()** function description.

GetPhysicalAddress

Retrieves the memory-mapped address of the firmware volume. See the **GetPhysicalAddress()** function description.

GetBlockSize

Retrieves the size for a specific block. Also returns the number of consecutive similarly sized blocks. See the **GetBlockSize()** function description.

Read

Reads *n* bytes into a buffer from the firmware volume hardware. See the **Read()** function description.

Write

Writes *n* bytes from a buffer into the firmware volume hardware. See the **Write()** function description.

EraseBlocks

Erases specified block(s) and sets all values as indicated by the **EFI_FVB_ERASE_POLARITY** bit. See the **EraseBlocks()** function description. Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

ParentHandle

Handle of the parent firmware volume. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The Firmware Volume Block Protocol is the low-level interface to a firmware volume. File-level access to a firmware volume should not be done using the Firmware Volume Block Protocol. Normal access to a firmware volume must use the Firmware Volume Protocol. Typically, only the file system driver that produces the Firmware Volume Protocol will bind to the Firmware Volume Block Protocol.

The Firmware Volume Block Protocol provides the following:

- Byte-level read/write functionality.
- Block-level erase functionality.
- It further exposes device-hardening features, such as may be required to protect the firmware from unwanted overwriting and/or erasure.
- It is useful to layer a file system driver on top of the Firmware Volume Block Protocol. This file system driver produces the Firmware Volume Protocol, which provides file-level access to a firmware volume. The Firmware Volume Protocol abstracts the file system that is used to format the firmware volume and the hardware device-hardening features that may be present.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_ATTRIBUTES) (
  IN  CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL   *This,
  OUT EFI_FVB_ATTRIBUTES_2                        *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Attributes

Pointer to **EFI_FVB_ATTRIBUTES_2** in which the attributes and current settings are returned. Type **EFI_FVB_ATTRIBUTES_2** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

Description

The **GetAttributes()** function retrieves the attributes and current settings of the block.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.SetAttributes()

Summary

Modifies the current settings of the firmware volume according to the input parameter.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_SET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL   *This,
  IN OUT EFI_FVB_ATTRIBUTES_2                    *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Attributes

On input, *Attributes* is a pointer to **EFI_FVB_ATTRIBUTES_2** that contains the desired firmware volume settings. On successful return, it contains the new settings of the firmware volume. Type **EFI_FVB_ATTRIBUTES_2** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

Description

The **SetAttributes()** function sets configurable firmware volume attributes and returns the new settings of the firmware volume.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.
EFI_INVALID_PARAMETER	The attributes requested are in conflict with the capabilities as declared in the firmware volume header.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetPhysicalAddress ()

Summary

Retrieves the physical address of a memory-mapped firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_PHYSICAL_ADDRESS) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL  *This,
  OUT EFI_PHYSICAL_ADDRESS                      *Address
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Address

Pointer to a caller-allocated **EFI_PHYSICAL_ADDRESS** that, on successful return from **GetPhysicalAddress()**, contains the base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Description

The **GetPhysicalAddress()** function retrieves the base address of a memory-mapped firmware volume. This function should be called only for memory-mapped firmware volumes.

Status Codes Returned

EFI_SUCCESS	The firmware volume base address is returned.
EFI_UNSUPPORTED	The firmware volume is not memory mapped.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetBlockSize()

Summary

Retrieves the size in bytes of a specific block within a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_BLOCK_SIZE) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL  *This,
  IN EFI_LBA                                    Lba,
  OUT UINTN                                     *BlockSize,
  OUT UINTN                                     *NumberOfBlocks
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

Indicates the block for which to return the size. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

BlockSize

Pointer to a caller-allocated **UINTN** in which the size of the block is returned.

NumberOfBlocks

Pointer to a caller-allocated **UINTN** in which the number of consecutive blocks, starting with *Lba*, is returned. All blocks in this range have a size of *BlockSize*.

Description

The **GetBlockSize()** function retrieves the size of the requested block. It also returns the number of additional blocks with the identical size. The **GetBlockSize()** function is used to retrieve the block map (see **EFI_FIRMWARE_VOLUME_HEADER**).

Status Codes Returned

EFI_SUCCESS	The firmware volume base address is returned.
EFI_INVALID_PARAMETER	The requested LBA is out of range.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Read()

Summary

Reads the specified number of bytes into a buffer from the specified block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_READ) (
  IN   CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL  *This,
  IN   EFI_LBA                                    Lba,
  IN   UINTN                                      Offset,
  IN   OUT UINTN                                  *NumBytes,
  OUT  UINT8                                      *Buffer,
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

The starting logical block index from which to read. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

Offset

Offset into the block at which to begin reading.

NumBytes

Pointer to a **UINTN**. At entry, **NumBytes* contains the total size of the buffer. At exit, **NumBytes* contains the total number of bytes read.

Buffer

Pointer to a caller-allocated buffer that will be used to hold the data that is read.

Description

The **Read()** function reads the requested number of bytes from the requested block and stores them in the provided buffer.

Implementations should be mindful that the firmware volume might be in the *ReadDisabled* state. If it is in this state, the **Read()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the buffer.

The **Read()** function must also prevent spanning block boundaries. If a read is requested that would span a block boundary, the read must read up to the boundary but not beyond. The output parameter *NumBytes* must be set to correctly indicate the number of bytes actually read. The caller must be aware that a read may be partially completed.

Status Codes Returned

EFI_SUCCESS	The firmware volume was read successfully and contents are in *Buffer*.
EFI_BAD_BUFFER_SIZE	Read attempted across an LBA boundary. On output, *NumBytes* contains the total number of bytes returned in *Buffer*.
EFI_ACCESS_DENIED	The firmware volume is in the *ReadDisabled* state.
EFI_DEVICE_ERROR	The block device is not functioning correctly and could not be read.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Write()

Summary

Writes the specified number of bytes from the input buffer to the block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_WRITE) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL   *This,
  IN EFI_LBA                                     Lba,
  IN UINTN                                       Offset,
  IN OUT UINTN                                   *NumBytes,
  IN UINT8                                       *Buffer
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

The starting logical block index to write to. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

Offset

Offset into the block at which to begin writing.

NumBytes

Pointer to a **UINTN**. At entry, **NumBytes* contains the total size of the buffer. At exit, **NumBytes* contains the total number of bytes actually written.

Buffer

Pointer to a caller-allocated buffer that contains the source for the write.

Description

The **Write()** function writes the specified number of bytes from the provided buffer to the specified block and offset.

If the firmware volume is sticky write, the caller must ensure that all the bits of the specified range to write are in the **EFI_FVB_ERASE_POLARITY** state before calling the **Write()** function, or else the result will be unpredictable. This unpredictability arises because, for a sticky-write firmware volume, a write may negate a bit in the **EFI_FVB_ERASE_POLARITY** state but it cannot flip it back again. In general, before calling the **Write()** function, the caller should call the **EraseBlocks()** function first to erase the specified block to write. A block erase cycle will transition bits from the **(NOT)EFI_FVB_ERASE_POLARITY** state back to the **EFI_FVB_ERASE_POLARITY** state.

Implementations should be mindful that the firmware volume might be in the *WriteDisabled* state. If it is in this state, the **Write()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the firmware volume.

The **Write()** function must also prevent spanning block boundaries. If a write is requested that spans a block boundary, the write must store up to the boundary but not beyond. The output parameter *NumBytes* must be set to correctly indicate the number of bytes actually written. The caller must be aware that a write may be partially completed.

All writes, partial or otherwise, must be fully flushed to the hardware before the **Write()** service returns.

Status Codes Returned

EFI_SUCCESS	The firmware volume was written successfully.
EFI_BAD_BUFFER_SIZE	The write was attempted across an LBA boundary. On output, *NumBytes* contains the total number of bytes actually written.
EFI_ACCESS_DENIED	The firmware volume is in the *WriteDisabled* state.
EFI_DEVICE_ERROR	The block device is malfunctioning and could not be written.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.EraseBlocks()

Summary

Erases and initializes a firmware volume block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_ERASE_BLOCKS) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL  *This,
  …
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

…

The variable argument list is a list of tuples. Each tuple describes a range of LBAs to erase and consists of the following:

- An **EFI_LBA** that indicates the starting LBA
- A **UINTN** that indicates the number of blocks to erase

The list is terminated with an **EFI_LBA_LIST_TERMINATOR**. Type **EFI_LBA_LIST_TERMINATOR** is defined in “Related Definitions” below.

For example, the following indicates that two ranges of blocks (5–7 and 10–11) are to be erased:

```
EraseBlocks (This, 5, 3, 10, 2, EFI_LBA_LIST_TERMINATOR);
```

Description

The **EraseBlocks()** function erases one or more blocks as denoted by the variable argument list. The entire parameter list of blocks must be verified before erasing any blocks. If a block is requested that does not exist within the associated firmware volume (it has a larger index than the last block of the firmware volume), the **EraseBlocks()** function must return the status code **EFI_INVALID_PARAMETER** without modifying the contents of the firmware volume.

Implementations should be mindful that the firmware volume might be in the *WriteDisabled* state. If it is in this state, the **EraseBlocks()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the firmware volume.

All calls to **EraseBlocks()** must be fully flushed to the hardware before the **EraseBlocks()** service returns.

Related Definitions

```
//**************************************************
// EFI_LBA_LIST_TERMINATOR
//**************************************************
#define EFI_LBA_LIST_TERMINATOR 0xFFFFFFFFFFFFFFFF
```

Status Codes Returned

EFI_SUCCESS	The erase request was successfully completed.
EFI_ACCESS_DENIED	The firmware volume is in the *WriteDisabled* state.
EFI_DEVICE_ERROR	The block device is not functioning correctly and could not be written. The firmware device may have been partially erased.
EFI_INVALID_PARAMETER	One or more of the LBAs listed in the variable argument list do not exist in the firmware volume.

3.4.3 Guided Section Extraction Protocol

EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL

Summary

If a GUID-defined section is encountered when doing section extraction, the section extraction driver calls the appropriate instance of the GUIDed Section Extraction Protocol to extract the section stream contained therein.

GUID

Typically, protocol interface structures are identified by associating them with a GUID. Each instance of a protocol with a given GUID must have the same interface structure. While all instances of the GUIDed Section Extraction Protocol must have the same interface structure, they do not all have the same GUID. The GUID that is associated with an instance of the GUIDed Section Extraction Protocol is used to correlate it with the GUIDed section type that it is intended to process.

Protocol Interface Structure

```
typedef struct _EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL {
  EFI_EXTRACT_GUIDED_SECTION     ExtractSection;
} EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL;
```

Parameters

ExtractSection

Takes the GUIDed section as input and produces the section stream data. See the **ExtractSection()** function description.

EFI_GUIDED_SECTION_EXTRACTION_PROCOCOL.ExtractSection()

Summary

Processes the input section and returns the data contained therein along with the authentication status.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EXTRACT_GUIDED_SECTION)(
  IN CONST EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL *This,
  IN CONST VOID                  *InputSection,
  OUT VOID                       **OutputBuffer,
  OUT UINTN                      *OutputSize,
  OUT UINT32                     *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** instance.

InputSection

Buffer containing the input GUIDed section to be processed.

OutputBuffer

**OutputBuffer* is allocated from boot services pool memory and contains the new section stream. The caller is responsible for freeing this buffer.

OutputSize

A pointer to a caller-allocated **UINTN** in which the size of **OutputBuffer* allocation is stored. If the function returns anything other than **EFI_SUCCESS**, the value of **OutputSize* is undefined.

AuthenticationStatus

A pointer to a caller-allocated **UINT32** that indicates the authentication status of the output buffer. If the input section's *GuidedSectionHeader.Attributes* field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit as clear, **AuthenticationStatus* must return zero. Both local bits (19:16) and aggregate bits (3:0) in *AuthenticationStatus* are returned by **ExtractSection()**. These bits reflect the status of the extraction operation. The bit pattern in both regions must be the same, as the local and aggregate authentication statuses have equivalent meaning at this level. If the function returns anything other than **EFI_SUCCESS**, the value of **AuthenticationStatus* is undefined.

Description

The **ExtractSection()** function processes the input section and allocates a buffer from the pool in which it returns the section contents.

If the section being extracted contains authentication information (the section's **GuidedSectionHeader.Attributes** field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit set), the values returned in *AuthenticationStatus* must reflect the results of the authentication operation.

Depending on the algorithm and size of the encapsulated data, the time that is required to do a full authentication may be prohibitively long for some classes of systems. To indicate this, use **EFI_SECURITY_POLICY_PROTOCOL_GUID**, which may be published by the security policy driver (see the *Platform Initialization Driver Execution Environment Core Interface Specification* for more details and the GUID definition). If the **EFI_SECURITY_POLICY_PROTOCOL_GUID** exists in the handle database, then, if possible, full authentication should be skipped and the section contents simply returned in the *OutputBuffer*. In this case, the **EFI_AUTH_STATUS_PLATFORM_OVERRIDE** bit *AuthenticationStatus* must be set on return. See "Related Definitions" in **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()** on page 84 for the definition of type **EFI_AUTH_STATUS_PLATFORM_OVERRIDE**.

ExtractSection() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **ExtractSection()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The *InputSection* was successfully processed and the section contents were returned.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The GUID in *InputSection* does not match this instance of the GUIDed Section Extraction Protocol.

3.5 SMM

3.5.1 SMM Firmware Volume Protocol

EFI_SMM_FIRMWARE_VOLUME_PROTOCOL

Summary

The Firmware Volume Protocol provides file-level access to the firmware volume in SMM.

GUID

```
#define EFI_SMM_FIRMWARE_VOLUME_PROTOCOL_GUID { \
 0x19e9da84, 0x72b, 0x4274, 0xb3, 0x2e, 0xc, 0x8, 0x2, 0xe7,
0x17, 0xa5 \
}
```

Prototype

Same as **EFI_FIRMWARE_VOLUME2_PROTOCOL**;

Description

The Firmware Volume Protocol provides file-level access to the firmware volume in SMM.

The function API is same as DXE version **EFI_FIRMWARE_VOLUME2_PROTOCOL**.

3.5.2 SMM Firmware Volume Block Protocol

EFI_SMM_FIRMWARE_VOLUME_BLOCK_PROTOCOL

Summary

This optional protocol provides control over block-oriented firmware devices in SMM.

GUID

```
#define EFI_SMM_FIRMWARE_VOLUME_BLOCK_PROTOCOL_GUID { \
 0xd326d041, 0xbd31, 0x4c01, 0xb5, 0xa8, 0x62, 0x8b, 0xe8, 0x7f,
0x06, 0x53 \
}
```

Prototype

Same as **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL**;

Description

This optional protocol provides control over block-oriented firmware devices in SMM.

The function API is same as DXE version **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL**.

4 HOB Design Discussion

4.1 Explanation of HOB Terms

Because HOBs are the key architectural mechanism that is used to hand off system information in the early preboot stages and because not all implementations of the PI Architecture will use the Pre-EFI Initialization (PEI) and Driver Execution Environment (DXE) phases, this specification refrains from using the PEI and DXE nomenclature used in other PI specifications.

Instead, this specification uses the following terms to refer to the phases that deal with HOBs:

- HOB producer phase
- HOB consumer phase

The *HOB producer phase* is the preboot phase in which HOBs and the HOB list are created. The *HOB consumer phase* is the preboot phase to which the HOB list is passed and then consumed.

If the PI Architecture implementation incorporates the PEI and DXE, the HOB producer phase is the PEI phase and the HOB consumer phase is the DXE phase. The producer and consumer can change, however, depending on the implementation.

The following table translates the terminology used in this specification with that used in other PI specifications.

Table 14. Translation of HOB Specification Terminology

Term Used in the HOB Specification	Term Used in Other PI Specifications
HOB producer phase	PEI phase
HOB consumer phase	DXE phase
executable content in the HOB producer phase	Pre-EFI Initialization Module (PEIM)
hand-off into the HOB consumer phase	DXE Initial Program Load (IPL) PEIM or DXE IPL PEIM-to-PEIM Interface (PPI)
platform boot-policy phase	Boot Device Selection (BDS) phase

4.2 HOB Overview

The HOB producer phase provides a simple mechanism to allocate memory for data storage during the phase’s execution. The data store is architecturally defined and described by HOBs. This data store is also passed to the HOB producer phase when it is invoked from the HOB producer phase.

The basic container of data storage is named a *Hand-Off Block,* or HOB. HOBs are allocated sequentially in memory that is available to executable content in the HOB producer phase. There are a series of services that facilitate HOB manipulation. The sequential list of HOBs in memory will be referred to as the *HOB list.*

For definitions of the various HOB types, see section 5 below. The construction semantics are described in section 4.5 below.

4.3 Example HOB Producer Phase Memory Map and Usage

Figure 11 shows an example of the HOB producer phase memory map and its usage. This map is a possible means by which to subdivide the region.

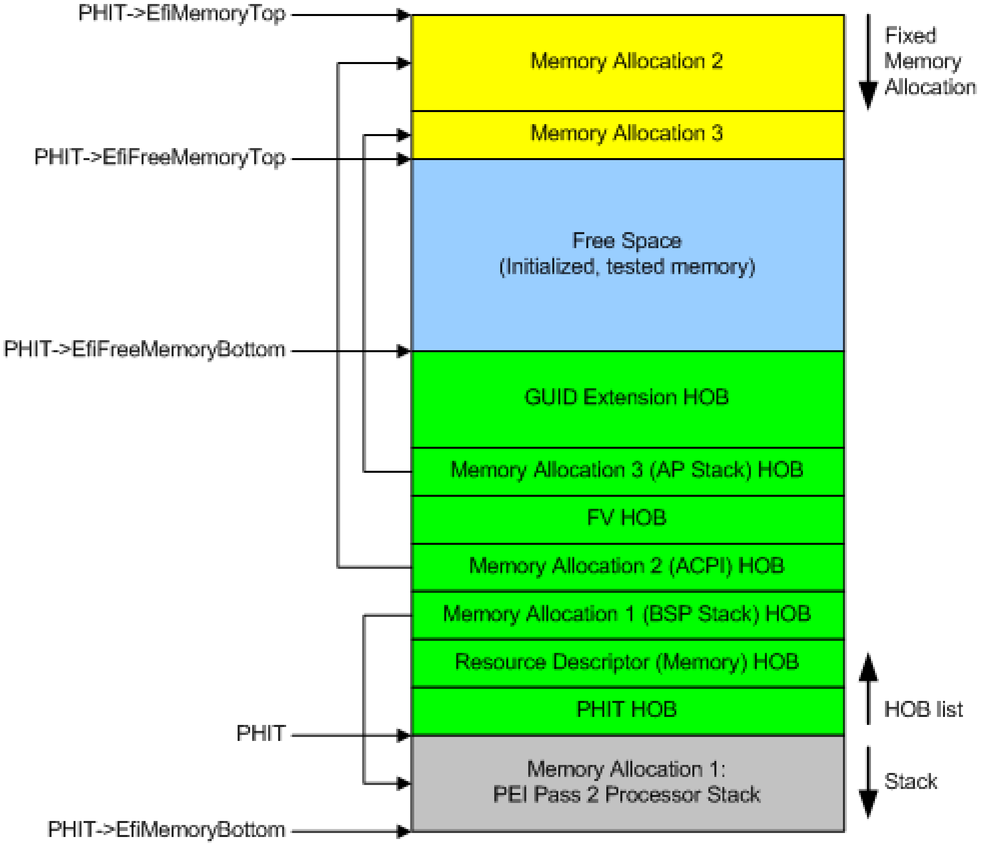

PHIT->EfiMemoryTop
PHIT->EfiFreeMemoryTop
PHIT->EfiFreeMemoryBottom
PHIT
PHIT->EfiMemoryBottom
Memory Allocation 2
Memory Allocation 3
Free Space
(Initialized, tested memory)
GUID Extension HOB
Memory Allocation 3 (AP Stack) HOB
FV HOB
Memory Allocation 2 (ACPI) HOB
Memory Allocation 1 (BSP Stack) HOB
Resource Descriptor (Memory) HOB
PHIT HOB
Memory Allocation 1:
PEI Pass 2 Processor Stack
Fixed
Memory
Allocation
HOB list
Stack

Figure 11. Example HOB Producer Phase Memory Map and Usage

4.4 HOB List

The first HOB in the HOB list must be the Phase Handoff Information Table (PHIT) HOB. The last HOB in the HOB list must be the End of HOB List HOB.

Only HOB producer phase components are allowed to make additions or changes to HOBs. Once the HOB list is passed into the HOB consumer phase, it is effectively read only. The ramification of a read-only HOB list is that handoff information, such as boot mode, must be handled in a distinguished fashion. For example, if the HOB consumer phase were to engender a recovery condition, it would not update the boot mode but instead would implement the action using a special type of reset call. The HOB list contains system state data at the time of HOB producer–to–HOB producer handoff and does not represent the current system state during the HOB consumer phase.

4.5 Constructing the HOB List

4.5.1 Constructing the Initial HOB List

The HOB list is initially built by the HOB producer phase. The HOB list is created in memory that is present, initialized, and tested. Once the initial HOB list has been created, the physical memory cannot be remapped, interleaved, or otherwise moved by a subsequent software agent.

The HOB producer phase **must** build the following three HOBs in the initial HOB list before exposing the list to other modules:

- The PHIT HOB
- A memory allocation HOB describing where the boot-strap processor (BSP) stack for permanent memory is located

  or

  A memory allocation HOB describing where the BSP store for permanent memory is located (Itanium® processor family only)
- A resource descriptor HOB that describes a physical memory range encompassing the HOB producer phase memory range with its attributes set as present, initialized, and tested

The HOB list creator may build more HOBs into the initial HOB list, such as additional HOBs to describe other physical memory ranges. There can also be additional modules, which might include a HOB producer phase–specific HOB to record memory errors discovered during initialization.

When the HOB producer phase completes its list creation, it exposes a pointer to the PHIT HOB to other modules.

4.5.2 HOB Construction Rules

HOB construction must obey the following rules:

1. All HOBs must start with a HOB generic header. This requirement allows users to locate the HOBs in which they are interested while skipping the rest. See the **EFI_HOB_GENERIC_HEADER** definition.
2. HOBs may contain boot services data that is available during the HOB producer and consumer phases only until the HOB consumer phase is terminated.
3. HOBs may be relocated in system memory by the HOB consumer phase. HOBs must not contain pointers to other data in the HOB list, including that in other HOBs. The table must be able to be copied without requiring internal pointer adjustment.
4. All HOBs must be multiples of 8 bytes in length. This requirement meets the alignment restrictions of the Itanium® processor family.
5. The PHIT HOB must always begin on an 8-byte boundary. Due to this requirement and requirement #4 in this list, all HOBs will begin on an 8-byte boundary.
6. HOBs are added to the end of the HOB list. HOBs can only be added to the HOB list during the HOB producer phase, not the HOB consumer phase.
7. HOBs cannot be deleted. The generic HOB header of each HOB must describe the length of the HOB so that the next HOB can be found. A private GUIDed HOB may provide a mechanism to

mark some or its entire contents invalid; however, this mechanism is beyond the scope of this document.

Note: *The HOB list must be valid (i.e., no HOBs "under construction") when any HOB producer phase service is invoked. Another HOB producer phase component's function might walk the HOB list, and if a HOB header contains invalid data, it might cause unreliable operation.*

4.5.3 Adding to the HOB List

To add a HOB to the HOB list, HOB consumer phase software must obtain a pointer to the PHIT HOB (start of the HOB list) and follow these steps:

1. Determine *NewHobSize*, where *NewHobSize* is the size in bytes of the HOB to be created.
2. Check free memory to ensure that there is enough free memory to allocate the new HOB. This test is performed by checking that *NewHobSize* <= PHIT->*EfiFreeMemoryTop* - PHIT->*EfiFreeMemoryBottom*).
3. Construct the HOB at PHIT->*EfiFreeMemoryBottom*.
4. Set PHIT->*EfiFreeMemoryBottom* = PHIT->*EfiFreeMemoryBottom* + *NewHobSize*.

5 HOB Code Definitions

5.1 HOB Introduction

This section contains the basic definitions of various HOBs. All HOBs consist of a generic header, **EFI_HOB_GENERIC_HEADER**, that specifies the type and length of the HOB. Each HOB has additional data beyond the generic header, according to the HOB type. The following data types and structures are defined in this section:

- **EFI_HOB_GENERIC_HEADER**
- **EFI_HOB_HANDOFF_INFO_TABLE**
- **EFI_HOB_MEMORY_ALLOCATION**
- **EFI_HOB_MEMORY_ALLOCATION_STACK**
- **EFI_HOB_MEMORY_ALLOCATION_BSP_STORE**
- **EFI_HOB_MEMORY_ALLOCATION_MODULE**
- **EFI_HOB_RESOURCE_DESCRIPTOR**
- **EFI_HOB_GUID_TYPE**
- **EFI_HOB_FIRMWARE_VOLUME**
- **EFI_HOB_FIRMWARE_VOLUME2**
- **EFI_HOB_FIRMWARE_VOLUME3**
- **EFI_HOB_CPU**
- **EFI_HOB_MEMORY_POOL**
- **EFI_HOB_UEFI_CAPSULE**
- **EFI_HOB_TYPE_UNUSED**
- **EFI_HOB_TYPE_END_OF_HOB_LIST**

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 data structure definition:

- **EFI_HOB_MEMORY_ALLOCATION_HEADER**
- **EFI_RESOURCE_TYPE**
- **EFI_RESOURCE_ATTRIBUTE_TYPE**

5.2 HOB Generic Header

EFI_HOB_GENERIC_HEADER

Summary

Describes the format and size of the data inside the HOB. All HOBs must contain this generic HOB header.

Prototype

```
typedef struct _EFI_HOB_GENERIC_HEADER{
  UINT16  HobType;
  UINT16  HobLength;
  UINT32  Reserved;
} EFI_HOB_GENERIC_HEADER;
```

Parameters

HobType

Identifies the HOB data structure type. See “Related Definitions” below for the HOB types that are defined in this specification.

HobLength

The length in bytes of the HOB.

Reserved

For this version of the specification, this field must always be set to zero.

Description

All HOBs have a common header that is used for the following:

- Traversing to the next HOB
- Describing the format and size of the data inside the HOB

Related Definitions

The following values for *HobType* are defined by this specification.

```
//*******************************************************
// HobType values
//*******************************************************

#define EFI_HOB_TYPE_HANDOFF              0x0001
#define EFI_HOB_TYPE_MEMORY_ALLOCATION    0x0002
#define EFI_HOB_TYPE_RESOURCE_DESCRIPTOR  0x0003
#define EFI_HOB_TYPE_GUID_EXTENSION       0x0004
#define EFI_HOB_TYPE_FV                   0x0005
#define EFI_HOB_TYPE_CPU                  0x0006
#define EFI_HOB_TYPE_MEMORY_POOL          0x0007
#define EFI_HOB_TYPE_FV2                  0x0009
#define EFI_HOB_TYPE_LOAD_PEIM_UNUSED     0x000A
#define EFI_HOB_TYPE_UEFI_CAPSULE         0x000B
#define EFI_HOB_TYPE_FV3                  0x000C
#define EFI_HOB_TYPE_UNUSED               0xFFFE
#define EFI_HOB_TYPE_END_OF_HOB_LIST      0xffff
```

Other values for *HobType* are reserved for future use by this specification.

5.3 PHIT HOB

EFI_HOB_HANDOFF_INFO_TABLE (PHIT HOB)

Summary

Contains general state information used by the HOB producer phase. This HOB must be the first one in the HOB list.

Prototype

```
typedef struct _EFI_HOB_HANDOFF_INFO_TABLE {
  EFI_HOB_GENERIC_HEADER  Header;
  UINT32                  Version;
  EFI_BOOT_MODE           BootMode;
  EFI_PHYSICAL_ADDRESS    EfiMemoryTop;
  EFI_PHYSICAL_ADDRESS    EfiMemoryBottom;
  EFI_PHYSICAL_ADDRESS    EfiFreeMemoryTop;
  EFI_PHYSICAL_ADDRESS    EfiFreeMemoryBottom;
  EFI_PHYSICAL_ADDRESS    EfiEndOfHobList;
} EFI_HOB_HANDOFF_INFO_TABLE;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_HANDOFF**.

Version

The version number pertaining to the PHIT HOB definition. See “Related Definitions” below for the version numbers defined by this specification. This value is 4 bytes in length to provide an 8-byte aligned entry when it is combined with the 4-byte *BootMode*.

BootMode

The system boot mode as determined during the HOB producer phase. Type **EFI_BOOT_MODE** is a **UINT32**; if the PI Architecture-compliant implementation incorporates the PEI phase, the possible bit values are defined in the *Platform Initialization Pre-EFI Initialization Core Interface Specification* (PEI CIS).

EfiMemoryTop

The highest address location of memory that is allocated for use by the HOB producer phase. This address must be 4-KiB aligned to meet page restrictions of UEFI. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

EfiMemoryBottom

The lowest address location of memory that is allocated for use by the HOB producer phase.

EfiFreeMemoryTop

The highest address location of free memory that is currently available for use by the HOB producer phase. This address must be 4-KiB aligned to meet page restrictions of UEFI.

EfiFreeMemoryBottom

The lowest address location of free memory that is available for use by the HOB producer phase.

EfiEndOfHobList

The end of the HOB list.

Description

The Phase Handoff Information Table (PHIT) HOB must be the first one in the HOB list. A pointer to this HOB is available to a HOB producer phase component through some service. This specification commonly refers to this HOB as the *PHIT HOB,* or sometimes the *handoff HOB.*

The HOB consumer phase reads the PHIT HOB during its initialization.

Related Definitions

```
//*****************************************************
// Version values
//*****************************************************

#define EFI_HOB_HANDOFF_TABLE_VERSION  0x0009
```

5.4 Memory Allocation HOB

5.4.1 Memory Allocation HOB

EFI_HOB_MEMORY_ALLOCATION

Summary

Describes all memory ranges used during the HOB producer phase that exist outside the HOB list. This HOB type describes how memory is used, not the physical attributes of memory.

Prototype

```
typedef struct _EFI_HOB_MEMORY_ALLOCATION {
  EFI_HOB_GENERIC_HEADER            Header;
  EFI_HOB_MEMORY_ALLOCATION_HEADER  AllocDescriptor;
  //
  // Additional data pertaining to the “Name” Guid memory
  // may go here.
  //
} EFI_HOB_MEMORY_ALLOCATION;
```

Parameters

Header

The HOB generic header. **Header.HobType = EFI_HOB_TYPE_MEMORY_ALLOCATION**.

AllocDescriptor

An instance of the **EFI_HOB_MEMORY_ALLOCATION_HEADER** that describes the various attributes of the logical memory allocation. The type field will be used for subsequent inclusion in the UEFI memory map. Type **EFI_HOB_MEMORY_ALLOCATION_HEADER** is defined in “Related Definitions” below.

Description

The memory allocation HOB is used to describe memory usage outside the HOB list. The HOB consumer phase does not make assumptions about the contents of the memory that is allocated by the memory allocation HOB, and it will not move the data unless it has explicit knowledge of the memory allocation HOB’s *Name* (**EFI_GUID**). Memory may be allocated in either the HOB producer phase memory area or other areas of present and initialized system memory.

The HOB consumer phase reads all memory allocation HOBs and allocates memory into the system memory map based on the following fields of **EFI_HOB_MEMORY_ALLOCATION_HEADER** of each memory allocation HOB:

- *MemoryBaseAddress*
- *MemoryLength*
- *MemoryType*

The HOB consumer phase does not parse the GUID-specific data identified by the *Name* field of each memory allocation HOB, except for a specific set of memory allocation HOBs that defined by this specification. A HOB consumer phase driver that corresponds to the specific *Name* GUIDed memory allocation HOB can parse the HOB list to find the specifically named memory allocation HOB and then manipulate the memory space as defined by the usage model for that GUID.

Note: *Special design care should be taken to ensure that two HOB consumer phase components do not modify memory space that is described by a memory allocation HOB, because unpredictable behavior might result.*

This specification defines a set of memory allocation HOBs that are architecturally used to allocate memory used by the HOB producer and consumer phases. Additionally, the following memory allocation HOBs are defined specifically for use by the final stage of the HOB producer phase to describe the processor state prior to handoff into the HOB consumer phase:

- BSP stack memory allocation HOB
- BSP store memory allocation HOB
- Memory allocation module HOB

Related Definitions

```
//*******************************************************
// EFI_HOB_MEMORY_ALLOCATION_HEADER
//*******************************************************

typedef struct _EFI_HOB_MEMORY_ALLOCATION_HEADER {
  EFI_GUID              Name;
  EFI_PHYSICAL_ADDRESS  MemoryBaseAddress;
  UINT64                MemoryLength;
  EFI_MEMORY_TYPE       MemoryType;  // UINT32
  UINT8                 Reserved[4]; // Padding for Itanium®
                                     // processor family
} EFI_HOB_MEMORY_ALLOCATION_HEADER;
```

Name

A GUID that defines the memory allocation region's type and purpose, as well as other fields within the memory allocation HOB. This GUID is used to define the additional data within the HOB that may be present for the memory allocation HOB. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

MemoryBaseAddress

The base address of memory allocated by this HOB. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

MemoryLength

The length in bytes of memory allocated by this HOB.

MemoryType

Defines the type of memory allocated by this HOB. The memory type definition follows the **EFI_MEMORY_TYPE** definition. Type **EFI_MEMORY_TYPE** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Reserved

For this version of the specification, this field will always be set to zero.

Note: *MemoryBaseAddress and MemoryLength must each have 4-KiB granularity to meet the page size requirements of UEFI.*

5.4.2 Boot-Strap Processor (BSP) Stack Memory Allocation HOB

EFI_HOB_MEMORY_ALLOCATION_STACK

Summary

Describes the memory stack that is produced by the HOB producer phase and upon which all post-memory-installed executable content in the HOB producer phase is executing.

GUID

```
#define EFI_HOB_MEMORY_ALLOC_STACK_GUID  \
  {0x4ed4bf27, 0x4092, 0x42e9, 0x80, 0x7d, 0x52, 0x7b, \
  0x1d, 0x0, 0xc9, 0xbd}
```

Prototype

```
typedef struct _EFI_HOB_MEMORY_ALLOCATION_STACK {
  EFI_HOB_GENERIC_HEADER            Header;
  EFI_HOB_MEMORY_ALLOCATION_HEADER  AllocDescriptor;
} EFI_HOB_MEMORY_ALLOCATION_STACK;
```

Parameters

Header

The HOB generic header. *Header.HobType =* **EFI_HOB_TYPE_MEMORY_ALLOCATION**.

AllocDescriptor

An instance of the **EFI_HOB_MEMORY_ALLOCATION_HEADER** that describes the various attributes of the logical memory allocation. The type field will be used for subsequent inclusion in the UEFI memory map. Type **EFI_HOB_MEMORY_ALLOCATION_HEADER** is defined in **EFI_HOB_MEMORY_ALLOCATION**.

Description

This HOB describes the memory stack that is produced by the HOB producer phase and upon which all post-memory-installed executable content in the HOB producer phase is executing. It is necessary for the hand-off into the HOB consumer phase to know this information so that it can appropriately map this stack into its own execution environment and describe it in any subsequent memory maps.

The HOB consumer phase reads this HOB during its initialization. The HOB consumer phase may elect to move or relocate the BSP's stack to meet size and location requirements that are defined by the HOB consumer phase's implementation. Therefore, other HOB consumer phase components cannot rely on the BSP stack memory allocation HOB to describe where the BSP stack is located during execution of the HOB consumer phase.

Note: *BSP stack memory allocation HOB must be valid at the time of hand off to the HOB consumer phase. If BSP stack is reallocated during HOB producer phase, the component that reallocates the stack must also update BSP stack memory allocation HOB.*

The BSP stack memory allocation HOB without any additional qualification describes either of the following:

- The stack that is currently consumed by the BSP.
- The processor that is currently executing the HOB producer phase and its executable content.
- The model for the PI architecture and the HOB producer phase is that of a single-threaded execution environment, so it is this single, distinguished thread of control whose environment is described by this HOB. The Itanium® processor family has the additional requirement of having to describe the value of the **BSPSTORE (AR18)** (“Backing Store Pointer Store”) register, which holds the successive location in memory where the Itanium processor family Register Stack Engine (RSE) will spill its values.
- In addition, Itanium®-based systems feature a system architecture where all processors come out of reset and execute the reset path concurrently. As such, the stack resources that are consumed by these alternate agents need to be described even though they are not responsible for executing the main thread of control through the HOB producer and consumer phases.

5.4.3 Boot-Strap Processor (BSP) BSPSTORE Memory Allocation HOB

EFI_HOB_MEMORY_ALLOCATION_BSP_STORE

Note: *This HOB is valid for the Itanium® processor family only.*

Summary

Defines the location of the boot-strap processor (BSP) BSPStore (“Backing Store Pointer Store”) register overflow store.

GUID

```
#define EFI_HOB_MEMORY_ALLOC_BSP_STORE_GUID  \
  {0x564b33cd, 0xc92a, 0x4593, 0x90, 0xbf, 0x24, 0x73, \
  0xe4, 0x3c, 0x63, 0x22}
```

Prototype

```
typedef struct _EFI_HOB_MEMORY_ALLOCATION_BSP_STORE {
  EFI_HOB_GENERIC_HEADER            Header;
  EFI_HOB_MEMORY_ALLOCATION_HEADER  AllocDescriptor;
} EFI_HOB_MEMORY_ALLOCATION_BSP_STORE;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_MEMORY_ALLOCATION**.

AllocDescriptor

An instance of the **EFI_HOB_MEMORY_ALLOCATION_HEADER** that describes the various attributes of the logical memory allocation. The type field will be used for subsequent inclusion in the UEFI memory map. Type **EFI_HOB_MEMORY_ALLOCATION_HEADER** is defined in the HOB type **EFI_HOB_MEMORY_ALLOCATION**.

Description

The HOB consumer phase reads this HOB during its initialization. The HOB consumer phase may elect to move or relocate the BSP’s register store to meet size and location requirements that are defined by the HOB consumer phase’s implementation. Therefore, other HOB consumer phase components cannot rely on the BSP store memory allocation HOB to describe where the BSP store is located during execution of the HOB consumer phase.

Note: *BSP BSPSTORE memory allocation HOB must be valid at the time of hand off to the HOB consumer phase. If BSP BSPSTORE is reallocated during HOB producer phase, the component that reallocates the stack must also update BSP BSPSTORE memory allocation HOB.*

This HOB is valid for the Itanium processor family only.

5.4.4 Memory Allocation Module HOB

EFI_HOB_MEMORY_ALLOCATION_MODULE

Summary

Defines the location and entry point of the HOB consumer phase.

GUID

```
#define EFI_HOB_MEMORY_ALLOC_MODULE_GUID  \
  {0xf8e21975, 0x899, 0x4f58, 0xa4, 0xbe, 0x55, 0x25, \
  0xa9, 0xc6, 0xd7, 0x7a}
```

Prototype

```
typedef struct {
  EFI_HOB_GENERIC_HEADER              Header;
  EFI_HOB_MEMORY_ALLOCATION_HEADER    MemoryAllocationHeader;
  EFI_GUID                            ModuleName;
  EFI_PHYSICAL_ADDRESS                EntryPoint;
} EFI_HOB_MEMORY_ALLOCATION_MODULE;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_MEMORY_ALLOCATION**.

MemoryAllocationHeader

An instance of the **EFI_HOB_MEMORY_ALLOCATION_HEADER** that describes the various attributes of the logical memory allocation. The type field will be used for subsequent inclusion in the UEFI memory map. Type **EFI_HOB_MEMORY_ALLOCATION_HEADER** is defined in the HOB type **EFI_HOB_MEMORY_ALLOCATION**.

ModuleName

The GUID specifying the values of the firmware file system name that contains the HOB consumer phase component. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

EntryPoint

The address of the memory-mapped firmware volume that contains the HOB consumer phase firmware file. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Description

The HOB consumer phase reads the memory allocation module HOB during its initialization. This HOB describes the memory location of the HOB consumer phase. The HOB consumer phase should use the information to create the image handle for the HOB consumer phase.

5.5 Resource Descriptor HOB

EFI_HOB_RESOURCE_DESCRIPTOR

Summary

Describes the resource properties of all fixed, nonrelocatable resource ranges found on the processor host bus during the HOB producer phase.

Prototype

```
typedef struct _EFI_HOB_RESOURCE_DESCRIPTOR {
  EFI_HOB_GENERIC_HEADER       Header;
  EFI_GUID                     Owner;
  EFI_RESOURCE_TYPE            ResourceType;
  EFI_RESOURCE_ATTRIBUTE_TYPE  ResourceAttribute;
  EFI_PHYSICAL_ADDRESS         PhysicalStart;
  UINT64                       ResourceLength;
} EFI_HOB_RESOURCE_DESCRIPTOR;
```

Parameters

Header

The HOB generic header. *Header.HobType =* **EFI_HOB_TYPE_RESOURCE_DESCRIPTOR**.

Owner

A GUID representing the owner of the resource. This GUID is used by HOB consumer phase components to correlate device ownership of a resource.

ResourceType

Resource type enumeration as defined by **EFI_RESOURCE_TYPE**. Type **EFI_RESOURCE_TYPE** is defined in “Related Definitions” below.

ResourceAttribute

Resource attributes as defined by **EFI_RESOURCE_ATTRIBUTE_TYPE**. Type **EFI_RESOURCE_ATTRIBUTE_TYPE** is defined in “Related Definitions” below.

PhysicalStart

Physical start address of the resource region. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

ResourceLength

Number of bytes of the resource region.

Description

The resource descriptor HOB describes the resource properties of all fixed, nonrelocatable resource ranges found on the processor host bus during the HOB producer phase. This HOB type does not describe how memory is used but instead describes the attributes of the physical memory present.

The HOB consumer phase reads all resource descriptor HOBs when it established the initial Global Coherency Domain (GCD) map. The minimum requirement for the HOB producer phase is that executable content in the HOB producer phase report one of the following:

- The resources that are necessary to start the HOB consumer phase
- The fixed resources that are not captured by HOB consumer phase driver components that were started prior to the dynamic system configuration performed by the platform boot-policy phase

For example, executable content in the HOB producer phase should report any physical memory found during the HOB producer phase. Another example is reporting the Boot Firmware Volume (BFV) that contains firmware volume(s). Executable content in the HOB producer phase does not need to report fixed system resources such as I/O port 70h/71h (real-time clock) because these fixed resources can be allocated from the GCD by a platform-specific chipset driver loading in the HOB consumer phase prior to the platform boot-policy phase, for example.

Current thinking is that the GCD does not track the HOB's *Owner* GUID, so a HOB consumer phase component that assumes ownership of a device's resource must deallocate the resource initialized by the HOB producer phase from the GCD before attempting to assign the devices resource to itself in the HOB consumer phase.

Related Definitions

There can only be a single *ResourceType* field, characterized as follows.

```
//*******************************************************
// EFI_RESOURCE_TYPE
//*******************************************************

typedef UINT32 EFI_RESOURCE_TYPE;

#define EFI_RESOURCE_SYSTEM_MEMORY           0x00000000
#define EFI_RESOURCE_MEMORY_MAPPED_IO        0x00000001
#define EFI_RESOURCE_IO                      0x00000002
#define EFI_RESOURCE_FIRMWARE_DEVICE         0x00000003
#define EFI_RESOURCE_MEMORY_MAPPED_IO_PORT   0x00000004
#define EFI_RESOURCE_MEMORY_RESERVED         0x00000005
#define EFI_RESOURCE_IO_RESERVED             0x00000006
#define EFI_RESOURCE_MAX_MEMORY_TYPE         0x00000007
```

The following table describes the fields listed in the above definition.

EFI_RESOURCE_SYSTEM_MEMORY	Memory that persists out of the HOB producer phase.
EFI_RESOURCE_MEMORY_MAPPED_IO	Memory-mapped I/O that is programmed in the HOB producer phase.
EFI_RESOURCE_IO	Processor I/O space.
EFI_RESOURCE_FIRMWARE_DEVICE	Memory-mapped firmware devices.
EFI_RESOURCE_MEMORY_MAPPED_IO_PORT	Memory that is decoded to produce I/O cycles.
EFI_RESOURCE_MEMORY_RESERVED	Reserved memory address space.
EFI_RESOURCE_IO_RESERVED	Reserved I/O address space.

EFI_RESOURCE_MAX_MEMORY_TYPE	Any reported HOB value of this type or greater should be deemed illegal. This value could increase with successive revisions of this specification, so the “illegality” will also be based upon the revision field of the PHIT HOB.

The *ResourceAttribute* field is characterized as follows:

```
//*******************************************************
// EFI_RESOURCE_ATTRIBUTE_TYPE
//*******************************************************

typedef UINT32 EFI_RESOURCE_ATTRIBUTE_TYPE;


// These types can be ORed together as needed.
//
// The following attributes are used to describe settings
//
#define EFI_RESOURCE_ATTRIBUTE_PRESENT      0x00000001
#define EFI_RESOURCE_ATTRIBUTE_INITIALIZED  0x00000002
#define EFI_RESOURCE_ATTRIBUTE_TESTED       0x00000004

#define EFI_RESOURCE_ATTRIBUTE_READ_PROTECTED    0x00000080
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTED   0x00000100
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTED
                                                 0x00000200
#define EFI_RESOURCE_ATTRIBUTE_PERSISTENT        0x00800000
#define EFI_RESOURCE_ATTRIBUTE_MORE_RELIABLE     0x02000000

// The rest of the attributes are used to describe capabilities
//
#define EFI_RESOURCE_ATTRIBUTE_SINGLE_BIT_ECC    0x00000008
#define EFI_RESOURCE_ATTRIBUTE_MULTIPLE_BIT_ECC  0x00000010
#define EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_1    0x00000020
#define EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_2    0x00000040
#define EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE       0x00000400
#define EFI_RESOURCE_ATTRIBUTE_WRITE_COMBINEABLE 0x00000800
#define EFI_RESOURCE_ATTRIBUTE_WRITE_THROUGH_CACHEABLE
                                                 0x00001000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_BACK_CACHEABLE
                                                 0x00002000
#define EFI_RESOURCE_ATTRIBUTE_16_BIT_IO         0x00004000
#define EFI_RESOURCE_ATTRIBUTE_32_BIT_IO         0x00008000
#define EFI_RESOURCE_ATTRIBUTE_64_BIT_IO         0x00010000
#define EFI_RESOURCE_ATTRIBUTE_UNCACHED_EXPORTED 0x00020000
#define EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTED
                                                 0x00040000

#define EFI_RESOURCE_ATTRIBUTE_READ_PROTECTABLE
                                                 0x00100000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE
                                                 0x00200000
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE
```

```
                                                 0x00400000
#define EFI_RESOURCE_ATTRIBUTE_PERSISTABLE       0x01000000
#define EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE
                                                 0x00080000
```

Table 15. EFI_RESOURCE_ATTRIBUTE_TYPE fields

EFI_RESOURCE_ATTRIBUTE_PRESENT	Physical memory attribute: The memory region exists.
EFI_RESOURCE_ATTRIBUTE_INITIALIZED	Physical memory attribute: The memory region has been initialized.
EFI_RESOURCE_ATTRIBUTE_TESTED	Physical memory attribute: The memory region has been tested.
EFI_RESOURCE_ATTRIBUTE_SINGLE_BIT_ECC	Physical memory attribute: The memory region supports single-bit ECC.
EFI_RESOURCE_ATTRIBUTE_MULTIPLE_BIT_ECC	Physical memory attribute: The memory region supports multibit ECC.
EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_1	Physical memory attribute: The memory region supports reserved ECC.
EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_2	Physical memory attribute: The memory region supports reserved ECC.
EFI_RESOURCE_ATTRIBUTE_READ_PROTECTED	Physical memory protection attribute: The memory region is read protected.
EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTED	Physical memory protection attribute: The memory region is write protected.This is typically used as memory cacheability attribute today. NOTE: Since PI spec 1.4, please use EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTED as Physical write protected attribute, and EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTED means Memory cacheability attribute: The memory supports being programmed with a write-protected cacheable attribute.
EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTED	Physical memory protection attribute: The memory region is execution protected.
EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTED	Physical memory protection attribute: The memory region is write protected.
EFI_RESOURCE_ATTRIBUTE_PERSISTENT	Physical memory persistence attribute. This memory is configured for byte-addressable non-volatility.
EFI_RESOURCE_ATTRIBUTE_MORE_RELIABLE	Physical memory relative reliability attribute. This memory provides higher reliability relative to other memory in the system. If all memory has the same reliability, then this bit is not used.

EFI_RESOURCE_ATTRIBUTE_ UNCACHEABLE	Memory cacheability attribute: The memory does not support caching.
EFI_RESOURCE_ATTRIBUTE_READ_PROTECTABLE	Memory capability attribute: The memory supports being protected from processor reads.
EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE	Memory capability attribute: The memory supports being protected from processor writes.. This is typically used as memory cacheability attribute today. NOTE: Since PI spec 1.4, please use EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE as Memory capability attribute: The memory supports being protected from processor writes, and EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE means Memory cacheability attribute: The memory supports being programmed with a write-protected cacheable attribute.
EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE	Memory capability attribute: The memory supports being protected from processor execution.
EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE	Memory capability attribute: The memory supports being protected from processor writes.
EFI_RESOURCE_ATTRIBUTE_PERSISTABLE	Memory capability attribute. This memory supports byte-addressable non-volatility.
EFI_RESOURCE_ATTRIBUTE_WRITE_ THROUGH_CACHEABLE	Memory cacheability attribute: The memory supports being programmed with a write-through cacheable attribute.
EFI_RESOURCE_ATTRIBUTE_WRITE_ COMBINEABLE	Memory cacheability attribute: The memory supports a write-combining attribute.
EFI_RESOURCE_ATTRIBUTE_WRITE_ BACK_CACHEABLE	Memory cacheability attribute: The memory region supports being configured as cacheable with a write-back policy. Reads and writes that hit in the cache do not propagate to main memory. Dirty data is written back to main memory when a new cache line is allocated.
EFI_RESOURCE_ATTRIBUTE_16_ BIT_IO	Memory physical attribute: The memory supports 16-bit I/O.
EFI_RESOURCE_ATTRIBUTE_32_ BIT_IO	Memory physical attribute: The memory supports 32-bit I/O.
EFI_RESOURCE_ATTRIBUTE_64_ BIT_IO	Memory physical attribute: The memory supports 64-bit I/O.
EFI_RESOURCE_ATTRIBUTE_ UNCACHED_EXPORTED	Memory cacheability attribute: The memory region is uncacheable and exported and supports the fetch and add semaphore mechanism.

Table 16 specifies the resource attributes applicable to each resource type.

Table 16. HOB Producer Phase Resource Types

EFI_RESOURCE_ATTRIBUTE_TYPE	HOB Producer Phase System Memory	HOB Producer Phase Memory-Mapped I/O	HOB Producer Phase I/O
Present	X		
Initialized	X		
Tested	X		
SingleBitEcc	X		
MultipleBitEcc	X		
EccReserved1	X		
EccReserved2	X		
ReadProtected	X	X	
WriteProtected	X	X	
ExecutionProtected	X		
ReadOnlyProtected	X	X	
Uncacheable	X	X	
ReadProtectable	X	X	
WriteProtectable	X	X	
ExecutionProtectable	X		
ReadOnlyProtectable	X	X	
WriteThroughCacheable	X	X	
WriteCombineable	X	X	
WriteBackCacheable	X	X	
16bitIO			X
32bitIO			X
64bitIO			X
UncachedExported	X	X	

5.6 GUID Extension HOB

EFI_HOB_GUID_TYPE

Summary

Allows writers of executable content in the HOB producer phase to maintain and manage HOBs whose types are not included in this specification. Specifically, writers of executable content in the HOB producer phase can generate a GUID and name their own HOB entries using this module-specific value.

Prototype

```
typedef struct _EFI_HOB_GUID_TYPE {
  EFI_HOB_GENERIC_HEADER  Header;
  EFI_GUID                Name;

  //
  // Guid specific data goes here
  //
} EFI_HOB_GUID_TYPE;
```

Parameters

Header

The HOB generic header. *Header.HobType =* **EFI_HOB_TYPE_GUID_EXTENSION**.

Name

A GUID that defines the contents of this HOB. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The GUID extension HOB allows writers of executable content in the HOB producer phase to create their own HOB definitions using a GUID. This HOB type should be used by all executable content in the HOB producer phase to define implementation-specific data areas that are not architectural. This HOB type may also pass implementation-specific data from executable content in the HOB producer phase to drivers in the HOB consumer phase.

A HOB consumer phase component such as a HOB consumer phase driver will read the GUID extension HOB during the HOB consumer phase. The HOB consumer phase component must inherently know the GUID for the GUID extension HOB for which it is scanning the HOB list. This knowledge establishes a contract on the HOB's definition and usage between the executable content in the HOB producer phase and the HOB consumer phase driver.

5.7 Firmware Volume HOB

EFI_HOB_FIRMWARE_VOLUME

Summary

Details the location of firmware volumes that contain firmware files.

Prototype

```
typedef struct {
  EFI_HOB_GENERIC_HEADER  Header;
  EFI_PHYSICAL_ADDRESS    BaseAddress;
  UINT64                  Length;
} EFI_HOB_FIRMWARE_VOLUME;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_FV**.

BaseAddress

The physical memory-mapped base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Length

The length in bytes of the firmware volume.

Description

The firmware volume HOB details the location of firmware volumes that contain firmware files. It includes a base address and length. In particular, the HOB consumer phase will use these HOBs to discover drivers to execute and the hand-off into the HOB consumer phase will use this HOB to discover the location of the HOB consumer phase firmware file.

The firmware volume HOB is produced in the following ways:

- By the executable content in the HOB producer phase in the Boot Firmware Volume (BFV) that understands the size and layout of the firmware volume(s) that are present in the platform.
- By a module that has loaded a firmware volume from some media into memory. The firmware volume HOB details this memory location.

Firmware volumes described by the firmware volume HOB must have a firmware volume header as described in this specification.

The HOB consumer phase consumes all firmware volume HOBs that are presented by the HOB producer phase for use by its read-only support for the PI Firmware Image Format. The HOB producer phase is required to describe any firmware volumes that may contain the HOB consumer phase or platform drivers that are required to discover other firmware volumes.

EFI_HOB_FIRMWARE_VOLUME2

Summary

Details the location of a firmware volume which was extracted from a file within another firmware volume.

Prototype

```
typedef struct {
  EFI_HOB_GENERIC_HEADER  Header;
  EFI_PHYSICAL_ADDRESS    BaseAddress;
  UINT64                  Length;
  EFI_GUID                FvName;
  EFI_GUID                FileName;
} EFI_HOB_FIRMWARE_VOLUME2;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_FV2**.

BaseAddress

The physical memory-mapped base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *Unified Extensible Firmware Interface Specification*, version 2.0.

Length

The length in bytes of the firmware volume.

FvName

The name of the firmware volume.

FileName

The name of the firmware file which contained this firmware volume.

Description

The firmware volume HOB details the location of a firmware volume that was extracted prior to the HOB consumer phase from a file within a firmware volume. By recording the volume and file name, the HOB consumer phase can avoid processing the same file again.

This HOB is created by a module that has loaded a firmware volume from another file into memory. This HOB details the base address, the length, the file name and volume name.

The HOB consumer phase consumes all firmware volume HOBs that are presented by the HOB producer phase for use by its read-only support for the PI Firmware Image format.

EFI_HOB_FIRMWARE_VOLUME3

Summary

Details the location of a firmware volume including authentication information, for both standalone and extracted firmware volumes.

Prototype

```
typedef struct {
   EFI_HOB_GENERIC_HEADER   Header;
   EFI_PHYSICAL_ADDRESS     BaseAddress;
   UINT64                   Length;
   UINT32                   AuthenticationStatus;
   BOOLEAN                  ExtractedFv;
   EFI_GUID                 FvName;
   EFI_GUID                 FileName;
} EFI_HOB_FIRMWARE_VOLUME3;
```

Parameters

Header

The HOB generic header. **Header.HobType = EFI_HOB_TYPE_FV3**.

BaseAddress

The physical memory-mapped base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the Unified Extensible Firmware Interface Specification.

Length

The length in bytes of the firmware volume.

AuthenticationStatus

The authentication status. See Related Definitions of **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()** for more information.

ExtractedFv

TRUE if the FV was extracted as a file within another firmware volume. **FALSE** otherwise.

FvName

The name GUID of the firmware volume. Valid only if *IsExtractedFv* is **TRUE**.

FileName

The name GUID of the firmware file which contained this firmware volume. Valid only if *IsExtractedFv* is **TRUE**.

Description

The firmware volume HOB details the location of firmware volumes that contain firmware files. It includes a base address and length. In particular, the HOB consumer phase will use these HOBs to discover drivers to execute and the hand-off into the HOB consumer phase will use this HOB to discover the location of the HOB consumer phase firmware file.

The firmware volume HOB is produced in the following ways:

- By the executable content in the HOB producer phase in the Boot Firmware Volume (BFV) that understands the size and layout of the firmware volume(s) that are present in the platform.
- By a module that has loaded a firmware volume from some media into memory. The firmware volume HOB details this memory location.
- By a module that has extracted the firmware volume from a file within a firmware file system. By recording the volume and file name, the HOB consumer phase can avoid processing the same file again.

Firmware volumes described by the firmware volume HOB must have a firmware volume header as described in this specification.

The HOB consumer phase consumes all firmware volume HOBs that are presented by the HOB producer phase for use by its read-only support for the PI Firmware Image Format. The HOB consumer phase must provide appropriate authentication data reflecting *AuthenticationStatus* for clients accessing the corresponding firmware volumes. The HOB producer phase is required to describe any firmware volumes that may contain the HOB consumer phase or platform drivers that are required to discover other firmware volumes.

5.8 CPU HOB

EFI_HOB_CPU

Summary

Describes processor information, such as address space and I/O space capabilities.

Prototype

```
typedef struct _EFI_HOB_CPU {
  EFI_HOB_GENERIC_HEADER  Header;
  UINT8                   SizeOfMemorySpace;
  UINT8                   SizeOfIoSpace;
  UINT8                   Reserved[6];
} EFI_HOB_CPU;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_CPU**.

SizeOfMemorySpace

Identifies the maximum physical memory addressability of the processor.

SizeOfIoSpace

Identifies the maximum physical I/O addressability of the processor.

Reserved

For this version of the specification, this field will always be set to zero.

Description

The CPU HOB is produced by the processor executable content in the HOB producer phase. It describes processor information, such as address space and I/O space capabilities. The HOB consumer phase consumes this information to describe the extent of the GCD capabilities.

5.9 Memory Pool HOB

EFI_HOB_MEMORY_POOL

Summary

Describes pool memory allocations.

Prototype

```
typedef struct _EFI_HOB_MEMORY_POOL {
  EFI_HOB_GENERIC_HEADER       Header;
} EFI_HOB_MEMORY_POOL;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_MEMORY_POOL**.

Description

The memory pool HOB is produced by the HOB producer phase and describes pool memory allocations. The HOB consumer phase should be able to ignore these HOBs. The purpose of this HOB is to allow for the HOB producer phase to have a simple memory allocation mechanism within the HOB list. The size of the memory allocation is stipulated by the *HobLength* field in **EFI_HOB_GENERIC_HEADER**.

5.10 UEFI Capsule HOB

EFI_HOB_UEFI_ CAPSULE

Summary

Details the location of coalesced each UEFI capsule memory pages.

Prototype

```
typedef struct {
     EFI_HOB_GENERIC_HEADER  Header;
     EFI_PHYSICAL_ADDRESS    BaseAddress;
     UINT64                  Length;
} EFI_HOB_UEFI_CAPSULE;
```

Parameters

Header

The HOB generic header where **Header.HobType = EFI_HOB_TYPE_UEFI_CAPSULE.**

BaseAddress

The physical memory-mapped base address of an UEFI capsule. This value is set to point to the base of the contiguous memory of the UEFI capsule.

The length of the contiguous memory in bytes

Description

Each UEFI capsule HOB details the location of a UEFI capsule. It includes a base address and length which is based upon memory blocks with a **EFI_CAPSULE_HEADER** and the associated *CapsuleImageSize*-based payloads. These HOB's shall be created by the PEI PI firmware sometime after the UEFI *UpdateCapsule* service invocation with the **CAPSULE_FLAGS_POPULATE_SYSTEM_TABLE** flag set in the **EFI_CAPSULE_HEADER**.

5.11 Unused HOB

EFI_HOB_TYPE_UNUSED

Summary

Indicates that the contents of the HOB can be ignored.

Prototype

```
#define EFI_HOB_TYPE_UNUSED            0xFFFE
```

Description

This HOB type means that the contents of the HOB can be ignored. This type is necessary to support the simple, allocate-only architecture of HOBs that have no delete service. The consumer of the HOB list should ignore HOB entries with this type field.

An agent that wishes to make a HOB entry ignorable should set its type to the prototype defined above.

5.12 End of HOB List HOB

EFI_HOB_TYPE_END_OF_HOB_LIST

Summary

Indicates the end of the HOB list. This HOB must be the last one in the HOB list.

Prototype

```
#define EFI_HOB_TYPE_END_OF_HOB_LIST      0xffff
```

Description

This HOB type indicates the end of the HOB list. This HOB type must be the last HOB type in the HOB list and terminates the HOB list. A HOB list should be considered ill formed if it does not have a final HOB of type **EFI_HOB_TYPE_END_OF_HOB_LIST**.

5.13 SMRAM Memory Hob

EFI_SMRAM_HOB_DESCRIPTOR_BLOCK

Summary

This is a special GUID extension Hob to describe SMRAM memory regions.

GUID

```
#define EFI_SMM_SMRAM_MEMORY_GUID { \
 0x6dadf1d1, 0xd4cc, 0x4910, 0xbb, 0x6e, 0x82, 0xb1, 0xfd, 0x80,
0xff, 0x3d \
}
```

Prototype

```
typedef struct {
  UINT32                          NumberOfSmmReservedRegions;
  EFI_SMRAM_DESCRIPTOR            Descriptor[1];
} EFI_SMRAM_HOB_DESCRIPTOR_BLOCK;
```

Parameters

NumberOfSmmReservedRegions

Designates the number of possible regions in the system that can be usable for SMRAM.

Descriptor

Used to describe the candidate regions for SMRAM that are supported by this platform.

Description

The GUID extension hob is to describe SMRAM memory regions supported by the platform.

6 Status Codes

6.1 Status Codes Overview

This specification defines the status code architecture that is required for an implementation of the Platform Initialization (PI) specifications (hereafter referred to as the “PI Architecture”). Status codes enable system components to report information about their current state. This specification does the following:

- Describes the basic components of status codes
- Defines the status code classes; their subclasses; and the progress, error, and debug code operations for each
- Provides code definitions for the data structures that are common to all status codes
- Provides code definitions for the status code classes; subclasses; progress, error, and debug code enumerations; and extended error data that are architecturally required by the PI Architecture.

The basic definition of a status code is contained in the **ReportStatusCode()** definition in volume 2 of this specification.

6.1.1 Organization of the Status Codes Specification

This specification is organized as listed below. Because status codes are just one component of a PI Architecture-based firmware solution, there are a number of references to the PI Specifications throughout this document.

Table 17. Organization of This Specification

Book	Description
Status Codes Overview	Provides a high-level explanation of status codes and the status code classes and subclasses that are defined in this specification.
Status Code Classes	Provides detailed explanations of the defined status code classes.
Code Definitions	Provides the code definitions for all status code classes; subclasses; extended error data structures; and progress, error, and debug code enumerations that are included in this specification.

6.2 Terms

The following terms are used throughout this document:

debug code

Data produced by various software entities that contains information specifically intended to assist in debugging. The format of the debug code data is governed by this specification.

error code

Data produced by various software entities that indicates an abnormal condition. The format of the error code data is governed by this specification.

progress code

Data produced by various software entities that indicates forward progress. The format of the progress code data is governed by this specification.

status code

One of the three types of codes: progress code, error code, or debug code.

status code driver

The driver that produces the Status Code Runtime Protocol (**EFI_STATUS_CODE_PROTOCOL**). The status code driver receives status codes and notifies registered listeners upon receipt. Status codes handled by this driver are different from the **EFI_STATUS** returned by various functions. The term **EFI_STATUS** is defined in the *UEFI Specification.*

6.3 Types of Status Codes

For each entity classification (class/subclass pair) there are three sets of operations:

- Progress codes
- Error codes
- Debug codes

For progress codes, operations correspond to activities related to the component classification. For error codes, operations correspond to exception conditions (errors). For debug codes, operations correspond to the basic nature of the debug information.

The values 0x00–0x0FFF are common operations that are shared by all subclasses in a class. There are also subclass-specific operations/error codes. Out of the subclass-specific operations, the values 0x1000–0x7FFF are reserved by this specification. The remaining values (0x8000–0xFFFF) are not defined by this specification and OEMs can assign meaning to values in this range. The combination of class and subclass operations provides the complete set of operations that may be reported by an entity. The figure below demonstrates the hierarchy of class and subclass and progress, error, and debug operations.

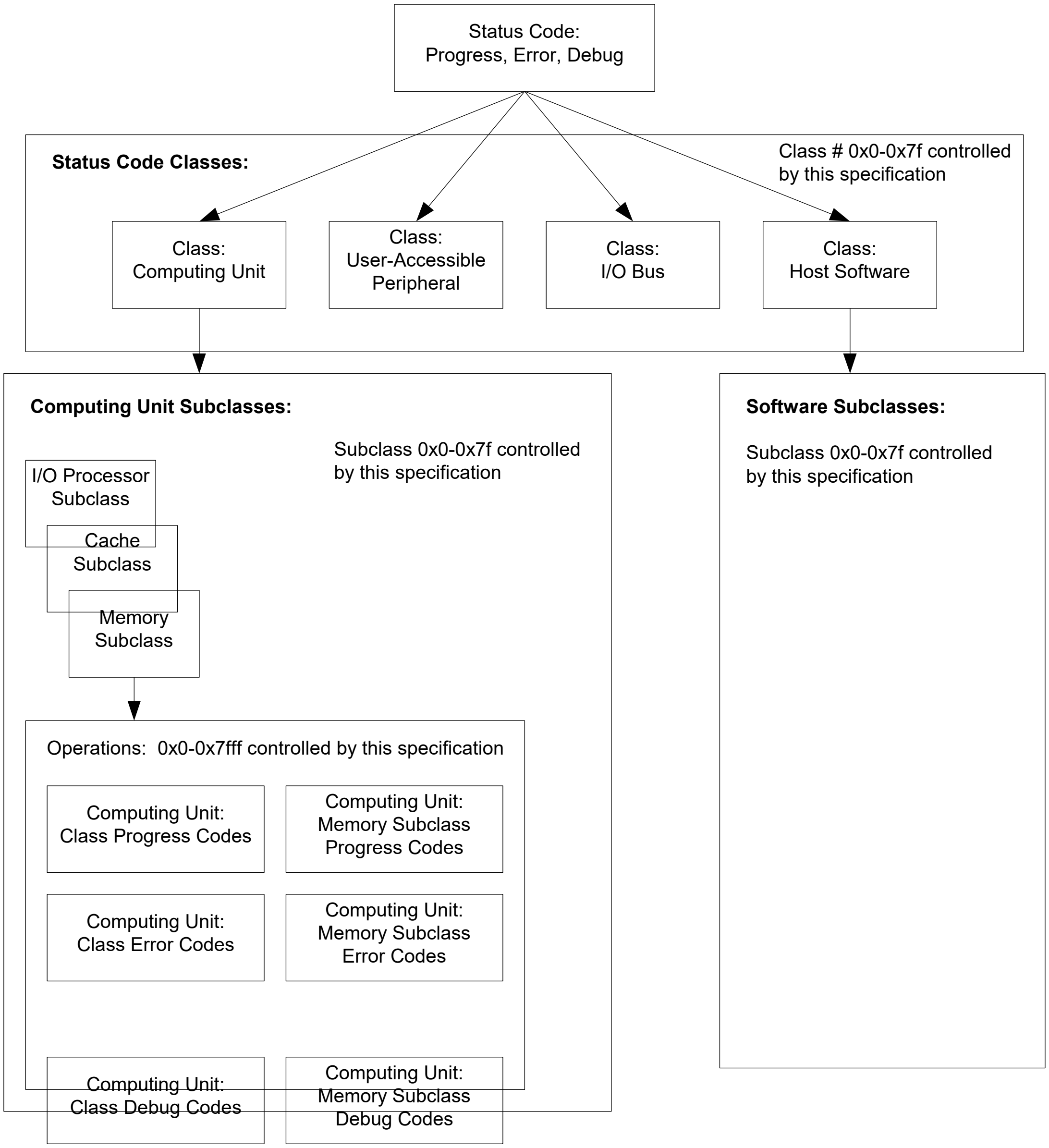

Status Code:
Progress, Error, Debug
Status Code Classes:
Class # 0x0-0x7f controlled by this specification
Class: Computing Unit
Class: User-Accessible Peripheral
Class: I/O Bus
Class: Host Software
Computing Unit Subclasses:
Subclass 0x0-0x7f controlled by this specification
I/O Processor Subclass
Cache Subclass
Memory Subclass
Operations: 0x0-0x7fff controlled by this specification
Computing Unit: Class Progress Codes
Computing Unit: Memory Subclass Progress Codes
Computing Unit: Class Error Codes
Computing Unit: Memory Subclass Error Codes
Computing Unit: Class Debug Codes
Computing Unit: Memory Subclass Debug Codes
Software Subclasses:
Subclass 0x0-0x7f controlled by this specification

Figure 12. Hierarchy of Status Code Operations

The organization of status codes, progress versus error, class, subclass, and operation facilitate a flexible reporting of status codes. In the simplest case, reporting the status code might only convey that an event occurred. In a slightly more complex system, it might be possible to report the class and if it is a progress, error, or debug Code. In such a case, it is at least possible to understand that the system is executing a software activity or that an error occurred with a computing unit. If more reporting capability is present, the error could be isolated to include the subclass—for example, an error occurred related to memory, or the system is currently executing the PEI Foundation software. If yet more capability is present, information about the type of error or activity is available—for example, single-bit ECC error or PEIM dispatch in progress. If the reporting capability is complete, it can provide the detailed error information about the single-bit ECC error, including the location and a string describing the failure. A large spectrum of consumer capability can be supported with a single interface for the producers of progress and error information.

6.3.1 Status Code Classes

The PI architecture defines four classes of status codes—three classes for hardware and one class for software. These classes are listed in the table below and described in detail in the rest of this section. Each class is made up of several subclasses, which are also defined later in this section.

See Code Definitions for all the definitions of all data types and enumerations listed in this section.

Table 18. Class Definitions

Type of Class	Class Name	Data Type Name
Hardware	Computing Unit	EFI_COMPUTING_UNIT
	User-Accessible Peripheral	EFI_PERIPHERAL
	I/O Bus	EFI_IO_BUS
Software	Host Software	EFI_SOFTWARE

Class/subclass pairing should be able to classify any system entity, whether software or hardware. For example, the boot-strap processor (BSP) in a system would be a member of the computing unit class and host processor subclass, while a graphics processor would also be a member of the computing unit class, but a member of the I/O processor subclass.

6.3.2 Instance Number

Because a system may contain multiple entities matching a class/subclass pairing, there is an *instance number*. Instance numbers have different meanings for different classes. However, an instance number of 0xFFFFFFFF always indicates that instance information is unavailable, not applicable, or not provided.

Valid instance numbers start from 0. So a 4-processor server would logically have four instances of the class/subclass pairing, computing unit/host processor, instance numbers 0 to 3.

Due to the complexity of system design, it is outside of the scope of this specification how to pair instance numbers with the actual component—for instance, determining which processor is number 3. However, this specification mandates that the numbering be consistent with the other agents in the system. For example, the processor numbering scheme that is followed by status codes must be consistent with the one followed by the ACPI tables.

6.4 Hardware Classes

6.4.1 Computing Unit Class

The Computing Unit class covers components directly related to system computational capabilities. Subclasses correspond to types of computational devices and resources. See the following for the computing unit class:

- Instance Number
- Progress Code Operations
- Error Code Operations
- Defined Subclasses

6.4.1.1 Instance Number

The instance number refers to the computing unit's geographic location in some manner. An instance number of 0xFFFFFFFF means that the instance number information is not available or the provider of the information is not interested in providing the instance number.

6.4.1.2 Progress Code Operations

All computing unit subclasses share the operation codes listed in the table below. See Progress Code Definitions in Code Definitions: Computing Unit Class for the definitions of these progress codes.

Table 19. Progress Code Operations: Computing Unit Class

Operation	Description	Extended Data
EFI_CU_PC_INIT_BEGIN	General computing unit initialization begins. No details regarding operation are made available.	See subclass.
EFI_CU_PC_INIT_END	General computing unit initialization ends. No details regarding operation are made available.	See subclass.
0x0002–0x0FFF	Reserved for future use by this specification for Computing Class progress codes.	NA
0x1000–0x7FFF	Reserved for subclass use. See the subclass definitions within this specification for value definitions.	NA
0x8000–0xFFFF	Reserved for OEM use.	OEM defined.

6.4.1.3 Error Code Operations

All computing unit subclasses share the error codes listed in the table below. See Error Code Definitions in section 6.7.1 for the definitions of these error codes.

Table 20. Error Code Operations: Computing Unit Class

Operation	Description	Extended Data
EFI_CU_EC_NON_SPECIFIC	No error details available.	See subclass.
EFI_CU_EC_DISABLED	Instance is disabled.	See subclass.
EFI_CU_EC_NOT_SUPPORTED	Instance is not supported.	See subclass.
EFI_CU_EC_NOT_DETECTED	Instance not detected when it was expected to be present.	See subclass.
EFI_CU_EC_NOT_CONFIGURED	Instance could not be properly or completely initialized or configured.	See subclass.
0x0005–0x0FFF	Reserved for future use by this specification for Computing Class error codes.	NA
0x1000–0x7FFF	Subclass defined: See the subclass definitions within this specification.	NA
0x8000–0xFFFF	Reserved for OEM use.	OEM defined.

6.4.1.4 Subclasses

6.4.1.4.1 Defined Subclasses

The table below lists the subclasses in the Computing Unit class. The following topics describe each subclass in more detail.

See Subclass Definitions in Code Definitions: Computing Unit Class for the definitions of these subclasses.

Table 21. Computing Unit Class: Subclasses

Subclass	Code Name	Description
Unspecified	EFI_COMPUTING_UNIT_UNSPECIFIED	The computing unit type is unknown, undefined, or unspecified.
Host processor	EFI_COMPUTING_UNIT_HOST_ PROCESSOR	The computing unit is a full-service central processing unit.
Firmware processor	EFI_COMPUTING_UNIT_FIRMWARE_ PROCESSOR	The computing unit is a limited service processor, typically designed to handle tasks of limited scope.
I/O processor	EFI_COMPUTING_UNIT_IO_PROCESSO R	The computing unit is a processor designed specifically to handle I/O transactions.
Cache	EFI_COMPUTING_UNIT_CACHE	The computing unit is a cache. All types of cache qualify.
Memory	EFI_COMPUTING_UNIT_MEMORY	The computing unit is memory. Many types of memory qualify.
Chipset	EFI_COMPUTING_UNIT_CHIPSET	The computing unit is a chipset component.
0x07–0x7F	Reserved for future use by this specification.	
0x80–0xFF	Reserved for OEM use.	

6.4.1.4.2 Unspecified Subclass

This subclass can be used for any computing unit type of component that does not belong in one of the other subclasses.

See section 6.7.1.1for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

Table 22. Progress and Error Code Operations: Computing Unit Unspecified Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.1.4.3 Host Processor Subclass

This subclass is used for computing units that provide the system's main processing power and their associated hardware. These are general-purpose processors capable of a wide range of functionality. The instance number matches the processor handle number that is assigned to the processor by the Multiprocessor (MP) Services Protocol. They often contain multiple levels of embedded cache.

See Subclass Definitions in section 6.7.1.1for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 23. Progress and Error Code Operations: Host Processor Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_CU_HP_PC_POWER_ON_INIT	Power-on initialization	None
	EFI_CU_HP_PC_CACHE_INIT	Embedded cache initialization including cache controller hardware and cache memory.	EFI_CACHE_INIT_DATA
Progress (cont.)	EFI_CU_HP_PC_RAM_INIT	Embedded RAM initialization	None
	EFI_CU_HP_PC_MEMORY_ CONTROLLER_INIT	Embedded memory controller initialization	None
	EFI_CU_HP_PC_IO_INIT	Embedded I/O complex initialization	None
	EFI_CU_HP_PC_BSP_SELECT	BSP selection	None
	EFI_CU_HP_PC_BSP_RESELECT	BSP reselection	None
	EFI_CU_HP_PC_AP_INIT	AP initialization (this operation is performed by the current BSP)	None
	EFI_CU_HP_PC_SMM_INIT	SMM initialization	None
	0x000B–0x7FFF	Reserved for future use by this specification	NA
Error	EFI_CU_EC_DISABLED	Instance is disabled. This is a standard error code for this class.	EFI_COMPUTING_UNIT_ CPU_DISABLED_ERROR_ DATA
	EFI_CU_HP_EC_INVALID_TYPE	Instance is not a valid type.	None
	EFI_CU_HP_EC_INVALID_SPEED	Instance is not a valid speed.	None
	EFI_CU_HP_EC_MISMATCH	Mismatch detected between two instances.	EFI_HOST_PROCESSOR_ MISMATCH_ERROR_DATA
	EFI_CU_HP_EC_TIMER_EXPIRED	A watchdog timer expired.	None
	EFI_CU_HP_EC_SELF_TEST	Instance detected an error during BIST	None
	EFI_CU_HP_EC_INTERNAL	Instance detected an IERR.	None

	EFI_CU_HP_EC_THERMAL	An over temperature condition was detected with this instance.	EFI_COMPUTING_UNIT_ THERMAL_ERROR_DATA
Error (cont.)	EFI_CU_HP_EC_LOW_VOLTAGE	Voltage for this instance dropped below the low voltage threshold.	EFI_COMPUTING_UNIT_ VOLTAGE_ERROR_DATA
	EFI_CU_HP_EC_HIGH_VOLTAGE	Voltage for this instance surpassed the high voltage threshold	EFI_COMPUTING_UNIT_ VOLTAGE_ERROR_DATA
	EFI_CU_HP_EC_CACHE	The instance suffered a cache failure.	None
	EFI_CU_HP_EC_MICROCODE_ UPDATE	Instance microcode update failed	EFI_COMPUTING_UNIT_ MICROCODE_ UPDATE_ERROR_DATA
	EFI_CU_HP_EC_CORRECTABLE	Correctable error detected	None
	EFI_CU_HP_EC_UNCORRECTABLE	Uncorrectable ECC error detected	None
	EFI_CU_HP_EC_NO_MICROCODE_UPD ATE	No matching microcode update is found	None
	0x100D–0x7FFF	Reserved for future use by this specification	NA

Related Definitions

See the following topics in section 6.7.1.1 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.1 for definitions of the extended error data listed above.

6.4.1.4.4 Firmware Processor Subclass

This subclass applies to processors other than the Host Processors that provides services to the system.

See section 6.7.1.1 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 24. Progress and Error Code Operations: Service Processor Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_CU_FP_EC_HARD_FAIL	Firmware processor detected a hardware error during initialization.	None
	EFI_CU_FP_EC_SOFT_FAIL	Firmware processor detected an error during initialization. E.g. Firmware processor NVRAM contents are invalid.	None
	EFI_CU_FP_EC_COMM_ERROR	The host processor encountered an error while communicating with the firmware processor.	None
	0x1004–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.1 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

6.4.1.4.5 I/O Processor Subclass

This subclass applies to system I/O processors and their associated hardware. These processors are typically designed to offload I/O tasks from the central processors in the system. Examples would include graphics or I20 processors. The subclass is identical to the host processor subclass. See Host Processor Subclass for more information.

See section 6.7.1.1 for the definition of this subclass.

6.4.1.4.6 Cache Subclass

The cache subclass applies to any external/system level caches. Any cache embedded in a computing unit would not be counted in this subclass, but would be considered a member of that computing unit subclass.

See Subclass Definitions in section 6.7.1 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 25. Progress and Error Code Operations: Cache Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_CU_CACHE_PC_PRESENCE_DETECT	Detecting cache presence.	None
	EFI_CU_CACHE_PC_CONFIGURATION	Configuring cache.	None
	0x1002–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_CU_CACHE_EC_INVALID_TYPE	Instance is not a valid type.	None
	EFI_CU_CACHE_EC_INVALID_SPEED	Instance is not a valid speed.	None
	EFI_CU_CACHE_EC_INVALID_SIZE	Instance size is invalid.	None
	EFI_CU_CACHE_EC_MISMATCH	Instance does not match other caches.	None
	0x1004–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.1 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

6.4.1.4.7 Memory Subclass

The memory subclass applies to any external/system level memory and associated hardware. Any memory embedded in a computing unit would not be counted in this subclass, but would be considered a member of that computing unit subclass.

See Subclass Definitions in section 6.7.1 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

For all operations and errors, the instance number specifies the DIMM number unless stated otherwise. Some of the operations may affect multiple memory devices and multiple memory controllers. The specification provides mechanisms (**EFI_MULTIPLE_MEMORY_DEVICE_OPERATION** and others) to describe such group operations. See **EFI_STATUS_CODE_DIMM_NUMBER** in section 6.7.1 for details.

Table 26. Progress and Error Code Operations: Memory Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_CU_MEMORY_PC_SPD_READ	Reading configuration data (e.g. SPD) from memory devices.	None
	EFI_CU_MEMORY_PC_PRESENCE_DETECT	Detecting presence of memory devices (e.g. DIMMs).	None
	EFI_CU_MEMORY_PC_TIMING	Determining optimum configuration e.g. timing for memory devices.	None
	EFI_CU_MEMORY_PC_CONFIGURING	Initial configuration of memory device and memory controllers.	None
	EFI_CU_MEMORY_PC_OPTIMIZING	Programming the memory controller and memory devices with optimized settings.	None
Progress (cont.)	EFI_CU_MEMORY_PC_INIT	Memory initialization such as ECC initialization.	EFI_MEMORY_RANGE_EXTENDED_DATA
	EFI_CU_MEMORY_PC_TEST	Performing memory test.	EFI_MEMORY_RANGE_EXTENDED_DATA
	0x1007–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_CU_MEMORY_EC_INVALID_TYPE	Instance is not a valid type.	None
	EFI_CU_MEMORY_EC_INVALID_SPEED	Instance is not a valid speed.	None
	EFI_CU_MEMORY_EC_CORRECTABLE	Correctable error detected.	EFI_MEMORY_EXTENDED_ERROR_DATA
	EFI_CU_MEMORY_EC_UNCORRECTABLE	Uncorrectable error detected. This included memory miscomparisions during the memory test.	EFI_MEMORY_EXTENDED_ERROR_DATA
	EFI_CU_MEMORY_EC_SPD_FAIL	Instance SPD failure detected.	None
	EFI_CU_MEMORY_EC_INVALID_SIZE	Instance size is invalid.	None
	EFI_CU_MEMORY_EC_MISMATCH	Mismatch detected between two instances.	EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA
	EFI_CU_MEMORY_EC_S3_RESUME_FAIL	Resume from S3 failed.	None
	EFI_CU_MEMORY_EC_UPDATE_FAIL	Flash Memory Update failed.	None

	EFI_CU_MEMORY_EC_NONE_DETECTED	Memory was not detected in the system. Instance field is ignored.	None
Error (cont.)	EFI_CU_MEMORY_EC_NONE_USEFUL	No useful memory was detected in the system. E.g., Memory was detected, but cannot be used due to errors. Instance field is ignored.	None
	0x1009–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.1 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See section 6.7.1.4 for definitions of the extended error data listed above.

6.4.1.4.8 Chipset Subclass

This subclass can be used for any chipset components and their related hardware.

See Subclass Definitions in section 6.7.1 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Computing Unit class, the table below lists the additional codes for this subclass.

Table 27. Progress and Error Code Operations: Chipset Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_CHIPSET_PC_PEI_CAR_SB_INIT	South Bridge initialization prior to memory detection	None
	EFI_CHIPSET_PC_PEI_CAR_NB_INIT	North Bridge initialization prior to memory detection	None
	EFI_CHIPSET_PC_PEI_MEM_SB_INIT	South Bridge initialization after memory detection	None
	EFI_CHIPSET_PC_PEI_MEM_NB_INIT	North Bridge initialization after memory detection	None
	EFI_CHIPSET_PC_DXE_HB_INIT	PCI Host Bridge DXE initialization	None
	EFI_CHIPSET_PC_DXE_NB_INIT	North Bridge DXE initialization	None
	EFI_CHIPSET_PC_DXE_NB_SMM_INIT	North Bridge specific SMM initialization in DXE	None
	EFI_CHIPSET_PC_DXE_SB_RT_INIT	Initialization of the South Bridge specific UEFI Runtime Services	None
	EFI_CHIPSET_PC_DXE_SB_INIT	South Bridge DXE initialization	None
	EFI_CHIPSET_PC_DXE_SB_SMM_INIT	South Bridge specific SMM initialization in DXE	None
	EFI_CHIPSET_PC_DXE_SB_DEVICES_INIT	Initialization of the South Bridge devices	None
Progress	0x100B–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_CHIPSET_EC_BAD_BATTERY	Bad battery status has been detected	None
	EFI_CHIPSET_EC_DXE_NB_ERROR	North Bridge initialization error in DXE	None
	EFI_CHIPSET_EC_DXE_NB_ERROR	South Bridge initialization error in DXE	None
	EFI_CHIPSET_EC_INTRUDER_DETECT	Physical access to inner system components that are not accessible during normal system operation is detected.	None
Error	0x1004–0x7FFF	Reserved for future use by this specification.	

Related Definitions

None.

6.4.2 User-Accessible Peripheral Class

The User-Accessible Peripheral class refers to any peripheral with which the user interacts. Subclass elements correspond to general classes of peripherals. See the following for the User-Accessible Peripheral class:

- Instance Number
- Progress Code Operations
- Error Code Operations
- Defined Subclasses

6.4.2.1 Instance Number

The instance number refers to the peripheral's geographic location in some manner. Instance number of 0 means that instance number information is not available or the provider of the information is not interested in providing the instance number.

6.4.2.2 Progress Code Operations

All peripheral subclasses share the operation codes listed in the table below. See Progress Code Definitions for the definitions of these progress codes.

Table 28. Progress Code Operations: User-Accessible Peripheral Class

Operation	Description	Extended Data
EFI_P_PC_INIT	General Initialization. No details regarding operation are made available.	See subclass.
EFI_P_PC_RESET	Resetting the peripheral.	See subclass.
EFI_P_PC_DISABLE	Disabling the peripheral.	See subclass.
EFI_P_PC_PRESENCE_DETECT	Detecting the presence.	See subclass.
EFI_P_PC_ENABLE	Enabling the peripheral.	See subclass.
EFI_P_PC_RECONFIG	Reconfiguration.	See subclass.
EFI_P_PC_DETECTED	Peripheral was detected.	See subclass.
EFI_P_PC_REMOVED	Peripheral was unplugged, ejected or otherwise removed from the system.	See subclass.
0x0007–0x0FFF	Reserved for future use by this specification for Peripheral Class progress codes.	NA
0x1000–0x7FFF	Reserved for subclass use. See the subclass definitions within this specification for value definitions.	See subclass.
0x8000–0xFFFF	Reserved for OEM use.	NA

6.4.2.3 Error Code Operations

All peripheral subclasses share the error codes listed in the table below. See section 6.7.2 for the definitions of these error codes.

Table 29. Error Code Operations: User-Accessible Peripheral Class

Operation	Description	Extended Data
EFI_P_EC_NON_SPECIFIC	No error details available.	See subclass
EFI_P_EC_DISABLED	Instance is disabled.	See subclass
EFI_P_EC_NOT_SUPPORTED	Instance is not supported.	See subclass
EFI_P_EC_NOT_DETECTED	Instance not detected when it was expected to be present.	See subclass
EFI_P_EC_NOT_CONFIGURED	Instance could not be properly or completely initialized or configured.	See subclass
EFI_P_EC_INTERFACE_ERROR	An error occurred with the peripheral interface.	See subclass
EFI_P_EC_CONTROLLER_ERROR	An error occurred with the peripheral controller.	See subclass
EFI_P_EC_INPUT_ERROR	An error occurred getting input from the peripheral.	See subclass.
EFI_P_EC_OUTPUT_ERROR	An error occurred putting output to the peripheral.	See subclass.
EFI_P_EC_RESOURCE_CONFLICT	A resource conflict exists with this instance's resource requirements.	See EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA for all subclasses.
0x0006–0x0FFF	Reserved for future use by this specification for User-Accessible Peripheral class error codes.	NA
0x1000–0x7FFF	See the subclass definitions within this specification.	See subclass
0x8000–0xFFFF	Reserved for OEM use.	NA

6.4.2.4 Subclasses

6.4.2.4.1 Defined Subclasses

The table below lists the subclasses in the User-Accessible Peripheral class. The following topics describe each subclass in more detail.

See Subclass Definitions in section 6.7.2 for the definitions of these subclasses.

Table 30. Defined Subclasses: User-Accessible Peripheral Class

Subclass	Code Name	Description
Unspecified	EFI_PERIPHERAL_UNSPECIFIED	The peripheral type is unknown, undefined, or unspecified.
Keyboard	EFI_PERIPHERAL_KEYBOARD	The peripheral referred to is a keyboard.
Mouse	EFI_PERIPHERAL_MOUSE	The peripheral referred to is a mouse.
Local console	EFI_PERIPHERAL_LOCAL_CONSOLE	The peripheral referred to is a console directly attached to the system.
Remote console	EFI_PERIPHERAL_REMOTE_CONSOLE	The peripheral referred to is a console that can be remotely accessed.
Serial port	EFI_PERIPHERAL_SERIAL_PORT	The peripheral referred to is a serial port.
Parallel port	EFI_PERIPHERAL_PARALLEL_PORT	The peripheral referred to is a parallel port.
Fixed media	EFI_PERIPHERAL_FIXED_MEDIA	The peripheral referred to is a fixed media device—e.g., an IDE hard disk drive.
Removable media	EFI_PERIPHERAL_REMOVABLE_MEDIA	The peripheral referred to is a removable media device—e.g., a DVD-ROM drive.
Audio input	EFI_PERIPHERAL_AUDIO_INPUT	The peripheral referred to is an audio input device—e.g., a microphone.
Audio output	EFI_PERIPHERAL_AUDIO_OUTPUT	The peripheral referred to is an audio output device—e.g., speakers or headphones.
LCD device	EFI_PERIPHERAL_LCD_DEVICE	The peripheral referred to is an LCD device.
Network device	EFI_PERIPHERAL_NETWORK	The peripheral referred to is a network device—e.g., a network card.
Docking Station	EFI_PERIPHERAL_DOCKING	The peripheral referred to is a docking station
0x0E–0x7F	Reserved for future use by this specification.	
0x80–0xFF	Reserved for OEM use.	

6.4.2.4.2 Unspecified Subclass

This subclass applies to any user-accessible peripheral not belonging to any of the other subclasses.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 31. Progress and Error Code Operations: Peripheral Unspecified Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.3 Keyboard Subclass

This subclass applies to any keyboard style interfaces. *ExtendedData* contains the device path to the keyboard device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 32. Progress and Error Code Operations: Keyboard Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_P_KEYBOARD_PC_CLEAR_BUFFER	Clearing the input keys from keyboard.	The device path to the keyboard device. See EFI_DEVICE_PATH_ EXTENDED_DATA
	EFI_P_KEYBOARD_PC_SELF_TEST	Keyboard self-test.	The device path to the keyboard device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
	0x1002–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_P_KEYBOARD_EC_LOCKED	The keyboard input is locked.	The device path to the keyboard device. See EFI_DEVICE_PATH_ EXTENDED_DATA
	EFI_P_KEYBOARD_EC_STUCK_KEY	A stuck key was detected.	The device path to the keyboard device. See EFI_DEVICE_PATH_ EXTENDED_DATA
	EFI_P_KEYBOARD_EC_BUFFER_FULL	Keyboard buffer is full.	The device path to the keyboard device. See EFI_DEVICE_PATH_EXTENDED_DATA
	0x1003–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.2s for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.2 for definitions of the extended error data listed above.

6.4.2.4.4 Mouse Subclass

This subclass applies to any mouse or pointer peripherals. *ExtendedData* contains the device path to the mouse device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 33. Progress and Error Code Operations: Mouse Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_P_MOUSE_PC_SELF_TEST	Mouse self-test.	The device path to the mouse device. See EFI_DEVICE_PATH_EXTENDED_DATA.
	0x1001–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_P_MOUSE_EC_LOCKED	The mouse input is locked.	The device path to the mouse device. See EFI_DEVICE_PATH_EXTENDED_DATA
	0x1001–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.2 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.2 for definitions of the extended error data listed above.

6.4.2.4.5 Local Console Subclass

This subclass applies to all console devices directly connected to the system. This would include VGA/UGA devices. *`ExtendedData`* contains the device path to the console device as defined in **`EFI_DEVICE_PATH_EXTENDED_DATA`** and the instance is ignored. LCD devices have their own subclass.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 34. Progress and Error Code Operations: Local Console Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.6 Remote Console Subclass

This subclass applies to any console not directly connected to the system. This would include consoles displayed via serial or LAN connections. *ExtendedData* contains the device path to the console device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 35. Progress and Error Code Operations: Remote Console Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.7 Serial Port Subclass

This subclass applies to devices attached to a system serial port, such as a modem. *ExtendedData* contains the device path to the device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 36. Progress and Error Code Operations: Serial Port Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_P_SERIAL_PORT_PC_CLEAR_BUFFER	Clearing the serial port input buffer.	The device handle. See EFI_DEVICE_PATH_EXTENDED_DATA.
	0x1001–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.2 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.2 for definitions of the extended error data listed above.

6.4.2.4.8 Parallel Port Subclass

This subclass applies to devices attached to a system parallel port, such as a printer. *ExtendedData* contains the device path to the device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 37. Progress and Error Code Operations: Parallel Port Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.9 Fixed Media Subclass

This subclass applies to fixed media peripherals such as hard drives. These peripherals are capable of producing the **EFI_BLOCK_IO** Protocol. *ExtendedData* contains the device path to the device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 38. Progress and Error Code Operations: Fixed Media Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.10 Removable Media Subclass

This subclass applies to removable media peripherals such as floppy disk drives or LS-120 drives. These peripherals are capable of producing the **EFI_BLOCK_IO** Protocol. *ExtendedData* contains the device path to the device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA** and the instance is ignored.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 39. Progress and Error Code Operations: Removable Media Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.11 Audio Input Subclass

This subclass applies to audio input devices such as microphones.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 40. Progress and Error Code Operations: Audio Input Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.12 Audio Output Subclass

This subclass applies to audio output devices like speakers or headphones.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 41. Progress and Error Code Operations: Audio Output Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.13 LCD Device Subclass

This subclass applies to LCD display devices attached to the system.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 42. Progress and Error Code Operations: LCD Device Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.2.4.14 Network Device Subclass

This subclass applies to network adapters attached to the system. These devices are capable of producing standard UEFI networking protocols such as the **EFI_SIMPLE_NETWORK** Protocol.

See Subclass Definitions in section 6.7.2 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the User-Accessible Peripheral class, the table below lists the additional codes for this subclass.

Table 43. Progress and Error Code Operations: Network Device Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3 I/O Bus Class

The I/O bus class covers hardware buses irrespective of any software protocols that are used. At a broad level, everything that connects the computing unit to the user peripheral can be covered by this class. Subclass elements correspond to industry-standard hardware buses. See the following for the I/O Bus class:

- Instance Number
- Progress Code Operations
- Error Code Operations
- Defined Subclasses

6.4.3.1 Instance Number

The instance number is ignored and the *ExtendedData* describes the device path to the controller or the device as defined in **EFI_DEVICE_PATH_EXTENDED_DATA**.

6.4.3.2 Progress Code Operations

All I/O bus subclasses share the operation codes listed in the table below. See Progress Code Definitions in section 6.7.3 for the definitions of these progress codes.

Table 44. Progress Code Operations: I/O Bus Class

Operation	Description	Extended Data
EFI_IOB_PC_INIT	General initialization. No details regarding operation are made available.	The device path corresponding to the host bus controller (the controller that produces this bus). For the PCI bus, it is the PCI root bridge. The format of the device path extended data is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_RESET	Resetting the bus. Generally, this operation resets all the devices on the bus as well.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_DISABLE	Disabling all the devices on the bus prior to enumeration.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_DETECT	Detecting devices on the bus.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_ENABLE	Configuring the bus and enabling device on the bus.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_RECONFIG	Bus reconfiguration including resource re-enumeration.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
EFI_IOB_PC_HOTPLUG	A hot-plug event was detected on the bus and the hot-plugged device was initialized.	The device path corresponding to the host controller (the controller that produces this bus). The format is defined in EFI_DEVICE_PATH_EXTENDED_DATA.
0x0007–0x0FFF	Reserved for future use by this specification for I/O Bus class progress codes.	NA
0x1000–0x7FFF	Reserved for subclass use. See the subclass definitions within this specification for value definitions.	NA
0x8000–0xFFFF	Reserved for OEM use.	OEM defined.

6.4.3.3 Error Code Operations

All I/O bus subclasses share the error codes listed in the table below. See Error Code Definitions in section 6.7.3 for the definitions of these error codes.

Table 45. Error Code Operations: I/O Bus Class

Operation	Description	Extended Data
EFI_IOB_EC_NON_SPECIFIC	No error details available	None.
EFI_IOB_EC_DISABLED	A device is disabled due to bus-level errors.	The device path corresponding to the device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_NOT_SUPPORTED	A device is not supported on this bus.	The device path corresponding to the device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_NOT_DETECTED	Instance not detected when it was expected to be present.	The device path corresponding to the device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_NOT_CONFIGURED	Instance could not be properly or completely initialized/configured.	The device path corresponding to the device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_INTERFACE_ERROR	An error occurred with the bus interface.	The device path corresponding to the failing device. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_CONTROLLER_ERROR	An error occurred with the host bus controller (the controller that produces this bus).	The device path corresponding to the bus controller. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_READ_ERROR	A bus specific error occurred getting input from a device on the bus.	The device path corresponding to the failing device or the closest device path. See EFI_DEVICE_PATH_
EFI_IOB_EC_WRITE_ERROR	An error occurred putting output to the bus.	The device path corresponding to the failing device or the closest device path. See EFI_DEVICE_PATH_ EXTENDED_DATA.
EFI_IOB_EC_RESOURCE_CONFLICT	A resource conflict exists with this instance's resource requirements.	See EFI_RESOURCE_ALLOC_ FAILURE ERROR_DATA.
0x000A–0x0FFF	Reserved for future use by this specification for I/O Bus class error codes.	NA
0x1000–0x7FFF	See the subclass definitions within this specification.	NA
0x8000–0xFFFF	Reserved for OEM use.	NA

6.4.3.4 Subclasses

6.4.3.4.1 Defined Subclasses

The table below lists the subclasses in the . The following topics describe each subclass in more detail.

See Subclass Definitions in section 6.7.3 for the definitions of these subclasses.

Table 46. Defined Subclasses: I/O Bus Class

Subclass	Code Name	Description
Unspecified	EFI_IO_BUS_UNSPECIFIED	The bus type is unknown, undefined, or unspecified.
PCI	EFI_IO_BUS_PCI	The bus is a PCI bus.
USB	EFI_IO_BUS_USB	The bus is a USB bus.
InfiniBand* architecture	EFI_IO_BUS_IBA	The bus is an IBA bus.
AGP	EFI_IO_BUS_AGP	The bus is an AGP bus.
PC card	EFI_IO_BUS_PC_CARD	The bus is a PC Card bus.
Low pin count (LPC)	EFI_IO_BUS_LPC	The bus is a LPC bus.
SCSI	EFI_IO_BUS_SCSI	The bus is a SCSI bus.
ATA/ATAPI/SATA	EFI_IO_BUS_ATA_ATAPI	The bus is a ATA/ATAPI bus.
Fibre Channel	EFI_IO_BUS_FC	The bus is an EC bus.
IP network	EFI_IO_BUS_IP_NETWORK	The bus is an IP network bus.
SMBus	EFI_IO_BUS_SMBUS	The bus is a SMBUS bus.
I2C	EFI_IO_BUS_I2C	The bus is an I2C bus.
0x0D–0x7F	Reserved for future use by this specification.	
0x80–0xFF	Reserved for OEM use.	

6.4.3.4.2 Unspecified Subclass

This subclass applies to any I/O bus not belonging to any of the other I/O bus subclasses.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 47. Progress and Error Code Operations: I/O Bus Unspecified Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.4.3 PCI Subclass

This subclass applies to PCI buses and devices. It also includes different variations of PCI bus including PCI-X and PCI Express.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 48. Progress and Error Code Operations: PCI Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_IOB_PCI_BUS_ENUM	Enumerating buses under a root bridge.	The device path corresponding to the PCI root bridge. See EFI_DEVICE_PATH_ EXTENDED_DATA.
	EFI_IOB_PCI_RES_ALLO C	Allocating resources to devices under a host bridge.	The host bridge handle as defined in EFI_DEVICE_HANDLE_ EXTENDED_DATA.
	EFI_IOB_PCI_HPC_INIT	Initializing a PCI hot-plug controller.	The device path to the controller as defined in EFI_DEVICE_PATH_ EXTENDED_DATA.
	0x1003–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_IOB_PCI_EC_PERR	Parity error; see PCI Specification.	The device path to the controller that generated the PERR. The data format is defined in EFI_DEVICE_PATH_ EXTENDED_DATA.
	EFI_IOB_PCI_EC_SERR	System error; see PCI Specification.	The device path to the controller that generated the SERR. The data format is defined in EFI_DEVICE_PATH_ EXTENDED_DATA.
	0x1002–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topics in section 6.7.2 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.3 for definitions of the extended error data listed above.

6.4.3.5 USB Subclass

This subclass applies to USB buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 49. Progress and Error Code Operations: USB Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.1 InfiniBand* Architecture Subclass

This subclass applies to InfiniBand* (IBA) buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 50. Progress and Error Code Operations: IBA Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.2 AGP Subclass

This subclass applies to AGP buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 51. Progress and Error Code Operations: AGP Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.3 PC Card Subclass

This subclass applies to PC Card buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 52. Progress and Error Code Operations: PC Card Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.4 LPC Subclass

This subclass applies to LPC buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 53. Progress and Error Code Operations: LPC Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.5 SCSI Subclass

This subclass applies to SCSI buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 54. Progress and Error Code Operations: SCSI Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.6 ATA/ATAPI/SATA Subclass

This subclass applies to ATA and ATAPI buses and devices. It also includes Serial ATA (SATA) buses.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 55. Progress and Error Code Operations: ATA/ATAPI/SATA Subclass

Type of code	Operation	Description	Extended data
Progress	EFI_IOB_ATA_BUS_SMART_ENABLE	SMART is enabled on the storage device	NA
	EFI_IOB_ATA_BUS_SMART_DISABLE	SMART is disabled on the storage device	NA
	EFI_IOB_ATA_BUS_SMART_OVERTHRESHOLD	SMART records are over threshold on the storage device	NA
	EFI_IOB_ATA_BUS_SMART_UNDERTHRESHOLD	SMART records are under threshold on the storage device	NA
	0x1004–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_IOB_ATA_BUS_SMART_NOTSUPPORTED	SMART is not supported on the storage device	NA
	EFI_IOB_ATA_BUS_SMART_DISABLED	SMART is disabled on the storage device	NA
	0x1002–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.7 Fibre Channel (FC) Subclass

This subclass applies to Fibre Channel buses and devices.

See Subclass Definitions in section 6.7.3for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 56. Progress and Error Code Operations: FC Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.8 IP Network Subclass

This subclass applies to IP network buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 57. Progress and Error Code Operations: IP Network Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.9 3SMBus Subclass

This subclass applies to SMBus buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 58. Progress and Error Code Operations: SMBus Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.4.3.5.10 I2C Subclass

This subclass applies to I2C buses and devices.

See Subclass Definitions in section 6.7.3 for the definition of this subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the I/O Bus class, the table below lists the additional codes for this subclass.

Table 59. Progress and Error Code Operations: I2C Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5 Software Classes

6.5.1 Host Software Class

The Host Software class covers any software-generated codes. Subclass elements correspond to common software types in a PI Architecture system. See the following for the Host Software class:

- Instance Number
- Progress Code Operations
- Error Code Operations
- Defined Subclasses

6.5.2 Instance Number

The instance number is not used for software subclasses unless otherwise stated.

6.5.3 Progress Code Operations

All host software subclasses share the operation codes listed in the table below. See Progress Code Definitions in section 6.7.4 for the definitions of these progress codes.

Table 60. Progress Code Operations: Host Software Class

Operation	Description	Extended Data
EFI_SW_PC_INIT	General initialization. No details regarding operation are made available.	None.
EFI_SW_PC_LOAD	Loading a software module in the preboot phase by using LoadImage() or an equivalent PEI service. May include a PEIM, DXE drivers, UEFI application, etc.	Handle identifying the module. There will be an instance of EFI_LOADED_IMAGE_PROTOCOL on this handle. See EFI_DEVICE_HANDLE_ EXTENDED_DATA.
EFI_SW_PC_INIT_BEGIN	Initializing software module by using StartImage() or an equivalent PEI service.	Handle identifying the module. There will be an instance of EFI_LOADED_IMAGE_PROTOCOL on this handle. See EFI_DEVICE_HANDLE_ EXTENDED_DATA.
EFI_SW_PC_INIT_END	Software module returned control back after initialization.	Handle identifying the module. There will be an instance of EFI_LOADED_IMAGE_PROTOCOL on this handle. See EFI_DEVICE_HANDLE_ EXTENDED_DATA.
EFI_SW_PC_ AUTHENTICATE_BEGIN	Performing authentication (passwords, biometrics, etc.).	None.
EFI_SW_PC_ AUTHENTICATE_END	Authentication completed.	None.
EFI_SW_PC_INPUT_WAIT	Waiting for user input.	None.
EFI_SW_PC_USER_SETUP	Executing user setup.	None.
0x0008–0x0FFF	Reserved for future use by this specification for Host Software class progress codes.	NA
0x1000–0x7FFF	Reserved for subclass use. See the subclass definitions within this specification for value definitions.	NA
0x8000–0xFFFF	Reserved for OEM use.	NA

6.5.4 Error Code Operations

All host software subclasses share the error codes listed in the table below. See Error Code Definitions in section 6.7.4 for the definitions of these progress codes.

Table 61. Error Code Operations: Host Software Class

Operation	Description	Extended Data
EFI_SW_EC_NON_SPECIFIC	No error details are available.	None
EFI_SW_EC_LOAD_ERROR	The software module load failed.	Handle identifying the module. There will be an instance of EFI_LOADED_IMAGE_PROTOCOL on this handle. See EFI_DEVICE_HANDLE_EXTENDED_ DATA.
EFI_SW_EC_INVALID_ PARAMETER	An invalid parameter was passed to the instance.	None.
EFI_SW_EC_UNSUPPORTED	An unsupported operation was requested.	None.
EFI_SW_EC_INVALID_BUFFER	The instance encountered an invalid buffer (too large, small, or nonexistent).	None.
EFI_SW_EC_OUT_OF_ RESOURCES	Insufficient resources exist.	None.
EFI_SW_EC_ABORTED	The instance was aborted.	None.
EFI_SW_EC_ILLEGAL_ SOFTWARE_STATE	The instance detected an illegal software state.	See EFI_DEBUG_ ASSERT_DATA
EFI_SW_EC_ILLEGAL_ HARDWARE_STATE	The instance detected an illegal hardware state.	None.
EFI_SW_EC_START_ERROR	The software module returned an error when started via StartImage() or equivalent.	Handle identifying the module. There will be an instance of EFI_LOADED_IMAGE_PROTOCOL on this handle. See EFI_DEVICE_HANDLE_EXTENDED_ DATA.
EFI_SW_EC_BAD_DATE_TIME	The system date/time is invalid	None.
EFI_SW_EC_CFG_INVALID	Invalid configuration settings were detected.	None.
EFI_SW_EC_CFG_CLR_ REQUEST	User requested that configuration defaults be loaded (via a physical jumper, for example).	None.
EFI_SW_EC_CFG_DEFAULT	Configuration defaults were loaded.	None.
EFI_SW_EC_PWD_INVALID	Invalid password settings were detected.	None.
EFI_SW_EC_PWD_CLR_ REQUEST	User requested that the passwords be cleared (via a physical jumper, for example).	None.
EFI_SW_EC_PWD_CLEARED	Passwords were cleared.	None.

Operation	Description	Extended Data
EFI_SW_EC_EVENT_LOG_FULL	System event log is full.	None.
EFI_SW_EC_WRITE_PROTECTED	The device cannot be written to.	EFI_DEVICE_PATH_EXTENDED_DATA if device path to the write protected device is available; otherwise, none.
EFI_SW_EC_FV_CORRUPTED	Corrupted Firmware Volume is detected.	EFI_DEVICE_PATH_EXTENDED_DATA if device path to the corrupted firmware volume is available; otherwise, none.
0x0014–0x00FF	Reserved for future use by this specification for Host Software class error codes.	None.
0x0100–0x01FF	Unexpected EBC exceptions.	See EFI_STATUS_CODE_EXCEP_EXTENDED_DATA.
0x0200–0x02FF	Unexpected IA-32 processor exceptions.	See EFI_STATUS_CODE_EXCEP_EXTENDED_DATA.
0x0300–0x03FF	Unexpected Itanium® processor family exceptions.	See EFI_STATUS_CODE_EXCEP_EXTENDED_DATA.
0x0400–0x7FFF	See the subclass definitions within this specification.	
0x8000–0xFFFF	Reserved for OEM use.	

6.5.5 Subclasses

6.5.5.1 Defined Subclasses

The table below lists the subclasses in the Host Software class. The following topics describe each subclass in more detail.

See Subclass Definitions in section 6.7.4 for the definitions of these subclasses.

Table 62. Defined Subclasses: Host Software Class

Subclass	Code Name	Description
Unspecified	EFI_SOFTWARE_UNSPECIFIED	The software type is unknown, undefined, or unspecified.
Security (SEC)	EFI_SOFTWARE_SEC	The software is a part of the SEC phase.
PEI Foundation	EFI_SOFTWARE_PEI_CORE	The software is the PEI Foundation module.
PEI module	EFI_SOFTWARE_PEI_MODULE	The software is a PEIM.
DXE Foundation	EFI_SOFTWARE_DXE_CORE	The software is the DXE Foundation module.
DXE Boot Service driver	EFI_SOFTWARE_DXE_BS_DRIVER	The software is a DXE Boot Service driver. Boot service drivers are not available once ExitBootServices() is called.
DXE Runtime Service driver	EFI_SOFTWARE_DXE_RT_DRIVER	The software is a DXE Runtime Service driver. These drivers execute during runtime phase.
SMM driver	EFI_SOFTWARE_SMM_DRIVER	The software is a SMM driver.
EFI application	EFI_SOFTWARE_EFI_APPLICATION	The software is a UEFI application.
OS loader	EFI_SOFTWARE_EFI_OS_LOADER	The software is an OS loader.
Runtime (RT)	EFI_SOFTWARE_EFI_RT	The software is a part of the RT phase.
EBC exception	EFI_SOFTWARE_EBC_EXCEPTION	The status code is directly related to an EBC exception.
IA-32 exception	EFI_SOFTWARE_IA32_EXCEPTION	The status code is directly related to an IA-32 exception.
Itanium® processor family exception	EFI_SOFTWARE_IPF_EXCEPTION	The status code is directly related to an Itanium processor family exception.
x64 software exception	EFI_SOFTWARE_X64_EXCEPTION	The status code is directly related to anx64 exception.
ARM software exception	EFI_SOFTWARE_ARM_EXCEPTION	The status code is directly related to an ARM exception whilst executing in AArch32 state
ARM AArch64 exception	EFI_SOFTWARE_AARCH64_EXCEPTION	The status code is directly related to an ARM exception whilst executing in AArch64 state.
RISC-V software exception	EFI_SOFTWARE_RISCV_EXCEPTION	The status code is directly related to RISC-V exception.
PEI Services	EFI_SOFTWARE_PEI_SERVICE	The status code is directly related to a PEI Services function.
EFI Boot Services	EFI_SOFTWARE_EFI_BOOT_SERVICE	The status code is directly related to a UEFI Boot Services function.
EFI Runtime Services	EFI_SOFTWARE_EFI_RUNTIME_SERVICE	The status code is directly related to a UEFI Runtime Services function.

DXE Services	EFI_SOFTWARE_EFI_DXE_SERVICE	The status code is directly related to a DXE Services function.
0x13–0x7F	Reserved for future use by this specification.	NA
0x80–0xFF	Reserved for OEM use.	NA

6.5.5.2 Unspecified Subclass

This subclass applies to any software entity not belonging to any of the other software subclasses. It may also be used if the caller is unable to determine the exact subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

Table 63. Progress and Error Code Operations: Host Software Unspecified Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5.5.3 SEC Subclass

This subclass applies to the Security (SEC) phase in software.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass. In most platforms, status code services may be unavailable during the SEC phase.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 64. Progress and Error Code Operations: SEC Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_SEC_PC_ENTRY_POINT	Entry point of the phase.	None
	EFI_SW_SEC_PC_HANDOFF_TO_NEXT	Handing off to the next phase	None
	0x1002–0x7FFF	Reserved for future use by this specification.	Reserved for future use by this specification.
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

6.5.5.4 PEI Foundation Subclass

This subclass applies to the PEI Foundation. The PEI Foundation is responsible for starting and ending the PEI phase as well as dispatching Pre-EFI Initialization Modules (PEIMs).

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 65. Progress and Error Code Operations: PEI Foundation Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_PEI_CORE_PC_ENTRY_POINT	Entry point of the phase.	None
	EFI_SW_PEI_CORE_PC_HANDOFF_TO_NEXT	Handing off to the next phase (DXE).	None
	EFI_SW_PEI_CORE_PC_RETURN_TO_LAST	Returning to the last phase.	None
	0x1003–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_SW_PEI_CORE_EC_DXE_CORRUPT	Unable to hand off to DXE because the DXE Foundation could not be found.	None
	EFI_SW_PEI_CORE_EC_DXEIPL_NOT_FOUND	DXE IPL PPI could not be found.	None
	EFI_SW_PEI_CORE_EC_MEMORY_NOT_INSTALLED	PEIM dispatching is over and InstallPeiMemory() PEI Service has not been called	None
	0x1003–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

6.5.5.5 PEI Module Subclass

This subclass applies to Pre-EFI Initialization Modules (PEIMs).

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 66. Progress and Error Code Operations: PEI Module Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_PEI_PC_RECOVERY_BEGIN	Crisis recovery has been initiated.	NULL
	EFI_SW_PEI_PC_CAPSULE_LOAD	Found a recovery capsule. About to load the recovery capsule.	NULL
	EFI_SW_PEI_PC_CAPSULE_START	Loaded the recovery capsule. About to hand off control to the capsule.	NULL
	EFI_SW_PEI_PC_RECOVERY_USER	Recovery was forced by the user via a jumper, for example. Reported by the PEIM that detects the jumpers and updates the boot mode.	NULL
	EFI_SW_PEI_PC_RECOVERY_AUTO	Recovery was forced by the software based on some policy. Reported by the PEIM that updates the boot mode to force recovery.	NULL
	EFI_SW_PEI_PC_S3_BOOT_SCRIPT	S3 boot script execution	NULL
	EFI_SW_PEI_PC_OS_WAKE	Calling OS S3 wake up vector	NULL
	EFI_SW_PEI_PC_S3_STARTED	Signifies beginning of the EFI_PEI_S3_RESUME_PPI.S3RestoreConfig() execution.	NULL
	0x1008–0x7FFF	Reserved for future use by this specification.	NULL
Error	EFI_SW_PEI_EC_NO_RECOVERY_CAPSULE	Unable to continue with the crisis recovery because no recovery capsule was found.	NULL
	EFI_SW_PEI_EC_INVALID_CAPSULE_DESCRIPTOR	An invalid or corrupt capsule descriptor was detected.	NULL
	EFI_SW_PEI_EC_S3_RESUME_PPI_NOT_FOUND	S3 Resume PPI is not found	NULL
	EFI_SW_PEI_EC_S3_BOOT_SCRIPT_ERROR	Error during boot script execution	NULL

Type of Code	Operation	Description	Extended Data
	EFI_SW_PEI_EC_S3_OS_WAKE_ERROR	Error related to the OS wake up vector (no valid vector found or vector returned control back to the firmware)	NULL
	EFI_SW_PEI_EC_S3_RESUME_FAILED	Unspecified S3 resume failure	NULL
	EFI_SW_PEI_EC_RECOVERY_PPI_NOT_FOUND	Recovery failed because Recovery Module PPI is not found	NULL
	EFI_SW_PEI_EC_RECOVERY_FAILED	Unspecified Recovery failure	NULL
	EFI_SW_PEI_EC_S3_RESUME_ERROR	Error during S3 resume process.	NULL
	EFI_SW_PEI_EC_INVALID_CAPSULE	Invalid capsule is detected.	NULL
	0x100A–0x7FFF	Reserved for future use by this specification.	

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

6.5.5.6 DXE Foundation Subclass

This subclass applies to DXE Foundation software. The DXE Foundation is responsible for providing core services, dispatching DXE drivers, and calling the Boot Device Selection (BDS) phase.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 67. Progress and Error Code Operations: DXE Foundation Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_DXE_CORE_PC_ENTRY_POINT	Entry point of the phase.	None
	EFI_SW_DXE_CORE_PC_HANDOFF _TO_NEXT	Handing off to the next phase (Runtime).	None
	EFI_SW_DXE_CORE_PC_RETURN_ TO_LAST	Returning to the last phase.	None
	EFI_SW_DXE_CORE_PC_START_DRIVER	Calling the Start() function of the EFI_DRIVER_BINDIN G Protocol.	See EFI_STATUS_ CODE_START_ EXTENDED_DATA
	EFI_SW_DXE_CORE_PC_ARCH_READY	All architectural protocols are available	None
	0x1005–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_SW_DXE_CORE_EC_NO_ARCH	Driver dispatching is over and some of the architectural protocols are not available	None
	0x1001–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

See Extended Error Data in section 6.7.4 for definitions of the extended error data listed above.

6.5.5.7 DXE Boot Service Driver Subclass

This subclass applies to DXE boot service drivers. If a driver provides both boot services and runtime services, it is considered a runtime service driver.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 68. Progress and Error Code Operations: DXE Boot Service Driver Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_DXE_BS_PC_LEGACY_OPROM_INIT	Initializing a legacy Option ROM (OpROM).	See EFI_LEGACY_OPROM_EXTENDED_DATA.
	EFI_SW_DXE_BS_PC_READY_TO_BOOT_EVENT	The EFI_EVENT_GROUP_READY_TO_BOOT event was signaled. See the *UEFI Specification*.	None
	EFI_SW_DXE_BS_PC_LEGACY_BOOT_EVENT	The event with GUID EFI_EVENT_LEGACY_BOOT_GUID was signaled. See the DXE CIS.	None
	EFI_SW_DXE_BS_PC_EXIT_BOOT_SERVICES_EVENT	The EVT_SIGNAL_EXIT_BOOT_SERVICES event was signaled. See the *UEFI Specification*.	None
	EFI_SW_DXE_BS_PC_VIRTUAL_ADDRESS_CHANGE_EVENT	The EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE event was signaled. See the *UEFI Specification.*	None
	EFI_SW_DXE_BS_PC_VARIABLE_SERVICES_INIT	Variable Services initialization	None
	EFI_SW_DXE_BS_PC_VARIABLE_RECLAIM	Variable Services driver is about to start variable store reclaim process (a process of storage optimization by removing stale data).	None
	EFI_SW_DXE_BS_PC_CONFIG_RESET	Configuration is about to be reset	None
	EFI_SW_DXE_BS_PC_CSM_INIT	Compatibility(legacy BIOS) Support Module initialization.	None
	0x100A–0x7FFF	Reserved for future use by this specification.	NA

Type of Code	Operation	Description	Extended Data
Error	EFI_SW_DXE_BS_EC_LEGACY_ OPROM_NO_SPACE	Not enough memory available to shadow a legacy option ROM.	See EFI_LEGACY_OPROM_ EXTENDED_DATA . RomImageBase corresponds to the ROM image in the regular memory as opposed to shadow RAM.
	EFI_SW_DXE_BS_EC_INVALID_PASSWORD	Invalid password has been provided	None
	EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR	Error during boot option loading (LoadImage returned error)	EFI_RETURN_STATUS_EXTENDED_DATA
	EFI_SW_DXE_BS_EC_BOOT_OPTION_FAILED	Error during boot option launch (StartImage returned error)	EFI_RETURN_STATUS_EXTENDED_DATA
	EFI_SW_DXE_BS_EC_INVALID_IDE_ PASSWORD	Invalid hard driver password has been provided	None
Progress	EFI_SW_DXE_BS_PC_ATTEMPT_BOOT_ORDER_EVENT	Attempting boot from options defined in the BootOrder list	None
	0x1005–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions
- Error Code Definitions

See Extended Error Data in section 6.7.4 for definitions of the extended error data listed above.

6.5.5.8 DXE Runtime Service Driver Subclass

This subclass applies to DXE runtime service drivers.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

Table 69. Progress and Error Code Operations: DXE Runtime Service Driver Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5.5.9 SMM Driver Subclass

This subclass applies to SMM code.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

Table 70. **Progress and Error Code Operations: SMM Driver Subclass**

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5.5.10 EFI Application Subclass

This subclass applies to UEFI applications.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

Table 71. Progress and Error Code Operations: UEFI Application Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5.5.11 OS Loader Subclass

This subclass applies to any OS loader application. Although OS loaders are UEFI applications, they are very special cases and merit a separate subclass.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

Table 72. Progress and Error Code Operations: OS Loader Subclass

Type of Code	Operation	Description	Extended Data
Progress	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

None.

6.5.6 Runtime (RT) Subclass

This subclass applies to runtime software. Runtime software is made up of the UEFI-aware operating system and the non-UEFI software running under the operating system environment. Other firmware components, such as SAL code or ASL code, are also executing during this phase but cannot call a UEFI runtime service. Hence no codes are reserved for them.

Progress and Error Code Operations

- In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 73. Progress and Error Code Operations: Runtime Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_RT_PC_ENTRY_POINT	Entry point of the phase.	None
	EFI_SW_RT_PC_RETURN_TO_LAST	Returning to the last phase.	None
	EFI_SW_RT_PC_HANDOFF_TO_NEXT		
	0x1003–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

6.5.6.1 PEI Services Subclass

This subclass applies to any PEI Service present in the PEI Services Table.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass. These progress codes are reported by the code that provides the specified boot service and not by the module that invokes the given boot service.

Many of the descriptions below refer to the *Platform Initialization Pre-EFI Initialization Core Interface Specification*, or PEI CIS. Also, see "Related Definitions" below for links to the definitions of code listed in this table.

Table 74. Progress and Error Code Operations: PEI Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_PS_PC_INSTALL_PPI	Install a PPI. See the PEI CIS.	None.
	EFI_SW_PS_PC_REINSTALL_PPI	Reinstall a PPI. See the PEI CIS.	None.
	EFI_SW_PS_PC_LOCATE_PPI	Locate an existing PPI. See the PEI CIS.	None.
	EFI_SW_PS_PC_NOTIFY_PPI	Install a notification callback. See the PEI CIS.	None.
	EFI_SW_PS_PC_GET_BOOT_MODE	Get the current boot mode. See the PEI CIS.	None.
	EFI_SW_PS_PC_SET_BOOT_MODE	Set the current boot mode. See the PEI CIS.	None.
	EFI_SW_PS_PC_GET_HOB_LIST	Get the HOB list. See the PEI CIS.	None.
	EFI_SW_PS_PC_CREATE_HOB	Create a HOB. See the PEI CIS.	None.
Progress (cont.)	EFI_SW_PS_PC_FFS_FIND_ NEXT_VOLUME	Find the next FFS formatted firmware volume. See the PEI CIS.	None.
	EFI_SW_PS_PC_FFS_FIND_NEXT_ FILE	Find the next FFS file. See the PEI CIS.	None
	EFI_SW_PS_PC_FFS_FIND_ SECTION_DATA	Find a section in an FFS file. See the PEI CIS.	None.
	EFI_SW_PS_PC_INSTALL_PEI_ MEMORY	Install the PEI memory. See the PEI CIS.	None.
	EFI_SW_PS_PC_ALLOCATE_PAGES	Allocate pages from the memory heap. See the PEI CIS.	None.
	EFI_SW_PS_PC_ALLOCATE_POOL	Allocate from the memory heap. See the PEI CIS.	None.
	EFI_SW_PS_PC_COPY_MEM	Copy memory. See the PEI CIS.	None
	EFI_SW_PS_PC_SET_MEM	Set a memory range to a specific value. See the PEI CIS.	None.
	EFI_SW_PS_PC_RESET_SYSTEM	System reset. See the PEI CIS.	None
	EFI_SW_PS_PC_FFS_FIND_FILE_BY_NAME	Find a file in a firmware volume by name. See the PEI CIS.	None
	EFI_SW_PS_PC_FFS_GET_FILE_INFO	Get information about a file in a firmware volume. See the PEI CIS.	None
	EFI_SW_PS_PC_FFS_GET_VOLUME_INFO	Get information about a firmware volume. See the PEI CIS.	None

Type of Code	Operation	Description	Extended Data
	EFI_SW_PS_PC_FFS_REGISTER_FOR_SHADOW	Register a module to be shadowed after permanent memory is discovered. See the PEI CIS.	None
	0x1017-0x7fff	Reserved for future use by this specification.	NA
Error	EFI_SW_PS_EC_RESET_NOT_AVAILABLE	ResetSystem() PEI Service is failed because Reset PPI is not available	None
	EFI_SW_PS_EC_MEMORY_INSTALLED_TWICE	InstallPeiMemory() PEI Service is called more than once	None
	0x1002–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

6.5.6.2 Boot Services Subclass

This subclass applies to any boot service present in the UEFI Boot Services Table.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass. These progress codes are reported by the code that provides the specified boot service and not by the module that invokes the given boot service.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 75. Progress and Error Code Operations: Boot Services Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_BS_PC_RAISE_TPL	Raise the task priority level service; see UEFI Specification. This code is an invalid operation because the status code driver uses this boot service. The status code driver cannot report its own status codes.	None.
	EFI_SW_BS_PC_RESTORE_TPL	Restore the task priority level service; see UEFI Specification. This code is an invalid operation because the status code driver uses this boot service. The status code driver cannot report its own status codes.	None.
	EFI_SW_BS_PC_ALLOCATE_PAGE	Allocate page service; see UEFI Specification.	None.
	EFI_SW_BS_PC_FREE_PAGES	Free page service; see UEFI Specification.	None.
	EFI_SW_BS_PC_GET_MEMORY_MAP	Get memory map service; see UEFI Specification.	None.
	EFI_SW_BS_PC_ALLOCATE_POOL	Allocate pool service; see UEFI Specification.	None.
	EFI_SW_BS_PC_FREE_POOL	Free pool service; see UEFI Specification.	None.
	EFI_SW_BS_PC_CREATE_EVENT	CreateEvent service; see UEFI Specification.	None.
	EFI_SW_BS_PC_SET_TIMER	Set timer service; see UEFI Specification.	None.
	EFI_SW_BS_PC_WAIT_FOR_EVENT	Wait for event service; see UEFI Specification.	None.
Progress (cont.)	EFI_SW_BS_PC_SIGNAL_EVENT	Signal event service; see UEFI Specification. This code is an invalid operation because the status code driver uses this boot service. The status code driver cannot report its own status codes.	None.
	EFI_SW_BS_PC_CLOSE_EVENT	Close event service; see UEFI Specification.	None.
	EFI_SW_BS_PC_CHECK_EVENT	Check event service; see UEFI Specification.	None.
	EFI_SW_BS_PC_INSTALL_ PROTOCOL_INTERFACE	Install protocol interface service; see UEFI Specification.	None.
	EFI_SW_BS_PC_REINSTALL_ PROTOCOL_INTERFACE	Reinstall protocol interface service; see UEFI Specification.	None.

Type of Code	Operation	Description	Extended Data
	EFI_SW_BS_PC_UNINSTALL_ PROTOCOL_INTERFACE	Uninstall protocol interface service; see UEFI Specification.	None.
	EFI_SW_BS_PC_HANDLE_ PROTOCOL	Handle protocol service; see UEFI Specification.	None.
	EFI_SW_BS_PC_PC_HANDLE_ PROTOCOL	PC handle protocol service; see UEFI Specification.	None.
	EFI_SW_BS_PC_REGISTER_ PROTOCOL_ NOTIFY	Register protocol notify service; see UEFI Specification.	None.
	EFI_SW_BS_PC_LOCATE_HANDLE	Locate handle service; see UEFI Specification.	None.
	EFI_SW_BS_PC_INSTALL_ CONFIGURATION_TABLE	Install configuration table service; see UEFI Specification.	None.
	EFI_SW_BS_PC_LOAD_IMAGE	Load image service; see UEFI Specification.	None.
	EFI_SW_BS_PC_START_IMAGE	Start image service; see UEFI Specification.	None.
	EFI_SW_BS_PC_EXIT	Exit service; see UEFI Specification.	None.
	EFI_SW_BS_PC_UNLOAD_IMAGE	Unload image service; see UEFI Specification.	None.
	EFI_SW_BS_PC_EXIT_BOOT_ SERVICES	Exit boot services service; see UEFI Specification.	None.
	EFI_SW_BS_PC_GET_NEXT_ MONOTONIC_COUNT	Get next monotonic count service; see UEFI Specification.	None.
	EFI_SW_BS_PC_STALL	Stall service; see UEFI Specification.	None.
	EFI_SW_BS_PC_SET_WATCHDOG_ TIMER	Set watchdog timer service; see UEFI Specification.	None.
	EFI_SW_BS_PC_CONNECT_ CONTROLLER	Connect controller service; see UEFI Specification.	None.
Progress (cont.)	EFI_SW_BS_PC_DISCONNECT_ CONTROLLER	Disconnect controller service; see UEFI Specification.	None.
	EFI_SW_BS_PC_OPEN_PROTOCOL	Open protocol service; see UEFI Specification.	None.
	EFI_SW_BS_PC_CLOSE_PROTOCOL	Close protocol service; see UEFI Specification.	None.
	EFI_SW_BS_PC_OPEN_PROTOCOL_ INFORMATION	Open protocol Information service; see UEFI Specification.	None.
	EFI_SW_BS_PC_PROTOCOLS_PER_ HANDLE	Protocols per handle service; see UEFI Specification.	None.
	EFI_SW_BS_PC_LOCATE_HANDLE_ BUFFER	Locate handle buffer service; see UEFI Specification.	None.

Type of Code	Operation	Description	Extended Data
	EFI_SW_BS_PC_LOCATE_PROTOCOL	Locate protocol service; see UEFI Specification.	None.
	EFI_SW_BS_PC_INSTALL_MULTIPLE_PROTOCOL_INTERFACES	Install multiple protocol interfaces service; see UEFI Specification.	None.
	EFI_SW_BS_PC_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES	Uninstall multiple protocol interfaces service; see UEFI Specification.	None.
	EFI_SW_BS_PC_CALCULATE_CRC_32	Calculate CRC32 service; see UEFI Specification.	None.
	EFI_SW_BS_PC_COPY_MEM	Copy memory; see UEFI Specification.	None.
	EFI_SW_BS_PC_SET_MEM	Set memory to a specific value; see UEFI Specification.	None.
	EFI_SW_BS_PC_CREATE_EVENT_EX	Create an event and, optionally, associate it with an event group. See the UEFI Specification.	None.
	0x102b-0x7fff	Reserved for future use by this specification.	NA.
Error	0x1000 – 0x7FFF	Reserved for future use by this specification.	NA.

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

6.5.6.3 Runtime Services Subclass

This subclass applies to any runtime service present in the UEFI Runtime Services Table.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass. For obvious reasons, the runtime service **ReportStatusCode()** cannot report status codes related to the progress of the **ReportStatusCode()** function.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 76. Progress and Error Code Operations: Runtime Services Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_RS_PC_GET_TIME	Get time service; see UEFI Specification.	None.
	EFI_SW_RS_PC_SET_TIME	Set time service; see UEFI Specification.	None
	EFI_SW_RS_PC_GET_WAKEUP_TIME	Get wakeup time service; see UEFI Specification.	None
	EFI_SW_RS_PC_SET_WAKEUP_TIME	Set wakeup time service; see UEFI Specification.	None
	EFI_SW_RS_PC_SET_VIRTUAL_ ADDRESS_MAP	Set virtual address map service; see UEFI Specification.	None
	EFI_SW_RS_PC_CONVERT_POINTER	Convert pointer service; see UEFI Specification.	None
	EFI_SW_RS_PC_GET_ VARIABLE	Get variable service; see UEFI Specification.	None
	EFI_SW_RS_PC_GET_NEXT_ VARIABLE_NAME	Get next variable name service; see UEFI Specification.	None
	EFI_SW_RS_PC_SET_VARIABLE	Set variable service; see UEFI Specification.	None
	EFI_SW_RS_PC_GET_NEXT_HIGH_ MONOTONIC_COUNT	Get next high monotonic count service; see UEFI Specification.	None
	EFI_SW_RS_PC_RESET_SYSTEM	Reset system service; see UEFI Specification.	None
	EFI_SW_RS_PC_UPDATE_CAPSULE	Update a capsule. See the UEFI Specification.	None
	EFI_SW_RS_PC_QUERY_CAPSULE_CAPABILITIES	Query firmware support for capsulate capabilities. See the UEFI specification.	None
	EFI_SW_RS_PC_QUERY_VARIABLE_INFO	Query firmware support for EFI variables. See the UEFI specification.	None
	0x100E	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

- Progress Code Definitions

6.5.6.4 DXE Services Subclass

This subclass applies to any DXE Service that present in the UEFI DXE Services Table.

Progress and Error Code Operations

In addition to the standard progress and error codes that are defined for the Host Software class, the table below lists the additional codes for this subclass.

See "Related Definitions" below for links to the definitions of code listed in this table.

Table 77. Progress and Error Code Operations: DXE Services Subclass

Type of Code	Operation	Description	Extended Data
Progress	EFI_SW_DS_PC_ADD_ MEMORY_SPACE	Add memory to GCD. See DXE CIS.	None
	EFI_SW_DS_PC_ALLOCATE_MEMORY_ SPACE	Allocate memory from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_FREE_MEMORY_SPACE	Free memory from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_REMOVE_MEMORY_ SPACE	Remove memory from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_GET_MEMORY_SPACE_ DESCRIPTOR	Get memory descriptor from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_SET_MEMORY_SPACE_ ATTRIBUTES	Set attributes of memory in GCD. See DXE CIS.	None
	EFI_SW_DS_PC_GET_MEMORY_SPACE_ MAP	Get map of memory space from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_ADD_IO_SPACE	Add I/O to GCD. See DXE CIS.	None
	EFI_SW_DS_PC_ALLOCATE_IO_SPACE	Allocate I/O from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_FREE_IO_SPACE	Free I/O from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_REMOVE_IO_SPACE	Remove I/O space from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_GET_IO_SPACE_ DESCRIPTOR	Get I/O space descriptor from GCD. See DXE CIS.	None
	EFI_SW_DS_PC_GET_IO_SPACE_MAP	Get map of I/O space from the GCD. See DXE CIS.	None
	EFI_SW_DS_PC_DISPATCH	Dispatch DXE drivers from a firmware volume. See DXE CIS.	None
	EFI_SW_DS_PC_SCHEDULE	Clear the schedule on request flag for a driver. See DXE CIS.	None
	EFI_SW_DS_PC_TRUST	Promote a file to trusted state. See DXE CIS.	None
	EFI_SW_DS_PC_PROCESS_FIRMWARE_ VOLUME	Dispatch all drivers in a firmware volume. See DXE CIS.	None
	0x1011–0x7FFF	Reserved for future use by this specification.	NA
Error	0x1000–0x7FFF	Reserved for future use by this specification.	NA

Related Definitions

See the following topic in section 6.7.4 for definitions of the subclass-specific operations listed above:

Progress Code Definitions

6.6 Code Definitions

This section provides the code definitions for the following data types and structures for status codes:

- Data structures and types that are common to all status codes
- Progress, error, and debug codes that are common to all classes
- Class definitions
- Subclass definitions for each status code class
- Extended error data

This section defines the data structures that are common to all status codes. For class- and subclass-specific information, see section 6.7.

6.6.1 Data Structures

See the **ReportStatusCode()** declaration in Volume 2 of this specification for definitions and details on the following basic data structures:

- **EFI_STATUS_CODE_TYPE** and defined severities
- **EFI_PROGRESS_CODE**
- **EFI_ERROR_CODE**
- **EFI_DEBUG_CODE**
- **EFI_STATUS_CODE_VALUE**

6.6.2 Extended Data Header

EFI_STATUS_CODE_DATA

Summary

The definition of the status code extended data header. The data will follow *HeaderSize* bytes from the beginning of the structure and is *Size* bytes long.

Related Definitions

```
typedef struct {
  UINT16    HeaderSize;
  UINT16    Size;
  EFI_GUID  Type;
} EFI_STATUS_CODE_DATA;
```

Parameters

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. The standard GUIDs and their associated data structures are defined in this specification.

Description

The status code information may be accompanied by optional extended data. The extended data begins with a header. The header contains a `Type` field that represents the format of the extended data following the header. This specification defines two GUIDs and their meaning. If these GUIDs are used, the extended data contents must follow this specification. Extended data formats that are not compliant with this specification are permitted, but they must use different GUIDs. The format of the extended data header is defined in *Platform Initialization DXE CIS*, but it is duplicated here for convenience.

EFI_STATUS_CODE_DATA_TYPE_STRING_GUID

Summary

Defines a string type of extended data.

GUID

```
#define EFI_STATUS_CODE_DATA_TYPE_STRING_GUID \
  { 0x92D11080, 0x496F, 0x4D95, 0xBE, 0x7E, 0x03, 0x74, \
  0x88, 0x38, 0x2B, 0x0A }
```

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                         DataHeader;
  EFI_STRING_TYPE                              StringType;
  EFI_STATUS_CODE_STRING                       String;
} EFI_STATUS_CODE_STRING_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, **DataHeader.Size** should be **sizeof (EFI_STATUS_CODE_STRING_DATA) – HeaderSize**, and *DataHeader.Type* should be **EFI_STATUS_CODE_DATA_TYPE_STRING_GUID**.

StringType

Specifies the format of the data in *String*. Type **EFI_STRING_TYPE** is defined in "Related Definitions" below.

String

A pointer to the extended data. The data follows the format specified by *StringType*. Type **EFI_STRING_TYPE** is defined in "Related Definitions" below.

Description

This data type defines a string type of extended data. A string can accompany any status code. The string can provide additional information about the status code. The string can be ASCII, Unicode, or a Human Interface Infrastructure (HII) token/GUID pair.

Related Definitions

```
//**********************************************
// EFI_STRING_TYPE
//**********************************************

typedef enum {
  EfiStringAscii,
  EfiStringUnicode,
  EfiStringToken
} EFI_STRING_TYPE;
```

EfiStringAscii

A **NULL**-terminated ASCII string.

EfiStringUnicode

A double **NULL**-terminated Unicode string.

EfiStringToken

An **EFI_STATUS_CODE_STRING_TOKEN** representing the string. The actual string can be obtained by querying the HII database.

```
//**********************************************
// EFI_STATUS_CODE_STRING_TOKEN
//**********************************************

//
// HII string token
//
typedef struct {
  EFI_HII_HANDLE    Handle;
  EFI_STRING_ID     Token;
} EFI_STATUS_CODE_STRING_TOKEN;
```

Handle

The HII package list which contains the string. *Handle* is a dynamic value that may not be the same for different boots. Type **EFI_HII_HANDLE** is defined in **EFI_HII_DATABASE_PROTOCOL.NewPackageList()** in the *UEFI Specification.*

Token

When combined with *Handle*, the string token can be used to retrieve the string. Type **EFI_STRING_ID** is defined in **EFI_IFR_OP_HEADER** in the *UEFI Specification.*

```
//*********************************************************
// EFI_STATUS_CODE_STRING
//*********************************************************

//
// String structure
//
typedef union {
  CHAR8                         *Ascii;
  CHAR16                        *Unicode;
  EFI_STATUS_CODE_STRING_TOKEN  Hii;
} EFI_STATUS_CODE_STRING;
```

Ascii

ASCII formatted string.

Unicode

Unicode formatted string.

Hii

HII handle/token pair. Type **EFI_STATUS_CODE_STRING_TOKEN** is defined above.

EFI_STATUS_CODE_SPECIFIC_DATA_GUID

Summary

Indicates that the format of the accompanying data depends upon the status code value but follows this specification.

GUID

```
#define EFI_STATUS_CODE_SPECIFIC_DATA_GUID \
  {0x335984bd,0xe805,0x409a,0xb8,0xf8,0xd2,0x7e, \
  0xce,0x5f,0xf7,0xa6}
```

Description

This GUID indicates that the format of the accompanying data depends upon the status code value but follows this specification. This specification defines the format of the extended data for several status code values. For example, **EFI_DEBUG_ASSERT_DATA** defines the extended error data for the error code **EFI_SW_EC_ILLEGAL_SOFTWARE_STATE**. The agent reporting this error condition can use this GUID if the extended data follows the format defined in **EFI_DEBUG_ASSERT_DATA**.

If the consumer of the status code detects this GUID, it must look up the status code value to correctly interpret the contents of the extended data.

This specification declares certain ranges of status code values as OEM specific. Because this specification does not define the meaning of status codes in these ranges, the extended data for these cannot use this GUID. The OEM defining the meaning of the status codes is responsible for defining the GUID that is to be used for associated extended data.

6.6.3 Enumeration Schemes

6.6.3.1 Operation Code Enumeration Scheme

Summary

All operation codes (regardless of class and subclass) use the progress code partitioning scheme listed in the table below.

Table 78. Progress Code Enumeration Scheme

Operation	Description
0x0000–0x0FFF	These operation codes are common to all the subclasses in a given class. These values are used to represent operations that are common to all subclasses in a given class. For example, all the I/O buses in the I/O Bus subclasses share an operation code that represents the reset operation, which is a common operation for most buses. It is possible that certain operation codes in this range will not be applicable to certain subclasses. It is also possible that the format of the extended data will vary from one subclass to another. If the subclass does not define the format of the extended data, extended data is not required. These codes are reserved by this specification.
0x1000–0x7FFF	These operation codes are specific to the subclass and represent operations that are specific to the subclass. These codes are reserved by this specification.
0x8000–0xFFFF	Reserved for OEM use.

Prototype

```
//
// General partitioning scheme for Progress and Error Codes
// 0x0000-0x0FFF  - Shared by all subclasses in a given class
// 0x1000-0x7FFF  - Subclass Specific
// 0x8000-0xFFFF  - OEM specific
//
#define EFI_SUBCLASS_SPECIFIC              0x1000
#define EFI_OEM_SPECIFIC                   0x8000
```

6.6.3.2 Debug Code Enumeration Scheme

Summary

All classes share these debug operation codes. It is not currently expected that operation codes have a lot of meaning for debug information. Only one debug code is currently defined by this specification and it is shared by all classes and subclasses.

Table 79. Debug Code Enumeration Scheme

Debug Code	Description
0x0000–0x7FFF	Reserved for future use by this specification.
0x8000–0xFFFF	Reserved for OEM use.

Prototype

```
//
// Debug Code definitions for all classes and subclass
// Only one debug code is defined at this point and should
// be used for anything that gets sent to debug stream.
//
#define  EFI_DC_UNSPECIFIED      0x0
```

6.6.4 Common Extended Data Formats

This section specifies formats for the extended data included in a variety of status codes.

EFI_DEVICE_PATH_EXTENDED_DATA

Summary

Extended data about the device path, which is used for many errors and progress codes to point to the device.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                DataHeader;
  // EFI_DEVICE_PATH_PROTOCOL         DevicePath;
} EFI_DEVICE_PATH_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**. *DataHeader.Size* should be the size of variable-length *DevicePath*, and *DataHeader.Size* is zero for a virtual device that does not have a device path. *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

DevicePath

The device path to the controller or the hardware device. Note that this parameter is a variable-length device path structure and not a pointer to such a structure. This structure is populated only if it is a physical device. For virtual devices, the *Size* field in *DataHeader* is set to zero and this field is not populated.

Description

The device path is used to point to the physical device in case there is more than one device belonging to the same subclass. For example, the system may contain two USB keyboards and one PS/2* keyboard. The driver that parses the status code can use the device path extended data to differentiate between the three. The index field is not useful in this case because there is no standard numbering convention. Device paths are preferred over using device handles because device handles for a given device can change from one boot to another and do not mean anything beyond Boot Services time. In certain cases, the bus driver may not create a device handle for a given device if it detects a critical error. In these cases, the device path extended data can be used to refer to the device, but there may not be any device handles with an instance of **`EFI_DEVICE_PATH_PROTOCOL`** that matches *`DevicePath`*. The variable device path structure is included in this structure to make it self sufficient.

EFI_DEVICE_HANDLE_EXTENDED_DATA

Summary

Extended data about the device handle, which is used for many errors and progress codes to point to the device.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                    DataHeader;
  EFI_HANDLE                              Handle;
} EFI_DEVICE_HANDLE_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_DEVICE_HANDLE_EXTENDED_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Handle

The device handle.

Description

The handle of the device with which the progress or error code is associated. The handle is guaranteed to be accurate only at the time the status code is reported. Handles are dynamic entities between boots, so handles cannot be considered to be valid if the system has reset subsequent to the status code being reported. Handles may be used to determine a wide variety of useful information about the source of the status code.

EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA

Summary

This structure defines extended data describing a PCI resource allocation error.

Prototype

Note: *The following structure contains variable-length fields and cannot be defined as a C-style structure.*

```
typedef struct {
  EFI_STATUS_CODE_DATA                DataHeader;
  UINT32                              Bar;
  UINT16                              DevicePathSize;
  UINT16                              ReqResSize;
  UINT16                              AllocResSize;
  // EFI_DEVICE_PATH_PROTOCOL         DevicePath;
  // UINT8                            ReqRes[…];
  // UINT8                            AllocRes[…];

} EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **(***DevicePathSize + DevicePathSize + DevicePathSize* + **sizeof(UINT32) + 3 * sizeof (UINT16)),** and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID.**

Bar

The PCI BAR. Applicable only for PCI devices. Ignored for all other devices.

DevicePathSize

DevicePathSize should be zero if it is a virtual device that is not associated with a device path. Otherwise, this parameter is the length of the variable-length *DevicePath*.

ReqResSize

Represents the size the *ReqRes* parameter. *ReqResSize* should be zero if the requested resources are not provided as a part of extended data.

AllocResSize

Represents the size the *AllocRes* parameter. *AllocResSize* should be zero if the allocated resources are not provided as a part of extended data.

DevicePath

The device path to the controller or the hardware device that did not get the requested resources. Note that this parameter is the variable-length device path structure and not a pointer to this structure.

ReqRes

The requested resources in the format of an ACPI 2.0 resource descriptor. This parameter is not a pointer; it is the complete resource descriptor.

AllocRes

The allocated resources in the format of an ACPI 2.0 resource descriptor. This parameter is not a pointer; it is the complete resource descriptor.

Description

This extended data conveys details for a PCI resource allocation failure error. See the PCI specification and the ACPI specification for details on PCI resource allocations and the format for resource descriptors. This error does not detail why the resource allocation failed. It may be due to a bad resource request or a lack of available resources to satisfy a valid request. The variable device path structure and the resource structures are included in this structure to make it self sufficient.

6.7 Class Definitions

Summary

Classes correspond to broad types of system pieces. These types are chosen to provide a reasonable initial classification of the system entity whose status is represented. There are three classes of hardware and one class for software. These classes are listed in the table below. Each class is made up of several subclasses. See section 6.3for descriptions of each of these classes.

Table 80. Class Definitions

Type of Class	Class Name	Data Type Name
Hardware	Computing Unit	EFI_COMPUTING_UNIT
	User-Accessible Peripherals	EFI_PERIPHERAL
	I/O Bus	EFI_IO_BUS
Software	Host Software	EFI_SOFTWARE

Prototype

```
//
// Class definitions
// Values of 4-127 are reserved for future use by this
// specification.
// Values in the range 127-255 are reserved for OEM use.
//
#define EFI_COMPUTING_UNIT                      0x00000000
#define EFI_PERIPHERAL                          0x01000000
#define EFI_IO_BUS                              0x02000000
#define EFI_SOFTWARE                            0x03000000
```

6.7.1 Computing Unit Class

The table below lists the subclasses defined in the Computing Unit class. See the following section for their code definitions.

Table 81. Defined Subclasses: Computing Unit Class

Subclass	Code Name
Unspecified	EFI_COMPUTING_UNIT_UNSPECIFIED
Host processor	EFI_COMPUTING_UNIT_HOST_PROCESSOR
Firmware processor	EFI_COMPUTING_UNIT_FIRMWARE_PROCESSOR
Service processor	EFI_COMPUTING_UNIT_SERVICE_PROCESSOR
I/O processor	EFI_COMPUTING_UNIT_IO_PROCESSOR
Cache	EFI_COMPUTING_UNIT_CACHE
Memory	EFI_COMPUTING_UNIT_MEMORY
Chipset	EFI_COMPUTING_UNIT_CHIPSET

6.7.1.1 Subclass Definitions

Summary

Definitions for the Computing Unit subclasses. See Subclasses in section 6.7.1 for descriptions of these subclasses.

Prototype

```
//
// Computing Unit Subclass definitions.
// Values of 8-127 are reserved for future use by this
// specification.
// Values of 128-255 are reserved for OEM use.
//
#define EFI_COMPUTING_UNIT_UNSPECIFIED \
  (EFI_COMPUTING_UNIT | 0x00000000)
#define EFI_COMPUTING_UNIT_HOST_PROCESSOR \
  (EFI_COMPUTING_UNIT | 0x00010000)
#define EFI_COMPUTING_UNIT_FIRMWARE_PROCESSOR \
  (EFI_COMPUTING_UNIT | 0x00020000)
#define EFI_COMPUTING_UNIT_IO_PROCESSOR \
  (EFI_COMPUTING_UNIT | 0x00030000)
#define EFI_COMPUTING_UNIT_CACHE \
  (EFI_COMPUTING_UNIT | 0x00040000)
#define EFI_COMPUTING_UNIT_MEMORY \
  (EFI_COMPUTING_UNIT | 0x00050000)
#define EFI_COMPUTING_UNIT_CHIPSET \
  (EFI_COMPUTING_UNIT | 0x00060000)
```

6.7.1.2 Progress Code Definitions

Summary

Progress code definitions for the Computing Unit class and all subclasses. See Progress Code Operations in section 6.7.1 for descriptions of these progress codes.

The following subclasses define additional subclass-specific progress code operations, which are included below:

- Host processor
- Cache
- Memory

Prototype

```
//
// Computing Unit Class Progress Code definitions.
// These are shared by all subclasses.
//
#define EFI_CU_PC_INIT_BEGIN           0x00000000
#define EFI_CU_PC_INIT_END             0x00000001


//
// Computing Unit Unspecified Subclass Progress Code
// definitions.
//


//
// Computing Unit Host Processor Subclass Progress Code
// definitions.
//
#define EFI_CU_HP_PC_POWER_ON_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_HP_PC_CACHE_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_HP_PC_RAM_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CU_HP_PC_MEMORY_CONTROLLER_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_CU_HP_PC_IO_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_CU_HP_PC_BSP_SELECT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_CU_HP_PC_BSP_RESELECT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_CU_HP_PC_AP_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_CU_HP_PC_SMM_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)


//
// Computing Unit Firmware Processor Subclass Progress Code
// definitions.
//


//
// Computing Unit IO Processor Subclass Progress Code
// definitions.
//


//
// Computing Unit Cache Subclass Progress Code definitions.
//
#define EFI_CU_CACHE_PC_PRESENCE_DETECT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_CACHE_PC_CONFIGURATION \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)


//
// Computing Unit Memory Subclass Progress Code definitions.
//
#define EFI_CU_MEMORY_PC_SPD_READ \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_MEMORY_PC_PRESENCE_DETECT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_MEMORY_PC_TIMING \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CU_MEMORY_PC_CONFIGURING \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_CU_MEMORY_PC_OPTIMIZING \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_CU_MEMORY_PC_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_CU_MEMORY_PC_TEST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)


//
// Computing Unit Chipset Subclass Progress Code definitions.
//
#define EFI_CHIPSET_PC_PEI_CAR_SB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CHIPSET_PC_PEI_CAR_NB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CHIPSET_PC_PEI_MEM_SB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CHIPSET_PC_PEI_MEM_NB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_CHIPSET_PC_DXE_HB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_CHIPSET_PC_DXE_NB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
```

```
#define EFI_CHIPSET_PC_DXE_NB_SMM_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_CHIPSET_PC_DXE_SB_RT_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_CHIPSET_PC_DXE_SB_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_CHIPSET_PC_DXE_SB_SMM_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_CHIPSET_PC_DXE_SB_DEVICES_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
```

6.7.1.3 Error Code Definitions

Summary

Error code definitions for the Computing Unit class and all subclasses. See Error Code Operations in section 6.7.1 for descriptions of these error codes.

The following subclasses define additional subclass-specific error code operations, which are included below:

- Host processor
- Firmware processor
- Cache
- Memory

Prototype

```
//
// Computing Unit Class Error Code definitions.
// These are shared by all subclasses.
//
#define EFI_CU_EC_NON_SPECIFIC        0x00000000
#define EFI_CU_EC_DISABLED             0x00000001
#define EFI_CU_EC_NOT_SUPPORTED        0x00000002
#define EFI_CU_EC_NOT_DETECTED         0x00000003
#define EFI_CU_EC_NOT_CONFIGURED       0x00000004


//
// Computing Unit Unspecified Subclass Error Code definitions.
//


//
// Computing Unit Host Processor Subclass Error Code
definitions.
//
#define EFI_CU_HP_EC_INVALID_TYPE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_HP_EC_INVALID_SPEED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_HP_EC_MISMATCH \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CU_HP_EC_TIMER_EXPIRED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_CU_HP_EC_SELF_TEST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_CU_HP_EC_INTERNAL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_CU_HP_EC_THERMAL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_CU_HP_EC_LOW_VOLTAGE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_CU_HP_EC_HIGH_VOLTAGE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_CU_HP_EC_CACHE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_CU_HP_EC_MICROCODE_UPDATE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
#define EFI_CU_HP_EC_CORRECTABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000B)
#define EFI_CU_HP_EC_UNCORRECTABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000C)
```

```
#define EFI_CU_HP_EC_NO_MICROCODE_UPDATE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000D)

//
// Computing Unit Firmware Processor Subclass Error Code
// definitions.
//
#define EFI_CU_FP_EC_HARD_FAIL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_FP_EC_SOFT_FAIL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_FP_EC_COMM_ERROR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)


//
// Computing Unit IO Processor Subclass Error Code definitions.
//


//
// Computing Unit Cache Subclass Error Code definitions.
//
#define EFI_CU_CACHE_EC_INVALID_TYPE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_CACHE_EC_INVALID_SPEED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_CACHE_EC_INVALID_SIZE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CU_CACHE_EC_MISMATCH \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)


//
// Computing Unit Memory Subclass Error Code definitions.
//
#define EFI_CU_MEMORY_EC_INVALID_TYPE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_CU_MEMORY_EC_INVALID_SPEED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_CU_MEMORY_EC_CORRECTABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_CU_MEMORY_EC_UNCORRECTABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_CU_MEMORY_EC_SPD_FAIL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_CU_MEMORY_EC_INVALID_SIZE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_CU_MEMORY_EC_MISMATCH \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_CU_MEMORY_EC_S3_RESUME_FAIL\
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_CU_MEMORY_EC_UPDATE_FAIL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_CU_MEMORY_EC_NONE_DETECTED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_CU_MEMORY_EC_NONE_USEFUL \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)


//
// Computing Unit Chipset Subclass Error Code definitions.
//
#define EFI_CHIPSET_EC_BAD_BATTERY      \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000000)
#define EFI_CHIPSET_EC_DXE_NB_ERROR     \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000001)
#define EFI_CHIPSET_EC_DXE_SB_ERROR     \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000002)
#define EFI_CHIPSET_EC_INTRUDER_DETECT  \
   (EFI_SUBCLASS_SPECIFIC | 0x00000003)
```

6.7.1.4 Extended Data Formats

6.7.1.4.1 Host Processor Subclass

EFI_COMPUTING_UNIT_VOLTAGE_ERROR_DATA

Summary

This structure provides details about the computing unit voltage error.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                DataHeader;
  EFI_EXP_BASE10_DATA                 Voltage;
  EFI_EXP_BASE10_DATA                 Threshold;
} EFI_COMPUTING_UNIT_VOLTAGE_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA),** *DataHeader.Size* should be

sizeof (EFI_COMPUTING_UNIT_VOLTAGE_ERROR_DATA) - *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Voltage

The voltage value at the time of the error.

Threshold

The voltage threshold.

Description

This structure provides the voltage at the time of error. It also provides the threshold value indicating the minimum or maximum voltage that is considered an error. If the voltage is less than the threshold, the error indicates that the voltage fell below the minimum acceptable value. If the voltage is greater than the threshold, the error indicates that the voltage rose above the maximum acceptable value.

EFI_COMPUTING_UNIT_MICROCODE_UPDATE_ERROR_DATA

Summary

This structure provides details about the microcode update error.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                  DataHeader;
  UINT32                                Version;
} EFI_COMPUTING_UNIT_MICROCODE_UPDATE_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA),** *DataHeader.Size* should be **sizeof (EFI_COMPUTING_UNIT_MICROCODE_UPDATE_ERROR_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Version

The version of the microcode update from the header.

EFI_COMPUTING_UNIT_TIMER_EXPIRED_ERROR_DATA

Summary

This structure provides details about the computing unit timer expiration error.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                  DataHeader;
  EFI_EXP_BASE10_DATA                   TimerLimit;
} EFI_COMPUTING_UNIT_TIMER_EXPIRED_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA),** *DataHeader.Size* should be **sizeof (EFI_COMPUTING_UNIT_TIMER_EXPIRED_ERROR_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

TimerLimit

The number of seconds that the computing unit timer was configured to expire.

Description

The timer limit provides the timeout value of the timer prior to expiration.

EFI_HOST_PROCESSOR_MISMATCH_ERROR_DATA

Summary

This structure defines extended data for processor mismatch errors.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                              DataHeader;
  UINT32                                            Instance;
  UINT16                                            Attributes;
} EFI_HOST_PROCESSOR_MISMATCH_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_ HOST_PROCESSOR_MISMATCH_ERROR_DATA) -** ***HeaderSize*** , and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID.**

Instance

The unit number of the computing unit that does not match.

Attributes

The attributes describing the failure. See “Related Definitions” below for the type declarations.

Description

This provides information to indicate which processors mismatch, and how they mismatch. The status code contains the instance number of the processor that is in error. This structure's *Instance* indicates the second processor that does not match. This differentiation allows the consumer to determine which two processors do not match. The *Attributes* indicate what mismatch is being reported. Because *Attributes* is a bit field, more than one mismatch can be reported with one error code.

Related Definitions

```
//*************************************************************
// EFI_COMPUTING_UNIT_MISMATCH_ATTRIBUTES
//*************************************************************
//
// All other attributes are reserved for future use and
// must be initialized to 0.
//
#define EFI_COMPUTING_UNIT_MISMATCH_SPEED       0x0001
#define EFI_COMPUTING_UNIT_MISMATCH_FSB_SPEED   0x0002
#define EFI_COMPUTING_UNIT_MISMATCH_FAMILY      0x0004
#define EFI_COMPUTING_UNIT_MISMATCH_MODEL       0x0008
#define EFI_COMPUTING_UNIT_MISMATCH_STEPPING    0x0010
#define EFI_COMPUTING_UNIT_MISMATCH_CACHE_SIZE  0x0020
#define EFI_COMPUTING_UNIT_MISMATCH_OEM1        0x1000
#define EFI_COMPUTING_UNIT_MISMATCH_OEM2        0x2000
#define EFI_COMPUTING_UNIT_MISMATCH_OEM3        0x4000
#define EFI_COMPUTING_UNIT_MISMATCH_OEM4        0x8000
```

EFI_COMPUTING_UNIT_THERMAL_ERROR_DATA

Summary

This structure provides details about the computing unit thermal failure.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                   DataHeader;
  EFI_EXP_BASE10_DATA                    Temperature;
  EFI_EXP_BASE10_DATA                    Threshold;
} EFI_COMPUTING_UNIT_THERMAL_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_COMPUTING_UNIT_THERMAL_ERROR_DATA) -** *HeaderSize* , and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Temperature

The thermal value at the time of the error.

Threshold

The thermal threshold.

Description

This structure provides the temperature at the time of error. It also provides the threshold value indicating the minimum temperature that is considered an error.

EFI_CACHE_INIT_DATA

Summary

This structure provides cache initialization data.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                 DataHeader;
  UINT32                               Level;
  EFI_INIT_CACHE_TYPE                  Type;
} EFI_CACHE_INIT_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_CACHE_INIT_DATA) -** *HeaderSize* , and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Level

The cache level. Starts with 1 for level 1 cache.

Type

The type of cache. Type **EFI_INIT_CACHE_TYPE** is defined in "Related Definitions" below.

Description

This structure contains the cache level and type information.

Related Definitions

```
//*************************************************
// EFI_INIT_CACHE_TYPE
//*************************************************

// Valid cache types

typedef enum {
  EfiInitCacheDataOnly,
  EfiInitCacheInstrOnly,
  EfiInitCacheBoth,
  EfiInitCacheUnspecified
} EFI_INIT_CACHE_TYPE;
```

EFI_COMPUTING_UNIT_CPU_DISABLED_ERROR_DATA

Summary

This structure provides information about the disabled computing unit.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA    DataHeader;
  UINT32                  Cause;
  BOOLEAN                 SoftwareDisabled;
} EFI_COMPUTING_UNIT_CPU_DISABLED_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_COMPUTING_UNIT_CPU_DISABLED_ERROR_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Cause

The reason for disabling the processor. See "Related Definitions" below for defined values.

SoftwareDisabled

TRUE if the processor is disabled via software means such as not listing it in the ACPI tables. Such a processor will respond to Interprocessor Interrupts (IPIs). **FALSE** if the processor is hardware disabled, which means it is invisible to software and will not respond to IPIs.

Description

This structure provides details as to why and how the computing unit was disabled. The causes should cover the typical reasons a processor would be disabled. How the processor was disabled is important because there are distinct differences between hardware and software disabling.

Related Definitions

```
//*******************************************************************
// EFI_CPU_STATE_CHANGE_CAUSE
//*******************************************************************
typedef UINT32    EFI_CPU_STATE_CHANGE_CAUSE;

//
// The reason a processor was disabled
//
#define EFI_CPU_CAUSE_INTERNAL_ERROR          0x0001
#define EFI_CPU_CAUSE_THERMAL_ERROR           0x0002
#define EFI_CPU_CAUSE_SELFTEST_FAILURE        0x0004
#define EFI_CPU_CAUSE_PREBOOT_TIMEOUT         0x0008
#define EFI_CPU_CAUSE_FAILED_TO_START         0x0010
#define EFI_CPU_CAUSE_CONFIG_ERROR            0x0020
#define EFI_CPU_CAUSE_USER_SELECTION          0x0080
#define EFI_CPU_CAUSE_BY_ASSOCIATION          0x0100
#define EFI_CPU_CAUSE_UNSPECIFIED             0x8000
```

Table 82. Decription of EFI_CPU_STATE_CHANGE_CAUSE fields

EFI_CPU_CAUSE_INTERNAL_ERROR	The processor was disabled because it signaled an internal error (IERR).
EFI_CPU_CAUSE_THERMAL_ERROR	The processor was disabled because of a thermal error.
EFI_CPU_CAUSE_SELFTEST_FAILURE	The processor was disabled because it failed BIST.
EFI_CPU_CAUSE_PREBOOT_TIMEOUT	The processor started execution, but it timed out during a particular task and was therefore disabled.
EFI_CPU_CAUSE_FAILED_TO_START	The processor was disabled because it failed to start execution (FRB-3 timeout).
EFI_CPU_CAUSE_CONFIG_ERROR	The processor was disabled due to a configuration error.
EFI_CPU_CAUSE_USER_SELECTION	The processor state was changed due to user selection. Applicable to enabling and disabling of processors.
EFI_CPU_CAUSE_BY_ASSOCIATION	The processor state was changed due because it shared the state with another processor and the state of the other processor was changed.
EFI_CPU_CAUSE_UNSPECIFIED	The CPU state was changed due to unspecified reason. Applicable to enabling and disabling of processors.

Memory Subclass

EFI_MEMORY_EXTENDED_ERROR_DATA

Summary

This structure defines extended data describing a memory error.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                  DataHeader;
  EFI_MEMORY_ERROR_GRANULARITY          Granularity;
  EFI_MEMORY_ERROR_OPERATION            Operation;
  UINT32                                Syndrome;
  EFI_PHYSICAL_ADDRESS                  Address;
  UINTN                                 Resolution;
} EFI_MEMORY_EXTENDED_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_MEMORY_EXTENDED_ERROR_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Granularity

The error granularity type. Type **EFI_MEMORY_ERROR_GRANULARITY** is defined in "Related Definitions" below.

Operation

The operation that resulted in the error being detected. Type **EFI_MEMORY_ERROR_OPERATION** is defined in "Related Definitions" below.

Syndrome

The error syndrome, vendor-specific ECC syndrome, or CRC data associated with the error. If unknown, should be initialized to 0.

Address

The physical address of the error. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification.*

Resolution

The range, in bytes, within which the error address can be determined.

Description

This structure provides specific details about the memory error that was detected. It provides enough information so that consumers can identify the exact failure and provides enough information to enable corrective action if necessary.

Related Definitions

```
//*********************************************************
// EFI_MEMORY_ERROR_GRANULARITY
//*********************************************************
typedef UINT8 EFI_MEMORY_ERROR_GRANULARITY;

//
// Memory Error Granularities
//
#define EFI_MEMORY_ERROR_OTHER                  0x01
#define EFI_MEMORY_ERROR_UNKNOWN                0x02
#define EFI_MEMORY_ERROR_DEVICE                 0x03
#define EFI_MEMORY_ERROR_PARTITION              0x04


//*********************************************************
// EFI_MEMORY_ERROR_OPERATION
//*********************************************************
typedef UINT8 EFI_MEMORY_ERROR_OPERATION;

//
// Memory Error Operations
//
#define EFI_MEMORY_OPERATION_OTHER              0x01
#define EFI_MEMORY_OPERATION_UNKNOWN            0x02
#define EFI_MEMORY_OPERATION_READ               0x03
#define EFI_MEMORY_OPERATION_WRITE              0x04
#define EFI_MEMORY_OPERATION_PARTIAL_WRITE      0x05
```

EFI_STATUS_CODE_DIMM_NUMBER

Summary

This structure defines extended data describing a DIMM.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                          DataHeader;
  UINT16                                        Array;
  UINT16                                        Device;
} EFI_STATUS_CODE_DIMM_NUMBER;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_STATUS_CODE_DIMM_NUMBER) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Array

The memory array number.

Device

The device number within that *Array*.

Description

This extended data provides some context that consumers can use to locate a DIMM within the overall memory scheme. The *Array* and *Device* numbers may indicate a specific DIMM, or they may be populated with the group definitions in "Related Definitions" below.

Related Definitions

```
//
// Definitions to describe Group Operations
// Many memory init operations are essentially group
// operations.
//
#define  EFI_MULTIPLE_MEMORY_DEVICE_OPERATION      0xfffe
#define  EFI_ALL_MEMORY_DEVICE_OPERATION           0xffff
#define  EFI_MULTIPLE_MEMORY_ARRAY_OPERATION       0xfffe
#define  EFI_ALL_MEMORY_ARRAY_OPERATION            0xffff
```

Table 83. Definitions to describe Group Operations

EFI_MULTIPLE_MEMORY_DEVICE_OPERATION	A definition to describe that the operation is performed on multiple devices within the array.

EFI_ALL_MEMORY_DEVICE_OPERATION	A definition to describe that the operation is performed on all devices within the array.
EFI_MULTIPLE_MEMORY_ARRAY_OPERATION	A definition to describe that the operation is performed on multiple arrays.
EFI_ALL_MEMORY_ARRAY_OPERATION	A definition to describe that the operation is performed on all the arrays

EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA

Summary

This structure defines extended data describing memory modules that do not match.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                      DataHeader;
  EFI_STATUS_CODE_DIMM_NUMBER               Instance;
} EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Instance

The instance number of the memory module that does not match. See the definition for type **EFI_STATUS_CODE_DIMM_NUMBER**.

Description

This extended data may be used to convey the specifics of memory modules that do not match.

EFI_MEMORY_RANGE_EXTENDED_DATA

Summary

This structure defines extended data describing a memory range.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                   DataHeader;
  EFI_PHYSICAL_ADDRESS                   Start;
  EFI_PHYSICAL_ADDRESS                   Length;
} EFI_MEMORY_RANGE_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_MEMORY_RANGE_EXTENDED_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Start

The starting address of the memory range. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification.*

Length

The length in bytes of the memory range.

Description

This extended data may be used to convey the specifics of a memory range. Ranges are specified with a start address and a length.

6.7.2 User-Accessible Peripherals Class

The table below lists the subclasses defined in the User-Accessible Peripheral class. See the following subsection for their code definitions.

Table 84. Defined Subclasses: User-Accessible Peripheral Class

Subclass	Code Name
Unspecified	EFI_PERIPHERAL_UNSPECIFIED
Keyboard	EFI_PERIPHERAL_KEYBOARD
Mouse	EFI_PERIPHERAL_MOUSE
Local console	EFI_PERIPHERAL_LOCAL_CONSOLE
Remote console	EFI_PERIPHERAL_REMOTE_CONSOLE
Serial port	EFI_PERIPHERAL_SERIAL_PORT
Parallel port	EFI_PERIPHERAL_PARALLEL_PORT
Fixed media	EFI_PERIPHERAL_FIXED_MEDIA
Removable media	EFI_PERIPHERAL_REMOVABLE_MEDIA
Audio input	EFI_PERIPHERAL_AUDIO_INPUT
Audio output	EFI_PERIPHERAL_AUDIO_OUTPUT
LCD device	EFI_PERIPHERAL_LCD_DEVICE
Network device	EFI_PERIPHERAL_NETWORK
Docking Station	EFI_PERIPHERAL_DOCKING
0x0E–0x7F	Reserved for future use by this specification.
0x80–0xFF	Reserved for OEM use.

6.7.2.1 Subclass Definitions

Summary

Definitions for the User-Accessible Peripheral subclasses. See Subclasses in section 6.7.2 for descriptions of these subclasses.

Prototype

```
//
// Peripheral Subclass definitions.
// Values of 12-127 are reserved for future use by this
// specification.
// Values of 128-255 are reserved for OEM use.
//
#define EFI_PERIPHERAL_UNSPECIFIED \
  (EFI_PERIPHERAL | 0x00000000)
#define EFI_PERIPHERAL_KEYBOARD \
  (EFI_PERIPHERAL | 0x00010000)
#define EFI_PERIPHERAL_MOUSE \
  (EFI_PERIPHERAL | 0x00020000)
#define EFI_PERIPHERAL_LOCAL_CONSOLE \
  (EFI_PERIPHERAL | 0x00030000)
#define EFI_PERIPHERAL_REMOTE_CONSOLE \
  (EFI_PERIPHERAL | 0x00040000)
#define EFI_PERIPHERAL_SERIAL_PORT \
  (EFI_PERIPHERAL | 0x00050000)
#define EFI_PERIPHERAL_PARALLEL_PORT \
  (EFI_PERIPHERAL | 0x00060000)
#define EFI_PERIPHERAL_FIXED_MEDIA \
  (EFI_PERIPHERAL | 0x00070000)
#define EFI_PERIPHERAL_REMOVABLE_MEDIA \
  (EFI_PERIPHERAL | 0x00080000)
#define EFI_PERIPHERAL_AUDIO_INPUT \
  (EFI_PERIPHERAL | 0x00090000)
#define EFI_PERIPHERAL_AUDIO_OUTPUT \
  (EFI_PERIPHERAL | 0x000A0000)
#define EFI_PERIPHERAL_LCD_DEVICE \
  (EFI_PERIPHERAL | 0x000B0000)
#define EFI_PERIPHERAL_NETWORK \
  (EFI_PERIPHERAL | 0x000C0000)
#define EFI_PERIPHERAL_DOCKING \
  (EFI_PERIPHERAL | 0x000D0000)
```

6.7.2.2 Progress Code Definitions

Summary

Progress code definitions for the User-Accessible Peripheral class and all subclasses. See Progress Code Operations in section 6.7.2 for descriptions of these progress codes.

The following subclasses define additional subclass-specific progress code operations, which are included below:

- Keyboard
- Mouse

- Serial port

Prototype

```
//
// Peripheral Class Progress Code definitions.
// These are shared by all subclasses.
//
#define EFI_P_PC_INIT                      0x00000000
#define EFI_P_PC_RESET                     0x00000001
#define EFI_P_PC_DISABLE                   0x00000002
#define EFI_P_PC_PRESENCE_DETECT           0x00000003
#define EFI_P_PC_ENABLE                    0x00000004
#define EFI_P_PC_RECONFIG                  0x00000005
#define EFI_P_PC_DETECTED                  0x00000006
#define EFI_P_PC_REMOVED                   0x00000007


//
// Peripheral Class Unspecified Subclass Progress Code
// definitions.
//


//
// Peripheral Class Keyboard Subclass Progress Code definitions.
//
#define EFI_P_KEYBOARD_PC_CLEAR_BUFFER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_P_KEYBOARD_PC_SELF_TEST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)


//
// Peripheral Class Mouse Subclass Progress Code definitions.
//
#define EFI_P_MOUSE_PC_SELF_TEST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)


//
// Peripheral Class Local Console Subclass Progress Code
// definitions.
//


//
// Peripheral Class Remote Console Subclass Progress Code
// definitions.
//
```

```
//
// Peripheral Class Serial Port Subclass Progress Code
// definitions.
//
#define EFI_P_SERIAL_PORT_PC_CLEAR_BUFFER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)


//
// Peripheral Class Parallel Port Subclass Progress Code
// definitions.
//


//
// Peripheral Class Fixed Media Subclass Progress Code
// definitions.
//


//
// Peripheral Class Removable Media Subclass Progress Code
// definitions.
//


//
// Peripheral Class Audio Input Subclass Progress Code
// definitions.
//


//
// Peripheral Class Audio Output Subclass Progress Code
// definitions.
//


//
// Peripheral Class LCD Device Subclass Progress Code
// definitions.
//


//
```

```
// Peripheral Class Network Subclass Progress Code definitions.
//
```

6.7.2.3 Error Code Definitions

Summary

Error code definitions for the User-Accessible Peripheral class and all subclasses. See Error Code Operations in section 6.7.2 for descriptions of these error codes.

The following subclasses define additional subclass-specific error code operations, which are included below:

- Keyboard
- Mouse

Prototype

```
//
// Peripheral Class Error Code definitions.
// These are shared by all subclasses.
//
#define EFI_P_EC_NON_SPECIFIC          0x00000000
#define EFI_P_EC_DISABLED              0x00000001
#define EFI_P_EC_NOT_SUPPORTED         0x00000002
#define EFI_P_EC_NOT_DETECTED          0x00000003
#define EFI_P_EC_NOT_CONFIGURED        0x00000004
#define EFI_P_EC_INTERFACE_ERROR       0x00000005
#define EFI_P_EC_CONTROLLER_ERROR      0x00000006
#define EFI_P_EC_INPUT_ERROR           0x00000007
#define EFI_P_EC_OUTPUT_ERROR          0x00000008
#define EFI_P_EC_RESOURCE_CONFLICT \
  0x00000009


//
// Peripheral Class Unspecified Subclass Error Code definitions.
//


//
// Peripheral Class Keyboard Subclass Error Code definitions.
//
#define EFI_P_KEYBOARD_EC_LOCKED       \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000000)
#define EFI_P_KEYBOARD_EC_STUCK_KEY    \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000001)
#define EFI_P_KEYBOARD_EC_BUFFER_FULL  \
   (EFI_SUBCLASS_SPECIFIC | 0x00000002)


//
// Peripheral Class Mouse Subclass Error Code definitions.
//
#define EFI_P_MOUSE_EC_LOCKED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)


//
// Peripheral Class Local Console Subclass Error Code
// definitions.
//
```

```
//
// Peripheral Class Remote Console Subclass Error Code
// definitions.
//


//
// Peripheral Class Serial Port Subclass Error Code definitions.
//


//
// Peripheral Class Parallel Port Subclass Error Code
// definitions.
//


//
// Peripheral Class Fixed Media Subclass Error Code definitions.
//


//
// Peripheral Class Removable Media Subclass Error Code
// definitions.
//


//
// Peripheral Class Audio Input Subclass Error Code definitions.
//


//
// Peripheral Class Audio Output Subclass Error Code
// definitions.
//


//
// Peripheral Class LCD Device Subclass Error Code definitions.
//


//
// Peripheral Class Network Subclass Error Code definitions.
//
```

6.7.2.4 Extended Data Formats

The User-Accessible Peripheral class uses the following extended error data definitions:

- **EFI_DEVICE_PATH_EXTENDED_DATA**
- **EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA**

See section 6.6.4 for definitions.

6.7.3 I/O Bus Class

The table below lists the subclasses defined in the I/O Bus class. See Subclass Definitions for their code definitions.

Table 85. Defined Subclasses: I/O Bus Class

Subclass	Code Name
Unspecified	EFI_IO_BUS_UNSPECIFIED
PCI	EFI_IO_BUS_PCI
USB	EFI_IO_BUS_USB
InfiniBand* architecture	EFI_IO_BUS_IBA
AGP	EFI_IO_BUS_AGP
PC card	EFI_IO_BUS_PC_CARD
Low pin count (LPC)	EFI_IO_BUS_LPC
SCSI	EFI_IO_BUS_SCSI
ATA/ATAPI/SATA	EFI_IO_BUS_ATA_ATAPI
Fibre Channel	EFI_IO_BUS_FC
IP network	EFI_IO_BUS_IP_NETWORK
SMBus	EFI_IO_BUS_SMBUS
I2C	EFI_IO_BUS_I2C
0x0D–0x7F	Reserved for future use by this specification.
0x80–0xFF	Reserved for OEM use.

6.7.3.1 Subclass Definitions

Summary

Definitions for the I/O Bus subclasses. See Subclasses in section 6.7.3 for descriptions of these subclasses.

Prototype

```
//
// IO Bus Subclass definitions.
// Values of 14-127 are reserved for future use by this
// specification.
// Values of 128-255 are reserved for OEM use.
//
#define EFI_IO_BUS_UNSPECIFIED \
  (EFI_IO_BUS | 0x00000000)
#define EFI_IO_BUS_PCI \
  (EFI_IO_BUS | 0x00010000)
#define EFI_IO_BUS_USB \
  (EFI_IO_BUS | 0x00020000)
#define EFI_IO_BUS_IBA \
  (EFI_IO_BUS | 0x00030000)
#define EFI_IO_BUS_AGP \
  (EFI_IO_BUS | 0x00040000)
#define EFI_IO_BUS_PC_CARD \
  (EFI_IO_BUS | 0x00050000)
#define EFI_IO_BUS_LPC \
  (EFI_IO_BUS | 0x00060000)
#define EFI_IO_BUS_SCSI \
  (EFI_IO_BUS | 0x00070000)
#define EFI_IO_BUS_ATA_ATAPI \
  (EFI_IO_BUS | 0x00080000)
#define EFI_IO_BUS_FC \
  (EFI_IO_BUS | 0x00090000)
#define EFI_IO_BUS_IP_NETWORK \
  (EFI_IO_BUS | 0x000A0000)
#define EFI_IO_BUS_SMBUS \
  (EFI_IO_BUS | 0x000B0000)
#define EFI_IO_BUS_I2C \
  (EFI_IO_BUS | 0x000C0000)
```

6.7.3.2 Progress Code Definitions

Summary

Progress code definitions for the I/O Bus class and all subclasses. See Progress Code Operations in section 6.7.2 for descriptions of these progress codes.

The following subclasses define additional subclass-specific progress code operations, which are included below:

- PCI

Prototype

```
//
// IO Bus Class Progress Code definitions.
// These are shared by all subclasses.
//
typedef struct _EFI_SIO_PROTOCOL EFI_SIO_PROTOCOL;

#define EFI_IOB_PC_INIT                 0x00000000
#define EFI_IOB_PC_RESET                0x00000001
#define EFI_IOB_PC_DISABLE              0x00000002
#define EFI_IOB_PC_DETECT               0x00000003
#define EFI_IOB_PC_ENABLE               0x00000004
#define EFI_IOB_PC_RECONFIG             0x00000005
#define EFI_IOB_PC_HOTPLUG              0x00000006


//
// IO Bus Class Unspecified Subclass Progress Code definitions.
//


//
// IO Bus Class PCI Subclass Progress Code definitions.
//
#define EFI_IOB_PCI_BUS_ENUM \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_IOB_PCI_RES_ALLOC \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#defineEFI_IOB_PCI_HPC_INIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)


//
// IO Bus Class USB Subclass Progress Code definitions.
//


//
// IO Bus Class IBA Subclass Progress Code definitions.
//


//
// IO Bus Class AGP Subclass Progress Code definitions.
//
```

```
//
// IO Bus Class PC Card Subclass Progress Code definitions.
//


//
// IO Bus Class LPC Subclass Progress Code definitions.
//


//
// IO Bus Class SCSI Subclass Progress Code definitions.
//


//
// IO Bus Class ATA/ATAPI Subclass Progress Code definitions.
//
#define EFI_IOB_ATA_BUS_SMART_ENABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_IOB_ATA_BUS_SMART_DISABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_IOB_ATA_BUS_SMART_OVERTHRESHOLD \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_IOB_ATA_BUS_SMART_UNDERTHRESHOLD \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
//
// IO Bus Class FC Subclass Progress Code definitions.
//


//
// IO Bus Class IP Network Subclass Progress Code definitions.
//


//
// IO Bus Class SMBUS Subclass Progress Code definitions.
//


//
// IO Bus Class I2C Subclass Progress Code definitions.
//
```

6.7.3.3 Error Code Definitions

Summary

Error code definitions for the I/O Bus class and all subclasses. See Error Code Operations in section 6.7.2 for descriptions of these error codes.

The following subclasses define additional subclass-specific error code operations, which are included below:

- PCI

Prototype

```
// IO Bus Class Error Code definitions.
// These are shared by all subclasses.
//
#define EFI_IOB_EC_NON_SPECIFIC        0x00000000
#define EFI_IOB_EC_DISABLED            0x00000001
#define EFI_IOB_EC_NOT_SUPPORTED       0x00000002
#define EFI_IOB_EC_NOT_DETECTED        0x00000003
#define EFI_IOB_EC_NOT_CONFIGURED      0x00000004
#define EFI_IOB_EC_INTERFACE_ERROR     0x00000005
#define EFI_IOB_EC_CONTROLLER_ERROR    0x00000006
#define EFI_IOB_EC_READ_ERROR          0x00000007
#define EFI_IOB_EC_WRITE_ERROR         0x00000008
#define EFI_IOB_EC_RESOURCE_CONFLICT   0x00000009


//
// IO Bus Class Unspecified Subclass Error Code definitions.
//


//
// IO Bus Class PCI Subclass Error Code definitions.
//
#define EFI_IOB_PCI_EC_PERR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_IOB_PCI_EC_SERR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)


//
// IO Bus Class USB Subclass Error Code definitions.
//


//
// IO Bus Class IBA Subclass Error Code definitions.
//


//
// IO Bus Class AGP Subclass Error Code definitions.
//


//
// IO Bus Class PC Card Subclass Error Code definitions.
```

```
//


//
// IO Bus Class LPC Subclass Error Code definitions.
//


//
// IO Bus Class SCSI Subclass Error Code definitions.
//


//
// IO Bus Class ATA/ATAPI Subclass Error Code definitions.
//
#define EFI_IOB_ATA_BUS_SMART_NOTSUPPORTED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_IOB_ATA_BUS_SMART_DISABLED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)


//
// IO Bus Class FC Subclass Error Code definitions.
//


//
// IO Bus Class IP Network Subclass Error Code definitions.
//


//
// IO Bus Class SMBUS Subclass Error Code definitions.
//


//
// IO Bus Class I2C Subclass Error Code definitions.
//
```

6.7.3.4 Extended Data Formats

The I/O Bus class uses the following extended data definitions:

- **EFI_DEVICE_PATH_EXTENDED_DATA**
- **EFI_DEVICE_HANDLE_EXTENDED_DATA**
- **EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA**

See section 6.6.4 for definitions.

6.7.4 Software Classes

The table below lists the subclasses defined in the Host Software class. See Subclass Definitions for their code definitions.

Table 86. Defined Subclasses: Host Software Class

Subclass	Code Name
Unspecified	EFI_SOFTWARE_UNSPECIFIED
Security (SEC)	EFI_SOFTWARE_SEC
PEI Foundation	EFI_SOFTWARE_PEI_CORE
PEI module	EFI_SOFTWARE_PEI_MODULE
DXE Foundation	EFI_SOFTWARE_DXE_CORE
DXE Boot Service driver	EFI_SOFTWARE_DXE_BS_DRIVER
DXE Runtime Service driver	EFI_SOFTWARE_DXE_RT_DRIVER
SMM driver	EFI_SOFTWARE_SMM_DRIVER
EFI application	EFI_SOFTWARE_EFI_APPLICATION
OS loader	EFI_SOFTWARE_EFI_OS_LOADER
Runtime (RT)	EFI_SOFTWARE_EFI_RT
EBC exception	EFI_SOFTWARE_EBC_EXCEPTION
IA-32 exception	EFI_SOFTWARE_IA32_EXCEPTION
Itanium® processor family exception	EFI_SOFTWARE_IPF_EXCEPTION
PEI Services	EFI_SOFTWARE_PEI_SERVICE
EFI Boot Service	EFI_SOFTWARE_EFI_BOOT_SERVICE
EFI Runtime Service	EFI_SOFTWARE_EFI_RUNTIME_SERVICE
DXE Service	EFI_SOFTWARE_EFI_DXE_SERVICE
0x13–0x7F	Reserved for future use by this specification.
0x80–0xFF	Reserved for OEM use.
x64 software exception	EFI_SOFTWARE_X64_EXCEPTION
ARM AArch32 software exception	EFI_SOFTWARE_ARM_EXCEPTION
ARM AArch64 software exception	EFI_SOFTWARE_AARCH64_EXCEPTION
RISC-V software exception	EFI_SOFTWARE_RISCV_EXCEPTION

6.7.4.1 Subclass Definitions

Summary

Definitions for the Host Software subclasses. See Subclasses in section 6.5.1 for descriptions of these subclasses.

Prototype

```
//
// Software Subclass definitions.
// Values of 14-127 are reserved for future use by this
// specification.
// Values of 128-255 are reserved for OEM use.
//
#define EFI_SOFTWARE_UNSPECIFIED \
  (EFI_SOFTWARE | 0x00000000)
#define EFI_SOFTWARE_SEC \
  (EFI_SOFTWARE | 0x00010000)
#define EFI_SOFTWARE_PEI_CORE \
  (EFI_SOFTWARE | 0x00020000)
#define EFI_SOFTWARE_PEI_MODULE \
  (EFI_SOFTWARE | 0x00030000)
#define EFI_SOFTWARE_DXE_CORE \
  (EFI_SOFTWARE | 0x00040000)
#define EFI_SOFTWARE_DXE_BS_DRIVER \
  (EFI_SOFTWARE | 0x00050000)
#define EFI_SOFTWARE_DXE_RT_DRIVER \
  (EFI_SOFTWARE | 0x00060000)
#define EFI_SOFTWARE_SMM_DRIVER \
  (EFI_SOFTWARE | 0x00070000)
#define EFI_SOFTWARE_EFI_APPLICATION \
  (EFI_SOFTWARE | 0x00080000)
#define EFI_SOFTWARE_EFI_OS_LOADER\
  (EFI_SOFTWARE | 0x00090000)
#define EFI_SOFTWARE_RT \
  (EFI_SOFTWARE | 0x000A0000)
#define EFI_SOFTWARE_AL \
  (EFI_SOFTWARE | 0x000B0000)
#define EFI_SOFTWARE_EBC_EXCEPTION \
  (EFI_SOFTWARE | 0x000C0000)
#define EFI_SOFTWARE_IA32_EXCEPTION \
  (EFI_SOFTWARE | 0x000D0000)
#define EFI_SOFTWARE_IPF_EXCEPTION \
  (EFI_SOFTWARE | 0x000E0000)
#define EFI_SOFTWARE_PEI_SERVICE \
  (EFI_SOFTWARE | 0x000F0000
#define EFI_SOFTWARE_EFI_BOOT_SERVICE \
  (EFI_SOFTWARE | 0x00100000)
#define EFI_SOFTWARE_EFI_RUNTIME_SERVICE \
  (EFI_SOFTWARE | 0x00110000)
#define EFI_SOFTWARE_EFI_DXE_SERVICE \
  (EFI_SOFTWARE | 0x00120000)
#define EFI_SOFTWARE_X64_EXCEPTION \
  (EFI_SOFTWARE | 0x00130000)
```

```
#define EFI_SOFTWARE_ARM_EXCEPTION \
  (EFI_SOFTWARE | 0x00140000)
#define EFI_SOFTWARE_AARCH64_EXCEPTION \
  (EFI_SOFTWARE | 0x00150000)
```

6.7.4.2 Progress Code Definitions

Summary

Progress code definitions for the Host Software class and all subclasses. See Progress Code Operations in section 6.5.1 for descriptions of these progress codes.

The following subclasses define additional subclass-specific progress code operations, which are included below:

- SEC
- PEI Foundation
- PEI Module
- DXE Foundation
- DXE Boot Service Driver
- Runtime (RT)
- PEI Services
- Boot Services
- Runtime Services
- DXE Services

Prototype

```
//
// Software Class Progress Code definitions.
// These are shared by all subclasses.
//
#define EFI_SW_PC_INIT \
  0x00000000
#define EFI_SW_PC_LOAD \
  0x00000001
#define EFI_SW_PC_INIT_BEGIN \
  0x00000002
#define EFI_SW_PC_INIT_END \
  0x00000003
#define EFI_SW_PC_AUTHENTICATE_BEGIN \
  0x00000004
#define EFI_SW_PC_AUTHENTICATE_END \
  0x00000005
#define EFI_SW_PC_INPUT_WAIT \
  0x00000006
#define EFI_SW_PC_USER_SETUP \
  0x00000007


//
// Software Class Unspecified Subclass Progress Code
// definitions.
//


//
// Software Class SEC Subclass Progress Code definitions.
//
#define EFI_SW_SEC_PC_ENTRY_POINT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_SEC_PC_HANDOFF_TO_NEXT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)


//
// Software Class PEI Foundation Subclass Progress Code
// definitions.
//
#define EFI_SW_PEI_CORE_PC_ENTRY_POINT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_PEI_CORE_PC_HANDOFF_TO_NEXT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_PEI_CORE_PC_RETURN_TO_LAST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)


//
// Software Class PEI Module Subclass Progress Code definitions.
//
#define EFI_SW_PEI_PC_RECOVERY_BEGIN \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000000)
#define EFI_SW_PEI_PC_CAPSULE_LOAD   \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000001)
#define EFI_SW_PEI_PC_CAPSULE_START  \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000001)
#define EFI_SW_PEI_PC_RECOVERY_USER  \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000003)
#define EFI_SW_PEI_PC_RECOVERY_AUTO  \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000004)
#define EFI_SW_PEI_PC_S3_BOOT_SCRIPT \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000005)
#define EFI_SW_PEI_PC_OS_WAKE        \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000006
#define EFI_SW_PEI_PC_S3_STARTED     \
   (EFI_SUBCLASS_SPECIFIC | 0x00000007)


//
// Software Class DXE Foundation Subclass Progress Code
// definitions.
//
#define EFI_SW_DXE_CORE_PC_ENTRY_POINT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_DXE_CORE_PC_HANDOFF_TO_NEXT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_DXE_CORE_PC_RETURN_TO_LAST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_DXE_CORE_PC_START_DRIVER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_DXE_CORE_PC_ARCH_READY \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)

//
// Software Class DXE BS Driver Subclass Progress Code
// definitions.
//
#define EFI_SW_DXE_BS_PC_LEGACY_OPROM_INIT        \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000000)
#define EFI_SW_DXE_BS_PC_READY_TO_BOOT_EVENT      \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000001)
```

```
#define EFI_SW_DXE_BS_PC_LEGACY_BOOT_EVENT        \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000002)
#define EFI_SW_DXE_BS_PC_EXIT_BOOT_SERVICES_EVENT \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000003)
#define EFI_SW_DXE_BS_PC_VIRTUAL_ADDRESS_CHANGE_EVENT \
  (EFI_SUBCLASS_SPECIFIC  | 0x00000004)
#define EFI_SW_DXE_BS_PC_VARIABLE_SERVICES_INIT   \
   (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_DXE_BS_PC_VARIABLE_RECLAIM         \
   (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_DXE_BS_PC_ATTEMPT_BOOT_ORDER_EVENT \
(EFI_SUBCLASS_SPECIFIC    | 0x00000007)
#define EFI_SW_DXE_BS_PC_CONFIG_RESET             \
   (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_DXE_BS_PC_CSM_INIT                 \
   (EFI_SUBCLASS_SPECIFIC | 0x00000009)



//
// Software Class DXE RT Driver Subclass Progress Code
// definitions.
//


//
// Software Class SMM Driver Subclass Progress Code definitions.
//


//
// Software Class EFI Application Subclass Progress Code
// definitions.
//


//
// Software Class EFI OS Loader Subclass Progress Code
// definitions.
//


//
// Software Class EFI RT Subclass Progress Code definitions.
//
#define EFI_SW_RT_PC_ENTRY_POINT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_RT_PC_HANDOFF_TO_NEXT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_RT_PC_RETURN_TO_LAST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)

//
// Software Class X64 Exception Subclass Progress Code
// definitions.
//

//
```

```
// Software Class ARM Exception Subclass Progress Code
// definitions.


//
// Software Class EBC Exception Subclass Progress Code
// definitions.
//


//
// Software Class IA32 Exception Subclass Progress Code
// definitions.
//


//
// Software Class IPF Exception Subclass Progress Code
// definitions.
//


//
// Software Class PEI Services Subclass Progress Code
definitions.
//


#define EFI_SW_PS_PC_INSTALL_PPI \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_PS_PC_REINSTALL_PPI \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_PS_PC_LOCATE_PPI \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_PS_PC_NOTIFY_PPI \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_PS_PC_GET_BOOT_MODE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_PS_PC_SET_BOOT_MODE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_PS_PC_GET_HOB_LIST \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_PS_PC_CREATE_HOB \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_SW_PS_PC_FFS_FIND_NEXT_VOLUME \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_PS_PC_FFS_FIND_NEXT_FILE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_SW_PS_PC_FFS_FIND_SECTION_DATA\
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
#define EFI_SW_PS_PC_INSTALL_PEI_MEMORY \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000B)
#define EFI_SW_PS_PC_ALLOCATE_PAGES \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000C)
#define EFI_SW_PS_PC_ALLOCATE_POOL \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000D)
#define EFI_SW_PS_PC_COPY_MEM \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000E)
#define EFI_SW_PS_PC_SET_MEM \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000F)
#define EFI_SW_PS_PC_RESET_SYSTEM  \
  (EFI_SUBCLASS_SPECIFIC | 0x00000010)
#define EFI_SW_PS_PC_FFS_FIND_FILE_BY_NAME \
  (EFI_SUBCLASS_SPECIFIC|0x00000013)
#define EFI_SW_PS_PC_FFS_GET_FILE_INFO \
  (EFI_SUBCLASS_SPECIFIC|0x00000014)
#define EFI_SW_PS_PC_FFS_GET_VOLUME_INFO \
  (EFI_SUBCLASS_SPECIFIC|0x00000015)
#define EFI_SW_PS_PC_FFS_REGISTER_FOR_SHADOW \
  (EFI_SUBCLASS_SPECIFIC|0x00000016)
//
// Software Class EFI Boot Services Subclass Progress Code
// definitions.
//
#define EFI_SW_BS_PC_RAISE_TPL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_BS_PC_RESTORE_TPL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_BS_PC_ALLOCATE_PAGES \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_BS_PC_FREE_PAGES \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_BS_PC_GET_MEMORY_MAP \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_BS_PC_ALLOCATE_POOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_BS_PC_FREE_POOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_BS_PC_CREATE_EVENT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_SW_BS_PC_SET_TIMER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_BS_PC_WAIT_FOR_EVENT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
```

```
#define EFI_SW_BS_PC_SIGNAL_EVENT \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
#define EFI_SW_BS_PC_CLOSE_EVENT \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000B)
#define EFI_SW_BS_PC_CHECK_EVENT \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000C)
#define EFI_SW_BS_PC_INSTALL_PROTOCOL_INTERFACE\
  (EFI_SUBCLASS_SPECIFIC | 0x0000000D)
#define EFI_SW_BS_PC_REINSTALL_PROTOCOL_INTERFACE\
  (EFI_SUBCLASS_SPECIFIC | 0x0000000E)
#define EFI_SW_BS_PC_UNINSTALL_PROTOCOL_INTERFACE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000F)
#define EFI_SW_BS_PC_HANDLE_PROTOCOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000010)
#define EFI_SW_BS_PC_PC_HANDLE_PROTOCOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000011)
#define EFI_SW_BS_PC_REGISTER_PROTOCOL_NOTIFY\
  (EFI_SUBCLASS_SPECIFIC | 0x00000012)
#define EFI_SW_BS_PC_LOCATE_HANDLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000013)
#define EFI_SW_BS_PC_INSTALL_CONFIGURATION_TABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000014)
#define EFI_SW_BS_PC_LOAD_IMAGE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000015)
#define EFI_SW_BS_PC_START_IMAGE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000016)
#define EFI_SW_BS_PC_EXIT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000017)
#define EFI_SW_BS_PC_UNLOAD_IMAGE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000018)
#define EFI_SW_BS_PC_EXIT_BOOT_SERVICES \
  (EFI_SUBCLASS_SPECIFIC | 0x00000019)
#define EFI_SW_BS_PC_GET_NEXT_MONOTONIC_COUNT \
  (EFI_SUBCLASS_SPECIFIC | 0x0000001A)
#define EFI_SW_BS_PC_STALL \
  (EFI_SUBCLASS_SPECIFIC | 0x0000001B)
#define EFI_SW_BS_PC_SET_WATCHDOG_TIMER \
  (EFI_SUBCLASS_SPECIFIC | 0x0000001C)
#define EFI_SW_BS_PC_CONNECT_CONTROLLER \
  (EFI_SUBCLASS_SPECIFIC | 0x0000001D)
#define EFI_SW_BS_PC_DISCONNECT_CONTROLLER\
  (EFI_SUBCLASS_SPECIFIC | 0x0000001E)
#define EFI_SW_BS_PC_OPEN_PROTOCOL \
  (EFI_SUBCLASS_SPECIFIC | 0x0000001F)
#define EFI_SW_BS_PC_CLOSE_PROTOCOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000020)
#define EFI_SW_BS_PC_OPEN_PROTOCOL_INFORMATION\
  (EFI_SUBCLASS_SPECIFIC | 0x00000021)
#define EFI_SW_BS_PC_PROTOCOLS_PER_HANDLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000022)
#define EFI_SW_BS_PC_LOCATE_HANDLE_BUFFER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000023)
#define EFI_SW_BS_PC_LOCATE_PROTOCOL \
  (EFI_SUBCLASS_SPECIFIC | 0x00000024)
#define EFI_SW_BS_PC_INSTALL_MULTIPLE_INTERFACES \
  (EFI_SUBCLASS_SPECIFIC | 0x00000025)
#define EFI_SW_BS_PC_UNINSTALL_MULTIPLE_INTERFACES \
  (EFI_SUBCLASS_SPECIFIC | 0x00000026)
#define EFI_SW_BS_PC_CALCULATE_CRC_32 \
  (EFI_SUBCLASS_SPECIFIC | 0x00000027)
#define EFI_SW_BS_PC_COPY_MEM \
  (EFI_SUBCLASS_SPECIFIC | 0x00000028)
#define EFI_SW_BS_PC_SET_MEM \
  (EFI_SUBCLASS_SPECIFIC | 0x00000029)
#define EFI_SW_BS_PC_CREATE_EVENT_EX \
  (EFI_SUBCLASS_SPECIFIC|0x0000002a)


//
// Software Class EFI Runtime Services Subclass Progress Code
// definitions.
//
#define EFI_SW_RS_PC_GET_TIME \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_RS_PC_SET_TIME \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_RS_PC_GET_WAKEUP_TIME \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_RS_PC_SET_WAKEUP_TIME \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_RS_PC_SET_VIRTUAL_ADDRESS_MAP\
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_RS_PC_CONVERT_POINTER \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_RS_PC_GET_VARIABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_RS_PC_GET_NEXT_VARIABLE_NAME\
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_SW_RS_PC_SET_VARIABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_RS_PC_GET_NEXT_HIGH_MONOTONIC_COUNT\
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_SW_RS_PC_RESET_SYSTEM \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
```

```
#define EFI_SW_RS_PC_UPDATE_CAPSULE \
  (EFI_SUBCLASS_SPECIFIC| 0x0000000B)
#define EFI_SW_RS_PC_QUERY_CAPSULE_CAPABILITIES \
  (EFI_SUBCLASS_SPECIFIC| 0x0000000C)
#define EFI_SW_RS_PC_QUERY_VARIABLE_INFO \
  (EFI_SUBCLASS_SPECIFIC| 0x0000000D)


//
// Software Class EFI DXE Services Subclass Progress Code
// definitions
//
#define EFI_SW_DS_PC_ADD_MEMORY_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_DS_PC_ALLOCATE_MEMORY_SPACE\
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_DS_PC_FREE_MEMORY_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_DS_PC_REMOVE_MEMORY_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_DS_PC_GET_MEMORY_SPACE_DESCRIPTOR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_DS_PC_SET_MEMORY_SPACE_ATTRIBUTES\
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_DS_PC_GET_MEMORY_SPACE_MAP \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_DS_PC_ADD_IO_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_SW_DS_PC_ALLOCATE_IO_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_DS_PC_FREE_IO_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000009)
#define EFI_SW_DS_PC_REMOVE_IO_SPACE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000A)
#define EFI_SW_DS_PC_GET_IO_SPACE_DESCRIPTOR \
    (EFI_SUBCLASS_SPECIFIC | 0x0000000B)
#define EFI_SW_DS_PC_GET_IO_SPACE_MAP \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000C)
#define EFI_SW_DS_PC_DISPATCH\
  (EFI_SUBCLASS_SPECIFIC | 0x0000000D)
#define EFI_SW_DS_PC_SCHEDULE \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000E)
#define EFI_SW_DS_PC_TRUST \
  (EFI_SUBCLASS_SPECIFIC | 0x0000000F)
#define EFI_SW_DS_PC_PROCESS_FIRMWARE_VOLUME\
  (EFI_SUBCLASS_SPECIFIC | 0x00000010)
```

6.7.4.3 Error Code Definitions

Summary

Error code definitions for the Host Software class and all subclasses. See Error Code Operations in section 6.5.1 for descriptions of these error codes.

The following subclasses define additional subclass-specific error code operations, which are included below:

- PEI Foundation
- PEIM
- DxeBootServiceDriver
- EFI Byte Code (EBC) exception
- IA-32 exception
- Itanium® processor family exception
- ARM AArch32 and AArch64 exceptions

Prototype

```
//
// Software Class Error Code definitions.
// These are shared by all subclasses.
//
#define EFI_SW_EC_NON_SPECIFIC             0x00000000
#define EFI_SW_EC_LOAD_ERROR               0x00000001
#define EFI_SW_EC_INVALID_PARAMETER        0x00000002
#define EFI_SW_EC_UNSUPPORTED              0x00000003
#define EFI_SW_EC_INVALID_BUFFER           0x00000004
#define EFI_SW_EC_OUT_OF_RESOURCES         0x00000005
#define EFI_SW_EC_ABORTED                  0x00000006
#define EFI_SW_EC_ILLEGAL_SOFTWARE_STATE   0x00000007
#define EFI_SW_EC_ILLEGAL_HARDWARE_STATE   0x00000008
#define EFI_SW_EC_START_ERROR              0x00000009
#define EFI_SW_EC_BAD_DATE_TIME            0x0000000A
#define EFI_SW_EC_CFG_INVALID              0x0000000B
#define EFI_SW_EC_CFG_CLR_REQUEST          0x0000000C
#define EFI_SW_EC_CFG_DEFAULT              0x0000000D
#define EFI_SW_EC_PWD_INVALID              0x0000000E
#define EFI_SW_EC_PWD_CLR_REQUEST          0x0000000F
#define EFI_SW_EC_PWD_CLEARED              0x00000010
#define EFI_SW_EC_EVENT_LOG_FULL           0x00000011
#define EFI_SW_EC_ WRITE_PROTECTED         0x00000012
#define EFI_SW_EC_ FV_CORRUPTED            0x00000013


//
// Software Class Unspecified Subclass Error Code definitions.
//


//
// Software Class SEC Subclass Error Code definitions.
//


//
// Software Class PEI Foundation Subclass Error Code
// definitions.
//
#define EFI_SW_PEI_CORE_EC_DXE_CORRUPT \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_PEI_CORE_EC_DXEIPL_NOT_FOUND \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_PEI_CORE_EC_MEMORY_NOT_INSTALLED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
```

```
//
// Software Class PEI Module Subclass Error Code definitions.
//
#define EFI_SW_PEI_EC_NO_RECOVERY_CAPSULE        \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_PEI_EC_INVALID_CAPSULE_DESCRIPTOR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_PEI_EC_S3_RESUME_PPI_NOT_FOUND    \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_PEI_EC_S3_BOOT_SCRIPT_ERROR       \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_PEI_EC_S3_OS_WAKE_ERROR           \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_PEI_EC_S3_RESUME_FAILED          \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_PEI_EC_RECOVERY_PPI_NOT_FOUND    \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)
#define EFI_SW_PEI_EC_RECOVERY_FAILED (         \
  EFI_SUBCLASS_SPECIFIC | 0x00000007)
#define EFI_SW_PEI_EC_S3_RESUME_ERROR           \
   (EFI_SUBCLASS_SPECIFIC | 0x00000008)
#define EFI_SW_PEI_EC_INVALID_CAPSULE           \
   (EFI_SUBCLASS_SPECIFIC | 0x00000009)


//
// Software Class DXE Foundation Subclass Error Code
// definitions.
//
#define EFI_SW_DXE_CORE_EC_NO_ARCH \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)

//
// Software Class DXE Boot Service Driver Subclass Error Code
// definitions.
//
#define EFI_SW_DXE_BS_EC_LEGACY_OPROM_NO_SPACE\
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_DXE_BS_EC_INVALID_PASSWORD \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR \
  (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_DXE_BS_EC_BOOT_OPTION_FAILED \
  (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_DXE_BS_EC_INVALID_IDE_PASSWORD \
  (EFI_SUBCLASS_SPECIFIC | 0x00000004)
```

```
//
// Software Class DXE Runtime Service Driver Subclass Error Code
// definitions.
//


//
// Software Class SMM Driver Subclass Error Code definitions.
//


//
// Software Class EFI Application Subclass Error Code
// definitions.
//


//
// Software Class EFI OS Loader Subclass Error Code definitions.
//


//
// Software Class EFI RT Subclass Error Code definitions.
//


//
// Software Class EBC Exception Subclass Error Code definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI specification.
//
#define EFI_SW_EC_EBC_UNDEFINED \
  0x00000000
#define EFI_SW_EC_EBC_DIVIDE_ERROR \
  EXCEPT_EBC_DIVIDE_ERROR
#define EFI_SW_EC_EBC_DEBUG \
  EXCEPT_EBC_DEBUG
#define EFI_SW_EC_EBC_DEBUG \
  EXCEPT_EBC_DEBUG
#define EFI_SW_EC_EBC_BREAKPOINT \
  EXCEPT_EBC_BREAKPOINT
#define EFI_SW_EC_EBC_OVERFLOW \
  EXCEPT_EBC_OVERFLOW
#define EFI_SW_EC_EBC_INVALID_OPCODE \
  EXCEPT_EBC_INVALID_OPCODE
```

```
#define EFI_SW_EC_EBC_STACK_FAULT \
  EXCEPT_EBC_STACK_FAULT
#define EFI_SW_EC_EBC_ALIGNMENT_CHECK \
  EXCEPT_EBC_ALIGNMENT_CHECK
#define EFI_SW_EC_EBC_INSTRUCTION_ENCODING \
  EXCEPT_EBC_INSTRUCTION_ENCODING
#define EFI_SW_EC_EBC_BAD_BREAK \
  EXCEPT_EBC_BAD_BREAK
#define EFI_SW_EC_EBC_STEP                      EXCEPT_EBC_STEP


//
// Software Class IA32 Exception Subclass Error Code
// definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI specification.
//
#define EFI_SW_EC_IA32_DIVIDE_ERROR \
  EXCEPT_IA32_DIVIDE_ERROR
#define EFI_SW_EC_IA32_DEBUG \
  EXCEPT_IA32_DEBUG
#define EFI_SW_EC_IA32_NMI                      EXCEPT_IA32_NMI
#define EFI_SW_EC_IA32_BREAKPOINT \
  EXCEPT_IA32_BREAKPOINT
#define EFI_SW_EC_IA32_OVERFLOW \
  EXCEPT_IA32_OVERFLOW
#define EFI_SW_EC_IA32_BOUND \
  EXCEPT_IA32_BOUND
#define EFI_SW_EC_IA32_INVALID_OPCODE \
  EXCEPT_IA32_INVALID_OPCODE
#define EFI_SW_EC_IA32_DOUBLE_FAULT \
  EXCEPT_IA32_DOUBLE_FAULT
#define EFI_SW_EC_IA32_INVALID_TSS \
  EXCEPT_IA32_INVALID_TSS
#define EFI_SW_EC_IA32_SEG_NOT_PRESENT \
  EXCEPT_IA32_SEG_NOT_PRESENT
#define EFI_SW_EC_IA32_STACK_FAULT \
  EXCEPT_IA32_STACK_FAULT
#define EFI_SW_EC_IA32_GP_FAULT \
  EXCEPT_IA32_GP_FAULT
#define EFI_SW_EC_IA32_PAGE_FAULT \
  EXCEPT_IA32_PAGE_FAULT
#define EFI_SW_EC_IA32_FP_ERROR \
  EXCEPT_IA32_FP_ERROR
#define EFI_SW_EC_IA32_ALIGNMENT_CHECK \
  EXCEPT_IA32_ALIGNMENT_CHECK
#define EFI_SW_EC_IA32_MACHINE_CHECK \
  EXCEPT_IA32_MACHINE_CHECK
#define EFI_SW_EC_IA32_SIMD                     EXCEPT_IA32_SIMD


//
// Software Class IPF Exception Subclass Error Code definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI specification.
//
#define EFI_SW_EC_IPF_ALT_DTLB \
  EXCEPT_IPF_ALT_DTLB
#define EFI_SW_EC_IPF_DNESTED_TLB \
  EXCEPT_IPF_DNESTED_TLB
#define EFI_SW_EC_IPF_BREAKPOINT \
  EXCEPT_IPF_BREAKPOINT
#define EFI_SW_EC_IPF_EXTERNAL_INTERRUPT \
  EXCEPT_IPF_EXTERNAL_INTERRUPT
#define EFI_SW_EC_IPF_GEN_EXCEPT \
  EXCEPT_IPF_GEN_EXCEPT
#define EFI_SW_EC_IPF_NAT_CONSUMPTION \
  EXCEPT_IPF_NAT_CONSUMPTION
#define EFI_SW_EC_IPF_DEBUG_EXCEPT \
  EXCEPT_IPF_DEBUG_EXCEPT
#define EFI_SW_EC_IPF_UNALIGNED_ACCESS \
  EXCEPT_IPF_UNALIGNED_ACCESS
#define EFI_SW_EC_IPF_FP_FAULT \
  EXCEPT_IPF_FP_FAULT
#define EFI_SW_EC_IPF_FP_TRAP \
  EXCEPT_IPF_FP_TRAP
#define EFI_SW_EC_IPF_TAKEN_BRANCH \
  EXCEPT_IPF_TAKEN_BRANCH
#define EFI_SW_EC_IPF_SINGLE_STEP \
  EXCEPT_IPF_SINGLE_STEP


//
// Software Class PEI Service Subclass Error Code definitions.
//
#define EFI_SW_PS_EC_RESET_NOT_AVAILABLE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_PS_EC_MEMORY_INSTALLED_TWICE \
  (EFI_SUBCLASS_SPECIFIC | 0x00000001)

//
// Software Class EFI Boot Service Subclass Error Code
// definitions.
//
```

```
//
// Software Class EFI Runtime Service Subclass Error Code \
// definitions.


//
//
// Software Class EFI DXE Service Subclass Error Code \
// definitions.
//


#define EFI_SW_DXE_BS_PC_BEGIN_CONNECTING_DRIVERS \
  (EFI_SUBCLASS_SPECIFIC | 0x00000005)
#define EFI_SW_DXE_BS_PC_VERIFYING_PASSWORD \
  (EFI_SUBCLASS_SPECIFIC | 0x00000006)

//
// Software Class DXE RT Driver Subclass Progress Code
// definitions.
//
#define EFI_SW_DXE_RT_PC_S0 (EFI_SUBCLASS_SPECIFIC | 0x00000000)
#define EFI_SW_DXE_RT_PC_S1 (EFI_SUBCLASS_SPECIFIC | 0x00000001)
#define EFI_SW_DXE_RT_PC_S2 (EFI_SUBCLASS_SPECIFIC | 0x00000002)
#define EFI_SW_DXE_RT_PC_S3 (EFI_SUBCLASS_SPECIFIC | 0x00000003)
#define EFI_SW_DXE_RT_PC_S4 (EFI_SUBCLASS_SPECIFIC | 0x00000004)
#define EFI_SW_DXE_RT_PC_S5 (EFI_SUBCLASS_SPECIFIC | 0x00000005)


//
// Software Class X64 Exception Subclass Error Code definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI
// specification.
//
#define EFI_SW_EC_X64_DIVIDE_ERROR     EXCEPT_X64_DIVIDE_ERROR
#define EFI_SW_EC_X64_DEBUG            EXCEPT_X64_DEBUG
#define EFI_SW_EC_X64_NMI              EXCEPT_X64_NMI
#define EFI_SW_EC_X64_BREAKPOINT       EXCEPT_X64_BREAKPOINT
#define EFI_SW_EC_X64_OVERFLOW         EXCEPT_X64_OVERFLOW
#define EFI_SW_EC_X64_BOUND            EXCEPT_X64_BOUND
#define EFI_SW_EC_X64_INVALID_OPCODE   EXCEPT_X64_INVALID_OPCODE
#define EFI_SW_EC_X64_DOUBLE_FAULT     EXCEPT_X64_DOUBLE_FAULT
#define EFI_SW_EC_X64_INVALID_TSS      EXCEPT_X64_INVALID_TSS
#define EFI_SW_EC_X64_SEG_NOT_PRESENT \
  EXCEPT_X64_SEG_NOT_PRESENT
#define EFI_SW_EC_X64_STACK_FAULT      EXCEPT_X64_STACK_FAULT
#define EFI_SW_EC_X64_GP_FAULT         EXCEPT_X64_GP_FAULT
#define EFI_SW_EC_X64_PAGE_FAULT       EXCEPT_X64_PAGE_FAULT
#define EFI_SW_EC_X64_FP_ERROR         EXCEPT_X64_FP_ERROR
#define EFI_SW_EC_X64_ALIGNMENT_CHECK \
  EXCEPT_X64_ALIGNMENT_CHECK
#define EFI_SW_EC_X64_MACHINE_CHECK    EXCEPT_X64_MACHINE_CHECK
#define EFI_SW_EC_X64_SIMD             EXCEPT_X64_SIMD


//
// Software Class ARM Exception Subclass Error Code definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI
// specification.
//
#define EFI_SW_EC_ARM_RESET            EXCEPT_ARM_RESET
#define EFI_SW_EC_ARM_UNDEFINED_INSTRUCTION \
  EXCEPT_ARM_UNDEFINED_INSTRUCTION
#define EFI_SW_EC_ARM_SOFTWARE_INTERRUPT \
  EXCEPT_ARM_SOFTWARE_INTERRUPT
#define EFI_SW_EC_ARM_PREFETCH_ABORT \
  EXCEPT_ARM_PREFETCH_ABORT
#define EFI_SW_EC_ARM_DATA_ABORT       EXCEPT_ARM_DATA_ABORT
#define EFI_SW_EC_ARM_RESERVED         EXCEPT_ARM_RESERVED
#define EFI_SW_EC_ARM_IRQ              EXCEPT_ARM_IRQ
#define EFI_SW_EC_ARM_FIQ              EXCEPT_ARM_FIQ
#define EFI_SW_EC_AARCH64_SYNCHRONOUS_EXCEPTIONS \
  EXCEPT_AARCH64_SYNCHRONOUS_EXCEPTIONS
#define EFI_SW_EC_AARCH64_IRQ EXCEPT_AARCH64_IRQ
#define EFI_SW_EC_AARCH64_FIQ EXCEPT_AARCH64_FIQ
#define EFI_SW_EC_AARCH64_SERROR EXCEPT_AARCH64_SERROR

//
// Software Class RISC-V Exception Subclass Error Code
definitions.
// These exceptions are derived from the debug protocol
// definitions in the EFI specification.
//

#define EFI_SW_EC_RISCV_INST_MISALIGNED
EXCEPT_RISCV_INST_MISALIGNED
#define EFI_SW_EC_RISCV_INST_ACCESS_FAULT
EXCEPT_RISCV_INST_ACCESS_FAULT
#define EFI_SW_EC_RISCV_ILLEGAL_INSTEXCEPT_RISCV_ILLEGAL_INST
#define EFI_SW_EC_RISCV_BREAKPOINTEXCEPT_RISCV_ BREAKPOINT
```

```
#define EFI_SW_EC_RISCV_LOAD_ADDRESS_MISALIGNED\
EXCEPT_RISCV_LOAD_ADDRESS_MISALIGNED
#define EFI_SW_EC_RISCV_LOAD_ACCESS_FAULT
EXCEPT_RISCV_LOAD_ACCESS_FAULT
#define EFI_SW_EC_RISCV_STORE_AMO_ADDRESS_MISALIGNED\
EXCEPT_RISCV_STORE_AMO_ADDRESS_MISALIGNED
#define EFI_SW_EC_RISCV_STORE_AMO_ACCESS_FAULT \
EXCEPT_RISCV_STORE_AMO_ACCESS_FAULT
#define EFI_SW_EC_RISCV_ENV_CALL_FROM_UMODE
EXCEPT_RISCV_ENV_CALL_FROM_UMODE
#define EFI_SW_EC_RISCV_ENV_CALL_FROM_SMODE
EXCEPT_RISCV_ENV_CALL_FROM_SMODE
#define EFI_SW_EC_RISCV_ENV_CALL_FROM_HMOD
EXCEPT_RISCV_ENV_CALL_FROM_HMODE
#define EFI_SW_EC_RISCV_ENV_CALL_FROM_MMODE
EXCEPT_RISCV_ENV_CALL_FROM_MMODE
```

6.7.4.4 Extended Data Formats

In addition to the other class-specific error definitions in this subsection, the Host Software class uses the following extended error data definition:

- **EFI_DEVICE_HANDLE_EXTENDED_DAT**A

See section 6.6.4 for its definition.

EFI_DEBUG_ASSERT_DATA

Summary

This structure provides the assert information that is typically associated with a debug assertion failing.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA         DataHeader;
  UINT32                       LineNumber;
  UINT32                       FileNameSize;
  EFI_STATUS_CODE_STRING_DATA  *FileName;
} EFI_DEBUG_ASSERT_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_DEBUG_ASSERT_DATA) – *HeaderSize*** , and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

LineNumber

The line number of the source file where the fault was generated.

FileNameSize

The size in bytes of *FileName*.

FileName

A pointer to a **NULL**-terminated ASCII or Unicode string that represents the file name of the source file where the fault was generated. Type **EFI_STATUS_CODE_STRING_DATA** is defined in section 6.6.2.

Description

The data indicates the location of the assertion that failed in the source code. This information includes the file name and line number that are necessary to find the failing assertion in source code.

EFI_STATUS_CODE_EXCEP_EXTENDED_DATA

Summary

This structure defines extended data describing a processor exception error.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                        DataHeader;
  EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT        Context;
} EFI_STATUS_CODE_EXCEP_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_STATUS_CODE_EXCEP_EXTENDED_DATA) – *HeaderSize***, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

Context

The system context. Type **EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT** is defined in "Related Definitions" below.

Description

This extended data allows the processor context that is present at the time of the exception to be reported with the exception. The format and contents of the context data varies depending on the processor architecture.

Related Definitions

```
//**********************************************************
// EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT
//**********************************************************
typedef union {
  EFI_SYSTEM_CONTEXT_EBC      SystemContextEbc;
  EFI_SYSTEM_CONTEXT_IA32     SystemContextIa32;
  EFI_SYSTEM_CONTEXT_IPF      SystemContextIpf;
  EFI_SYSTEM_CONTEXT_X64      SystemContextX64;
  EFI_SYSTEM_CONTEXT_ARM      SystemContextArm;
  EFI_SYSTEM_CONTEXT_RISCV32  SystemContextRiscV32;
  EFI_SYSTEM_CONTEXT_RISCV64  SystemContextRiscV64;
  EFI_SYSTEM_CONTEXT_RISCV128 SystemContextRiscv128;
} EFI_STATUS_CODE_EXCEP_SYSTEM_CONTEXT;
```

SystemContextEbc

The context of the EBC virtual machine when the exception was generated. Type **EFI_SYSTEM_CONTEXT_EBC** is defined in **EFI_DEBUG_SUPPORT_PROTOCOL** in the *UEFI Specification*.

SystemContextIa32

The context of the IA-32 processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_IA32** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** in the *UEFI Specification.*

SystemContextIpf

The context of the Itanium® processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_IPF** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** in the *UEFI Specification.*

SystemContextX64

The context of the X64 processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_X64** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** in the *UEFI Specification*.

SystemContextArm

The context of the ARM processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_ARM** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** in the *UEFI Specification*.

SystemContextRiscV32

The context of the RISC-V RV32 processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_RISCV32** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** *in the UEFI Specification.*

SystemContextRiscV64

The context of the RISC-V RV64 processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_RISCV64** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** *in the UEFI Specification.*

SystemContextRiscV128

The context of the RISC-V RV128 processor when the exception was generated. Type **EFI_SYSTEM_CONTEXT_RISCV128** is defined in the **EFI_DEBUG_SUPPORT_PROTOCOL** *in the UEFI Specification.*

EFI_STATUS_CODE_START_EXTENDED_DATA

Summary

This structure defines extended data describing a call to a driver binding protocol start function.

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA           DataHeader;
  EFI_HANDLE                     ControllerHandle;
  EFI_HANDLE                     DriverBindingHandle;
  UINT16                         DevicePathSize;
  // EFI_DEVICE_PATH_PROTOCOL    RemainingDevicePath;
} EFI_STATUS_CODE_START_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data. *DataHeader.HeaderSize* should be **sizeof (EFI_STATUS_CODE_DATA)**, *DataHeader.Size* should be **sizeof (EFI_STATUS_CODE_START_EXTENDED_DATA) -** *HeaderSize*, and *DataHeader.Type* should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

ControllerHandle

The controller handle.

DriverBindingHandle

The driver binding handle.

DevicePathSize

The size of the *RemainingDevicePath*. It is zero if the **Start()** function is called with *RemainingDevicePath* **= NULL**. The *UEFI Specification* allows that the **Start()** function of bus drivers can be called in this way.

RemainingDevicePath

Matches the *RemainingDevicePath* parameter being passed to the **Start()** function. Note that this parameter is the variable-length device path and not a pointer to the device path.

Description

This extended data records information about a **Start()** function call. **Start()** is a member of the UEFI Driver Binding Protocol.

EFI_LEGACY_OPROM_EXTENDED_DATA

Summary

This structure defines extended data describing a legacy option ROM (OpROM).

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA                              DataHeader;
  EFI_HANDLE                                        DeviceHandle;
  EFI_PHYSICAL_ADDRESS                              RomImageBase;
} EFI_LEGACY_OPROM_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data:

DataHeader.HeaderSize should be **sizeof(EFI_STATUS_CODE_DATA)**,

DataHeader.Size should be **sizeof(EFI_LEGACY_OPROM_EXTENDED_DATA) -** *HeaderSize*, and

DataHeader.Type should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

DeviceHandle

The handle corresponding to the device that this legacy option ROM is being invoked.

RomImageBase

The base address of the shadowed legacy ROM image. May or may not point to the shadow RAM area. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification*.

Description

The device handle and ROM image base can be used by consumers to determine which option ROM failed. Due to the black-box nature of legacy option ROMs, the amount of information that can be obtained may be limited.

EFI_RETURN_STATUS_EXTENDED_DATA

Summary

This structure defines extended data describing an **EFI_STATUS** return value that stands for a failed function call (such as a UEFI boot service).

Prototype

```
typedef struct {
  EFI_STATUS_CODE_DATA DataHeader;
  EFI_STATUS           ReturnStatus;
} EFI_RETURN_STATUS_EXTENDED_DATA;
```

Parameters

DataHeader

The data header identifying the data:

DataHeader.HeaderSize should be **sizeof(EFI_STATUS_CODE_DATA)**,

DataHeader.Size should be **sizeof(EFI_RETURN_STATUS_EXTENDED_DATA) -** *HeaderSize*, and

DataHeader.Type should be **EFI_STATUS_CODE_SPECIFIC_DATA_GUID**.

ReturnStatus

The **EFI_STATUS** return value of the service or function whose failure triggered the reporting of the status code (generally an error code or a debug code).

Description

The extended data records the return value of a service or function call whose failure justifies reporting a status code (generally an error code or debug code).

7 Report Status Code Routers

7.1 Overview

This section provides the code definitions for the PPI and Protocols used in a Report Status Code Router. These interfaces allow multiple platform dependent drivers for displaying status code information to coexist without prior knowledge of one another.

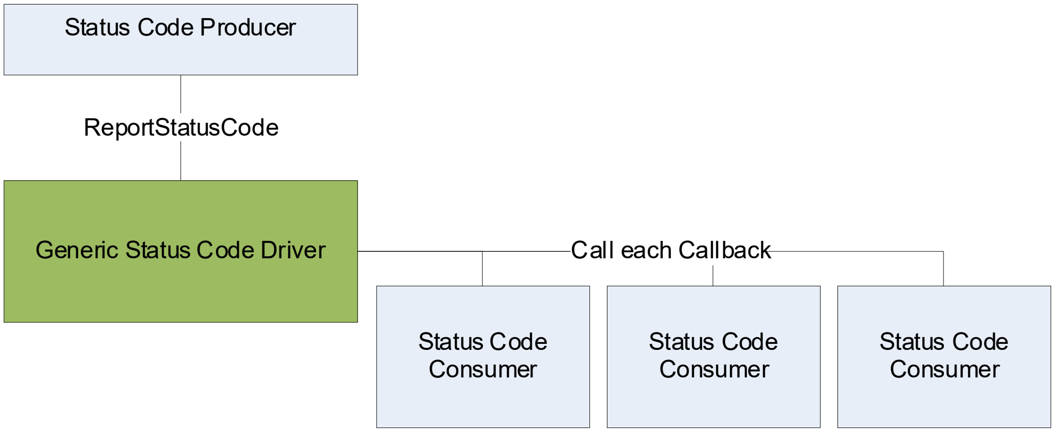


Figure 13. Status Code Services

There is a generic status code driver in each phase. In each case the driver consumes the Report Status Code Protocol and produces the Report Status Code Handler PPI or Protocol. Each consumer of the Report Status Code Handler PPI or Protocol will register a callback to receive notification of new Status Codes from the Generic Status Code Driver.

7.2 Code Definitions

7.2.1 Report Status Code Handler Protocol

EFI_RSC_HANDLER_PROTOCOL

Summary

Provide registering and unregistering services to status code consumers while in DXE.

GUID

```
#define EFI_RSC_HANDLER_PROTOCOL_GUID \
  { \
    0x86212936, 0xe76, 0x41c8, \
    0xa0, 0x3a, 0x2a, 0xf2, 0xfc, 0x1c, 0x39, 0xe2 \
  }
```

Protocol Interface Structure

```
typedef struct {
  EFI_RSC_HANDLER_REGISTER       Register;
  EFI_RSC_HANDLER_UNREGISTER     Unregister;
} EFI_RSC_HANDLER_PROTOCOL;
```

Members

Register

Register the callback for notification of status code messages.

Unregister

Unregister the callback.

Description

Once registered, status code messages will be forwarded to the callback. The callback must be unregistered before it is deallocated.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_RSC_HANDLER_CALLBACK) (
  IN EFI_STATUS_CODE_TYPE     CodeType,
  IN EFI_STATUS_CODE_VALUE    Value,
  IN UINT32                   Instance,
  IN EFI_GUID                 * CallerId,
  IN EFI_STATUS_CODE_DATA     * Data
  );
```

For parameter descriptions, function descriptions and status code values, see **ReportStatusCode()** in the *PI Specification,Volume 2, section 14.2.*

EFI_RSC_HANDLER_PROTOCOL.Register()

Summary

Register the callback function for **ReportStatusCode()** notification.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_RSC_HANDLER_REGISTER) (
  IN EFI_RSC_HANDLER_CALLBACK   Callback,
  IN EFI_TPL                       Tpl
  );
```

Parameters

Callback

A pointer to a function of type **EFI_RSC_HANDLER_CALLBACK** that is called when a call to **ReportStatusCode()** occurs.

Tpl

TPL at which callback can be safely invoked.

Description

When this function is called the function pointer is added to an internal list and any future calls to **ReportStatusCode()** will be forwarded to the *Callback* function. During the boot-services, this is the callback for which this service can be invoked. The report status code router will create an event such that the callback function is only invoked at the TPL for which it was registered. The entity that registers for the callback should also register for an event upon generation of exit boot services and invoke the unregister service.

If the handler does not have a TPL dependency, it should register for a callback at TPL high. The router infrastructure will support making callbacks at runtime, but the caller for runtime invocation must meet the following criteria:

1. must be a runtime driver type so that its memory is not reclaimed
2. not unregister at exit boot services so that the router will still have its callback address
3. the caller must be self-contained (eg. Not call out into any boot-service interfaces) and be runtime safe, in general.

Status Codes Returned

EFI_SUCCESS	Function was successfully registered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_OUT_OF_RESOURCES	The internal buffer ran out of space. No more functions can be registered.
EFI_ALREADY_STARTED	The function was already registered. It can't be registered again.

EFI_RSC_HANDLER_PROTOCOL.Unregister()

Summary

Remove a previously registered callback function from the notification list.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_RSC_HANDLER_UNREGISTER) (
  IN EFI_RSC_HANDLER_CALLBACK      Callback
  );
```

Parameters

Callback

A pointer to a function of type **EFI_RSC_HANDLER_CALLBACK** that is to be unregistered.

Description

A callback function must be unregistered before it is deallocated. It is important that any registered callbacks that are not runtime complaint be unregistered when **ExitBootServices()** is called.

Status Codes Returned

EFI_SUCCESS	The function was successfully unregistered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_NOT_FOUND	The callback function was not found to be unregistered.

7.2.2 Report Status Code Handler PPI

EFI_PEI_RSC_HANDLER_PPI

Summary

Provide registering and unregistering services to status code consumers.

GUID

```
#define EFI_PEI_RSC_HANDLER_PPI_GUID \
  { \
    0x65d394,   0x9951, 0x4144, \
    0x82, 0xa3, 0xa, 0xfc, 0x85, 0x79, 0xc2, 0x51 \
  }
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RSC_HANDLER_PPI {
  EFI_PEI_RSC_HANDLER_REGISTER       Register;
  EFI_PEI_RSC_HANDLER_UNREGISTER     Unregister;
} EFI_PEI_RSC_HANDLER_PPI;
```

Members

Register

Register the callback for notification of status code messages.

Unregister

Unregister the callback.

Description

Once registered, status code messages will be forwarded to the callback.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_RSC_HANDLER_CALLBACK) (
  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,
  IN CONST EFI_STATUS_CODE_DATA  *Data
);
```

For parameter descriptions, function descriptions and status code values, see **ReportStatusCode()** in the *PI specification Volume 1, section 4.5*.

EFI_PEI_RSC_HANDLER_PPI.Register()

Summary

Register the callback function for **ReportStatusCode()** notification.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_RSC_HANDLER_REGISTER) (
  IN EFI_PEI_RSC_HANDLER_CALLBACK  Callback
  );
```

Parameters

Callback

A pointer to a function of type **EFI_PEI_RSC_HANDLER_CALLBACK** that is called when a call to **ReportStatusCode()** occurs.

Description

When this function is called the function pointer is added to an internal list and any future calls to **ReportStatusCode()** will be forwarded to the *Callback* function.

Status Codes Returned

EFI_SUCCESS	Function was successfully registered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_OUT_OF_RESOURCES	The internal buffer ran out of space. No more functions can be registered.
EFI_ALREADY_STARTED	The function was already registered. It can't be registered again.

EFI_PEI_RSC_HANDLER_PPI.Unregister()

Summary

Remove a previously registered callback function from the notification list.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_RSC_HANDLER_UNREGISTER) (
  IN EFI_PEI_RSC_HANDLER_CALLBACK  Callback
  );
```

Parameters

Callback

A pointer to a function of type **EFI_PEI_RSC_HANDLER_CALLBACK** that is to be unregistered.

Description

ReportStatusCode() messages will no longer be forwarded to the *Callback* function.

Status Codes Returned

EFI_SUCCESS	The function was successfully unregistered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_NOT_FOUND	The callback function was not found to be unregistered.

7.2.3 SMM Report Status Code Handler Protocol

EFI_SMM_RSC_HANDLER_PROTOCOL

Summary

Provide registering and unregistering services to status code consumers while in DXE SMM.

GUID

```
#define EFI_SMM_RSC_HANDLER_PROTOCOL_GUID \
  { \
0x2ff29fa7, 0x5e80, 0x4ed9, 0xb3, 0x80, 0x1, 0x7d, 0x3c, 0x55,
0x4f, 0xf4
  }
```

Protocol Interface Structure

```
typedef struct _EFI_SMM_RSC_HANDLER_PROTOCOL {
  EFI_SMM_RSC_HANDLER_REGISTER       Register;
  SMM_RSC_HANDLER_UNREGISTER         Unregister;
} EFI_SMM_RSC_HANDLER_PROTOCOL;
```

Members

Register

Register the callback for notification of status code messages.

Unregister

Unregister the callback.

Description

Once registered, status code messages will be forwarded to the callback. The callback must be unregistered before it is deallocated.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_RSC_HANDLER_CALLBACK) (
  IN EFI_STATUS_CODE_TYPE     CodeType,
  IN EFI_STATUS_CODE_VALUE    Value,
  IN UINT32                   Instance,
  IN EFI_GUID                 * CallerId,
  IN EFI_STATUS_CODE_DATA     * Data
  );
```

For parameter descriptions, function descriptions and status code values, see **ReportStatusCode()** in the PI specification Volume 2, section 14.2.

EFI_SMM_RSC_HANDLER_PROTOCOL.Register()

Summary

Register the callback function for **ReportStatusCode()** notification.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_RSC_HANDLER_REGISTER) (
  IN EFI_SMM_RSC_HANDLER_CALLBACK      Callback
  );
```

Parameters

Callback

A pointer to a function of type **EFI_RSC_HANDLER_CALLBACK** that is called when a call to **ReportStatusCode()** occurs.

Description

When this function is called the function pointer is added to an internal list and any future calls to **ReportStatusCode()** will be forwarded to the *Callback* function.

Status Codes Returned

EFI_SUCCESS	Function was successfully registered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_OUT_OF_RESOURCES	The internal buffer ran out of space. No more functions can be registered.
EFI_ALREADY_STARTED	The function was already registered. It can't be registered again.

EFI_SMM_RSC_HANDLER_PROTOCOL.Unregister()

Summary

Remove a previously registered callback function from the notification list.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_RSC_HANDLER_UNREGISTER) (
  IN EFI_SMM_RSC_HANDLER_CALLBACK        Callback
  );
```

Parameters

Callback

A pointer to a function of type **EFI_SMM_RSC_HANDLER_CALLBACK** that is to be unregistered.

Description

A callback function must be unregistered before it is deallocated. It is important that any registered callbacks that are not runtime complaint be unregistered when **ExitBootServices()** is called.

Status Codes Returned

EFI_SUCCESS	The function was successfully unregistered.
EFI_INVALID_PARAMETER	The callback function was **NULL**.
EFI_NOT_FOUND	The callback function was not found to be unregistered.

8 PCD

8.1 PCD Protocol Definitions

8.1.1 PCD Protocol

EFI_PCD_PROTOCOL

Summary

A platform database that contains a variety of current platform settings or directives that can be accessed by a driver or application.

GUID

```
#define EFI_PCD_PROTOCOL_GUID \
  { 0x13a3f0f6, 0x264a, 0x3ef0, \
  { 0xf2, 0xe0, 0xde, 0xc5, 0x12, 0x34, 0x2f, 0x34 } }
```

Protocol Interface Structure

```
typedef struct _EFI_PCD_PROTOCOL {
  EFI_PCD_PROTOCOL_SET_SKU                SetSku;

  EFI_PCD_PROTOCOL_GET_8                  Get8;
  EFI_PCD_PROTOCOL_GET_16                 Get16;
  EFI_PCD_PROTOCOL_GET_32                 Get32;
  EFI_PCD_PROTOCOL_GET_64                 Get64;
  EFI_PCD_PROTOCOL_GET_POINTER            GetPtr;
  EFI_PCD_PROTOCOL_GET_BOOLEAN            GetBool;
  EFI_PCD_PROTOCOL_GET_SIZE               GetSize;

  EFI_PCD_PROTOCOL_SET_8                  Set8;
  EFI_PCD_PROTOCOL_SET_16                 Set16;
  EFI_PCD_PROTOCOL_SET_32                 Set32;
  EFI_PCD_PROTOCOL_SET_64                 Set64;
  EFI_PCD_PROTOCOL_SET_POINTER            SetPtr;
  EFI_PCD_PROTOCOL_SET_BOOLEAN            SetBool;

  EFI_PCD_PROTOCOL_CALLBACK_ON_SET        CallbackOnSet;
  EFI_PCD_PROTOCOL_CANCEL_CALLBACK        CancelCallback;
  EFI_PCD_PROTOCOL_GET_NEXT_TOKEN         GetNextToken;
  EFI_PCD_PROTOCOL_GET_NEXT_TOKEN_SPACE GetNextTokenSpace;
} EFI_PCD_PROTOCOL;
```

Parameters

SetSku

Establish a current SKU value for the PCD service to use for subsequent data Get/Set requests.

Get8

Retrieve an 8-bit value from the PCD service using a GUIDed token namespace.

Get16

Retrieve a 16-bit value from the PCD service using a GUIDed token namespace.

Get32

Retrieve a 32-bit value from the PCD service using a GUIDed token namespace.

Get64

Retrieve a 64-bit value from the PCD service using a GUIDed token namespace.

GetPtr

Retrieve a pointer to a value from the PCD service using a GUIDed token namespace. Can be used to retrieve an array of bytes that may represent a data structure, ASCII string, or Unicode string

GetBool

Retrieve a Boolean value from the PCD service using a GUIDed token namespace.

GetSize

Retrieve the size of a particular PCD Token value using a GUIDed token namespace.

Set8

Set an 8-bit value in the PCD service using a GUIDed token namespace

Set16

Set a 16-bit value in the PCD service using a GUIDed token namespace.

Set32

Set a 32-bit value in the PCD service using a GUIDed token namespace.

Set64

Set a 64-bit value in the PCD service using a GUIDed token namespace.

SetPtr

Set a pointer to a value in the PCD service using a GUIDed token namespace. Can be used to set an array of bytes that may represent a data structure, ASCII string, or Unicode string

SetBool

Set a Boolean value in the PCD service using a GUIDed token namespace.

CallBackOnSet

Establish a notification to alert when a particular PCD Token value is set.

CancelCallBackOnSet

Cancel a previously set notification for a particular PCD Token value.

GetNextToken

Retrieve the next token number that is contained in the PCD name-space.

GetNextTokenSpace

Retrieve the next valid PCD token namespace for a given name-space.

Description

Callers to this protocol must be at a **TPL_APPLICATION** task priority level.

This is the base PCD service API that provides an abstraction for accessing configuration content in the platform. It a seamless mechanism for extracting information regardless of where the information is stored (such as in Read-only data, or an EFI Variable).

This protocol allows access to data through size-granular APIs and provides a mechanism for a firmware component to monitor specific settings and be alerted when a setting is changed.

EFI_PCD_PROTOCOL.SetSku ()

Summary

Sets the SKU value for subsequent calls to set or get PCD token values.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PCD_PROTOCOL_SET_SKU) (
  IN UINTN           SkuId
  );
```

Parameters

SkuId

The SKU value to set.

Description

SetSku() sets the SKU Id to be used for subsequent calls to set or get PCD values. **SetSku()** is normally called only once by the system.

For each item (token), the database can hold a single value that applies to all SKUs, or multiple values, where each value is associated with a specific SKU Id. Items with multiple, SKU-specific values are called *SKU enabled*.

The SKU Id of zero is reserved as a default. For tokens that are not SKU enabled, the system ignores any set SKU Id and works with the single value for that token. For SKU-enabled tokens, the system will use the SKU Id set by the last call to SetSku(). If no SKU Id is set or the currently set SKU Id isn't valid for the specified token, the system uses the default SKU Id. If the system attempts to use the default SKU Id and no value has been set for that Id, the results are unpredictable.

EFI_PCD_PROTOCOL.Get8 ()

Summary

Retrieves an 8-bit value for a given PCD token.

Prototype

```
typedef
UINT8
(EFIAPI *EFI_PCD_PROTOCOL_GET_8) (
  IN CONST EFI_GUID                    *Guid,
  IN UINTN                             TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current byte-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.Get16 ()

Summary

Retrieves a 16-bit value for a given PCD token.

Prototype

```
typedef
UINT16
(EFIAPI *EFI_PCD_PROTOCOL_GET_16) (
  IN CONST EFI_GUID                    *Guid,
  IN UINTN                             TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current word-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.Get32 ()

Summary

Retrieves a 32-bit value for a given PCD token.

Prototype

```
typedef
UINT32
(EFIAPI *EFI_PCD_PROTOCOL_GET_32) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current 32-bit sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.Get64 ()

Summary

Retrieves a 64 -bit value for a given PCD token.

Prototype

```
typedef
UINT64
(EFIAPI *EFI_PCD_PROTOCOL_GET_64) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the 64-bit sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.GetPtr ()

Summary

Retrieves a pointer to a value for a given PCD token.

Prototype

```
typedef
VOID *
(EFIAPI *EFI_PCD_PROTOCOL_GET_POINTER) (
  IN CONST EFI_GUID                   *Guid,
  IN UINTN                            TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current pointer to the value for a PCD token number. Do not make any assumptions about the alignment of the pointer that is returned by this function call. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.GetBool ()

Summary

Retrieves a Boolean value for a given PCD token.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_PCD_PROTOCOL_GET_BOOLEAN) (
  IN CONST EFI_GUID                     *Guid,
  IN UINTN                              TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current BOOLEAN-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.GetSize ()

Summary

Retrieves the size of the value for a given PCD token.

Prototype

```
typedef
UINTN
(EFIAPI *EFI_PCD_PROTOCOL_GET_SIZE) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current size of a particular PCD token. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PCD_PROTOCOL.Set8 ()

Summary

Sets an 8-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_8) (
  IN CONST EFI_GUID       *Guid,
  IN UINTN                 TokenNumber,
  IN UINT 8                Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use *GetBool()* to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.Set16 ()

Summary

Sets a 16-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_16) (
  IN CONST EFI_GUID        *Guid,
  IN UINTN                 TokenNumber,
  IN UINT16                Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.The 128-bit unique value that designates the namespace from which to extract the value.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use `GetBool()` to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.Set32 ()

Summary

Sets a 32-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_32) (
  IN CONST EFI_GUID    *Guid,
  IN UINTN              TokenNumber,
  IN UINT32             Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.Set64 ()

Summary

Sets a 64-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_64) (
  IN CONST EFI_GUID       *Guid,
  IN UINTN                TokenNumber,
  IN UINT64               Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use `GetBool()` to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.SetPtr ()

Summary

Sets a value of a specified size for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_POINTER) (
  IN CONST EFI_GUID   *Guid,
  IN UINTN            TokenNumber,
  IN OUT UINTN        *SizeOfValue,
  IN VOID             *Buffer
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

SizeOfValue

The length of the value being set for the PCD token. If too large of a length is specified, upon return from this function the value of *SizeOfValue* will reflect the maximum size for the PCD token.

Buffer

A pointer to the buffer containing the value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. The *SizeofValue* parameter reflects the maximum size of the PCD token referenced. Use **GetSize()** to retrieve the current size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.SetBool ()

Summary

Sets a Boolean value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_SET_BOOLEAN) (
  IN CONST EFI_GUID             *Guid,
  IN UINTN                      TokenNumber,
  IN BOOLEAN                    Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

Token

NumberThe PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PCD_PROTOCOL.CallbackOnSet ()

Summary

Specifies a function to be called anytime the value of a designated token is changed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_CALLBACK_ON_SET) (
  IN CONST EFI_GUID                 *Guid, OPTIONAL
  IN UINTN                          CallBackToken,
  IN EFI_PCD_PROTOCOL_CALLBACK      CallBackFunction
  );
```

Parameters

Guid

The 128-bit unique value that designates which namespace to monitor. If **NULL**, use the standard platform namespace.

CallBackToken

The PCD token number to monitor.

CallBackFunction

The function prototype called when the value associated with the *CallBackToken* is set.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_PCD_PROTOCOL_CALLBACK) {
  IN   EFI_GUID                   *Guid, OPTIONAL
  IN   UINTN                      CallBackToken,
  IN   OUT VOID                   *TokenData,
  IN   UINTN                      TokenDataSize
  };
```

Description

Specifies a function to be called anytime the value of a designated token is changed.

Status Codes Returned

EFI_SUCCESS	The PCD service has successfully established a call event for the *CallBackToken* requested.
EFI_NOT_FOUND	The PCD service could not find the referenced token number.

EFI_PCD_PROTOCOL.CancelCallback ()

Summary

Cancels a previously set callback function for a particular PCD token number.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_CANCEL_CALLBACK) (
  IN CONST EFI_GUID             *Guid, OPTIONAL
  IN UINTN                      CallBackToken,
  IN EFI_PCD_PROTOCOL_CALLBACK  CallBackFunction
  );
```

Parameters

Guid

The 128-bit unique value that designates which namespace to monitor. If **NULL**, use the standard platform namespace.

CallBackToken

The PCD token number for which to cancel monitoring.

CallBackFunction

The function prototype that was originally passed to the *CallBackOnSet* function.

Description

Cancels a callback function that was set through a previous call to the *CallBackOnSet* function.

Status Codes Returned

EFI_SUCCESS	The PCD service has cancelled the call event associated with the *CallBackToken*.
EFI_INVALID_PARAMETER	The PCD service did not match the *CallBackFunction* to one that is currently being monitored.
EFI_NOT_FOUND	The PCD service could not find data the requested token number.

EFI_PCD_PROTOCOL.GetNextToken ()

Summary

Retrieves the next valid PCD token for a given namespace.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_GET_NEXT_TOKEN) (
  IN CONST EFI_GUID        *Guid, OPTIONAL
  IN UINTN                 *TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to retrieve the next token.

TokenNumber

A pointer to the PCD token number to use to find the subsequent token number. To retrieve the "first" token, have the pointer reference a *TokenNumber* value of 0.

Description

Gets the next valid token number in a given namespace. This is useful since the PCD infrastructure contains a sparse list of token numbers, and one cannot a priori know what token numbers are valid in the database.

Status Codes Returned

EFI_SUCCESS	The PCD service has retrieved the value requested
EFI_NOT_FOUND	The PCD service could not find data from the requested token number.

EFI_PCD_PROTOCOL.GetNextTokenSpace ()

Summary

Retrieves the next valid PCD token namespace for a given namespace.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCD_PROTOCOL_GET_NEXT_TOKEN_SPACE) (
  IN OUT CONST EFI_GUID         **Guid
  );
```

Parameters

Guid

An indirect pointer to **EFI_GUID**. On input it designates a known token namespace from which the search will start. On output, it designates the next valid token namespace on the platform. If **Guid* is **NULL**, then the GUID of the first token space of the current platform is returned. If the search cannot locate the next valid token namespace, an error is returned and the value of **Guid* is undefined.

Description

Gets the next valid token namespace for a given namespace. This is useful to traverse the valid token namespaces on a platform.

Status Codes Returned

EFI_SUCCESS	The PCD service retrieved the value requested.
EFI_NOT_FOUND	The PCD service could not find the next valid token namespace.

8.1.2 Get PCD Information Protocol

EFI_GET_PCD_INFO_PROTOCOL

Summary

The protocol that provides additional information about items that reside in the PCD database.

GUID

```
#define EFI_GET_PCD_INFO_PROTOCOL_GUID \
  { 0xfd0f4478, 0xefd, 0x461d, \
  { 0xba, 0x2d, 0xe5, 0x8c, 0x45, 0xfd, 0x5f, 0x5e } }
```

Protocol Interface Structure

```
typedef struct _EFI_GET_PCD_INFO_PROTOCOL {
  EFI_GET_PCD_INFO_PROTOCOL_GET_INFO    GetInfo;
  EFI_GET_PCD_INFO_PROTOCOL_GET_SKU     GetSku;
} EFI_GET_PCD_INFO_PROTOCOL;
```

Parameters

GetInfo

Retrieve additional information associated with a PCD.

GetSku

Retrieve the currently set SKU Id.

Description

Callers to this protocol must be at a **TPL_APPLICATION** task priority level.

This is the PCD service to use when querying for some additional data that can be contained in the PCD database.

EFI_GET_PCD_INFO_PROTOCOL.GetInfo ()

Summary

Retrieve additional information associated with a PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_PCD_INFO_PROTOCOL_GET_INFO) (
  IN CONST EFI_GUID    *Guid,
  IN UINTN             TokenNumber,
  OUT EFI_PCD_INFO     *PcdInfo
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

PcdInfo

The returned information associated with the requested *TokenNumber*. See related definitions below.

Description

GetInfo() retrieves additional information associated with a PCD token. This includes information such as the type of value the *TokenNumber* is associated with as well as possible human readable name that is associated with the token.

Related Definitions

```
typedef struct {
  EFI_PCD_TYPE  PcdType;
  UINTN         PcdSize;
  CHAR8         *PcdName;
} EFI_PCD_INFO;
```

PcdType

The returned information associated with the requested *TokenNumber*. If *TokenNumber* is 0, then *PcdType* is set to *EFI_PCD_TYPE_8*.

PcdSize

The size of the data in bytes associated with the *TokenNumber* specified. If *TokenNumber* is 0, then *PcdSize* is set 0.

PcdName

The null-terminated ASCII string associated with a given token. If the *TokenNumber* specified was 0, then this field corresponds to the null-terminated ASCII string associated with the token's namespace *Guid.* If NULL, there is no name associated with this request.

```
typedef enum {
  EFI_PCD_TYPE_8,
  EFI_PCD_TYPE_16,
  EFI_PCD_TYPE_32,
  EFI_PCD_TYPE_64,
  EFI_PCD_TYPE_BOOL,
  EFI_PCD_TYPE_PTR
 } EFI_PCD_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The PCD information was returned successfully
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_GET_PCD_INFO_PROTOCOL.GetSku ()

Summary

Retrieve the currently set SKU Id.

Prototype

```
typedef
UINTN
(EFIAPI *EFI_GET_PCD_INFO_PROTOCOL_GET_SKU) (
  VOID
  );
```

Description

GetSku() returns the currently set SKU Id. If the platform has not set at a SKU Id, then the default SKU Id value of 0 is returned. If the platform has set a SKU Id, then the currently set SKU Id is returned.

8.2 PCD PPI Definitions

8.2.1 PCD PPI

EFI_PEI_PCD_PPI

Summary

A platform database that contains a variety of current platform settings or directives that can be accessed by a driver or application.

GUID

```
#define EFI_PEI_PCD_PPI_GUID \
  { 0x1f34d25, 0x4de2, 0x23ad, \
  { 0x3f, 0xf3, 0x36, 0x35, 0x3f, 0xf3, 0x23, 0xf1 } }
```

PPI Structure

```
typedef struct {
  EFI_PEI_PCD_PPI_SET_SKU                SetSku;

  EFI_PEI_PCD_PPI_GET_8                  Get8;
  EFI_PEI_PCD_PPI_GET_16                 Get16;
  EFI_PEI_PCD_PPI_GET_32                 Get32;
  EFI_PEI_PCD_PPI_GET_64                 Get64;
  EFI_PEI_PCD_PPI_GET_POINTER            GetPtr;
  EFI_PEI_PCD_PPI_GET_BOOLEAN            GetBool;
  EFI_PEI_PCD_PPI_GET_SIZE               GetSize;

  EFI_PEI_PCD_PPI_SET_8                  Set8;
  EFI_PEI_PCD_PPI_SET_16                 Set16;
  EFI_PEI_PCD_PPI_SET_32                 Set32;
  EFI_PEI_PCD_PPI_SET_64                 Set64;
  EFI_PEI_PCD_PPI_SET_POINTER            SetPtr;
  EFI_PEI_PCD_PPI_SET_BOOLEAN            SetBool;

  EFI_PEI_PCD_PPI_CALLBACK_ON_SET        CallbackOnSet;
  EFI_PEI_PCD_PPI_CANCEL_CALLBACK        CancelCallback;
  EFI_PEI_PCD_PPI_GET_NEXT_TOKEN         GetNextToken;
  EFI_PEI_PCD_PPI_GET_NEXT_TOKEN_SPACE GetNextTokenSpace;
} EFI_PEI_PCD_PPI;
```

Parameters

SetSku

Establish a current SKU value for the PCD service to use for subsequent data Get/Set requests.

Get8

Retrieve an 8-bit value from the PCD service using a GUIDed token namespace.

Get16

Retrieve a 16-bit value from the PCD service using a GUIDed token namespace.

Get32

Retrieve a 32-bit value from the PCD service using a GUIDed token namespace.

Get64

Retrieve a 64-bit value from the PCD service using a GUIDed token namespace.

`GetPtr`

Retrieve a pointer to a value from the PCD service using a GUIDed token namespace. Can be used to retrieve an array of bytes that represents a data structure, ASCII string, or Unicode string

`GetBool`

Retrieve a Boolean value from the PCD service using a GUIDed token namespace.

`GetBool`

Retrieve the size of a particular PCD Token value using a GUIDed token namespace.

`Set8`

Set an 8-bit value in the PCD service using a GUIDed token namespace.

`Set16`

Set a 16-bit value in the PCD service using a GUIDed token namespace.

`Set32`

Set a 32-bit value in the PCD service using a GUIDed token namespace.

`Set64`

Set a 64-bit value in the PCD service using a GUIDed token namespace.

`SetPtr`

Set a pointer to a value in the PCD service using a GUIDed token namespace. Can be used to set an array of bytes that represents a data structure, ASCII string, or Unicode string

`SetBool`

Set a Boolean value in the PCD service using a GUIDed token namespace.

`CallBackOnSet`

Establish a notification when a particular PCD Token value is set.

`CancelCallBackOnSet`

Cancel a previously set notification for a particular PCD Token value.

`GetNextToken`

Retrieve the next token number that is contained in the PCD name -space.

Description

This is the base PCD service API that provides an abstraction for accessing configuration content in the platform. It is a seamless mechanism for extracting information regardless of where the information is stored (such as in Read-only data in an EFI Variable).

This ppi provides access to data through size-granular APIs and provides a mechanism for a firmware component to monitor specific settings and be alerted when a setting is changed.

EFI_PEI_PCD_PPI.SetSku ()

Summary

Sets the SKU value for subsequent calls to set or get PCD token values.

Prototype

```
typedef
VOID
(EFIAPI *EFI_PEI_PCD_PPI_SET_SKU) (
  IN UINTN           SkuId
  );
```

Parameters

SkuId

The SKU value to set.

Description

SetSku() sets the SKU Id to be used for subsequent calls to set or get PCD values. **SetSku()** is normally called only once by the system.

For each item (token), the database can hold a single value that applies to all SKUs, or multiple values, where each value is associated with a specific SKU Id. Items with multiple, SKU-specific values are called *SKU enabled*.

The SKU Id of zero is reserved as a default. For tokens that are not SKU enabled, the system ignores any set SKU Id and works with the single value for that token. For SKU-enabled tokens, the system will use the SKU Id set by the last call to **SetSku()**. If no SKU Id is set or the currently set SKU Id isn't valid for the specified token, the system uses the default SKU Id. If the system attempts to use the default SKU Id and no value has been set for that Id, the results are unpredictable.

EFI_PEI_PCD_PPI.Get8 ()

Summary

Retrieves an 8-bit value for a given PCD token.

Prototype

```
typedef
UINT8
(EFIAPI *EFI_PEI_PCD_PPI_GET_8) (
  IN CONST EFI_GUID              *Guid,
  IN UINTN                        TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates which namespace to extract the value from.

TokenNumber

The PCD token number.

Description

Retrieves the current byte-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.Get16 ()

Summary

Retrieves a value for a given PCD token.

Prototype

```
typedef
UINT16
(EFIAPI *EFI_PEI_PCD_PPI_GET_16) (
  IN CONST EFI_GUID               *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current word-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.Get32 ()

Summary

Retrieves a 32-bit value for a given PCD token.

Prototype

```
typedef
UINT32
(EFIAPI *EFI_PEI_PCD_PPI_GET_32) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current 32-bit value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.Get64 ()

Summary

Retrieves a 64-bit value for a given PCD token.

Prototype

```
typedef
UINT64
(EFIAPI *EFI_PEI_PCD_PPI_GET_64) (
  IN CONST EFI_GUID              *Guid,
  IN UINTN                       TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the 64-bit value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.GetPtr ()

Summary

Retrieves a pointer to the value for a given PCD token.

Prototype

```
typedef
VOID *
(EFIAPI *EFI_PEI_PCD_PPI_GET_POINTER) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current pointer to the value for a PCD token number. There should not be any alignment assumptions about the pointer that is returned by this function call. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.GetBool ()

Summary

Retrieves a Boolean value for a given PCD token.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_PEI_PCD_PPI_GET_BOOLEAN) (
  IN CONST EFI_GUID            *Guid,
  IN UINTN                      TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current Boolean-sized value for a PCD token number. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.GetSize ()

Summary

Retrieves the size of the value for a given PCD token.

Prototype

```
typedef
UINTN
(EFIAPI *EFI_PEI_PCD_PPI_GET_SIZE) (
  IN CONST EFI_GUID              *Guid,
  IN UINTN                       TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Description

Retrieves the current size of a particular PCD token. If the *TokenNumber* is invalid, the results are unpredictable.

EFI_PEI_PCD_PPI.Set8 ()

Summary

Sets an 8-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_8) (
  IN CONST EFI_GUID            *Guid,
  IN UINTN                     TokenNumber,
  IN UINT8                     Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.Set16 ()

Summary

Sets a 16-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_16) (
  IN CONST EFI_GUID                *Guid,
  IN UINTN                         TokenNumber,
  IN UINT16                        Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.Set32 ()

Summary

Sets a 32-bit value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_32) (
  IN CONST EFI_GUID            *Guid,
  IN UINTN                     TokenNumber,
  IN UINT32                    Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.Set64 ()

Summary

Sets a 64-bit value for a given PCD token.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_64) (
  IN CONST EFI_GUID          *Guid,
  IN UINTN                   TokenNumber,
  IN UINT64                   Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.SetPtr ()

Summary

Sets a value of the specified size for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_POINTER) (
  IN CONST EFI_GUID          *Guid,
  IN UINTN                   TokenNumber,
  IN OUT UINTN               *SizeOfValue,
  IN VOID                    *Buffer
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

SizeOfValue

The length of the value being set for the PCD token. If too large of a length is specified, upon return from this function the value of *SizeOfValue* will reflect the maximum size for the PCD token.

Buffer

A pointer to the buffer containing the value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. The *SizeofValue* parameter reflects the maximum size of the PCD token referenced. Use **GetSize()** to retrieve the current size of the target data.

EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.SetBool()

Summary

Sets a Boolean value for a given PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_SET_BOOLEAN) (
  IN CONST EFI_GUID                 Guid,
  IN UINTN                          TokenNumber,
  IN BOOLEAN                        Value
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

Value

The value to set for the PCD token.

Description

When the PCD service sets a value, it will check to ensure that the size of the value being set is compatible with the Token's existing definition. If it is not, an error will be returned.

Status Codes Returned

EFI_SUCCESS	The PCD service has set the value requested
EFI_INVALID_PARAMETER	The PCD service determined that the size of the data being set was incompatible with a call to this function. Use **GetBool()** to retrieve the size of the target data.
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_PEI_PCD_PPI.CallbackOnSet ()

Summary

Specifies a function to be called anytime the value of a designated token is changed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_CALLBACK_ON_SET) (
  IN CONST EFI_GUID              *Guid, OPTIONAL
  IN UINTN                       CallBackToken,
  IN EFI_PEI_PCD_PPI_CALLBACK    CallBackFunction
  );
```

Parameters

Guid

The 128-bit unique value that designates which namespace to monitor. If **NULL**, use the standard platform namespace.

CallBackToken

The PCD token number to monitor.

CallBackFunction

The function prototype that will be called when the value associated with the *CallBackToken* is set.

Related Definitions

```
typedef
VOID
(EFIAPI * EFI_PEI_PCD_PPI_CALLBACK ) {
IN  EFI_GUID              *Guid, OPTIONAL,
IN  UINTN                 CallBackToken,
IN  OUT VOID              *TokenData,
IN  UINTN                 TokenDatSize
  };
```

Description

Specifies a function to be called anytime the value of a designated token is changed.

Status Codes Returned

EFI_SUCCESS	The PCD service has successfully established a call event for the *CallBackToken* requested.
EFI_NOT_FOUND	The PCD service could not find the referenced token number.

EFI_PEI_PCD_PPI.CancelCallback ()

Summary

Cancels a previously set callback function for a particular PCD token number.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_CANCEL_CALLBACK) (
  IN CONST EFI_GUID             *Guid, OPTIONAL
  IN UINTN                      CallBackToken,
  IN  EFI_PEI_PCD_PPI_CALLBACK  CallBackFunction
  );
```

Parameters

Guid

The 128-bit unique value that designates which namespace to monitor. If NULL, use the standard platform namespace.

CallBackToken

The PCD token number to cancel monitoring.

CallBackFunction

The function prototype that was originally passed to the *CallBackOnSet* function.

Description

Cancels a callback function that was set through a previous call to the *CallBackOnSet* function.

Status Codes Returned

EFI_SUCCESS	The PCD service has cancelled the call event associated with the *CallBackToken*.
EFI_INVALID_PARAMETER	The PCD service did not match the *CallBackFunction* to one that is currently being monitored.
EFI_NOT_FOUND	The PCD service could not find data the requested token number.

EFI_PEI_PCD_PPI.GetNextToken ()

Summary

Retrieves the next valid PCD token for a given namespace.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PPI_GET_NEXT_TOKEN) (
  IN CONST EFI_GUID          *Guid, OPTIONAL
  IN UINTN                   *TokenNumber
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

A pointer to the PCD token number to use to find the subsequent token number. To retrieve the “first” token, have the pointer reference a *TokenNumber* value of 0.

Description

This provides a means by which to get the next valid token number in a given namespace. This is useful since the PCD infrastructure has a sparse list of token numbers in it, and one cannot a priori know what token numbers are valid in the database.

Status Codes Returned

EFI_SUCCESS	The PCD service has retrieved the value requested
EFI_NOT_FOUND	The PCD service could not find data from the requested token number.

EFI_PEI_PCD_PPI.GetNextTokenSpace ()

Summary

Retrieves the next valid PCD token namespace for a given namespace.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_PCD_PROTOCOL_GET_NEXT_TOKEN_SPACE) (
  IN OUT CONST EFI_GUID          **Guid
  );
```

Parameters

Guid

An indirect pointer to *EFI_GUID*. On input it designates a known token namespace from which the search will start. On output, it designates the next valid token namespace on the platform. If **Guid* is **NULL**, then the GUID of the first token space of the current platform is returned. If the search cannot locate the next valid token namespace, an error is returned and the value of **Guid* is undefined.

Description

Gets the next valid token namespace for a given namespace. This is useful to traverse the valid token namespaces on a platform.

Status Codes Returned

EFI_SUCCESS	The PCD service retrieved the value requested.
EFI_NOT_FOUND	The PCD service could not find the next valid token namespace.

8.2.2 Get PCD Information PPI

EFI_GET_PCD_INFO_PPI

Summary

The PPI that provides additional information about items that reside in the PCD database.

GUID

```
#define EFI_GET_PCD_INFO_PPI_GUID \
  { 0xa60c6b59, 0xe459, 0x425d, \
  { 0x9c, 0x69, 0xb, 0xcc, 0x9c, 0xb2, 0x7d, 0x81 } }
```

Protocol Interface Structure

```
typedef struct _EFI_GET_PCD_INFO_PPI {
  EFI_GET_PCD_INFO_PPI_GET_INFO    GetInfo;
  EFI_GET_PCD_INFO_PPI_GET_SKU     GetSku;
} EFI_GET_PCD_INFO_PPI;
```

Parameters

GetInfo

Retrieve additional information associated with a PCD.

GetSku

Retrieve the currently set SKU Id.

Description

This is the PCD service to use when querying for some additional data that can be contained in the PCD database.

EFI_GET_PCD_INFO_PPI.GetInfo ()

Summary

Retrieve additional information associated with a PCD token.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_PCD_INFO_PPI_GET_INFO) (
  IN CONST EFI_GUID    *Guid,
  IN UINTN             TokenNumber,
  OUT EFI_PCD_INFO     *PcdInfo
  );
```

Parameters

Guid

The 128-bit unique value that designates the namespace from which to extract the value.

TokenNumber

The PCD token number.

PcdInfo

The returned information associated with the requested *TokenNumber*.

Description

GetInfo() retrieves additional information associated with a PCD token. This includes information such as the type of value the *TokenNumber* is associated with as well as possible human readable name that is associated with the token.

Status Codes Returned

EFI_SUCCESS	The PCD information was returned successfully
EFI_NOT_FOUND	The PCD service could not find the requested token number.

EFI_GET_PCD_INFO_PPI.GetSku ()

Summary

Retrieve the currently set SKU Id.

Prototype

```
typedef
UINTN
(EFIAPI *EFI_GET_PCD_INFO_PPI_GET_SKU) (
  VOID
  );
```

Description

GetSku() returns the currently set SKU Id. If the platform has not set at a SKU Id, then the default SKU Id value of 0 is returned. If the platform has set a SKU Id, then the currently set SKU Id is returned.

Platform Initialization (PI) Specification

Volume 4: Management Mode Core Interface

Version 1.7

January 2019

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

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

Revision History

Revision	**Mantis ID / Description**	**Date**
1.7	• 1848 PEI Core PEIM Migration Support Change • 1856 SecCore/PeiCore BFV Requirement Change • 1860 MM MP Protocol issues • 1885 Add extended data for EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR • 1889 New Status Codes • 1891 PEI delayed dispatch • 1892 new pi spec revision • 1955 set (*Attributes) when EFI_PEI_GET_VARIABLE2 returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL • 1967 New architectural PPI for PI PEI Core FV Location • 1969 Incompatible Pci Ignore Option Rom	January 2019
1.6 Errata A	• 1828 Add decorator 'OPTIONAL' for Attributes parameter of EFI_PEI_GET_VARIABLE2. • 1849 Issues in PI Spec Vol. 5 Ch. 18 • 1857 Specifies how notifications are passed from SEC to PEI. • 1884 Variadic API issue in S3 save API • 1907 Clarification of the EFI_MM_COMMUNICATION_PROTOCOL • 1940 Clarify EFI_MM_CONFIGURATION_PROTOCOL_GUID	December 2018
1.6	• 1567 Layered SPI bus • 1648 PI Binding for RISC-V • 1746 Add an FV Extended Header entry that contains the used size of the FV • 1763 MM Handler state notification protocol • 1764 Add additional alignment • 1768 Update the PI Spec to 1.6 • 1777 Update Revision History • 1778 Update front matter	April 2017
1.5 Errata A	• 1587 pre permanent memory page allocation • 1665 Incorrect status code for an AP calling EFI_MP_SERVICES_PROTOCOL.SwitchBSP() • 1734 Outdated EFI spec reference in Save State Write • 1735 Several copy & paste errors in Save State Write • 1747 Clarify that MM_ACCESS_PROTOCOL should cover all MMRAM region used by the platform	April 2017

Revision	Mantis ID / Description	Date
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 • 1593 coalesce language enhancements • 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 • 1607 Update MM guid def'n to match edkII impl • 1626 Add new Status Code for BDS Attempting UEFI BootOrder entries • 1628 Minor feedback for PI 1.5 Vol 4 SMM Draft • 1666 Graphics Device Info Hob	4/26/16
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

Revision	Mantis ID / Description	Date
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.	2/20/15
1.3 Errata A	• 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
1.3	• 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

Revision	Mantis ID / Description	Date
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 • 937SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 • 969Vol 1 errata: TE Header parameters	10/26/12
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
1.2.1	• 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

Revision	Mantis ID / Description	Date
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 • 737 Remove SMM Communication ACPI Table definition . • 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 • 753 SIO PEI and UEFI-Driver Model Architecture • 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

Revision	Mantis ID / Description	Date
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 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	• 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

Revision	Mantis ID / Description	Date
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	• 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

Revision	Mantis ID / Description	Date
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
1.1 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

Revision	Mantis ID / Description	Date
1.1 Errata	• 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	Revises typographical errors and minor omissions--see Errata for details	04/25/08
1.1 correction	Restore (missing) MP protocol	03/12/08
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

Revision	Mantis ID / Description	Date
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
1.0	Initial public release.	8/21/06

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.

Table of Contents

Revision History iii
Table of Contents xiii
List of Figures xxi
List of Tables xxii

1 Overview 1
1.1 Definition of Terms 1
1.2 Management Mode (MM) 2
1.3 MM Driver Execution Environment 2
1.4 Initializing Management Mode in MM Traditional Mode 3
1.4.1 SEC Initialization 4
1.4.2 PEI Initialization 4
1.4.3 DXE Initialization 4
1.5 Initializing Management Mode in MM Standalone Mode 6
1.5.1 Initializing MM Standalone Mode in PEI phase 6
1.5.2 Initializing MM Standalone Mode in SEC phase 8
1.6 Entering & Exiting MM 9
1.7 MM Traditional Drivers 9
1.7.1 MM Drivers 9
1.7.2 Combination MM/DXE Drivers 9
1.7.3 MM Standalone Drivers 10
1.7.4 MM_IMAGE_ENTRY_POINT 10
1.7.5 SOR and Dependency Expressions for SM 11
1.8 MM Traditional Driver Initialization 11
1.9 MM Standalone Driver Initialization 11
1.10 MM Traditional Driver Runtime 11
1.11 MM Standalone Driver Runtime 12
1.12 Dispatching MMI Handlers 12
1.13 MM Services 13
1.13.1 MM Driver Model 13
1.13.2 MM Protocols 13
1.14 MM UEFI Protocols 14
1.14.1 UEFI Protocols 14
1.14.2 MM Protocols 14

2 MM Foundation Entry Point 16
2.1 EFI_MM_ENTRY_POINT 16
2.2 MM_FOUNDATION_ENTRY_POINT 17

3 Management Mode System Table (MMST) 18
3.1 MMST Introduction 18
3.2 EFI_MM_SYSTEM_TABLE 18
MmInstallConfigurationTable() 24

MmAllocatePool() ... 26
MmFreePool() ... 27
MmAllocatePages() ... 28
MmFreePages() ... 29
MmStartupThisAp() ... 30
MmInstallProtocolInterface() ... 31
MmUninstallProtocolInterface() ... 32
MmHandleProtocol() ... 33
MmRegisterProtocolNotify() ... 34
MmLocateHandle() ... 36
MmLocateProtocol() ... 37
MmiManage() ... 38
MmiHandlerRegister() ... 40
MmiHandlerUnRegister() ... 43

4 MM Protocols ... 44

4.1 Introduction ... 44
4.2 Status Codes Services ... 44
EFI_MM_STATUS_CODE_PROTOCOL ... 44
EFI_MM_STATUS_CODE_PROTOCOL.ReportStatusCode() ... 45
4.3 CPU Save State Access Services ... 46
EFI_MM_CPU_PROTOCOL ... 46
EFI_MM_CPU_PROTOCOL.ReadSaveState() ... 48
AARCH32/AARCH64 REGISTER AVAILABILITY ... 54
EFI_MM_SAVE_STATE_ARM_CSR, EFI_MM_SAVE_STATE_AARCH64_CSR . 54
EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID ... 54
EFI_MM_SAVE_STATE_REGISTER_LMA ... 55
EFI_MM_CPU_PROTOCOL.WriteSaveState() ... 56
4.3.1 MM Save State IO Info ... 57
EFI_MM_SAVE_STATE_IO_INFO ... 57
4.4 MM CPU I/O Protocol ... 58
EFI_MM_CPU_IO_PROTOCOL ... 58
EFI_MM_CPU_IO_PROTOCOL.Mem() ... 60
EFI_MM_CPU_IO_PROTOCOL.Io() ... 62
4.5 MM PCI I/O Protocol ... 63
EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL ... 63
4.6 MM Ready to Lock Protocol ... 63
EFI_MM_READY_TO_LOCK_PROTOCOL ... 63
4.7 MM MP protocol ... 64
EFI_MM_MP_PROTOCOL ... 64
EFI_MM_MP_PROTOCOL.Revision ... 66
EFI_MM_MP_PROTOCOL.Attributes ... 66
EFI_MM_MP_PROTOCOL.GetNumberOfProcessors() ... 67
EFI_MM_MP_PROTOCOL.DispatchProcedure() ... 68
EFI_MM_MP_PROTOCOL.BroadcastProcedure() ... 70
EFI_MM_MP_PROTOCOL.SetStartupProcedure() ... 72

EFI_MM_MP_PROTOCOL.CheckOnProcedure() 73
EFI_MM_MP_PROTOCOL.WaitForProcedure() 74
4.8 MM Configuration Protocol 75
EFI_MM_CONFIGURATION_PROTOCOL 75
EFI_MM_CONFIGURATION_PROTOCOL.RegisterMmFoundationEntry() 76
4.9 MM End Of PEI Protocol 76
EFI_MM_END_OF_PEI_PROTOCOL 76
4.10 MM UEFI Ready Protocol 77
EFI_MM_UEFI_READY_PROTOCOL 77
4.11 MM Ready To Boot Protocol 77
EFI_MM_READY_TO_BOOT_PROTOCOL 77
4.12 MM Exit Boot Services Protocol 78
EFI_MM_EXIT_BOOT_SERVICES_PROTOCOL 78
4.13 MM Security Architecture Protocol 78
EFI_MM_SECURITY_ARCHITECTURE_PROTOCOL 78
4.14 MM End of DXE Protocol 79
EFI_MM_END_OF_DXE_PROTOCOL 79
4.15 MM Handler State Notification Protocol 79
EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL 79
EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL.
HandlerStateNotifierRegister 80
EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL.
HandlerStateNotifierUnregister 82
5 UEFI Protocols 85
5.1 Introduction 85
5.2 EFI MM Base Protocol 85
EFI_MM_BASE_PROTOCOL 85
EFI_MM_BASE_PROTOCOL.InMm() 87
EFI_MM_BASE_PROTOCOL.GetMmstLocation() 88
5.3 MM Access Protocol 88
EFI_MM_ACCESS_PROTOCOL 88
EFI_MM_ACCESS_PROTOCOL.Open() 90
EFI_MM_ACCESS_PROTOCOL.Close() 91
EFI_MM_ACCESS_PROTOCOL.Lock() 92
EFI_MM_ACCESS_PROTOCOL.GetCapabilities() 93
5.4 MM Control Protocol 95
EFI_MM_CONTROL_PROTOCOL 95
EFI_MM_CONTROL_PROTOCOL.Trigger() 97
EFI_MM_CONTROL_PROTOCOL.Clear() 99
5.5 MM Configuration Protocol 100
EFI_MM_CONFIGURATION_PROTOCOL 100
EFI_MM_CONFIGURATION_PROTOCOL.RegisterMmEntry() 102
5.6 DXE MM Ready to Lock Protocol 102
EFI_DXE_MM_READY_TO_LOCK_PROTOCOL 102
5.7 MM Communication Protocol 103
EFI_MM_COMMUNICATION_PROTOCOL 103

EFI_MM_COMMUNICATION_PROTOCOL.Communicate() 105

6 PI PEI PPIs.. 108

6.1 MM Access PPI.. 108

EFI_PEI_MM_ACCESS_PPI 108

EFI_PEI_MM_ACCESS_PPI.Open() 110

EFI_PEI_MM_ACCESS_PPI.Close().. 111

EFI_PEI_MM_ACCESS_PPI.Lock() 112

EFI_PEI_MM_ACCESS_PPI.GetCapabilities()....................................... 113

6.2 MM Control PPI... 114

EFI_PEI_MM_CONTROL_PPI.Trigger()... 116

EFI_PEI_MM_CONTROL_PPI.Clear()... 118

6.3 EFI DELAYED DISPATCH PPI (Required)... 119

EFI_DELAYED_DISPATCH_PPI (Required).. 119

EFI_DELAYED_DISPATCH_PPI.Register() 120

6.4 MM Configuration PPI.. 121

EFI_PEI_MM_CONFIGURATION_PPI 121

EFI_PEI_MM_CONFIGURATION_PPI.RegisterMmEntry()..................... 122

6.5 MM Communication PPI 123

EFI_PEI_MM_COMMUNICATION_PPI 123

EFI_PEI_MM_COMMUNICATION_PPI.Communicate() 124

7 MM Child Dispatch Protocols.. 125

7.1 Introduction 125

7.2 MM Software Dispatch Protocol.. 125

EFI_MM_SW_DISPATCH_PROTOCOL 125

EFI_MM_SW_DISPATCH_PROTOCOL.Register() 127

EFI_MM_SW_DISPATCH_PROTOCOL.UnRegister() 130

7.3 MM Sx Dispatch Protocol.. 130

EFI_MM_SX_DISPATCH_PROTOCOL 130

EFI_MM_SX_DISPATCH_PROTOCOL.Register() 132

EFI_MM_SX_DISPATCH_PROTOCOL.UnRegister() 134

7.4 MM Periodic Timer Dispatch Protocol... 134

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL 134

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL.Register() 136

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL.UnRegister() ... 139

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL.
GetNextShorterInterval() 140

7.5 MM USB Dispatch Protocol 140

EFI_MM_USB_DISPATCH_PROTOCOL 140

EFI_MM_USB_DISPATCH_PROTOCOL.Register() 142

EFI_MM_USB_DISPATCH_PROTOCOL.UnRegister() 144

7.6 MM General Purpose Input (GPI) Dispatch Protocol............................... 144

EFI_MM_GPI_DISPATCH_PROTOCOL 144

EFI_MM_GPI_DISPATCH_PROTOCOL.Register() 146

EFI_MM_GPI_DISPATCH_PROTOCOL.UnRegister() 148

7.7 MM Standby Button Dispatch Protocol 148

EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL..................... 148

EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL.Register() 150
EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL.UnRegister() 152
7.8 MM Power Button Dispatch Protocol 152
EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL 152
EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL. Register() 154
EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL.UnRegister() 156
7.9 MM IO Trap Dispatch Protocol.. 156
EFI_MM_IO_TRAP_DISPATCH_PROTOCOL 156
EFI_MM_IO_TRAP_DISPATCH_PROTOCOL.Register () 158
EFI_MM_IO_TRAP_DISPATCH_PROTOCOL.UnRegister ()........................... 160
7.10 HOBs 160
EFI_PEI_MM_CORE_GUID.. 160
8 Interactions with PEI, DXE, and BDS..162
8.1 Introduction 162
8.2 MM and DXE... 162
8.2.1 Software MMI Communication Interface (Method #1)...................................... 162
8.2.2 Software MMI Communication Interface (Method #2)...................................... 162
8.3 MM and PEI 163
8.3.1 Software MMI Communication Interface (Method #1)...................................... 163
9 Other Related Notes For Support Of MM Drivers....................................164
9.1 File Types 164
9.1.1 File Type EFI_FV_FILETYPE_MM 164
9.1.2 File Type EFI_FV_FILETYPE_COMBINED_MM_DXE 164
9.2 File Type EFI_FV_FILETYPE_MM_STANDALONE 165
9.3 File Section Types 165
9.3.1 File Section Type EFI_SECTION_MM_DEPEX.. 165
10 MCA/INIT/PMI Protocol 166
10.1 Machine Check and INIT 166
10.2 MCA Handling... 168
10.3 INIT Handling.. 170
10.4 PMI.. 171
10.5 Event Handlers 172
10.5.1 MCA Handlers.. 172
MCA Handler.. 172
10.5.2 INIT Handlers... 173
INIT Handler 173
10.5.3 PMI Handlers 174
PMI Handler 174
10.6 MCA PMI INIT Protocol... 174
EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterMcaHandler () 176
EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterInitHandler () 177
EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterPmiHandler ()..................... 178
11 Extended SAL Services 179
11.1 SAL Overview 179

11.2 Extended SAL Boot Service Protocol 181
EXTENDED_SAL_BOOT_SERVICE_PROTOCOL 181
EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableInfo() 183
EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableEntry() ... 185
EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddExtendedSalProc() 186
EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.ExtendedSalProc().......... 189
11.3 Extended SAL Service Classes 190
11.3.1 Extended SAL Base I/O Services Class 192
ExtendedSalIoRead 193
ExtendedSalIoWrite........ 195
ExtendedSalMemRead 197
ExtendedSalMemWrite........ 199
11.4 Extended SAL Stall Services Class 200
ExtendedSalStall 202
11.4.1 Extended SAL Real Time Clock Services Class 203
ExtendedSalGetTime 205
ExtendedSalSetTime........ 207
ExtendedSalGetWakeupTime 209
ExtendedSalSetWakeupTime 211
11.4.2 Extended SAL Reset Services Class 212
ExtendedSalResetSystem........ 214
11.4.3 Extended SAL PCI Services Class 215
ExtendedSalPciRead 217
ExtendedSalPciWrite........ 219
11.4.4 Extended SAL Cache Services Class 220
ExtendedSalCacheInit........ 221
ExtendedSalCacheFlush........ 223
11.4.5 Extended SAL PAL Services Class........ 224
ExtendedSalPalProc 225
ExtendedSalSetNewPalEntry........ 227
ExtendedSalGetNewPalEntry 229
ExtendedSalUpdatePal 231
11.4.6 Extended SAL Status Code Services Class........ 232
ExtendedSalReportStatusCode 233
11.4.7 Extended SAL Monotonic Counter Services Class 234
ExtendedSalGetNextHighMtc........ 236
11.4.8 Extended SAL Variable Services Class 237
ExtendedSalGetVariable 239
ExtendedSalGetNextVariableName 241
ExtendedSalSetVariable 243
ExtendedSalQueryVariableInfo 245
11.4.9 Extended SAL Firmware Volume Block Services Class 246
ExtendedSalRead 249
ExtendedSalWrite........ 251
ExtendedSalEraseBlock........ 253
ExtendedSalGetAttributes 255

ExtendedSalSetAttributes 257
ExtendedSalGetPhysicalAddress........ 259
ExtendedSalGetBlockSize 261
ExtendedSalEraseCustomBlockRange........ 263
11.4.10 Extended SAL MCA Log Services Class 264
ExtendedSalGetStateInfo........ 266
ExtendedSalGetStateInfoSize........ 268
ExtendedSalClearStateInfo........ 270
ExtendedSalGetStateBuffer 272
ExtendedSalSaveStateBuffer........ 274
11.4.11 Extended SAL Base Services Class 275
ExtendedSalSetVectors 278
ExtendedSalMcRendez........ 280
ExtendedSalMcSetParams 282
ExtendedSalGetVectors........ 284
ExtendedSalMcGetParams........ 286
ExtendedSalMcGetMcParams 288
ExtendedSalGetMcCheckinFlags........ 290
ExtendedSalGetPlatformBaseFreq 292
ExtendedSalRegisterPhysicalAddr........ 294
11.4.12 Extended SAL MP Services Class........ 295
ExtendedSalAddCpuData 297
ExtendedSalRemoveCpuData 299
ExtendedSalModifyCpuData 301
ExtendedSalGetCpuDataById........ 303
ExtendedSalGetCpuDataByIndex........ 305
ExtendedSalWhoiAmI 307
ExtendedSalNumProcessors 309
ExtendedSalSetMinState 311
ExtendedSalGetMinState........ 313
ExtendedSalPhysicalIdInfo........ 315
11.4.13 Extended SAL MCA Services Class 316
ExtendedSalMcaGetStateInfo........ 317
ExtendedSalMcaRegisterCpu 319

12 SMM SPI Protocol Stack 321

12.1 Design........ 321
12.2 SMM SPI Protocols........ 321
EFI_LEGACY _SPI_SMM_FLASH_PROTOCOL GUID 321
EFI_SPI_SMM_NOR_FLASH_PROTOCOL GUID........ 321
EFI- SPI- SMM- CONFIGURATION PROTOCOL GUID........ 322
EFI- SPI- SMM HC- PROTOCOL GUID........ 322
EFI LEGACY- SPI- SMM- CONTROLLER- PROTOCOL GUID 322

Appendix A
Management Mode Backward Compatibility Types 323

EFI_SMM_CONFIGURATION_PROTOCOL........ 329
EFI_SMM_CAPABILITIES2 329

EFI_SMM_INSIDE_OUT2 ... 329
EFI_SMM_SW_CONTEXT ... 330
EFI_SMM_SW_REGISTER_CONTEXT ... 330
EFI_SMM_PERIODIC_TIMER_REGISTER_CONTEXT ... 330
EFI_SMM_SAVE_STATE_IO_WIDTH ... 330
EFI_SMM_IO_WIDTH ... 331

List of Figures

Figure 1. MM Architecture 3
Figure 2. Example MM Initialization Components 6
Figure 3. MMI Handler Relationships 13
Figure 4. Published Protocols for IA-32 Systems 15
Figure 5. Early Reset, MCA and INIT flow 167
Figure 6. Basic MCA processing flow 168
Figure 7. PI MCA processing flow 168
Figure 8. PI architectural data in the min-state 169
Figure 9. PI INIT processing flow 171
Figure 10. PMI handling flow 171
Figure 11. SAL Calling Diagram 180

List of Tables

Table 1. Extended SAL Service Classes – EFI Runtime Services 191
Table 2. Extended SAL Service Classes – SAL Procedures 191
Table 3. Extended SAL Service Classes – Hardware Abstractions.................................... 191
Table 4. Extended SAL Service Classes – Other 191
Table 5. Extended SAL Base I/O Services Class 192
Table 6. Extended SAL Stall Services Class 201
Table 7. Extended SAL Real Time Clock Services Class.. 204
Table 8. Extended SAL Reset Services Class.. 213
Table 9. Extended SAL PCI Services Class 216
Table 10. Extended SAL Cache Services Class... 220
Table 11. Extended SAL PAL Services Class 224
Table 12. Extended SAL Status Code Services Class 232
Table 13. Extended SAL Monotonic Counter Services Class.. 235
Table 14. Extended SAL Variable Services Class 238
Table 15. Extended SAL Variable Services Class 247
Table 16. Extended SAL MP Services Class... 276
Table 17. Extended SAL MP Services Class... 295
Table 18. Extended SAL MCA Services Class 316

1 Overview

1.1 Definition of Terms

The following terms are used in the MM Core Interface Specification (CIS). See Glossary in the master help system for additional definitions.

IP

Instruction pointer.

IPI

Interprocessor Interrupt. This interrupt is the means by which multiple processors in a system or a single processor can issue APIC-directed messages for communicating with self or other processors.

MM

Management Mode. Generic term for a secure, isolated execution environment entered when a CPU core detects an MMI and jumps to the MM Entry Point within MMRAM. This can be implemented by System Management Mode on x86 processors and TrustZone on ARM processors.

MM Driver

A driver launched directly into MMRAM, with access to the MM interfaces.

MM Driver Initialization

The phase of MM Driver initialization which starts with the call to the driver's entry point and ends with the return from the driver's entry point.

MM Driver Runtime

The phase of MM Driver initialization which starts after the return from the driver's entry point.

MM Entry Point

When the CPU core(s) enter MM, they begin execution at a pre-defined addresses in a pre-defined operating mode. At some point later, they jump into the MM Foundation entry point.

MM handler

A DXE driver that is loaded into and executed from MMRAM. MM Handlers are dispatched during boot services time and invoked synchronously or asynchronously thereafter. MM handlers remain present during runtime.

MMI

Management Mode Invocation.The CPU instruction or high-priority interrupt which transitions CPU core(s) into MM via the MM Entry Point.

MMI Source.

The instruction, interrupt or exception which caused the CPU core(s) to enter MM. An MMI source can be detected, quiesced and disabled.

MMST

Management Mode System Table. Hand-off to handler.

MTRR

Memory Type Range Register.

RSM

Resume. The process by which a CPU exits MM.

1.2 Management Mode (MM)

Management Mode (MM) is a generic term used to describe a secure execution environment provided by the CPU and related silicon that is entered when the CPU detects a MMI. For x86 systems, this can be implemented with System Management Mode (SMM). For ARM systems, this can be implemented with TrustZone (TZ).

A MMI can be a CPU instruction or interrupt. Upon detection of a MMI, a CPU will jump to the MM Entry Point and save some portion of its state (the "save state") such that execution can be resumed.

The MMI can be generated synchronously by software or asynchronously by a hardware event. Each MMI source can be detected, cleared and disabled.

Some systems provide for special memory (Management Mode RAM or MMRAM) which is set aside for software running in MM. Usually the MMRAM is hidden during normal CPU execution, but this is not required. Usually, after MMRAM is hidden it cannot be exposed until the next system reset.

1.3 MM Driver Execution Environment

The MM Core Interface Specification describes the optional MM environment, which exists in parallel with the other PI Architecture phases into runtime.

The MM Core Interface Specification describes three pieces of the PI Management Mode architecture:

MM Dispatch

During DXE, the DXE Foundation works with the MM Foundation to schedule MM drivers for execution in the discovered firmware volumes.

MM Initialization

MM related code opens MMRAM, creates the MMRAM memory map, and launches the MM Foundation, which provides the necessary services to launch MM-related drivers. Then, sometime before boot, MMRAM is closed and locked. This piece may be completed during the SEC, PEI or DXE phases.

MMI Management

When an MMI generated, the MM environment is created and then the MMI sources are detected and MMI handlers called.

The figure below shows the MM architecture.

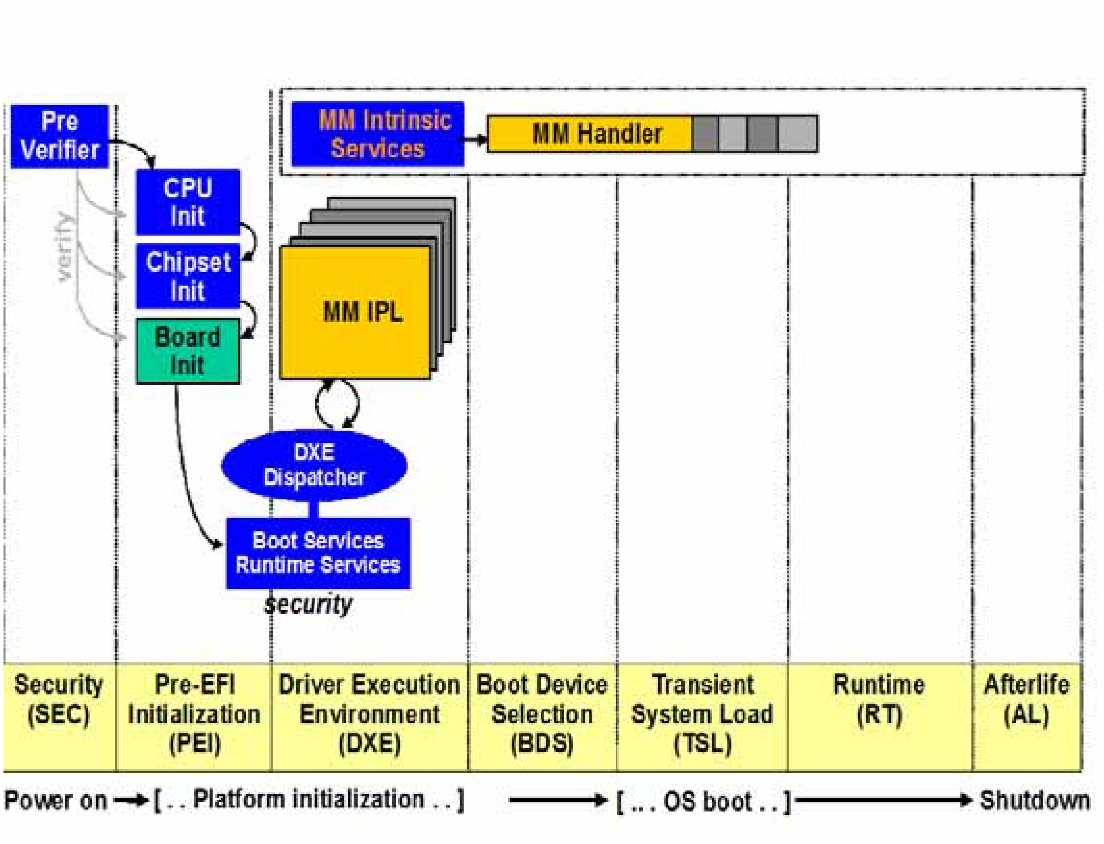

Pre Verifier
verify
CPU Init
Chipset Init
Board Init
MM Intrinsic Services
MM Handler
MM IPL
DXE Dispatcher
Boot Services Runtime Services
security
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Runtime (RT)
Afterlife (AL)
Power on → [. . Platform initialization . .] → [... OS boot . .] → Shutdown

Figure 1. MM Architecture

Note: *The MM architecture does not guarantee support for the execution of handlers written to the EFI Byte Code (EBC) specification.*

1.4 Initializing Management Mode in MM Traditional Mode

Management Mode initialization prepares the hardware for MMI generation and creates the necessary data structures for managing the MM resources such as MMRAM.

This specification supports three MM initialization models: SEC, PEI and DXE. This specification does not describe MM Dispatch or MMI handling during SEC or PEI. Previous versions of this specification only supported DXE Initialization.

1.4.1 SEC Initialization

In this model, the MM Entry Points are initialized and the MM Foundation is loaded into MMRAM during the SEC phase. Optionally, MMRAM is hidden and locked. Then, during the DXE phase, MM or MM/DXE drivers are loaded normally. This is detailed in the following steps:

1. The SEC code initializes the MM environment, including initializing the MM Entry Points, setting up MMRAM, initializing the MM Foundation in MMRAM.
2. Optionally, the SEC code hides and locks the MMRAM.
3. The SEC code produces the **EFI_SEC_HOB_DATA_PPI**, which produces a HOB with the GUID **EFI_PEI_MM_CORE_GUID**, and the **EFI_PEI_MM_CORE_LOADED** flag set which indicates that the MM Foundation is already installed.

After this, the steps follow those in DXE initialization. There is not architectural provision for loading MM-related drivers during the SEC phase.

1.4.2 PEI Initialization

In this model, the MM Entry Points are initialized and the MM Foundation is loaded into MMRAM during the PEI phase. Optionally, MMRAM is hidden and locked. Then, during the DXE phase, MM or MM/DXE drivers are loaded normally. This is detailed in the following steps:

1. The PEI code initializes the MM environment, including initializing the MM Entry Points, setting up MMRAM and initializing the MM Foundation in MMRAM.
2. Optionally, the PEI code hides and locks the MMRAM.
3. The PEI code produces the HOB with the GUID **EFI_PEI_MM_CORE_GUID**, and the **EFI_PEI_MM_CORE_LOADED** flag set, which indicates that the MM Foundation has already been installed.

After this, the steps follow those in DXE initialization. There is not architectural provision for loading MM-related drivers during the PEI phase.

1.4.3 DXE Initialization

It is initialized with the cooperation of several DXE drivers.

1. A DXE driver produces the **EFI_MM_ACCESS_PROTOCOL**, which describes the different MMRAM regions available in the system.
2. A DXE driver produces the **EFI_MM_CONTROL_PROTOCOL**, which allows synchronous MMIs to be generated.
3. A DXE driver (dependent on the **EFI_MM_ACCESS_PROTOCOL** and, perhaps, the **EFI_MM_CONTROL_PROTOCOL**), does the following:
   - If the **MM_CORE_LOADED** flag is not set in the **EFI_PEI_MM_CORE_GUID** HOB was not set, initializes the MM entry vector with the code necessary to meet the entry point requirements described in "Entering & Exiting MM".
   - If the **MM_CORE_LOADED** flag is not set in the **EFI_PEI_MM_CORE_GUID** HOB or that HOB does not exist, then produces the **EFI_MM_CONFIGURATION _PROTOCOL**, which describes those areas of MMRAM which should be excluded from the memory map.
   - NOTE: This implies that this DXE driver is completely optional if the **MM_CORE_LOADED** flag is set in the **EFI_PEI_MM_CORE_GUID** HOB.

4. The MM IPL DXE driver (dependent on the **EFI_MM_CONTROL_PROTOCOL**) does the following:
   - If **MM_CORE_LOADED** flags is set in the **EFI_PEI_MM_CORE_GUID** HOB, register for notification of the installation of the **EFI_MM_ACCESS_PROTOCOL** and the **EFI_MM_CONFIGURATION_PROTOCOL**. Once both are available, opens MMRAM and:
   - Creates the MMRAM heap, excluding any areas listed in **EFI_MM_CONFIGURATION_PROTOCOL** *MmramReservedRegions* field.
   - Loads the MM Foundation into MMRAM. The MM Foundation produces the MMST.
   - Invokes the **EFI_MM_CONFIGURATION_PROTOCOL***.RegisterMmEntry()* function with the MM Foundation entry point.
   - Publishes the **EFI_MM_BASE_PROTOCOL** in the UEFI Protocol Database
   - At this point MM is initially configured and MMIs can be generated.
   - Call the **Communicate()** member of the **EFI_MM_COMMUNICATION_PROTOCOL** with a buffer containing the **EFI_MM_INITIALIAZATION_HEADER** and the pointer to the UEFI System Table in the communication buffer. This gives the MM Core access to the UEFI Boot Services. Before this point, the MM Core must not use any UEFI services or protocols. NOTE: It also implies that the MM Core cannot find or dispatch any MM drivers from firmware volumes, since access to UEFI Boot Services is required to find instances for the Firmware Volume protocols.
   - Register for notification upon installation of the **EFI_DXE_MM_READY_TO_LOCK_PROTOCOL** in the UEFI protocol database.
5. During the remainder of the DXE phase, additional drivers may load and be initialized in MMRAM.
6. At some point prior to the processing of boot options, a DXE driver will install the **EFI_DXE_MM_READY_TO_LOCK_PROTOCOL** protocol in the UEFI protocol database. (outside of MM).
7. As a result, some DXE driver will cause the **EFI_MM_READY_TO_LOCK_PROTOCOL** protocol to be installed in the SM protocol database.
   - Optionally, close the MMRAM so that it is no longer visible using the **EFI_MM_ACCESS_PROTOCOL**. Closing MMRAM may not be supported on all platforms.
   - Optionally, lock the MMRAM so that its configuration can no longer be altered using the **EFI_MM_ACCESS_PROTCOL**. Locking MMRAM may not be supported on all platforms.

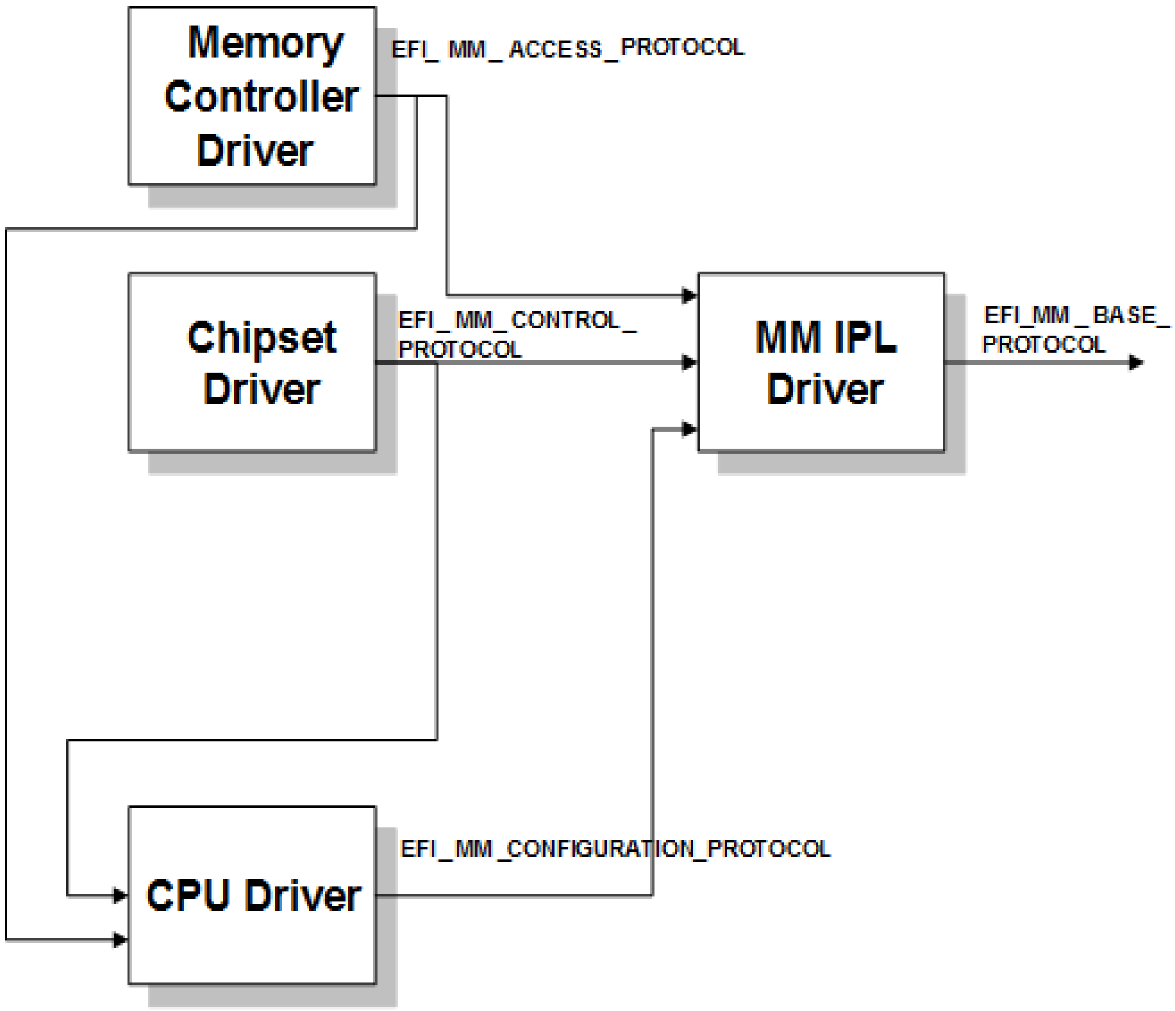


Figure 2. Example MM Initialization Components

1.5 Initializing Management Mode in MM Standalone Mode

1.5.1 Initializing MM Standalone Mode in PEI phase

Management Mode initialization prepares the hardware for MMI generation and creates the necessary data structures for managing the MM resources such as MMRAM. It is initialized with the cooperation of several DXE driver or PEIMs. Details below:

1. A PEIM produces the **EFI_PEI_MM_ACCESS_PPI**, which describes the different MMRAM regions available in the system.
2. A PEIM produces the **EFI_PEI_MM_CONTROL_PPI**, which allows synchronous MMIs to be generated.

3. A PEIM (dependent on the **EFI_PEI_MM_ACCESS_PPI** and, perhaps, the **EFI_PEI_MM_CONTROL_PPI**), does the following:
   - Initializes the MM entry vector with the code necessary to meet the entry point requirements described in "Entering & Exiting MM".
   - Produces the **EFI_MM_CONFIGURATION_PPI**, which describes those areas of MMRAM which should be excluded from the memory map.
   - The MM IPL PEIM (dependent on the **EFI_PEI_MM_ACCESS_PPI**, **EFI_PEI_MM_CONTROL_PPI** and **EFI_PEI_MM_CONFIGURATION_PPI**) does the following:
   - Opens MMRAM.
   - Creates the MMRAM heap, excluding any areas listed in **EFI_PEI_MM_CONFIGURATION_PPI** *MmramReservedRegions* field.
   - Loads the MM Foundation into MMRAM. The MM Foundation produces the MMST.
   - Invokes the **EFI_PEI_MM_CONFIGURATION_PPI.***RegisterMmEntry()* function with the MM Foundation entry point.
   - At this point MM is initially configured and MMIs can be generated.
   - •Publishes the **EFI_PEI_MM_COMMUNICATION_PPI**
4. During the remainder of the PEI phase, additional MM standalone drivers may load and be initialized in MMRAM.
5. During the remainder of the DXE phase, additional MM standalone drivers may load and be initialized in MMRAM.
6. A special MM IPL DXE driver does the following:
   - Communicate with MM Foundation and tell **EFI_SYSTEM_TABLE** pointer.
   - Publishes the **EFI_MM_BASE_PROTOCOL** in the UEFI Protocol Database
   - Publishes the **EFI_MM_COMMUNICATION_PROTOCOL** in the UEFI Protocol Database
7. During the remainder of the DXE phase, additional MM Traditional drivers may load and be initialized in MMRAM.
8. At some point prior to the processing of boot options, a DXE driver will install the **EFI_DXE_MM_READY_TO_LOCK_PROTOCOL** protocol in the UEFI protocol database. (outside of MM).
9. As a result, some DXE driver will cause the **EFI_MM_READY_TO_LOCK_PROTOCOL** protocol to be installed in the MM protocol database.
   - Optionally, close the MMRAM so that it is no longer visible using the **EFI_MM_ACCESS_PROTOCOL**. Closing MMRAM may not be supported on all platforms.
   - Optionally, lock the MMRAM so that its configuration can no longer be altered using the **EFI_MM_ACCESS_PROTCOL**. Locking MMRAM may not be supported on all platforms.

Note: *In order to support both MM standalone driver and MM traditional driver, the MM Foundation must have same calling convention as DXE phase, instead of PEI phase. It means, if PEI phase is 32bit,* *DXE phase is 64bit, then the MM Foundation must be 64bit. The 32bit MM IPL PEIM must have ability to launch 64bit MM Foundation.*

1.5.2 Initializing MM Standalone Mode in SEC phase

Standalone Mode can also be initialized in SEC phase. We take SEC phase initialization as example for MM Standalone Mode. Detail below:

1. SEC does the following:
   - Initializes the MM entry vector with the code necessary to meet the entry point requirements described in "Entering & Exiting MM".
   - Opens MMRAM.
   - Creates the MMRAM heap.
   - Loads the MM Foundation into MMRAM. The MM Foundation produces the MMST.
   - Invokes the **RegisterMmEntry()** function with the MM Foundation entry point.
   - At this point MM is initially configured and MMIs can be generated.
   - Optionally, closes MMRAM so that it is no longer visible.
   - Optionally, locks MMRAM so that its configuration can no longer be altered.
2. Then SEC Core can load PEI core as normal process.
3. A special MM IPL PEIM does the following:
   - Publishes the **EFI_PEI_MM_COMMUNICATION_PPI**
4. During the remainder of the PEI phase, additional MM standalone drivers may load and be initialized in MMRAM.
5. During the remainder of the DXE phase, additional MM standalone drivers may load and be initialized in MMRAM.
6. A special MM IPL DXE driver does the following:
   - Communicate with MM Foundation and tell **EFI_SYSTEM_TABLE** pointer.
   - Publishes the **EFI_MM_BASE_PROTOCOL** in the UEFI Protocol Database
   - Publishes the **EFI_MM_COMMUNICATION_PROTOCOL** in the UEFI Protocol Database
7. During the remainder of the DXE phase, additional MM traditional drivers may load and be initialized in MMRAM.
8. At some point prior to the processing of boot options, a DXE driver will install the **EFI_DXE_MM_READY_TO_LOCK_PROTOCOL** protocol in the UEFI protocol database. (outside of MM).
9. As a result, some DXE driver will cause the **EFI_MM_READY_TO_LOCK_PROTOCOL** protocol to be installed in the MM protocol database.

Note: *In order to support both MM standalone driver and MM traditional driver, the MM Foundation must have same calling convention as DXE phase, instead of SEC phase. It means, if SEC phase is*

32bit, DXE phase is 64bit, then the MM Foundation must be 64bit. The 32bit SEC must have ability to launch 64bit MM Foundation.

1.6 Entering & Exiting MM

The code at the entry vector must:

- Save any CPU state necessary for supporting the **EFI_MM_CPU_PROTOCOL**
- Save any CPU state so that the normal operation can be resumed.
- Select a single CPU to enter the MM Foundation.
- If an entry point has been registered via **RegisterMmEntry()**, switch to the same CPU mode as the MM Foundation and call the MM Foundation entry point.

The MM Foundation entry point must:

- Update the MMST with the CPU information passed to the entry point.
- Call all root MMI controller handlers using **MmiManage** (**NULL**)
- Return to the entry vector code.

After returning from the MM Foundation entry point, the code at the entry vector must:

- Restore any CPU state information necessary for normal operation.
- Resume normal operation

1.7 MM Traditional Drivers

There are two types of SM-related drivers: MM Drivers and Combination SM/DXE Drivers. Both types of drivers are initialized by calling their main entry point.

The entry point of the driver is the same as a *UEFI Specification* **EFI_IMAGE_ENTRY_POINT**.

1.7.1 MM Drivers

MM Drivers must have the file type **EFI_FV_FILETYPE_MM**. MM Drivers are launched once, directly into MMRAM in MM Traditional Mode. MM Drivers cannot be launched until the dependency expression in the file section **EFI_SECTION_MM_DEPEX** evaluates to true. This dependency expression can refer to both UEFI and SM protocols.

The entry point of the driver is the same as a *UEFI Specification* **EFI_IMAGE_ENTRY_POINT**.

1.7.2 Combination MM/DXE Drivers

Combination MM/DXE Drivers must have the file type **EFI_FV_FILETYPE_COMBINED_MM_DXE**. Combination Drivers are launched twice.

They are launched by the DXE Dispatcher as a normal DXE driver outside of MMRAM in MM Tradtional Mode after the dependency expression in the file section **EFI_SECTION_DXE_DEPEX** evaluates to true. As DXE Drivers, they have access to the normal UEFI interfaces.

Combination Drivers are also launched as MM Drivers inside of MMRAM after the dependency expression in the file section **EFI_SECTION_MM_DEPEX** evaluates to true. Combination Drivers have access to DXE, UEFI and SM services during MM Initialization. Combination Drivers have access to MM services during MM Runtime.

Combination Drivers can determine whether or not they are executing during MM Initialization or MM Runtime by locating the **EFI_MM_READY_TO_LOCK_MM_PROTOCOL**.

On the first load, the entry point of the driver is the same as a UEFI specification **EFI_IMAGE_ENTRY_POINT** since the driver is loaded by the DXE core.

On the second load, the entry point of the driver is the same as a *UEFI Specification* **EFI_IMAGE_ENTRY_POINT**.

1.7.3 MM Standalone Drivers

MM Standalone Drivers must have the file type **EFI_FV_FILETYPE_MM_STANDALONE**. MM Standalone Drivers are launched once, directly into MMRAM. MM Standalone Drivers cannot be launched until the dependency expression in the file section **EFI_SECTION_MM_DEPEX** evaluates to true. This dependency expression must refer to MM protocols.

The entry point of the driver is defined below as **MM_IMAGE_ENTRY_POINT**.

1.7.4 MM_IMAGE_ENTRY_POINT

Summary

This function is the main entry point to an MM Standalone Driver.

Prototype

```
typedef
VOID
(EFIAPI *MM_IMAGE_ENTRY_POINT) (
IN EFI_HANDLE              ImageHandle,
IN EFI_MM_SYSTEM_TABLE     *MmSystemTable
);
```

Parameters

ImageHandle

The handle allocated for the MM Standalone Driver.

MmSystemTable

A pointer to the MM System Table.

Description

This function is the entry point to an MM Standalone Driver. An MM Standalone Driver is loaded and relocated into MMRAM by MM Foundation. The first argument is the image's image handle. The second argument is a pointer to the MM system table.

1.7.5 SOR and Dependency Expressions for SM

The Apriori file can also contain DXE and SM FFS files. The implementation doesn't support SOR for the MM drivers, though.

1.8 MM Traditional Driver Initialization

An MM Driver's initialization phase begins when the driver has been loaded into MMRAM in MM Traditional Mode and its entry point is called. An MM Driver's initialization phase ends when the entry point returns.

During MM Driver initialization, MM Drivers have access to two sets of protocols: UEFI and SM. UEFI protocols are those which are installed and discovered using the UEFI Boot Services. UEFI protocols can be located and used by MM drivers only during MM Initialization. SM protocols are those which are installed and discovered using the Management Mode Services Table (MMST). SM protocols can be discovered by MM drivers during initialization time and accessed while inside of SM.

MM Drivers shall not use the following UEFI Boot Services during MM Driver Initialization:

- **`Exit()`**
- **`ExitBootServices()`**

1.9 MM Standalone Driver Initialization

An MM Standalone Driver's initialization phase begins when the driver has been loaded into MMRAM in MM Standalone Mode and its entry point is called. An MM Standalone Driver's initialization phase ends when the entry point returns.

During MM Standalone Driver initialization, MM Standalone Drivers can only access MM protocols. MM protocols are those which are installed and discovered using the Management Mode Services Table (MMST). MM protocols can be discovered by MM Drivers during initialization time and accessed while inside of MM.

1.10 MM Traditional Driver Runtime

During MM Driver runtime, MM Drivers only have access to MM protocols. In addition, depending on the platform architecture, memory areas outside of MMRAM may not be accessible to MM Drivers. Likewise, memory areas inside of MMRAM may not be accessible to UEFI drivers.

These MM Driver Runtime characteristics lead to several restrictions regarding the usage of UEFI services:

- UEFI interfaces and services which are located during MM Driver Initialization should not be called or referenced during MM Driver Runtime. This includes the EFI System Table, the UEFI Boot Services and the UEFI Runtime Services.
- Installed UEFI protocols should be uninstalled before exiting the driver entry point, or the UEFI protocol should refer to addresses which are not within MMRAM.

- Events created during MM Driver Initialization should be closed before exiting the drier entry point.

1.11 MM Standalone Driver Runtime

During MM Standalone Driver runtime, MM drivers only have access to MM protocols. In addition, depending on the platform architecture, memory areas outside of MMRAM may not be accessible to MM Drivers.

1.12 Dispatching MMI Handlers

MMI handlers are registered using the MMST's **`MmiHandlerRegister()`** function. MMI handlers fall into three categories:

RootMMI Controller Handlers

These are handlers for devices which directly control MMI generation for the CPU(s). The handlers have the ability to detect, clear and disable one or more MMI sources. They are registered by calling **`MmiHandlerRegister()`** with *`HandlerType`* set to NULL. After an MMI source has been detected, the Root MMI handler calls the Child MMI Controllers or MMI Handlers whose handler functions were registered using either an MM Child Dispatch protocols or using **`MmiHandlerRegister()`**. To call the latter, it calls **`Manage()`** with a GUID identifying the MMI source so that any registered Child MMI Handlers or Leaf MMI Handlers will be called. If the handler returns **`EFI_INTERRUPT_PENDING`**, it indicates that the interrupt source could not be quiesced. If possible, the Root MMI handler should disable and clear the MMI source. If the handler does not return an error, the Root MMI Handler should clear the MMI source.

Child MMI Controller Handlers

These are MMI handlers which handle a single interrupt source from a Root or Child MMI handler and, in turn, control one or more child MMI sources which can be detected, cleared and disabled. They are registered by calling the **`MmiHandlerRegister()`** function with *`HandlerType`* set to the GUID of the Parent MMI Controller MMI source. Handlers for this MMI handler's MMI sources are called in the same manner as Root MMI Handlers.

MMI Handlers

These MMI handlers perform basic software or hardware services based on the MMI source received. If the MMI handler manages a device outside the control of the Parent MMI Controller, it must make sure that the device is quiesced, especially if the device drives a level-active input.

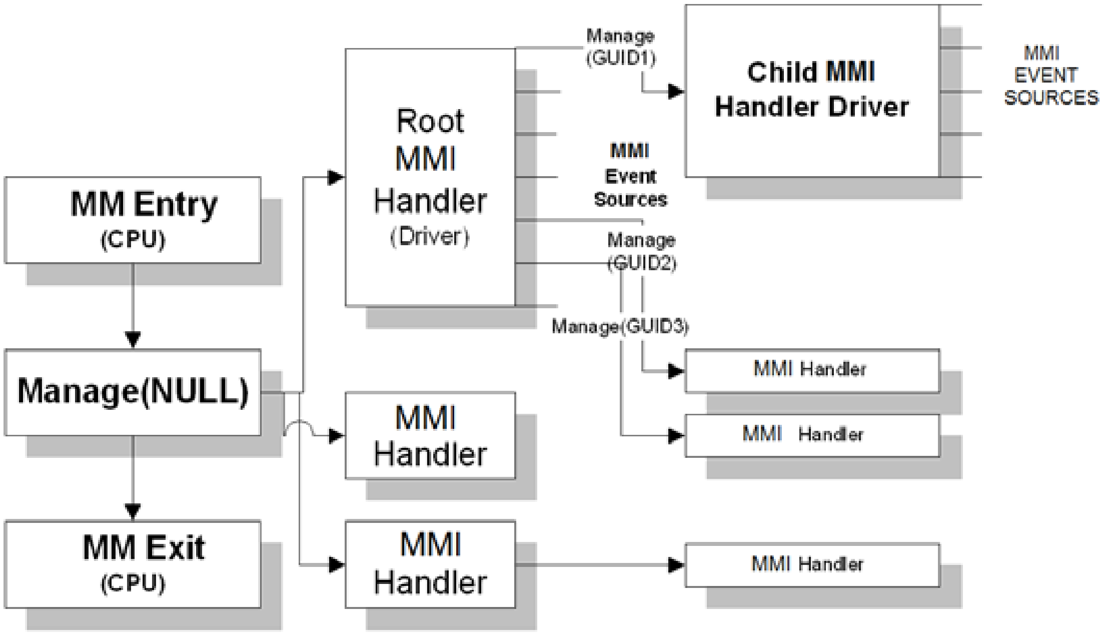


Figure 3. MMI Handler Relationships

1.13 MM Services

1.13.1 MM Driver Model

The MM Driver model has similar constraints to those of UEFI runtime drivers. Specifically, during MM Driver Runtime, the drivers must not use core protocol services. There will be MMST-based services, which the drivers can access, but the UEFI System Table and other protocols installed during boot services are not available.

Instead, the full collection of UEFI Boot Services and UEFI Runtime Services are available only during the MM Driver Initialization phase. This visibility is useful so that the MM Driver can leverage the rich set of UEFI services. This design makes the UEFI protocol database useful to these drivers while outside of SM and during their initial load within SM.

The MMST-based services that are available include the following:

- A minimal, blocking variant of the device I/O protocol
- A memory allocator from MM memory
- A minimal protocol database for protocols for use inside of SM.

These services are exposed by entries in the Management Mode System Table (MMST).

1.13.2 MM Protocols

Additional standard protocols are exposed as SM protocols and accessed using the protocol services provided by the MMST. They may be located during MM Driver Initialization or MM Driver Runtime. MM Driver. For example, the status code equivalent in MM is simply a UEFI protocol whose interface references an MM-based driver's service. Other MM Drivers locate this MM-based status code protocol and can use it during runtime to emit error or progress information.

1.14 MM UEFI Protocols

This section describes those protocols related to MM that are available through the UEFI boot services (called "UEFI Protocols") or through the MMST (called "MM Protocols").

1.14.1 UEFI Protocols

The system architecture of the MM driver is broken into the following pieces:

- MM Base Protocol
- MM Access Protocol
- MM Control Protocol

The *MM Base Protocol* will be published by the MM IPL driver which activates the MM Foundation for usage during the DXE phase.The *MM Access Protocol* understands the particular enable and locking mechanisms that memory controller might support while executing in MM.

The *MM Control Protocol* understands how to trigger synchronous MMIs either once or periodically.

1.14.2 MM Protocols

The following figure shows the MM protocols that are published for an IA-32 system.

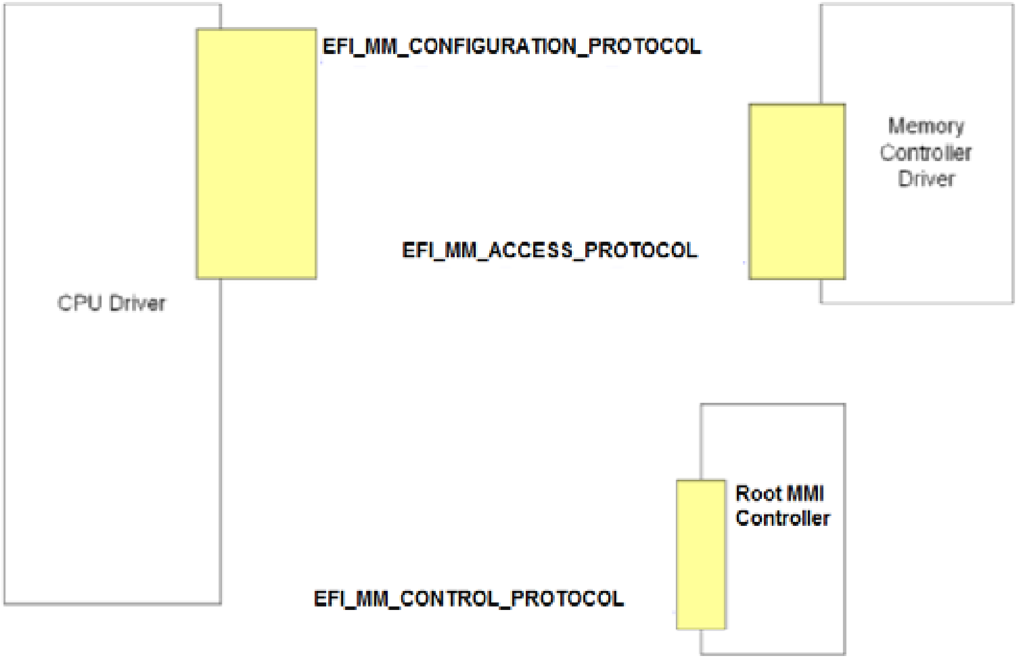


Figure 4. Published Protocols for IA-32 Systems

2 MM Foundation Entry Point

2.1 EFI_MM_ENTRY_POINT

Summary

This function is the main entry point to the MM Foundation.

Prototype

```
typedef
VOID
(EFIAPI *EFI_MM_ENTRY_POINT) (
  IN CONST EFI_MM_ENTRY_CONTEXT *MmEntryContext
  );
```

Parameters

MmEntryContext

Processor information and functionality needed by MM Foundation.

Description

This function is the entry point to the MM Foundation. The processor MM entry code will call this function with the processor information and functionality necessary for MM.

Related Definitions

```
typedef struct _EFI_MM_ENTRY_CONTEXT {
  EFI_MM_STARTUP_THIS_AP  MmStartupThisAp;
  UINTN                   CurrentlyExecutingCpu;
  UINTN                   NumberOfCpus;
  UINTN                   *CpuSaveStateSize;
  VOID                    **CpuSaveState;
} EFI_MM_ENTRY_CONTEXT;
```

MmStartupThisAp

Initiate a procedure on an application processor while in SM. See the **MmStartupThisAp()** function description.

CurrentlyExecutingCpu

A number between zero and the *NumberOfCpus* field. This field designates which processor is executing the MM Foundation.

NumberOfCpus

The number of current operational processors in the platform. This is a 1 based counter. This does not indicate the number of processors that entered MM.

CpuSaveStateSize

Points to an array, where each element describes the number of bytes in the corresponding save state specified by *CpuSaveState*. There are always *NumberOfCpus* entries in the array.

CpuSaveState

Points to an array, where each element is a pointer to a CPU save state. The corresponding element in *CpuSaveStateSize* specifies the number of bytes in the save state area. There are always *NumberOfCpus* entries in the array.

2.2 MM_FOUNDATION_ENTRY_POINT

Summary

This function is the image entry point of a standalone MM Foundation.

Prototype

```
typedef
VOID
(EFIAPI *MM_FOUNDATION_ENTRY_POINT) (
  IN CONST VOID            *HobStart
  );
```

Parameters

HobStart

A pointer to the HOB list.

Description

This function is the image entry point of a standalone MM Foundation. Standalone MM IPL passes *HobStart* to standalone MM Foundation. HOB list that describes the system state at the hand-off to the MM Foundation. At a minimum, this system state must include the following:

- PHIT HOB
- CPU HOB
- Description of MMRAM
- Description of one or more firmware volumes

MM Foundation can use MMRAM hob to build heap base upon MMRAM hob information. MM Foundation can use FV hob to dispatch standalone MM driver.

3 Management Mode System Table (MMST)

3.1 MMST Introduction

This section describes the Management Mode System Table (MMST). The MMST is a set of capabilities exported for use by all drivers that are loaded into Management Mode RAM (MMRAM).

The MMST is similar to the UEFI System Table. It is a fixed set of services and data that are designed to provide basic services for MM Drivers. The MMST is provided by the MM IPL driver, which also manages the following:

- Dispatch of drivers in MM
- Allocations of MMRAM
- Installation/discovery of MM protocols

3.2 EFI_MM_SYSTEM_TABLE

Summary

The Management Mode System Table (MMST) is a table that contains a collection of common services for managing MMRAM allocation and providing basic I/O services. These services are intended for both preboot and runtime usage.

Related Definitions

```
#define MM_MMST_SIGNATURE   EFI_SIGNATURE_32('S','M','S','T')
#define MM_SPECIFICATION_MAJOR_REVISION  1
#define MM_SPECIFICATION_MINOR_REVISION  60
#define EFI_MM_SYSTEM_TABLE_REVISION
((MM_SPECIFICATION_MAJOR_REVISION<<16) |
(MM_SPECIFICATION_MINOR_REVISION)

typedef struct _EFI_MM_SYSTEM_TABLE {
  EFI_TABLE_HEADER                    Hdr;

  CHAR16                              *MmFirmwareVendor;
  UINT32                              MmFirmwareRevision;

  EFI_MM_INSTALL_CONFIGURATION_TABLE
MmInstallConfigurationTable;

EFI_MM_CPU_IO_PROTOCOL                MmIo;

  //
  // Runtime memory service
  //
  EFI_ALLOCATE_POOL                  MmAllocatePool;
  EFI_FREE_POOL                      MmFreePool;
  EFI_ALLOCATE_PAGES                 MmAllocatePages;
  EFI_FREE_PAGES                     MmFreePages;

  //
  // MP service
  //
  EFI_MM_STARTUP_THIS_AP             MmStartupThisAp;

  //
  // CPU information records
  //
  UINTN                              CurrentlyExecutingCpu;
  UINTN                              NumberOfCpus;
UINTN                                *CpuSaveStateSize;
VOID                                 **CpuSaveState;

  //
  // Extensibility table
  //
  UINTN                              NumberOfTableEntries;
  EFI_CONFIGURATION_TABLE            *MmConfigurationTable;

//
// Protocol services
//
  EFI_INSTALL_PROTOCOL_INTERFACE     MmInstallProtocolInterface;
  EFI_UNINSTALL_PROTOCOL_INTERFACE
MmUninstallProtocolInterface;
  EFI_HANDLE_PROTOCOL                MmHandleProtocol;
  EFI_MM_REGISTER_PROTOCOL_NOTIFY    MmRegisterProtocolNotify;
EFI_LOCATE_HANDLE                    MmLocateHandle;
EFI_LOCATE_PROTOCOL                  MmLocateProtocol;

//
// MMI management functions
//
EFI_MM_INTERRUPT_MANAGE              MmiManage;
EFI_MM_INTERRUPT_REGISTER            MmiHandlerRegister;
EFI_MM_INTERRUPT_UNREGISTER          MmiHandlerUnRegister;
} EFI_MM_SYSTEM_TABLE;
```

Parameters

Hdr

The table header for the Management Mode System Table (MMST). This header contains the **MM_SMST_SIGNATURE, MM_MMST_SIGNATURE** and **EFI_MM_SYSTEM_TABLE_REVISION** values along with the size of the **EFI_MM_SYSTEM_TABLE** structure.

Note: *In the MM Foundation use of the* **EFI_TABLE_HEADER** *for the Management Mode Services Table (MMST), there is special treatment of the CRC32 field. This value is reserved for MM and should be set to zero*

MmFirmwareVendor

A pointer to a **NULL**-terminated Unicode string containing the vendor name. It is permissible for this pointer to be **NULL**.

MmFirmwareRevision

The particular revision of the firmware.

MmInstallConfigurationTable

Adds, updates, or removes a configuration table entry from the MMST. See the **MmInstallConfigurationTable()** function description.

MmIo

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices. The I/O services are provided by the driver which produces the MM CPU I/O Protocol. If that driver has not been loaded yet, this function pointer will return **EFI_UNSUPPORTED**.

MmAllocatePool

Allocates MMRAM.

MmFreePool

Returns pool memory to the system.

MmAllocatePages

Allocates pages from MMRAM.

MmFreePages

Returns pages of memory to the system.

MmStartupThisAp

Initiate a procedure on an application processor while in MM. See the **MmStartupThisAp()** function description. *MmStartupThisAp* may not be used during MM Driver Initialization, and MM and MM Driver must be considered "undefined". This service only defined while an MMI is being processed.

CurrentlyExecutingCpu

A number between zero and the value in the field *NumberOfCpus*. This field designates which processor is executing the MM infrastructure. *CurrentlyExecutingCpu* may not be used during MM Driver Initialization, and MM and MM Driver and must be considered "undefined". This field is only defined while an MMI is being processed.

NumberOfCpus

The number of possible processors in the platform. This is a 1 based counter. *NumberOfCpus* may not be used in the entry point of an MM MM Driver and must be considered "undefined". This field is only defined while an MMI is being processed.

CpuSaveStateSize

Points to an array, where each element describes the number of bytes in the corresponding save state specified by *CpuSaveState*. There are always *NumberOfCpus* entries in the array. *CpuSaveStateSize* may not be used during MM Driver Initialization Driver and must be considered "undefined". This field is only defined while an MMI is being processed.

CpuSaveState

Points to an array, where each element is a pointer to a CPU save state. The corresponding element in *CpuSaveStateSize* specifies the number of bytes in the save state area. There are always *NumberOfCpus* entries in the array.

CpuSaveState may not be used during MM Driver Initialization MM Driver and must be considered "undefined". This field is only defined while an MMI is being processed.

NumberOfTableEntries

The number of UEFI Configuration Tables in the buffer *MmConfigurationTable*.

MmConfigurationTable

A pointer to the UEFI Configuration Tables. The number of entries in the table is *NumberOfTableEntries*. Type **EFI_CONFIGURATION_TABLE** is defined in the *UEFI Specification*, section 4.6.

MmInstallProtocolInterface

Installs an MM protocol interface on a device handle. Type **EFI_INSTALL_PROTOCOL_INTERFACE** is defined in the *UEFI Specification*, section 4.4.

MmUninstallProtocolInterface

Removes an MM protocol interface from a device handle. Type **EFI_UNINSTALL_PROTOCOL_INTERFACE** is defined in the *UEFI Specification*, section 4.4.

MmHandleProtocol

Queries a handle to determine if it supports a specified MM protocol. Type **EFI_HANDLE_PROTOCOL** is defined in the *UEFI Specification*, section 4.4.

MmRegisterProtocolNotify

Registers a callback routine that will be called whenever an interface is installed for a specified MM protocol.

MmLocateHandle

Returns an array of handles that support a specified MM protocol. Type **EFI_LOCATE_HANDLE** is defined in the *UEFI Specification*, section 4.4.

MmLocateProtocol

Returns the first installed interface for a specific MM protocol. Type **EFI_LOCATE_PROTOCOL** is defined in the *UEFI Specification*, section 4.4.

MmiManage

Manage MMI sources of a particular type.

MmiHandlerRegister

Registers an MMI handler for an MMI source.

MmiHandlerUnRegister

Unregisters an MMI handler for an MMI source.

Description

The *CurrentlyExecutingCpu* parameter is a value that is less than the *NumberOfCpus* field. The *CpuSaveState* is a pointer to an array of CPU save states in MMRAM.   The *CurrentlyExecutingCpu* can be used as an index to locate the respective save-state for which the given processor is executing, if so desired.

The **EFI_MM_SYSTEM_TABLE** provides support for MMRAM allocation. The functions have the same function prototypes as those found in the UEFI Boot Services, but are only effective in allocating and freeing MMRAM. Drivers cannot allocate or free UEFI memory using these services. Drivers cannot allocate or free MMRAM using the UEFI Boot Services. The functions are:

- **`MmAllocatePages()`**
- **`MmFreePages()`**
- **`MmAllocatePool()`**
- **`MmFreePool()`**

The **`EFI_MM_SYSTEM_TABLE`** provides support for MM protocols, which are runtime protocols designed to execute exclusively inside of MM. Drivers cannot access protocols installed using the UEFI Boot Services through this interface. Drivers cannot access protocols installed using these interfaces through the UEFI Boot Services interfaces.

Five of the standard protocol-related functions from the UEFI boot services table are provided in the MMST and perform in a similar fashion. These functions are required to be available until the **`EFI_MM_READY_TO_LOCK_PROTOCOL`** notification has been installed. The functions are:

- **`MmInstallProtocolInterface()`**
- **`MmUninstallProtocolInterface()`**
- **`MmLocateHandle()`**
- **`MmHandleProtocol()`**
- **`MmLocateProtocol()`**.

Noticeably absent are services which support the UEFI driver model. The function **`MmRegisterProtocolNotify()`**, works in a similar fashion to the UEFI function except that it does not use an event.

MmInstallConfigurationTable()

Summary

Adds, updates, or removes a configuration table entry from the Management Mode System Table (MMST).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_INSTALL_CONFIGURATION_TABLE) (
  IN CONST EFI_MM_SYSTEM_TABLE    *SystemTable,
  IN CONST EFI_GUID               *Guid,
  IN VOID                         *Table,
  IN UINTN                        TableSize
  )
```

Parameters

SystemTable

A pointer to the Management Mode System Table (MMST).

Guid

A pointer to the GUID for the entry to add, update, or remove.

Table

A pointer to the buffer of the table to add.

TableSize

The size of the table to install.

Description

The **MmInstallConfigurationTable()** function is used to maintain the list of configuration tables that are stored in the MMST. The list is stored as an array of (GUID, Pointer) pairs. The list must be allocated from pool memory with *PoolType* set to **EfiRuntimeServicesData**.

If *Guid* is not a valid GUID, **EFI_INVALID_PARAMETER** is returned. If *Guid* is valid, there are four possibilities:

- If *Guid* is not present in the MMST and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is added to the MMST. See Note below.
- If *Guid* is not present in the MMST and *Table* is **NULL**, then **EFI_NOT_FOUND** is returned.
- If *Guid* is present in the MMST and *Table* is not **NULL**, then the (*Guid*, *Table*) pair is updated with the new *Table* value.
- If *Guid* is present in the MMST and *Table* is **NULL**, then the entry associated with *Guid* is removed from the MMST.

If an add, modify, or remove operation is completed, then **EFI_SUCCESS** is returned.

Note: *If there is not enough memory to perform an add operation, then* **`EFI_OUT_OF_RESOURCES`** *is returned.*

Status Codes Returned

EFI_SUCCESS	The (*`Guid`*, *`Table`*) pair was added, updated, or removed.
EFI_INVALID_PARAMETER	*`Guid`* is not valid.
EFI_NOT_FOUND	An attempt was made to delete a nonexistent entry.
EFI_OUT_OF_RESOURCES	There is not enough memory available to complete the operation.

MmAllocatePool()

Summary

Allocates pool memory from MMRAM.

Prototype

Type **EFI_ALLOCATE_POOL** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.2.

Description

The **MmAllocatePool()** function allocates a memory region of *Size* bytes from memory of type *PoolType* and returns the address of the allocated memory in the location referenced by *Buffer*. This function allocates pages from **EfiConventionalMemory** as needed to grow the requested pool type. All allocations are eight-byte aligned.

The allocated pool memory is returned to the available pool with the **MmFreePool()** function.

Note: *All allocations of MMRAM should use* **EfiRuntimeServicesCode** *or* **EfiRuntimeServicesData**.

Status Codes Returned

EFI_SUCCESS	The requested number of bytes was allocated.
EFI_OUT_OF_RESOURCES	The pool requested could not be allocated.
EFI_INVALID_PARAMETER	*PoolType* was invalid.

MmFreePool()

Summary

Returns pool memory to the system.

Prototype

Type **EFI_FREE_POOL** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.2.

Description

The **MmFreePool()** function returns the memory specified by *Buffer* to the MMRAM heap. The *Buffer* that is freed must have been allocated by **MmAllocatePool().**

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Buffer* was invalid.

MmAllocatePages()

Summary

Allocates page memory from MMRAM.

Prototype

Type **EFI_ALLOCATE_PAGES** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.2.

Description

The **MmAllocatePages()** function allocates the requested number of pages from the MMRAM heap and returns a pointer to the base address of the page range in the location referenced by *Memory*. The function scans the MM memory map to locate free pages. When it finds a physically contiguous block of pages that is large enough and also satisfies the allocation requirements of *Type*, it changes the memory map to indicate that the pages are now of type *MemoryType*.

All allocations of MMRAM should use **EfiRuntimeServicesCode** or **EfiRuntimeServicesData**.

Allocation requests of *Type*

- **AllocateAnyPages** allocate any available range of pages that satisfies the request. On input, the address pointed to by *Memory* is ignored.
- **AllocateMaxAddress** allocate any available range of pages whose uppermost address is less than or equal to the address pointed to by *Memory* on input.
- **AllocateAddress** allocate pages at the address pointed to by *Memory* on input.

Status Codes Returned

EFI_SUCCESS	The requested pages were allocated.
EFI_OUT_OF_RESOURCES	The pages could not be allocated.
EFI_INVALID_PARAMETER	*Type* is not **AllocateAnyPages** or **AllocateMaxAddress** or **AllocateAddress.**
EFI_INVALID_PARAMETER	*MemoryType* is in the range **EfiMaxMemoryType** …0x7FFFFFFF.
EFI_NOT_FOUND	The requested pages could not be found.

MmFreePages()

Summary

Returns pages of memory to the system.

Protocol

Type **EFI_FREE_PAGES** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.2.

Description

The **MmFreePages()** function returns memory allocated by **MmAllocatePages()** to the MMRAM heap.

Status Codes Returned

EFI_SUCCESS	The requested memory pages were freed.
EFI_NOT_FOUND	The requested memory pages were not allocated with **MmAllocatePages()**.
EFI_NOT_FOUND	EFI_INVALID_PARAMETER *Memory* is not a page-aligned address or *Pages* is invalid.

MmStartupThisAp()

Summary

This service lets the caller to get one distinct application processor (AP) to execute a caller-provided code stream while in MM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_STARTUP_THIS_AP) (
  IN   EFI_AP_PROCEDURE           Procedure
  IN   UINTN                      CpuNumber,
  IN   OUT VOID                   *ProcArguments OPTIONAL
  );
```

Parameters

Procedure

A pointer to the code stream to be run on the designated AP of the system. Type **EFI_AP_PROCEDURE** is defined below.

CpuNumber

The zero-based index of the processor number of the AP on which the code stream is supposed to run. If the processor number points to the current processor, then it will not run the supplied code.

ProcArguments

Allows the caller to pass a list of parameters to the code that is run by the AP. It is an optional common mailbox between APs and the caller to share information.

Related Definitions

See Volume 2, **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs**, Related definitions.

Description

This function is used to dispatch one specific, healthy, enabled, and non-busy AP out of the processor pool to the code stream that is provided by the caller while in MM. The recovery of a failed AP is optional and the recovery mechanism is implementation dependent.

Status Codes Returned

EFI_SUCCESS	The call was successful and the return parameters are valid.
EFI_INVALID_PARAMETER	The input arguments are out of range.
EFI_INVALID_PARAMETER	The CPU requested is not available on this MMI invocation.
EFI_INVALID_PARAMETER	The CPU cannot support an additional service invocation.

MmInstallProtocolInterface()

Summary

Installs a MM protocol interface on a device handle. If the handle does not exist, it is created and added to the list of handles in the system.

Prototype

Type **EFI_INSTALL_PROTOCOL_INTERFACE** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.3.1.

Description

The **MmInstallProtocolInterface()** function installs a protocol interface (a GUID/Protocol Interface structure pair) on an MM device handle. The same GUID cannot be installed more than once onto the same handle. If installation of a duplicate GUID on a handle is attempted, an **EFI_INVALID_PARAMETER** will result. Installing a protocol interface allows other MM MM Drivers to locate the *Handle*, and the interfaces installed on it.

When a protocol interface is installed, the firmware calls all notification functions that have registered to wait for the installation of *Protocol.* For more information, see the **MmRegisterProtocolNotify()** function description.

Status Codes Returned

EFI_SUCCESS	The protocol interface was installed.
EFI_OUT_OF_RESOURCES	Space for a new handle could not be allocated.
EFI_INVALID_PARAMETER	*Handle* is **NULL.**
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_INVALID_PARAMETER	*InterfaceType* is not **EFI_NATIVE_INTERFACE**.
EFI_INVALID_PARAMETER	*Protocol* is already installed on the handle specified by *Handle*.

MmUninstallProtocolInterface()

Summary

Removes a MM protocol interface from a device handle.

Prototype

Type **EFI_UNINSTALL_PROTOCOL_INTERFACE** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.3.1.

Description

The **MmUninstallProtocolInterface()** function removes a protocol interface from the handle on which it was previously installed. The *Protocol* and *Interface* values define the protocol interface to remove from the handle.

The caller is responsible for ensuring that there are no references to a protocol interface that has been removed. If the last protocol interface is removed from a handle, the handle is freed and is no longer valid.

Status Codes Returned

EFI_SUCCESS	The interface was removed.
EFI_NOT_FOUND	The interface was not found.
EFI_ACCESS_DENIED	The interface was not removed because the interface is still being used by a driver.
EFI_INVALID_PARAMETER	*Handle i*s not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.

MmHandleProtocol()

Summary

Queries a handle to determine if it supports a specified MM protocol.

Prototype

Type **EFI_HANDLE_PROTOCOL** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.3.1.

Description

The **MmHandleProtocol()** function queries *Handle* to determine if it supports *Protocol*. If it does, then, on return, *Interface* points to a pointer to the corresponding Protocol Interface.

Interface can then be passed to any protocol service to identify the context of the request.

Status Codes Returned

EFI_SUCCESS	The interface information for the specified protocol was returned.
EFI_UNSUPPORTED	The device does not support the specified protocol.
EFI_INVALID_PARAMETER	*Handle* is not a valid **EFI_HANDLE**.
EFI_INVALID_PARAMETER	*Protocol* is **NULL**.
EFI_INVALID_PARAMETER	*Interface* is **NULL**.

MmRegisterProtocolNotify()

Summary

Register a callback function be called when a particular protocol interface is installed.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_REGISTER_PROTOCOL_NOTIFY)(
  IN  CONST EFI_GUID       *Protocol,
  IN  EFI_MM_NOTIFY_FN     Function,
  IN OUT VOID              **Registration
);
```

Parameters

Protocol

The unique ID of the protocol for which the event is to be registered. Type **EFI_GUID** is defined in the **InstallProtocolInterface()** function description.

Function

Points to the notification function, which is described below.

Registration

A pointer to a memory location to receive the registration value. This value must be saved and used by the notification function to retrieve the list of handles that have added a protocol interface of type *Protocol*.

Description

The **MmRegisterProtocolNotify()** function creates a registration *Function* that is to be called whenever a protocol interface is installed for *Protocol* by **MmInstallProtocolInterface()**.

When *Function* has been called, the **MmLocateHandle()** function can be called to identify the newly installed handles that support *Protocol*. The *Registration* parameter in **MmRegisterProtocolNotify()** corresponds to the *SearchKey* parameter in **MmLocateHandle()**. Note that the same handle may be returned multiple times if the handle reinstalls the target protocol ID multiple times.

If *Function* == NULL and *Registration* is an existing registration, then the callback is unhooked. **Protocol* must be validated it with **Registration*. If *Registration* is not found then **EFI_NOT_FOUND** is returned.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_NOTIFY_FN)(
  IN CONST EFI_GUID    *Protocol,
  IN VOID              *Interface,
  IN EFI_HANDLE       Handle
  );
```

Protocol

Points to the protocol's unique identifier.

Interface

Points to the interface instance.

Handle

The handle on which the interface was installed.

Status Codes Returned

EFI_SUCCESS	Successfully returned the registration record that has been added or unhooked.
EFI_INVALID_PARAMETER	*Protocol*is NULL or *Registration* is NULL.
EFI_OUT_OF_RESOURCES	Not enough memory resource to finish the request.
EFI_NOT_FOUND	If the registration is not found when Function == NULL

MmLocateHandle()

Summary

Returns an array of handles that support a specified protocol.

Prototype

Type **EFI_LOCATE_HANDLE** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.3.1.

Description

The **MmLocateHandle()** function returns an array of handles that match the *SearchType* request. If the input value of *BufferSize* is too small, the function returns **EFI_BUFFER_TOO_SMALL** and updates *BufferSize* to the size of the buffer needed to obtain the array.

Status Codes Returned

EFI_SUCCESS	The array of handles was returned.
EFI_NOT_FOUND	No handles match the search.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small for the result. *BufferSize* has been updated with the size needed to complete the request.
EFI_INVALID_PARAMETER	*SearchType* is not a member of **EFI_LOCATE_SEARCH_TYPE**.
EFI_INVALID_PARAMETER	*SearchType* is **ByRegisterNotify** and *SearchKey* is **NULL**.
EFI_INVALID_PARAMETER	*SearchType* is **ByProtocol** and *Protocol* is **NULL**.
EFI_INVALID_PARAMETER	One or more matches are found and *BufferSize* is **NULL**.
EFI_INVALID_PARAMETER	*BufferSize* is large enough for the result and *Buffer* is **NULL**.

MmLocateProtocol()

Summary

Returns the first MM protocol instance that matches the given protocol.

Prototype

Type **EFI_LOCATE_PROTOCOL** is defined in the *UEFI Specification*, section 4.4. The function description is found in the *UEFI Specification*, section 6.3.1.

Description

The **MmLocateProtocol()** function finds the first device handle that support *Protocol*, and returns a pointer to the protocol interface from that handle in *Interface*. If no protocol instances are found, then *Interface* is set to **NULL**.

If *Interface* is **NULL**, then **EFI_INVALID_PARAMETER** is returned.

If *Registration* is **NULL**, and there are no handles in the handle database that support *Protocol*, then **EFI_NOT_FOUND** is returned.

If *Registration* is not **NULL**, and there are no new handles for *Registration*, then **EFI_NOT_FOUND** is returned.

Status Codes Returned

EFI_SUCCESS	A protocol instance matching *Protocol* was found and returned in *Interface*.
EFI_INVALID_PARAMETER	*Interface* is **NULL**.
EFI_NOT_FOUND	No protocol instances were found that match *Protocol* and *Registration* .

MmiManage()

Summary

Manage MMI of a particular type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_INTERRUPT_MANAGE)(
 IN CONST EFI_GUID       *HandlerType,
 IN CONST VOID           *Context          OPTIONAL,
 IN OUT VOID             *CommBuffer       OPTIONAL,
 IN OUT UINTN            *CommBufferSize   OPTIONAL
 );
```

Parameters

HandlerType

Points to the handler type or NULL for root MMI handlers.

Context

Points to an optional context buffer. The format of the contents of the context buffer depends on *HandlerType*.

CommBuffer

Points to the optional communication buffer. The format of the contents of the communication buffer depends on *HandlerType*. The contents of the buffer (and its size) may be altered if **EFI_SUCCESS** is returned.

CommBufferSize

Points to the size of the optional communication buffer. The size of the buffer may be altered if **EFI_SUCCESS** is returned.

Description

This function will call the registered handler functions which match the specified invocation type.

If NULL is passed in *HandlerType*, then only those registered handler functions which passed NULL as their *HandlerType* will be called. If NULL is passed in *HandlerType*, then Context should be NULL, *CommBuffer* should point to an instance of **EFI_MM_ENTRY_CONTEXT** and *CommBufferSize* should point to the size of that structure. Type **EFI_MM_ENTRY_CONTEXT** is defined in “Related Definitions” below.

If at least one of the handlers returns **EFI_WARN_INTERRUPT_SOURCE_QUIESCED** or **EFI_SUCCESS** then the function will return **EFI_SUCCESS**. If a handler returns **EFI_SUCCESS** and *HandlerType* is not NULL then no additional handlers will be processed.

If a handler returns **EFI_INTERRUPT_PENDING** and *HandlerType* is not NULL then no additional handlers will be processed and **EFI_INTERRUPT_PENDING** will be returned.

If all the handlers returned **EFI_WARN_INTERRUPT_SOURCE_PENDING** then **EFI_WARN_INTERRUPT_SOURCE_PENDING** will be returned.

If no handlers of *HandlerType* are found then **EFI_NOT_FOUND** will be returned.

Status Codes Returned

EFI_WARN_INTERRUPT_SOURCE_PENDING	The MMI was processed successfully but the MMI source not quiesced.
EFI_INTERRUPT_PENDING	One or more MMI sources could not be quiesced.
EFI_NOT_FOUND	The MMI was not handled and the MMI source was not quiesced.
EFI_SUCCESS	The MMI was handled and the MMI source was quiesced.

MmiHandlerRegister()

Summary

Registers a handler to execute within MM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_INTERRUPT_REGISTER) (
  IN  EFI_MM_HANDLER_ENTRY_POINT    Handler,
  IN  CONST EFI_GUID                *HandlerType OPTIONAL,
  OUT EFI_HANDLE                    *DispatchHandle
  );
```

Parameters

Handler

Handler service function pointer. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in “Related Definitions” below.

HandlerType

Points to an **EFI_GUID** which describes the type of invocation that this handler is for or **NULL** to indicate a root MMI handler.

DispatchHandle

On return, contains a unique handle which can be used to later unregister the handler function. It is also passed to the handler function itself.

Description

This service allows the registration of a MMI handling function from within MM.

The handler should have the **EFI_MM_HANDLER_ENTRY_POINT** interface defined in “Related Definitions” below.

Related Definitions

```
//*************************************************
// EFI_MM_HANDLER_ENTRY_POINT
//*************************************************

typedef
EFI_STATUS
(EFIAPI *EFI_MM_HANDLER_ENTRY_POINT) (
  IN EFI_HANDLE       DispatchHandle,
  IN CONST VOID      *Context           OPTIONAL,
  IN OUT VOID        *CommBuffer        OPTIONAL,
  IN OUT UINTN       *CommBufferSize    OPTIONAL
  );
```

DispatchHandle

The unique handle assigned to this handler by **MmiHandlerRegister()**.Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Context

Points to the optional handler context which was specified when the handler was registered.

CommBuffer

A pointer to a collection of data in memory that will be conveyed from a non-MM environment into an MM environment. The buffer must be contiguous, physically mapped, and be a physical address.

CommBufferSize

The size of the *CommBuffer*.

MmiHandlerRegister() returns one of two status codes:

Status Codes Returned (MmiHandlerRegister)

EFI_SUCCESS	MMI handler added successfully.
EFI_INVALID_PARAMETER	Handler is **NULL** or *DispatchHandle* is **NULL**

EFI_MM_HANDLER_ENTRY_POINT returns one of four status codes:

Status Codes Returned (EFI_MM_HANDLER_ENTRY_POINT)

EFI_SUCCESS	The MMI was handled and the MMI source the MMI source was quiesced. No other handlers should still be called.
EFI_WARN_INTERRUPT_SOURCE_QUIESCED	The MMI source has been quiesced but other handlers should still be called.
EFI_WARN_INTERRUPT_SOURCE_PENDING	The MMI source is still pending and other handlers should still be called.

EFI_INTERRUPT_PENDING	The MMI source could not be quiesced.

MmiHandlerUnRegister()

Summary

Unregister a handler in MM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_INTERRUPT_UNREGISTER)(
  IN EFI_HANDLE                              DispatchHandle,
  );
```

Parameters

DispatchHandle

The handle that was specified when the handler was registered.

Description

This function unregisters the specified handler function.

Status Codes Returned

EFI_SUCCESS	Handler function was successfully unregistered.
EFI_INVALID_PARAMETER	*DispatchHandle* does not refer to a valid handle.

4 MM Protocols

4.1 Introduction

There is a share-nothing model that is employed between the management-mode application and the boot service/runtime UEFI environment. As such, a minimum set of services needs to be available to the boot service agent.

The services described in this section coexist with a foreground pre-boot or runtime environment. The latter can include both UEFI and non-UEFI aware operating systems. As such, the implementation of these services must save and restore any "shared" resources with the foreground environment or only use resources that are private to the MM code.

4.2 Status Codes Services

EFI_MM_STATUS_CODE_PROTOCOL

Summary

Provides status code services from MM.

GUID

```
#define EFI_MM_STATUS_CODE_PROTOCOL_GUID \
  { 0x6afd2b77, 0x98c1, 0x4acd, 0xa6, 0xf9, 0x8a, 0x94, \
  0x39, 0xde, 0xf, 0xb1 }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_STATUS_CODE_PROTOCOL {
  EFI_MM_REPORT_STATUS_CODE   ReportStatusCode;
} EFI_MM_STATUS_CODE_PROTOCOL;
```

Parameters

ReportStatusCode

Allows for the MM agent to produce a status code output. See the **ReportStatusCode()** function description.

Description

The **EFI_MM_STATUS_CODE_PROTOCOL** provides the basic status code services while in MMRAM.

EFI_MM_STATUS_CODE_PROTOCOL.ReportStatusCode()

Summary

Service to emit the status code in MM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_REPORT_STATUS_CODE) (
  IN CONST EFI_MM_STATUS_CODE_PROTOCOL *This,
  IN EFI_STATUS_CODE_TYPE               CodeType,
  IN EFI_STATUS_CODE_VALUE              Value,
  IN UINT32                             Instance,
  IN CONST EFI_GUID                     *CallerId,
  IN EFI_STATUS_CODE_DATA               *Data OPTIONAL
  );
```

Parameters

This

Points to this instance of the **EFI_MM_STATUS_CODE_PROTOCOL**.

CodeType

Indicates the type of status code being reported. Type **EFI_STATUS_CODE_TYPE** is defined in "Related Definitions" below.

Value

Describes the current status of a hardware or software entity. This status 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

The **EFI_MM_STATUS_CODE_PROTOCOL.ReportStatusCode()** function enables a driver to emit a status code while in MM. The reason that there is a separate protocol definition from the DXE variant of this service is that the publisher of this protocol will provide a service that is capability of coexisting with a foreground operational environment, such as an operating system after the termination of boot services.

In case of an error, the caller can specify the severity. In most cases, the entity that reports the error may not have a platform-wide view and may not be able to accurately assess the impact of the error condition. The MM MM Driver that produces the Status Code MM Protocol is responsible for assessing the true severity level based on the reported severity and other information. This MM MM Driver may perform platform specific actions based on the type and severity of the status code being reported.

If *Data* is present, the driver treats it as read only data. The driver must copy *Data* to a local buffer in an atomic operation before performing any other actions. This is necessary to make this function re-entrant. The size of the local buffer may be limited. As a result, some of the *Data* can be lost. The size of the local buffer should at least be 256 bytes in size. Larger buffers will reduce the probability of losing part of the *Data*. If all of the local buffers are consumed, then this service may not be able to perform the platform specific action required by the status code being reported. As a result, if all the local buffers are consumed, the behavior of this service is undefined.

If the *CallerId* parameter is not **NULL**, then it is required to point to a constant GUID. In other words, the caller may not reuse or release the buffer pointed to by *CallerId.*

Status Codes Returned

EFI_SUCCESS	The function completed successfully
EFI_DEVICE_ERROR	The function should not be completed due to a device error.

4.3 CPU Save State Access Services

EFI_MM_CPU_PROTOCOL

Summary

Provides access to CPU-related information while in MM.

GUID

```
#define EFI_MM_CPU_PROTOCOL_GUID \
  { 0xeb346b97, 0x975f, 0x4a9f, \
  0x8b, 0x22, 0xf8, 0xe9, 0x2b, 0xb3, 0xd5, 0x69 }
```

Prototype

```
typedef struct _EFI_MM_CPU_PROTOCOL {
  EFI_MM_READ_SAVE_STATE  ReadSaveState;
  EFI_MM_WRITE_SAVE_STATE WriteSaveState;
} EFI_MM_CPU_PROTOCOL;
```

Members

ReadSaveState

Read information from the CPU save state. See **ReadSaveState()** for more information.

WriteSaveState

Write information to the CPU save state. See **WriteSaveState()** for more information.

Description

This protocol allows MM Drivers to access architecture-standard registers from any of the CPU save state areas. In some cases, difference processors provide the same information in the save state, but not in the same format. These so-called pseudo-registers provide this information in a standard format.

EFI_MM_CPU_PROTOCOL.ReadSaveState()

Summary

Read data from the CPU save state.

Prototype

```
typedef
  EFI_STATUS
(EFIAPI *EFI_MM_READ_SAVE_STATE (
  IN  CONST EFI_MM_CPU_PROTOCOL  *This,
  IN  UINTN                       Width,
  IN  EFI_MM_SAVE_STATE_REGISTER Register,
  IN  UINTN                       CpuIndex,
  OUT VOID                        *Buffer
  );
```

Parameters

Width

The number of bytes to read from the CPU save state. If the register specified by *Register* does not support the size specified by *Width*, then **EFI_INVALID_PARAMETER** is returned.

Register

Specifies the CPU register to read form the save state. The type **EFI_MM_SAVE_STATE_REGISTER** is defined in “Related Definitions” below. If the specified register is not implemented in the CPU save state map then **EFI_NOT_FOUND** error will be returned.

CpuIndex

Specifies the zero-based index of the CPU save state

**Buffer*

Upon return, this holds the CPU register value read from the save state.

Description

This function is used to read the specified number of bytes of the specified register from the CPU save state of the specified CPU and place the value into the buffer. If the CPU does not support the specified register *Register*, then **EFI_NOT_FOUND** should be returned. If the CPU does not support the specified register width *Width*, then **EFI_INVALID_PARAMETER** is returned.

Related Definitions

```
typedef enum {

  //
  // x86/X64 standard registers
  //
  EFI_MM_SAVE_STATE_REGISTER_GDTBASE       = 4,
  EFI_MM_SAVE_STATE_REGISTER_IDTBASE       = 5,
  EFI_MM_SAVE_STATE_REGISTER_LDTBASE       = 6,
  EFI_MM_SAVE_STATE_REGISTER_GDTLIMIT      = 7,
  EFI_MM_SAVE_STATE_REGISTER_IDTLIMIT      = 8,
  EFI_MM_SAVE_STATE_REGISTER_LDTLIMIT      = 9,
  EFI_MM_SAVE_STATE_REGISTER_LDTINFO       = 10,

  EFI_MM_SAVE_STATE_REGISTER_ES            = 20,
  EFI_MM_SAVE_STATE_REGISTER_CS            = 21,
  EFI_MM_SAVE_STATE_REGISTER_SS            = 22,
  EFI_MM_SAVE_STATE_REGISTER_DS            = 23,
  EFI_MM_SAVE_STATE_REGISTER_FS            = 24,
  EFI_MM_SAVE_STATE_REGISTER_GS            = 25,
  EFI_MM_SAVE_STATE_REGISTER_LDTR_SEL      = 26,
  EFI_MM_SAVE_STATE_REGISTER_TR_SEL        = 27,
  EFI_MM_SAVE_STATE_REGISTER_DR7           = 28,
  EFI_MM_SAVE_STATE_REGISTER_DR6           = 29,

  EFI_MM_SAVE_STATE_REGISTER_R8            = 30,
  EFI_MM_SAVE_STATE_REGISTER_R9            = 31,
  EFI_MM_SAVE_STATE_REGISTER_R10           = 32,
  EFI_MM_SAVE_STATE_REGISTER_R11           = 33,
  EFI_MM_SAVE_STATE_REGISTER_R12           = 34,
  EFI_MM_SAVE_STATE_REGISTER_R13           = 35,
  EFI_MM_SAVE_STATE_REGISTER_R14           = 36,
  EFI_MM_SAVE_STATE_REGISTER_R15           = 37,

  EFI_MM_SAVE_STATE_REGISTER_RAX           = 38,
  EFI_MM_SAVE_STATE_REGISTER_RBX           = 39,
  EFI_MM_SAVE_STATE_REGISTER_RCX           = 40,
  EFI_MM_SAVE_STATE_REGISTER_RDX           = 41,
  EFI_MM_SAVE_STATE_REGISTER_RSP           = 42,
  EFI_MM_SAVE_STATE_REGISTER_RBP           = 43,
  EFI_MM_SAVE_STATE_REGISTER_RSI           = 44,
  EFI_MM_SAVE_STATE_REGISTER_RDI           = 45,
  EFI_MM_SAVE_STATE_REGISTER_RIP           = 46,

  EFI_MM_SAVE_STATE_REGISTER_RFLAGS        = 51,
  EFI_MM_SAVE_STATE_REGISTER_CR0           = 52,
  EFI_MM_SAVE_STATE_REGISTER_CR3           = 53,
  EFI_MM_SAVE_STATE_REGISTER_CR4           = 54,

  EFI_MM_SAVE_STATE_REGISTER_FCW           = 256,
  EFI_MM_SAVE_STATE_REGISTER_FSW           = 257,
  EFI_MM_SAVE_STATE_REGISTER_FTW           = 258,
  EFI_MM_SAVE_STATE_REGISTER_OPCODE        = 259,
  EFI_MM_SAVE_STATE_REGISTER_FP_EIP        = 260,
  EFI_MM_SAVE_STATE_REGISTER_FP_CS         = 261,
  EFI_MM_SAVE_STATE_REGISTER_DATAOFFSET    = 262,
  EFI_MM_SAVE_STATE_REGISTER_FP_DS         = 263,
  EFI_MM_SAVE_STATE_REGISTER_MM0           = 264,
  EFI_MM_SAVE_STATE_REGISTER_MM1           = 265,
  EFI_MM_SAVE_STATE_REGISTER_MM2           = 266,
  EFI_MM_SAVE_STATE_REGISTER_MM3           = 267,
  EFI_MM_SAVE_STATE_REGISTER_MM4           = 268,
  EFI_MM_SAVE_STATE_REGISTER_MM5           = 269,
  EFI_MM_SAVE_STATE_REGISTER_MM6           = 270,
  EFI_MM_SAVE_STATE_REGISTER_MM7           = 271,
  EFI_MM_SAVE_STATE_REGISTER_XMM0          = 272,
  EFI_MM_SAVE_STATE_REGISTER_XMM1          = 273,
  EFI_MM_SAVE_STATE_REGISTER_XMM2          = 274,
  EFI_MM_SAVE_STATE_REGISTER_XMM3          = 275,
  EFI_MM_SAVE_STATE_REGISTER_XMM4          = 276,
  EFI_MM_SAVE_STATE_REGISTER_XMM5          = 277,
  EFI_MM_SAVE_STATE_REGISTER_XMM6          = 278,
  EFI_MM_SAVE_STATE_REGISTER_XMM7          = 279,
  EFI_MM_SAVE_STATE_REGISTER_XMM8          = 280,
  EFI_MM_SAVE_STATE_REGISTER_XMM9          = 281,
  EFI_MM_SAVE_STATE_REGISTER_XMM10         = 282,
  EFI_MM_SAVE_STATE_REGISTER_XMM11         = 283,
  EFI_MM_SAVE_STATE_REGISTER_XMM12         = 284,
  EFI_MM_SAVE_STATE_REGISTER_XMM13         = 285,
  EFI_MM_SAVE_STATE_REGISTER_XMM14         = 286,
  EFI_MM_SAVE_STATE_REGISTER_XMM15         = 287,

  //
  // Pseudo-Registers
  //
  EFI_MM_SAVE_STATE_REGISTER_IO            = 512,
  EFI_MM_SAVE_STATE_REGISTER_LMA           = 513,
  EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID  = 514,

//
// ARM Registers. X0 corresponds to R0
//

EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X0 = 1024,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X1 = 1025,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X2 = 1026,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X3 = 1027,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X4 = 1028,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X5 = 1029,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X6 = 1030,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X7 = 1031,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X8 = 1032,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X9 = 1033,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X10 = 1034,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X11 = 1035,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X12 = 1036,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X13 = 1037,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X14 = 1038,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X15 = 1039,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X16 = 1040,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X17 = 1041,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X18 = 1042,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X19 = 1043,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X20 = 1044,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X21 = 1045,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X22 = 1046,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X23 = 1047,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X24 = 1048,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X25 = 1049,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X26 = 1050,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X27 = 1051,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X28 = 1052,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X29 = 1053,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X30 = 1054,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_X31 = 1055,

EFI_SMM_SAVE_STATE_REGISTER_AARCH64_FP = 1053, // x29 - Frame
Pointer
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_LR = 1054, // x30 - Link
Register
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SP = 1055, // x31 - Stack
Pointer

// AArch64 EL1 Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ELR_EL1 = 1300,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ESR_EL1 = 1301,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_FAR_EL1 = 1302,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ISR_EL1 = 1303,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_MAIR_EL1 = 1304,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_MIDR_EL1 = 1305,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_MPIDR_EL1 = 1306,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SCTLR_EL1 = 1307,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SP_EL0 = 1308,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SP_EL1 = 1309,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SPSR_EL1 = 1310,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TCR_EL1 = 1311,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TPIDR_EL0 = 1312,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TPIDR_EL1 = 1313,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TPIDRRO_EL0 = 1314,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TTBR0_EL1 = 1315,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TTBR1_EL1 = 1316,

// AArch64 EL2 Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ELR_EL2 = 1320,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ESR_EL2 = 1321,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_FAR_EL2 = 1322,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_HACR_EL2 = 1333,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_HCR_EL2 = 1334,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_HPFAR_EL2 = 1335,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_MAIR_EL2 = 1336,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SCTLR_EL2 = 1337,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SP_EL2 = 1338,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SPSR_EL2 = 1339,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TCR_EL2 = 1340,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TPIDR_EL2 = 1341,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TTBR0_EL2 = 1342,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_VTCR_EL2 = 1343,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_VTTBR_EL2 = 1344,

// AArch64 EL3 Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ELR_EL3 = 1350,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_ESR_EL3 = 1351,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_FAR_EL3 = 1352,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_MAIR_EL3 = 1353,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SCTLR_EL3 = 1354,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SP_EL3 = 1355,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_SPSR_EL3 = 1356,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TCR_EL3 = 1357,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TPIDR_EL3 = 1358,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_TTBR0_EL3 = 1359,


// 32-bit aliases for Rx->Xx
EFI_SMM_SAVE_STATE_REGISTER_ARM_R0 = 1024,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R1 = 1025,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R2 = 1026,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R3 = 1027,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R4 = 1028,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R5 = 1029,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R6 = 1030,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R7 = 1031,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R8 = 1032,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R9 = 1033,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R10 = 1034,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R11 = 1035,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R12 = 1036,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R13 = 1037,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R14 = 1038,
EFI_SMM_SAVE_STATE_REGISTER_ARM_R15 = 1039,
// Unique AArch32 Registers
EFI_SMM_SAVE_STATE_REGISTER_ARM_SP = 1037,  // alias for R13
EFI_SMM_SAVE_STATE_REGISTER_ARM_LR = 1038, // alias for R14
EFI_SMM_SAVE_STATE_REGISTER_ARM_PC = 1040, // alias for R15

// AArch32 EL1 Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_ARM_DFAR = 1222,
EFI_SMM_SAVE_STATE_REGISTER_ARM_DFSR = 1223,
EFI_SMM_SAVE_STATE_REGISTER_ARM_IFAR = 1224,
EFI_SMM_SAVE_STATE_REGISTER_ARM_ISR = 1225,
EFI_SMM_SAVE_STATE_REGISTER_ARM_MAIR0 = 1226,
EFI_SMM_SAVE_STATE_REGISTER_ARM_MAIR1 = 1227,
EFI_SMM_SAVE_STATE_REGISTER_ARM_MIDR = 1228,
EFI_SMM_SAVE_STATE_REGISTER_ARM_MPIDR = 1229,
EFI_SMM_SAVE_STATE_REGISTER_ARM_NMRR = 1230,
EFI_SMM_SAVE_STATE_REGISTER_ARM_PRRR = 1231,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SCTLR_NS = 1231,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR = 1232,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_abt = 1233,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_fiq = 1234,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_irq = 1235,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_svc = 1236,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_und = 1237,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TPIDRPRW = 1238,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TPIDRURO = 1239,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TPIDRURW = 1240,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TTBCR = 1241,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TTBR0 = 1242,
EFI_SMM_SAVE_STATE_REGISTER_ARM_TTBR1 = 1243,
EFI_SMM_SAVE_STATE_REGISTER_ARM_DACR = 1244,

// AArch32 EL1 Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_ARM_ELR_hyp = 1245,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HAMAIR0 = 1246,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HAMAIR1 = 1247,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HCR = 1248,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HCR2 = 1249,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HDFAR = 1250,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HIFAR = 1251,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HPFAR = 1252,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HSR = 1253,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HTCR = 1254,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HTPIDR = 1255,
EFI_SMM_SAVE_STATE_REGISTER_ARM_HTTBR = 1256,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_hyp = 1257,
EFI_SMM_SAVE_STATE_REGISTER_ARM_VTCR = 1258,
EFI_SMM_SAVE_STATE_REGISTER_ARM_VTTBR = 1259,
EFI_SMM_SAVE_STATE_REGISTER_ARM_DACR32_EL2 = 1260,

// AArch32 EL2 Secure Context System Registers
EFI_SMM_SAVE_STATE_REGISTER_ARM_SCTLR_S = 1261,
EFI_SMM_SAVE_STATE_REGISTER_ARM_SPSR_mon = 1262,

// Context System Registers: 32768 - 65535
EFI_SMM_SAVE_STATE_REGISTER_ARM_CSR = 32768,
EFI_SMM_SAVE_STATE_REGISTER_AARCH64_CSR = 32768
} EFI_SMM_SAVE_STATE_REGISTER;

} EFI_MM_SAVE_STATE_REGISTER;
```

AARCH32/AARCH64 REGISTER AVAILABILITY

Depending on the platform policy, not all registers may be available in the MM Save State. These registers will return the status code **EFI_NOT_FOUND** when calling **ReadSaveState()** or **WriteSaveState()**. In some cases this may be done to protect sensitive information in the non-secure execution environment.

EFI_MM_SAVE_STATE_ARM_CSR, EFI_MM_SAVE_STATE_AARCH64_CSR

The Read/Write interface can be used to retrieve AARCH32/AARCH64 Context System Registers that were saved upon entry to MM. These registers have the CPU Register Index starting with **EFI_MM_SAVE_STATE_ARM_CSR**. The actual CPU register index for a specific CSR register is calculated by adding the encoding of the MRS instruction, bits 5:19, to **EFI_MM_SAVE_STATE_REGISTER_ARM_CSR**. That is: (MRSIntruction[5:19] << 5 + **EFI_MM_SAVE_STATE_ARM_CSR**). See the *UEFI Specification*, Table 275 in Appendix N for more information.

EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID

The Read/Write interface for the pseudo-register **EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID** follows these rules:

For **ReadSaveState()**:

The pseudo-register only supports the 64-bit size specified by *Width*.

If the processor is in SM at the time the MMI occurred, the pseudo register value **EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID** is returned in Buffer. The value should match the *ProcessorId* value, as described in the **EFI_PROCESSOR_INFORMATION** record defined in Volume 2 of the *Platform Initialization Specification*.

For **WriteSaveState**():

Write operations to this pseudo-register are ignored.

EFI_MM_SAVE_STATE_REGISTER_LMA

The Read/Write interface for the pseudo-register **EFI_MM_SAVE_STATE_REGISTER_LMA** follows these rules:

For **ReadSaveState()**:

The pseudo-register only supports the single Byte size specified by *Width*. If the processor acts in 32-bit mode at the time the MMI occurred, the pseudo register value **EFI_MM_SAVE_STATE_REGISTER_LMA_32BIT** is returned in *Buffer*. Otherwise, **EFI_MM_SAVE_STATE_REGISTER_LMA_64BIT** is returned in *Buffer*.

```
#define EFI_MM_SAVE_STATE_REGISTER_LMA_32BIT = 32
#define EFI_MM_SAVE_STATE_REGISTER_LMA_64BIT = 64
```

For **WriteSaveState()**:

Write operations to this pseudo-register are ignored.

Status Codes Returned

EFI_SUCCESS	The register was read or written from Save State
EFI_NOT_FOUND	The register is not defined for the Save State of Processor
EFI_NOT_FOUND	The processor is not in SM.
EFI_INVALID_PARAMETER	Input parameters are not valid. For ex: Processor No or register width is not correct.*This* or *Buffer* is **NULL**.

EFI_MM_CPU_PROTOCOL.WriteSaveState()

Summary

Write data to the CPU save state.

Prototype

```
typedef
  EFI_STATUS
(EFIAPI *EFI_MM_WRITE_SAVE_STATE (
  IN  CONST EFI_MM_CPU_PROTOCOL   *This,
   IN  UINTN                        Width,
   IN  EFI_MM_SAVE_STATE_REGISTER Register,
   IN  UINTN                        CpuIndex,
   IN  CONST VOID                   *Buffer
   );
```

Parameters

Width

The number of bytes to write to the CPU save state. If the register specified by *Register* does not support the size specified by *Width*, then **EFI_INVALID_PARAMETER** s returned.

Register

Specifies the CPU register to write to the save state. The type **EFI_MM_SAVE_STATE_REGISTER** is defined in **ReadSaveState()**above. If the specified register is not implemented in the CPU save state map then **EFI_NOT_FOUND** error will be returned.

CpuIndex

Specifies the zero-based index of the CPU save state.

Buffer

Upon entry, this holds the new CPU register value.

Description

This function is used to write the specified number of bytes of the specified register to the CPU save state of the specified CPU and place the value into the buffer. If the CPU does not support the specified register *Register*, then **EFI_NOT_FOUND** should be returned. If the CPU does not support the specified register width *Width*, then **EFI_INVALID_PARAMETER** is returned.

Status Codes Returned

EFI_SUCCESS	The register was read or written from Save State
EFI_NOT_FOUND	The register *Register* is not defined for the Save State of Processor
EFI_INVALID_PARAMETER	Input parameters are not valid. For example: *ProcessorIndex* or *Width* is not correct. *This* or *Buffer* is **NULL**.

4.3.1 MM Save State IO Info

EFI_MM_SAVE_STATE_IO_INFO

Summary

Describes the I/O operation which was in process when the MMI was generated.

Prototype

```
typedef struct _EFI_MM_SAVE_STATE_IO_INFO {
  UINT64                          IoData;
  UINT16                          IoPort;
  EFI_MM_SAVE_STATE_IO_WIDTH     IoWidth;
  EFI_MM_SAVE_STATE_IO_TYPE      IoType;
} EFI_MM_SAVE_STATE_IO_INFO
```

Parameters

IoData

For input instruction (IN, INS), this is data read before the MMI occurred. For output instructions (OUT, OUTS) this is data that was written before the MMI occurred. The width of the data is specified by *IoWidth*. The data buffer is allocated by the Called MMfunction, and it is the Caller's responsibility to free this buffer.

IoPort

The I/O port that was being accessed when the MMI was triggered.

IoWidth

Defines the size width (UINT8, UINT16, UINT32, UINT64) for *IoData*. See Related Definitions.

IoType

Defines type of I/O instruction. See Related Definitions.

Description

This is the structure of the data which is returned when **ReadSaveState()** is called with **EFI_MM_SAVE_STATE_REGISTER_IO**. If there was no I/O then **ReadSaveState()** will return **EFI_NOT_FOUND**.

Related Definitions

```
typedef enum {
  EFI_MM_SAVE_STATE_IO_WIDTH_UINT8        = 0,
  EFI_MM_SAVE_STATE_IO_WIDTH_UINT16       = 1,
  EFI_MM_SAVE_STATE_IO_WIDTH_UINT32       = 2,
  EFI_MM_SAVE_STATE_IO_WIDTH_UINT64       = 3
} EFI_MM_SAVE_STATE_IO_WIDTH


typedef enum {
  EFI_MM_SAVE_STATE_IO_TYPE_INPUT         = 1,
  EFI_MM_SAVE_STATE_IO_TYPE_OUTPUT        = 2,
  EFI_MM_SAVE_STATE_IO_TYPE_STRING        = 4,
  EFI_MM_SAVE_STATE_IO_TYPE_REP_PREFIX    = 8
} EFI_MM_SAVE_STATE_IO_TYPE
```

4.4 MM CPU I/O Protocol

EFI_MM_CPU_IO_PROTOCOL

Summary

Provides CPU I/O and memory access within SM

GUID

```
#define EFI_MM_CPU_IO_PROTOCOL_GUID \
  { 0x3242a9d8, 0xce70, 0x4aa0, \
  0x95, 0x5d, 0x5e, 0x7b, 0x14, 0xd, 0xe4, 0xd2 }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_CPU_IO_PROTOCOL {
  EFI_MM_IO_ACCESS    Mem;
  EFI_MM_IO_ACCESS    Io;
} EFI_MM_CPU_IO_PROTOCOL;
```

Parameters

Mem

Allows reads and writes to memory-mapped I/O space. See the **Mem()** function description. Type **EFI_MM_IO_ACCESS** is defined in “Related Definitions” below.

Io

Allows reads and writes to I/O space. See the **Io()** function description. Type **EFI_MM_IO_ACCESS** is defined in “Related Definitions” below.

Description

The **EFI_MM_CPU_IO_PROTOCOL** service provides the basic memory, I/O, and PCI interfaces that are used to abstract accesses to devices.

The interfaces provided in **EFI_MM_CPU_IO_PROTOCOL** are for performing basic operations to memory and I/O. The **EFI_MM_CPU_IO_PROTOCOL** can be thought of as the bus driver for the system. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

Related Definitions

```
//*******************************************************
// EFI_MM_IO_ACCESS
//*******************************************************
typedef struct {
  EFI_MM_CPU_IO  Read;
  EFI_MM_CPU_IO  Write;
} EFI_MM_IO_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_MM_CPU_IO_PROTOCOL.Mem()

Summary

Enables a driver to access device registers in the memory space.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_MM_CPU_IO (
  IN CONST EFI_MM_CPU_IO_PROTOCOL    *This,
  IN EFI_MM_IO_WIDTH                 Width,
  IN UINT64                          Address,
  IN UINTN                           Count,
  IN OUT VOID                        *Buffer
  );
```

Parameters

This

The **EFI_MM_CPU_IO_PROTOCOL** instance.

Width

Signifies the width of the I/O operations. Type **EFI_MM_IO_WIDTH** is defined in "Related Definitions" below.

Address

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

Count

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

Buffer

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

Description

The **EFI_MM_CPU_IO.Mem()** function enables a driver to access device registers in the memory.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues that the bus, device, platform, or type of I/O might require. For example, on IA-32 platforms, width requests of **MM_IO_UINT64** do not work.

The *Address* field is the bus relative address as seen by the device on the bus.

Related Definitions

```
//*******************************************************
// EFI_MM_IO_WIDTH
//*******************************************************

typedef enum {
  MM_IO_UINT8  = 0,
  MM_IO_UINT16 = 1,
  MM_IO_UINT32 = 2,
  MM_IO_UINT64 = 3
} EFI_MM_IO_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_UNSUPPORTED	The *Address* is not valid for this system.
EFI_INVALID_PARAMETER	*Width* or *Count*, or both, were invalid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_MM_CPU_IO_PROTOCOL.Io()

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_MM_CPU_IO) (
  IN CONST EFI_MM_CPU_IO_PROTOCOL    *This,
  IN EFI_MM_IO_WIDTH                  Width,
  IN UINT64                           Address,
  IN UINTN                            Count,
  IN OUT VOID                         *Buffer
  );
```

Parameters

This

The **EFI_MM_CPU_IO_PROTOCOL** instance.

Width

Signifies the width of the I/O operations. Type **EFI_MM_IO_WIDTH** is defined in **Mem()**.

Address

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

Count

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

Buffer

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

Description

The **EFI_MM_CPU_IO.Io()** function enables a driver to access device registers in the I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for any alignment and I/O width issues which the bus, device, platform, or type of I/O might require. For example, on IA-32 platforms, width requests of **MM_IO_UINT64** do not work.

The caller must align the starting address to be on a proper width boundary.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the device.
EFI_UNSUPPORTED	The *Address* is not valid for this system.
EFI_INVALID_PARAMETER	*Width* or *Count*, or both, were invalid.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

4.5 MM PCI I/O Protocol

EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL

Summary

Provides access to PCI I/O, memory and configuration space inside of SM.

GUID

```
#define EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID \
  {0x8bc1714d, 0xffcb, 0x41c3, \
  0x89, 0xdc, 0x6c, 0x74, 0xd0, 0x6d, 0x98, 0xea}
```

Prototype

```
typedef EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL
EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

Description

This protocol provides the same functionality as the PCI Root Bridge I/O Protocol defined in the *UEFI Specfcation*, section 13.2, except that the functions for **Map()**, **Unmap()**, **Flush()**, **AllocateBuffer()**, **FreeBuffer()**, **SetAttributes()**, and **Configuration()** may return **EFI_UNSUPPORTED**.

4.6 MM Ready to Lock Protocol

EFI_MM_READY_TO_LOCK_PROTOCOL

Summary

Indicates that MM resources and services that should not be used by the third party code are about to be locked.

GUID

```
#define EFI_MM_READY_TO_LOCK_PROTOCOL_GUID \
  { 0x47b7fa8c, 0xf4bd, 0x4af6, \
  {0x82, 0x0, 0x33, 0x30, 0x86, 0xf0, 0xd2, 0xc8 } }
```

Prototype

```
NULL
```

Description

This protocol is a mandatory protocol published by the MM Foundation code when the system is preparing to lock certain resources and interfaces in anticipation of the invocation of 3rd party extensible modules. This protocol is an SM counterpart of the *DXE MM Ready to Lock Protocol*. This protocol prorogates resource locking notification into SM environment. This protocol is installed after installation of the *SM End of DXE Protocol*.

4.7 MM MP protocol

EFI_MM_MP_PROTOCOL

Summary

The MM MP protocol provides a set of functions to allow execution of procedures on processors that have entered MM. This protocol has the following properties:

- The caller can only invoke execution of a procedure on a processor, other than the caller, that has also entered MM.
- It is possible to invoke a procedure on multiple processors.
- Supports blocking and non-blocking modes of operation.

GUID

```
// {5D5450D7-990C-4180-A803-8E63F0608307}
#define EFI_MM_MP_PROTOCOL_GUID  \
   { 0x5d5450d7, 0x990c, 0x4180,
   { 0xa8, 0x3, 0x8e, 0x63, 0xf0, 0x60, 0x83, 0x7 } };
```

Protocol

```
typedef struct _EFI_MM_MP_PROTOCOL {
  UINT32                              Revision,
  UINT32                              Attributes,
  EFI_MM_ GET_NUMBER_OF_PROCESSORS   GetNumberOfProcessors,
  EFI_MM_DISPATCH_PROCEDURE           DispatchProcedure,
  EFI_MM_BROADCAST_PROCEDURE          BroadcastProcedure,
  EFI_MM_SET_STARTUP_PROCEDURE        SetStartupProcedure,
  EFI_CHECK_FOR_PROCEDURE             CheckOnProcedure,
  EFI_WAIT_FOR_PROCEDURE              WaitForProcedure,
}EFI_MM_MP_PROTOCOL;
```

Members

Revision

Revision information for the interface

Attributes

Provides information about the capabilities of the implementation.

GetNumberOfProcessors

Return the number of processors in the system.

DispatchProcedure

Run a procedure on one AP.

BroadcastProcedure

Run a procedure on all processors except the caller.

SetStartupProcedure

Provide a procedure to be executed when an AP starts up from power state where core context and configuration is lost.

CheckOnProcedure

Check whether a procedure on one or all APs has completed.

WaitForProcedure

Wait until a procedure on one or all APs has completed execution.

EFI_MM_MP_PROTOCOL.Revision

Summary

For implementations compliant with this revision of the specification this value must be 0.

EFI_MM_MP_PROTOCOL.Attributes

Summary

This parameter takes the following format:

Field	Number of bits	Bit Offset	Description
Timeout support flag	1	0	This bit describes whether timeouts are supported in **DispatchProcedure** and **BroadcastProcedure** functions. This bit is set to one if timeouts are supported in **DispatchProcedure** and **BroadcastProcedure**. This bit is set to zero if timeouts are not supported in **DispatchProcedure** and **BroadcastProcedure**. In implementations where timeouts are not supported, timeout values are always treated as infinite. See **EFI_MM_MP_TIMEOUT_SUPPORTED** in Related Definitions below.
Reserved	31	1	Reserved must be zero.

EFI_MM_MP_PROTOCOL.GetNumberOfProcessors()

Summary

This service retrieves the number of logical processor in the platform.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_ GET_NUMBER_OF_PROCESSORS) (
    IN  CONST EFI_MM_MP_PROTOCOL  *This,
    OUT UINTN                     *NumberOfProcessors
  );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

NumberOfProcessors

Pointer to the total number of logical processors in the system, including the BSP and all APs.

Status Codes Returned

EFI_SUCCESS	The number of processors was retrieved successfully
EFI_INVALID_PARAMETER	*NumberOfProcessors* is **NULL**

EFI_MM_MP_PROTOCOL.DispatchProcedure()

Summary

This service allows the caller to invoke a procedure one of the application processors (AP). This function uses an optional token parameter to support blocking and non-blocking modes. If the token is passed into the call, the function will operate in a non-blocking fashion and the caller can check for completion with **CheckOnProcedure** or **WaitForProcedure**.

Prototype

```
        typedef
        EFI_STATUS
        (EFIAPI *EFI_MM_DISPATCH_PROCEDURE) (
            IN  CONST EFI_MM_MP_PROTOCOL      *This,
            IN  EFI_AP_PROCEDURE2             Procedure,
            IN  UINTN                         CpuNumber,
            IN  UINTN                         TimeoutInMicroseconds,
            IN  OUT VOID                      *ProcedureArguments
     OPTIONAL,
            IN  OUT MM_COMPLETION             *Token,
            IN  OUT EFI_STATUS                *CPUStatus,
          );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

Procedure

A pointer to the procedure to be run on the designated target AP of the system. Type **EFI_AP_PROCEDURE2** is defined below in related definitions.

CpuNumber

The zero-based index of the processor number of the target AP, on which the code stream is supposed to run. If the number points to the calling processor then it will not run the supplied code.

TimeoutInMicroseconds

Indicates the time limit in microseconds for this AP to finish execution of *Procedure*, either for blocking or non-blocking mode. Zero means infinity. If the timeout expires before this AP returns from *Procedure*, then *Procedure* on the AP is terminated. If the timeout expires in blocking mode, the call returns **EFI_TIMEOUT**. If the timeout expires in non-blocking mode, the timeout determined can be through **CheckOnProcedure** or **WaitForProcedure**.

Note that timeout support is optional. Whether an implementation supports this feature, can be determined via the **Attributes** data member.

ProcedureArguments

Allows the caller to pass a list of parameters to the code that is run by the AP. It is an optional common mailbox between APs and the caller to share information.

Token

This is an optional parameter that allows the caller to execute the procedure in a blocking or non-blocking fashion. If it is **NULL** the call is blocking, and the call will not return until the AP has completed the procedure. If the token is not **NULL**, the call will return immediately. The caller can check whether the procedure has completed with **CheckOnProcedure** or **WaitForProcedure**.

Type **MM_COMPLETION** is defined below in related definitions.

CpuStatus

This optional pointer may be used to get the status code returned by **Procedure** when it completes execution on the target AP, or with **EFI_TIMEOUT** if the **Procedure** fails to complete within the optional timeout. The implementation will update this variable with **EFI_NOT_READY** prior to starting **Procedure** on the target AP.

Status Codes Returned

EFI_SUCCESS	In the blocking case, this indicates that Procedure has completed execution on the target AP. In the non-blocking case this indicates that the procedure has been successfully scheduled for execution on the target AP.
EFI_INVALID_PARAMETER	The input arguments are out of range. Either the target AP is the caller of the function, or the Procedure or Token is NULL
EFI_NOT_READY	If the target AP is busy executing another procedure
EFI_ALREADY_STARTED	Before the AP procedure associated with the Token is finished, the same Token cannot be used to dispatch or broadcast another procedure.
EFI_TIMEOUT	In blocking mode, the timeout expired before the specified AP has finished.

EFI_MM_MP_PROTOCOL.BroadcastProcedure()

Summary

This service allows the caller to invoke a procedure on all running application processors (AP) except the caller. This function uses an optional token parameter to support blocking and non-blocking modes. If the token is passed into the call, the function will operate in a non-blocking fashion and the caller can check for completion with **CheckOnProcedure** or **WaitForProcedure**.

It is not necessary for the implementation to run the procedure on every processor on the platform. Processors that are powered down in such a way that they cannot respond to interrupts, may be excluded from the broadcast.

Prototype

```
        typedef
        EFI_STATUS
        (EFIAPI *EFI_MM_BROADCAST_PROCEDURE) (
            IN   CONST EFI_MM_MP_PROTOCOL      *This,
            IN   EFI_AP_PROCEDURE2             Procedure,
            IN   UINTN                         TimeoutInMicroseconds,
            IN   OUT VOID                      *ProcedureArguments
     OPTIONAL,
            IN   OUT MM_COMPLETION            *Token,
            IN   OUT EFI_STATUS                *CPUStatus,
          );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

Procedure

A pointer to the code stream to be run on the APs that have entered MM. Type **EFI_AP_PROCEDURE2** is defined below in related definitions.

TimeoutInMicroseconds

Indicates the time limit in microseconds for the APs to finish execution of Procedure, either for blocking or non-blocking mode. Zero means infinity. If the timeout expires before all APs return from *Procedure*, then *Procedure* on the failed APs is terminated. If the timeout expires in blocking mode, the call returns **EFI_TIMEOUT**. If the timeout expires in non-blocking mode, the timeout determined can be through **CheckOnProcedure** or **WaitForProcedure**.

Note that timeout support is optional. Whether an implementation supports this feature can be determined via the **Attributes** data member.

ProcedureArguments

Allows the caller to pass a list of parameters to the code that is run by the AP. It is an optional common mailbox between APs and the caller to share information.

Token

This is an optional parameter that allows the caller to execute the procedure in a blocking or non-blocking fashion. If it is **NULL** the call is blocking, and the call will not return until the AP has completed the procedure. If the token is not **NULL**, the call will return immediately. The caller can check whether the procedure has completed with **CheckOnProcedure** or **WaitForProcedure**.

Type **MM_COMPLETION** is defined below in related definitions

CPUStatus

This optional pointer may be used to get the individual status returned by every AP that participated in the broadcast. This parameter if used provides the base address of an array to hold the **EFI_STATUS** value of each AP in the system. The size of the array can be ascertained by the **GetNumberOfProcessors** function.

As mentioned above, the broadcast may not include every processor in the system. Some implementations may exclude processors that have been powered down in such a way that they are not responsive to interrupts. Additionally the broadcast excludes the processor which is making the **BroadcastProcedure** call. For every excluded processor, the array entry must contain a value of **EFI_NOT_STARTED**.

Status Codes Returned

EFI_SUCCESS	In the blocking case, this indicates that *Procedure* has completed execution on the APs. In the non-blocking case this indicates that the procedure has been successfully scheduled for execution on the APs.
EFI_INVALID_PARAMETER	*Procedure* or *Token* is **NULL**
EFI_NOT_READY	If a target AP is busy executing another procedure
EFI_TIMEOUT	In blocking mode, the timeout expired before all enabled APs have finished.
EFI_ALREADY_STARTED	Before the AP procedure associated with the Token is finished, the same Token cannot be used to dispatch or broadcast another procedure.

EFI_MM_MP_PROTOCOL.SetStartupProcedure()

Summary

This service allows the caller to set a startup procedure that will be executed when an AP powers up from a state where core configuration and context is lost. The procedure is execution has the following properties:

- The procedure executes before the processor is handed over to the operating system.
- All processors execute the same startup procedure.
- The procedure may run in parallel with other procedures invoked through the functions in this protocol, or with processors that are executing an MM handler or running in the operating system.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_STARTUP_ PROCEDURE) (
    IN  CONST EFI_MM_MP_PROTOCOL   *This,
    IN  EFI_AP_PROCEDURE           Procedure,
    IN  OUT VOID                  *ProcedureArguments OPTIONAL,
  );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

Procedure

A pointer to the code stream to be run on the designated target AP of the system. Type **EFI_AP_PROCEDURE** is defined below in Volume 2 with the related definitions of **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs**.

If caller may pass a value of **NULL** to deregister any existing startup procedure.

ProcedureArguments

Allows the caller to pass a list of parameters to the code that is run by the AP. It is an optional common mailbox between APs and the caller to share information.

Status Codes Returned

EFI_SUCCESS	The *Procedure* has been set successfully.

EFI_MM_MP_PROTOCOL.CheckOnProcedure()

Summary

When non-blocking execution of a procedure on an AP is invoked with **DispatchProcedure**, via the use of a token, this function can be used to check for completion of the procedure on the AP. The function takes the token that was passed into the **DispatchProcedure** call. If the procedure is complete, and therefore it is now possible to run another procedure on the same AP, this function returns **EFI_SUCESS**. In this case the status returned by the procedure that executed on the AP is returned in the token's *Status* field. If the procedure has not yet completed, then this function returns **EFI_NOT_READY**.

When a non-blocking execution of a procedure is invoked with **BroadcastProcedure**, via the use of a token, this function can be used to check for completion of the procedure on all the broadcast APs. The function takes the token that was passed into the **BroadcastProcedure** call. If the procedure is complete on all broadcast APs this function returns **EFI_SUCESS**. If the procedure has not yet completed on the broadcast APs, the function returns **EFI_NOT_READY**.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_CHECK_ON_PROCEDURE)
    IN  CONST EFI_MM_MP_PROTOCOL          *This,
    IN  OUT MM_COMPLETION                 Token
  );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

Token

This parameter describes the token that was passed into **DispatchProcedure** or **BroadcastProcedure**.

Type **MM_COMPLETION** is defined below in related definitions.

Status Codes Returned

EFI_SUCCESS	*Procedure* has completed.
EFI_NOT_READY	The *Procedure* has not completed.
EFI_INVALID_PARAMETER	*Token* is **NULL**
EFI_NOT_FOUND	*Token* is not currently in use for a non-blocking call

EFI_MM_MP_PROTOCOL.WaitForProcedure()

Summary

When a non-blocking execution of a procedure on an AP is invoked via **DispatchProcedure**, this function will block the caller until the remote procedure has completed on the designated AP. The non-blocking procedure invocation is identified by the *Token* parameter, which must match the token that used when **DispatchProcedure** was called.

When a non-blocking execution of a procedure on an AP is invoked via **BroadcastProcedure** this function will block the caller until the remote procedure has completed on all of the APs that entered MM. The non-blocking procedure invocation is identified by the *Token* parameter, which must match the token that used when **BroadcastProcedure** was called.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_WAIT_FOR_PROCEDURE)
    IN   CONST EFI_MM_MP_PROTOCOL          *This,
    IN   OUT MM_COMPLETION                 Token,
  );
```

Parameters

This

The **EFI_MM_MP_PROTOCOL** instance.

Token

This parameter describes token that was passed into **DispatchProcedure** or **BroadcastProcedure**.

Type **MM_COMPLETION** is defined below in related definitions.

Status Codes Returned

EFI_SUCCESS	The procedure has completed.
EFI_INVALID_PARAMETER	*Token* is **NULL**
EFI_NOT_FOUND	*Token* is not currently in use for a non-blocking call

Related Definitions

EFI_AP_PROCEDURE is defined in Volume 2, with **EFI_MP_SERVICES_PROTOCOL.StartupAllAPs** Related Definitions.

```
// Attribute flags
#define EFI_MM_MP_TIMEOUT_SUPPORTED  0x1

// Procedure callback
typedef
EFI_STATUS
(EFIAPI                  *EFI_AP_PROCEDURE2)(
IN VOID                  *ProcedureArgument
);

// completion token
typedef VOID             * MM_COMPLETION;
```

4.8 MM Configuration Protocol

EFI_MM_CONFIGURATION_PROTOCOL

Summary

Register MM Foundation entry point.

GUID

```
#define EFI_MM_CONFIGURATION_PROTOCOL_GUID { \
  0x26eeb3de, 0xb689, 0x492e, \
  0x80, 0xf0, 0xbe, 0x8b, 0xd7, 0xda, 0x4b, 0xa7
}
```

Prototype

```
typedef struct _EFI_MM_CONFIGURATION_PROTOCOL {
EFI_MM_REGISTER_MM_FOUNDATION_ENTRY RegisterMmFoundationEntry;
} EFI_MM_CONFIGURATION_PROTOCOL;
```

Members

RegisterMmFoundationEntry

A function to register the MM Foundation entry point.

Description

This Protocol is an MM Protocol published by a standalone MM CPU driver to allow MM Foundation register MM Foundation entry point. If a platform chooses to let MM Foundation load standalone MM CPU driver for MM relocation, this protocol must be produced this standalone MM CPU driver.

The **RegisterMmFoundationEntry()** function allows the MM Foundation to register the MM Foundation entry point with the MM entry vector code.

EFI_MM_CONFIGURATION_PROTOCOL.RegisterMmFoundationEntry( )

Summary

Register the MM Foundation entry point in MM standalone mode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_REGISTER_MM_FOUNDATION_ENTRY) (
 IN CONST EFI_MM_CONFIGURATION_PROTOCOL      *This,
 IN EFI_MM_ENTRY_POINT                       MmEntryPoint
 )
```

Parameters

This

The **EFI_MM_CONFIGURATION_PROTOCOL** instance.

MmEntryPoint

MM Foundation entry point.

Description

This function registers the MM Foundation entry point with the processor code. This entry point will be invoked by the MM Processor entry code as defined in section 2.5.

Status Codes Returned

EFI_SUCCESS	The entry-point was successfully registered.

4.9 MM End Of PEI Protocol

EFI_MM_END_OF_PEI_PROTOCOL

Summary

Indicate that the UEFI/PI firmware is about to exit PEI phase.

GUID

```
#define EFI_MM_END_OF_PEI_PROTOCOL_GUID { \
 0xf33e1bf3, 0x980b, 0x4bfb, 0xa2, 0x9a, 0xb2, 0x9c, 0x86, 0x45,
0x37, 0x32 \
}
```

Prototype

```
NULL
```

Description

This protocol is a MM Protocol published by a standalone MM Foundation code if MM Foundation is loaded in PEI phase. This protocol should be installed immediately after DXE IPL installs **EFI_PEI_END_OF_PEI_PHASE_PPI**.

4.10 MM UEFI Ready Protocol

EFI_MM_UEFI_READY_PROTOCOL

Summary

Indicate that the UEFI/PI firmware is in UEFI phase and **EFI_SYSTEM_TABLE** is ready to use.

GUID

```
#define EFI_MM_UEFI_READY_PROTOCOL_GUID { \
 0xc63a953b, 0x73b0, 0x482f, 0x8d, 0xa6, 0x76, 0x65, 0x66, 0xf6,
0x5a, 0x82 \
}
```

Prototype

```
NULL
```

Description

This protocol is a MM Protocol published by a standalone MM Foundation code after DXE MM IPL communicates with MM Foundation to tell MM Foundation UEFI system table location. After that tradition MM driver can be dispatched.

4.11 MM Ready To Boot Protocol

EFI_MM_READY_TO_BOOT_PROTOCOL

Summary

Indicate that the UEFI/PI firmware is about to load and execute a boot option.

GUID

```
#define EFI_MM_READY_TO_BOOT_PROTOCOL_GUID { \
 0x6e057ecf, 0xfa99, 0x4f39, 0x95, 0xbc, 0x59, 0xf9, 0x92, 0x1d,
0x17, 0xe4 \
}
```

Prototype

```
NULL
```

Description

This protocol is a MM Protocol published by a standalone MM Foundation code, when UEFI/PI firmware is about to load and execute a boot option. There is an associated event GUID that is signaled for the DXE drivers called **EFI_EVENT_GROUP_READY_TO_BOOT**.

4.12 MM Exit Boot Services Protocol

EFI_MM_EXIT_BOOT_SERVICES_PROTOCOL

Summary

Indicate that the UEFI/PI firmware is about to enter UEFI runtime phase.

GUID

```
#define EFI_MM_EXIT_BOOT_SERVICES_PROTOCOL_GUID { \
0x296eb418, 0xc4c8, 0x4e05, 0xab, 0x59, 0x39, 0xe8, 0xaf, 0x56,
0xf0, 0xa \
}
```

Prototype

```
NULL
```

Description

This protocol is a MM Protocol published by a standalone MM Foundation code, when UEFI/PI firmware is about to enter UEFI runtime phase. There is an associated event GUID that is signaled for the DXE drivers called **EFI_EVENT_GROUP_EXIT_BOOT_SERVICES**.

4.13 MM Security Architecture Protocol

EFI_MM_SECURITY_ARCHITECTURE_PROTOCOL

Summary

Abstracts security-specific functions from the MM Foundation for purposes of handling GUIDed section encapsulations in standalone mode. This protocol must be produced by a MM driver and may only be consumed by the MM Foundation and any other MM drivers that need to validate the authentication of files.

GUID

```
#define EFI_MM_SECURITY_ARCH_PROTOCOL_GUID  { \
 0xb48e70a3, 0x476f, 0x486d, 0xb9, 0xc0, 0xc2, 0xd0, 0xf8, 0xb9,
0x44, 0xd9 \
}
```

Prototype

Same as **EFI_SECURITY_ARCH_PROTOCOL.**

Description

The **EFI_MM_SECURITY_ARCH_PROTOCOL** is used to abstract platform-specific policy from the MM Foundation in standalone mode. This includes locking flash upon failure to authenticate, attestation logging, and other exception operations.

The usage is same as DXE **EFI_SECURITY_ARCH_PROTOCOL**.

4.14 MM End of DXE Protocol

EFI_MM_END_OF_DXE_PROTOCOL

Summary

Indicates end of the execution phase when all of the components are under the authority of the platform manufacturer.

GUID

```
#define EFI_MM_END_OF_DXE_PROTOCOL_GUID \
{ 0x24e70042, 0xd5c5, 0x4260, \
{ 0x8c, 0x39, 0xa, 0xd3, 0xaa, 0x32, 0xe9, 0x3d } }
```

Prototype

NULL

Description

This protocol is a mandatory protocol published by MM Foundation code. This protocol is an MM counterpart of the End of DXE Event. This protocol prorogates End of DXE notification into MM environment. This protocol is installed prior to installation of the MM Ready to Lock Protocol.

4.15 MM Handler State Notification Protocol

EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL

Summary

Register or unregister a MM Handler State notification function

GUID

```
// {30C8340F-4C30-41D9-BFAE-444ACB2C1F76}
#define EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL_GUID \
 {0x30c8340f, 0x4c30, 0x41d9, {0xbf, 0xae, 0x44, 0x4a, \
                                0xcb, 0x2c, 0x1f, 0x76}}
```

Prototype

```
typedef
struct _EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL {
  EFI_MM_HANDLER_STATE_NOTIFIER_REGISTER
HandlerStateNotifierRegister;
  EFI_MM_HANDLER_STATE_NOTIFIER_UNREGISTER
HandlerStateNotifierUnregister;
} EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL;
```

Members

HandlerStateNotifierRegister

Register notification function

HandlerStateNotifierUnRegister

Un-register previously registered notification function

Description

This protocol is an MM Protocol published by Standalone MM Foundation code if MM Foundation is loaded in SEC Phase. This protocol must be installed before any MM Standalone or Traditional drivers are initialized in MMRAM.

The MM Handler State notification protocol provides a set of functions to allow registration and un-registration of a notification procedure that is invoked whenever a MM Handler is registered or un-registered through an invocation of *MmiHandlerRegister()* or *MmiHandlerUnRegister()* services in the MMST.

EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL. HandlerStateNotifierRegister

Summary

Register notification function

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_HANDLER_STATE_NOTIFIER_REGISTER)(
  IN  EFI_MM_HANDLER_STATE_NOTIFY_FN  Notifier,
  OUT VOID                            **Registration
);
```

Parameters

Notifier

Pointer to function to invoke whenever *MmiHandlerRegister()* or *MmiHandlerUnRegister()* in the MMST are called. Type **EFI_MM_HANDLER_STATE_NOTIFY_FN** is defined in "Related Definitions" below

Registration

Returns the registration record that has been successfully added

Description

This service registers a function (*Notifier*) which will be called whenever a MM Handler is registered or un-registered through invocations of *MmiHandlerRegister()* or *MmiHandlerUnRegister()* in the MMST. On a successful return, *Registration* contains a record that can be later used to unregister the *Notifier* function.

Related Definitions

```
//*************************************************
// EFI_MM_HANDLER_STATE_NOTIFY_FN
//*************************************************
typedef
EFI_STATUS
(EFIAPI *EFI_MM_HANDLER_STATE_NOTIFY_FN)(
  IN EFI_MM_HANDLER_ENTRY_POINT     Handler,
  IN CONST EFI_GUID                *HandlerType OPTIONAL,
  IN EFI_MM_HANDLER_STATE           HandlerState
  );
```

Handler

MM Handler function pointer. The *Handler* parameter is the same as the one passed to *MmiHandlerRegister()*.Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in Volume 4 of the Platform Initialization Specification.

HandlerType

Points to an **EFI_GUID** which describes the type of invocation that this handler is for or **NULL** to indicate a root MMI handler. The *HandlerType* parameter is the same as the one passed to *MmiHandlerRegister()*.

HandlerState

Handler state i.e. registered or unregistered. Type **EFI_MM_HANDLER_STATE** is defined below. This parameter indicates whether **EFI_MM_HANDLER_STATE_NOTIFY_FN** has been invoked in response to *MmiHandlerRegister()*or *MmiHandlerUnRegister()*.

```
//************************************************
// EFI_MM_HANDLER_STATE
//************************************************
typedef enum {
  HandlerRegistered,
  HandlerUnregistered,
} EFI_MM_HANDLER_STATE;
```

Status Codes Returned (EFI_MM_HANDLER_STATE_NOTIFY_FN)

EFI_SUCCESS	Notification function was successfully invoked
EFI_INVALID_PARAMETER	*Handler* is **NULL** or *HandlerType* is unrecognized

Status Codes Returned (HandlerStateNotifierRegister)

EFI_SUCCESS	Notification function registered successfully
EFI_INVALID_PARAMETER	Registration is **NULL**
EFI_OUT_OF_RESOURCES	Not enough memory resource to finish the request

EFI_MM_HANDLER_STATE_NOTIFICATION_PROTOCOL. HandlerStateNotifierUnregister

Summary

Un-register notification function.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_HANDLER_STATE_NOTIFIER_UNREGISTER)(
  IN VOID                                     *Registration
);
```

Parameters

Registration

Registration record returned upon successfully registering the callback function

Description

This service un-registers a previously registered `Notifier` function. The function is identified by the `Registration` parameter. Subsequent invocations of `MmiHandlerRegister()` or `MmiHandlerUnRegister()` will not invoke the Notifier function.

Status Codes Returned (HandlerStateNotifierUnRegister)

EFI_SUCCESS	Notification function un-registered successfully
EFI_INVALID_PARAMETER	Registration is NULL
EFI_NOT_FOUND	Registration record not found

5 UEFI Protocols

5.1 Introduction

The services described in this Mode chapter describe a series of protocols that locate the MMST, manipulate the Management RAM (MMRAM) apertures, and generate MMIs. Some of these protocols provide only boot services while others have both boot services and runtime services.

The following protocols are defined in this chapter:

- **EFI_MM_BASE_PROTOCOL**
- **EFI_MM_ACCESS_PROTOCOL**
- **EFI_MM_CONTROL_PROTOCOL**
- **EFI_MM_CONFIGURATION_PROTOCOL**
- **EFI_MM_COMMUNICATION_PROTOCOL**

5.2 EFI MM Base Protocol

EFI_MM_BASE_PROTOCOL

Summary

This protocol is used to locate the MMST during MM Driver Initialization.

GUID

```
#define EFI_MM_BASE_PROTOCOL_GUID \
  { 0xf4ccbfb7, 0xf6e0, 0x47fd, \
  0x9d, 0xd4, 0x10, 0xa8, 0xf1, 0x50, 0xc1, 0x91 }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_BASE_PROTOCOL {
  EFI_MM_INSIDE_OUT                  InMm;
  EFI_MM_GET_MMST_LOCATION           GetMmstLocation;
} EFI_MM_BASE_PROTOCOL;
```

Parameters

InMm

Detects whether the caller is inside or outside of MMRAM. See the **InMm()** function description.

GetMmstLocation

Retrieves the location of the Management Mode System Table (MMST). See the **GetMmstLocation()** function description.

Description

The **EFI_MM_BASE_PROTOCOL** is provided by the MM IPL driver. It is a required protocol. It will be utilized by all MM Drivers to locate the MM infrastructure services and determine whether the driver is being invoked as a DXE or MM Driver.

EFI_MM_BASE_PROTOCOL.InMm()

Summary

Service to indicate whether the driver is currently executing in the MM Driver Initialization phase.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_INSIDE_OUT) (
  IN CONST EFI_MM_BASE_PROTOCOL  *This,
  OUT BOOLEAN                      *InMmram
  )
```

Parameters

This

The **EFI_MM_BASE_PROTOCOL** instance.

InMmram

Pointer to a Boolean which, on return, indicates that the driver is currently executing inside of MMRAM (TRUE) or outside of MMRAM (FALSE).

Description

This service returns whether the caller is being executed in the MM Driver Initialization phase. For MM Drivers, this will return **TRUE** in *InMmram* while inside the driver's entry point and otherwise **FALSE**. For combination MM/DXE drivers, this will return **FALSE** in the DXE launch. For the MM launch, it behaves as an MM Driver.

Status Codes Returned

EFI_SUCCESS	The call returned successfully.
EFI_INVALID_PARAMETER	*InMmram* was **NULL**.

EFI_MM_BASE_PROTOCOL.GetMmstLocation()

Summary

Returns the location of the Management Mode Service Table (MMST).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_GET_MMST_LOCATION) (
  IN     CONST EFI_MM_BASE_PROTOCOL  *This,
  IN OUT EFI_MM_SYSTEM_TABLE         **Mmst
  )
```

Parameters

This

The **EFI_MM_BASE_PROTOCOL** instance.

Mmst

On return, points to a pointer to the Management Mode Service Table (MMST).

Description

This function returns the location of the Management Mode Service Table (MMST). The use of the API is such that a driver can discover the location of the MMST in its entry point and then cache it in some driver global variable so that the MMST can be invoked in subsequent handlers.

Status Codes Returned

EFI_SUCCESS	The memory was returned to the system.
EFI_INVALID_PARAMETER	*Mmst* was invalid.
EFI_UNSUPPORTED	Not in MM.

5.3 MM Access Protocol

EFI_MM_ACCESS_PROTOCOL

Summary

This protocol is used to control the visibility of the MMRAM on the platform.

GUID

```
#define EFI_MM_ACCESS_PROTOCOL_GUID \
  { 0xc2702b74, 0x800c, 0x4131, \
  0x87, 0x46, 0x8f, 0xb5, 0xb8, 0x9c, 0xe4, 0xac }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_ACCESS_PROTOCOL {
  EFI_MM_OPEN            Open;
  EFI_MM_CLOSE           Close;
  EFI_MM_LOCK            Lock;
  EFI_MM_CAPABILITIES    GetCapabilities;
  BOOLEAN                LockState;
  BOOLEAN                OpenState;
} EFI_MM_ACCESS_PROTOCOL;
```

Parameters

Open

Opens the MMRAM. See the **Open()** function description.

Close

Closes the MMRAM. See the **Close()** function description.

Lock

Locks the MMRAM. See the **Lock()** function description.

GetCapabilities

Gets information about all MMRAM regions. See the **GetCapabilities()** function description.

LockState

Indicates the current state of the MMRAM. Set to **TRUE** if MMRAM is locked.

OpenState

Indicates the current state of the MMRAM. Set to **TRUE** if MMRAM is open.

Description

The **EFI_MM_ACCESS_PROTOCOL** abstracts the location and characteristics of MMRAM. The platform should report all MMRAM via **EFI_MM_ACCESS_PROTOCOL**. The principal functionality found in the memory controller includes the following:

- Exposing the MMRAM to all non-MM agents, or the "open" state
- Hiding the MMRAM to all but the MM agents, or the "closed" state
- Securing or "locking" the MMRAM, such that the settings cannot be changed by either boot service or runtime agents

EFI_MM_ACCESS_PROTOCOL.Open()

Summary

Opens the MMRAM area to be accessible by a boot-service driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_OPEN) (
  IN EFI_MM_ACCESS_PROTOCOL *This
  );
```

Parameters

This

The **EFI_MM_ACCESS_PROTOCOL** instance.

Description

This function “opens” MMRAM so that it is visible while not inside of MM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of MMRAM. The function should return **EFI_DEVICE_ERROR** if the MMRAM configuration is locked.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of MMRAM.
EFI_DEVICE_ERROR	MMRAM cannot be opened, perhaps because it is locked.

EFI_MM_ACCESS_PROTOCOL.Close()

Summary

Inhibits access to the MMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_CLOSE) (
  IN EFI_MM_ACCESS_PROTOCOL   *This
  );
```

Parameters

This

The **EFI_MM_ACCESS_PROTOCOL** instance.

Description

This function "closes" MMRAM so that it is not visible while outside of MM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of MMRAM.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of MMRAM.
EFI_DEVICE_ERROR	MMRAM cannot be closed.

EFI_MM_ACCESS_PROTOCOL.Lock()

Summary

Inhibits access to the MMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_LOCK) (
  IN EFI_MM_ACCESS_PROTOCOL *This
  );
```

Parameters

This

The **EFI_MM_ACCESS_PROTOCOL** instance.

Description

This function prohibits access to the MMRAM region. This function is usually implemented such that it is a write-once operation.

Status Codes Returned

EFI_SUCCESS	The device was successfully locked.
EFI_UNSUPPORTED	The system does not support locking of MMRAM.

EFI_MM_ACCESS_PROTOCOL.GetCapabilities()

Summary

Queries the memory controller for the regions that will support MMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_CAPABILITIES) (
  IN CONST EFI_MM_ACCESS_PROTOCOL    *This,
  IN OUT UINTN                       *MmramMapSize,
  IN OUT EFI_MMRAM_DESCRIPTOR        *MmramMap
  );
```

Parameters

This

The **EFI_MM_ACCESS_PROTOCOL** instance.

MmramMapSize

A pointer to the size, in bytes, of the *MmramMemoryMap* buffer. On input, this value is the size of the buffer that is allocated by the caller. On output, it is the size of the buffer that was returned by the firmware if the buffer was large enough, or, if the buffer was too small, the size of the buffer that is needed to contain the map.

MmramMap

A pointer to the buffer in which firmware places the current memory map. The map is an array of **EFI_MMRAM_DESCRIPTOR**s. Type **EFI_MMRAM_DESCRIPTOR** is defined in “Related Definitions” below.

Description

This function describes the MMRAM regions.

This data structure forms the contract between the MM Access and MM IPL drivers. There is an ambiguity when any MMRAM region is remapped. For example, on some chipsets, some MMRAM regions can be initialized at one physical address but is later accessed at another processor address. There is currently no way for the MM IPL driver to know that it must use two different addresses depending on what it is trying to do. As a result, initial configuration and loading can use the physical address *PhysicalStart* while MMRAM is open. However, once the region has been closed and needs to be accessed by agents in MM, the *CpuStart* address must be used.

This protocol publishes the available memory that the chipset can shroud for the use of installing code.

These regions serve the dual purpose of describing which regions have been open, closed, or locked. In addition, these regions may include overlapping memory ranges, depending on the chipset implementation. The latter might include a chipset that supports T-SEG, where memory near the top of the physical DRAM can be allocated for MMRAM too.

The key thing to note is that the regions that are described by the protocol are a subset of the capabilities of the hardware.

Related Definitions

```
//*******************************************************
//EFI_MMRAM_STATE
//*******************************************************
//
// Hardware state
//
#define  EFI_MMRAM_OPEN                        0x00000001
#define  EFI_MMRAM_CLOSED                      0x00000002
#define  EFI_MMRAM_LOCKED                      0x00000004
//
// Capability
//
#define  EFI_CACHEABLE                         0x00000008
//
// Logical usage
//
#define  EFI_ALLOCATED                         0x00000010
//
// Directive prior to usage
//
#define  EFI_NEEDS_TESTING                     0x00000020
#define  EFI_NEEDS_ECC_INITIALIZATION          0x00000040


//*******************************************************
// EFI_MMRAM_DESCRIPTOR
//*******************************************************
typedef struct _EFI_MMRAM_DESCRIPTOR {
  EFI_PHYSICAL_ADDRESS  PhysicalStart;
  EFI_PHYSICAL_ADDRESS  CpuStart;
  UINT64                PhysicalSize;
  UINT64                RegionState;
} EFI_MMRAM_DESCRIPTOR;
```

PhysicalStart

Designates the physical address of the MMRAM in memory. This view of memory is the same as seen by I/O-based agents, for example, but it may not be the address seen by the processors. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *UEFI Specification.*

CpuStart

Designates the address of the MMRAM, as seen by software executing on the processors. This address may or may not match *PhysicalStart*.

PhysicalSize

Describes the number of bytes in the MMRAM region.

RegionState

Describes the accessibility attributes of the MMRAM. These attributes include the hardware state (e.g., Open/Closed/Locked), capability (e.g., cacheable), logical allocation (e.g., allocated), and pre-use initialization (e.g., needs testing/ECC initialization).

Status Codes Returned

EFI_SUCCESS	The chipset supported the given resource.
EFI_BUFFER_TOO_SMALL	The *MmramMap* parameter was too small. The current buffer size needed to hold the memory map is returned in *MmramMapSize*.

5.4 MM Control Protocol

EFI_MM_CONTROL_PROTOCOL

Summary

This protocol is used initiate synchronous MMIs.

GUID

```
#define EFI_MM_CONTROL_PROTOCOL_GUID \
  { 0x843dc720, 0xab1e, 0x42cb, \
  0x93, 0x57, 0x8a, 0x0, 0x78, 0xf3, 0x56, 0x1b }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_CONTROL_PROTOCOL {
  EFI_MM_ACTIVATE             Trigger;
  EFI_MM_DEACTIVATE           Clear;
  EFI_MM_PERIOD               MinimumTriggerPeriod;
} EFI_MM_CONTROL_PROTOCOL;
```

Parameters

Trigger

Initiates the MMI. See the **Trigger()** function description.

Clear

Quiesces the MMI source. See the **Clear()** function description.

MinimumTriggerPeriod

Minimum interval at which the platform can set the period. A maximum is not specified. That is, the MM infrastructure code can emulate a maximum interval that is greater than the hardware capabilities by using software emulation in the MM infrastructure code. Type **EFI_MM_PERIOD** is defined in "Related Definitions" below.

Description

The **EFI_MM_CONTROL_PROTOCOL** is produced by a runtime driver. It provides an abstraction of the platform hardware that generates an MMI. There are often I/O ports that, when accessed, will generate the MMI. Also, the hardware optionally supports the periodic generation of these signals.

Related Definitions

```
//*************************************************
// EFI_MM_PERIOD
//*************************************************
typedef UINTN EFI_MM_PERIOD;
```

Note: *The period is in increments of 10 ns.*

EFI_MM_CONTROL_PROTOCOL.Trigger()

Summary

Invokes MMI activation from either the preboot or runtime environment.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_ACTIVATE) (
  IN CONST EFI_MM_CONTROL_PROTOCOL *This,
  IN OUT UINT8                     *CommandPort        OPTIONAL,
  IN OUT UINT8                     *DataPort           OPTIONAL,
  IN BOOLEAN                       Periodic            OPTIONAL,
  IN UINTN                          ActivationInterval OPTIONAL
  );
```

Parameters

This

The **EFI_MM_CONTROL_PROTOCOL** instance.

CommandPort

The value written to the command port; this value corresponds to the *SwMmiInputValue* in the *RegisterContext* parameter for the **Register()** function in the **EFI_MM_SW_DISPATCH_PROTOCOL** and in the *Context* parameter in the call to the **DispatchFunction**, see section 7.2.

DataPort

The value written to the data port; this value corresponds to the *DataPort* member in the *CommBuffer* parameter in the call to the **DispatchFunction**, see section 7.2.

Periodic

Optional mechanism to engender a periodic stream.

ActivationInterval

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function generates an MMI.

Status Codes Returned

EFI_SUCCESS	The MMI has been engendered.
EFI_DEVICE_ERROR	The timing is unsupported.
EFI_INVALID_PARAMETER	The activation period is unsupported.
EFI_INVALID_PARAMETER	The last periodic activation has not been cleared.
EFI_NOT_STARTED	The MM base service has not been initialized.

EFI_MM_CONTROL_PROTOCOL.Clear()

Summary

Clears any system state that was created in response to the **Trigger()** call.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_DEACTIVATE) (
  IN CONST EFI_MM_CONTROL_PROTOCOL  *This,
  IN BOOLEAN                        Periodic OPTIONAL
  );
```

Parameters

This

The **EFI_MM_CONTROL_PROTOCOL** instance.

Periodic

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function acknowledges and causes the deassertion of the MMI activation source.that was initiated by a preceding *Trigger* invocation.

The results of this function update the software state of the communication infrastructure in the runtime code, but it is ignorable from the perspective of the hardware state, though. This distinction stems from the fact that many implementations clear the hardware acknowledge in the MM-resident infrastructure itself and may also have other actions using that same activation hardware generated by MM Drivers. This clear-in MM distinction also avoids having the possible pathology of an asynchronous MMI being received in the time window between the RSM instruction (or other means of exiting MM) that followed the flows engendered by the *Trigger* and the subsequent non-MM resident runtime driver code invocation of the *Clear*.

Status Codes Returned

EFI_SUCCESS	The MMI has been engendered.
EFI_DEVICE_ERROR	The source could not be cleared.
EFI_INVALID_PARAMETER	The service did not support the *Periodic* input argument.

5.5 MM Configuration Protocol

EFI_MM_CONFIGURATION_PROTOCOL

Summary

Reports the portions of MMRAM regions which cannot be used for the MMRAM heap.

GUID

```
#define EFI_MM_CONFIGURATION_PROTOCOL_GUID \
  { 0x26eeb3de, 0xb689, 0x492e, \
  0x80, 0xf0, 0xbe, 0x8b, 0xd7, 0xda, 0x4b, 0xa7 }
```

Prototype

```
typedef struct _EFI_MM_CONFIGURATION_PROTOCOL {
  EFI_MM_RESERVED_MMRAM_REGION   *MmramReservedRegions;
  EFI_MM_REGISTER_MM_ENTRY      RegisterMmEntry;
} EFI_MM_CONFIGURATION_PROTOCOL;
```

Members

MmramReservedRegions

A pointer to an array MMRAM ranges used by the initial MM Entry Point code.

RegisterMmEntry

A function to register the MM Foundation entry point.

Description

This protocol is a mandatory protocol published by a DXE CPU driver to indicate which areas within MMRAM are reserved for use by the CPU for any purpose, such as stack, save state or MM Entry Point.

The *MmramReservedRegions* points to an array of one or more **EFI_MM_RESERVED_MMRAM_REGION** structures, with the last structure having the *MmramReservedSize* set to 0. An empty array would contain only the last structure.

The *RegisterMmEntry()* function allows the MM IPL DXE driver to register the MM Foundation entry point with the MM entry vector code.

Related Definitions

```
typedef struct _EFI_MM_RESERVED_MMRAM_REGION {
  EFI_PHYSICAL_ADDRESS MmramReservedStart;
  UINT64               MmramReservedSize;
} EFI_MM_RESERVED_MMRAM_REGION;
```

MmramReservedStart

Starting address of the reserved MMRAM area, as it appears while MMRAM is open. Ignored if *MmramReservedSize* is 0.

MmramReservedSize

Number of bytes occupied by the reserved MMRAM area. A size of zero indicates the last MMRAM area.

EFI_MM_CONFIGURATION_PROTOCOL.RegisterMmEntry()

Summary

Register the MM Foundation entry point.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_REGISTER_MM_ENTRY) (
  IN CONST EFI_MM_CONFIGURATION_PROTOCOL  *This,
  IN EFI_MM_ENTRY_POINT                   MmEntryPoint
  )
```

Parameters

This

The **EFI_MM_CONFIGURATION_PROTOCOL** instance.

MmEntryPoint

MM Foundation entry point.

Description

This function registers the MM Foundation entry point with the processor code. This entry point will be invoked by the MM Processor entry code as defined in section 2.5.

Status Codes Returned

EFI_SUCCESS	The entry-point was successfully registered.

5.6 DXE MM Ready to Lock Protocol

EFI_DXE_MM_READY_TO_LOCK_PROTOCOL

Summary

Indicates that MM is about to be locked.

GUID

```
#define EFI_DXE_MM_READY_TO_LOCK_PROTOCOL_GUID \
 { 0x60ff8964, 0xe906, 0x41d0, \
 0xaf, 0xed, 0xf2, 0x41, 0xe9, 0x74, 0xe0, 0x8e}
```

Prototype

```
NULL
```

Description

This protocol is a mandatory protocol published by PI platform code.

This protocol in tandem with the *End of DXE Even*t facilitates transition of the platform from the environment where all of the components are under the authority of the platform manufacturer to the environment where third party extensible modules such as UEFI drivers and UEFI applications are executed.

The protocol is published immediately after signaling of the *End of DXE Event*.

PI modules that need to lock or protect their resources in anticipation of the invocation of 3rd party extensible modules should register for notification on installation of this protocol and effect the appropriate protections in their notification handlers. For example, PI platform code may choose to use notification handler to lock MM by invoking **EFI_MM_ACCESS_PROTOCOL.Lock()** function.

5.7 MM Communication Protocol

EFI_MM_COMMUNICATION_PROTOCOL

Summary

This protocol provides a means of communicating between drivers outside of MM and MMI handlers inside of MM.

GUID

```
#define EFI_MM_COMMUNICATION_PROTOCOL_GUID \
  { 0xc68ed8e2, 0x9dc6, 0x4cbd, 0x9d, 0x94, 0xdb, 0x65, \
  0xac, 0xc5, 0xc3, 0x32 }
```

Prototype

```
typedef struct _EFI_MM_COMMUNICATION_PROTOCOL {
  EFI_MM_COMMUNICATE           Communicate;
} EFI_MM_COMMUNICATION_PROTOCOL;
```

Members

Communicate

Sends/receives a message for a registered handler. See the **Communicate()** function description.

Description

This protocol provides runtime services for communicating between DXE drivers and a registered MMI handler.

EFI_MM_COMMUNICATION_PROTOCOL.Communicate()

Summary

Communicates with a registered handler.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_COMMUNICATE) (
  IN CONST EFI_MM_COMMUNICATION_PROTOCOL  *This,
  IN OUT VOID                              *CommBuffer,
  IN OUT UINTN                             *CommSize OPTIONAL
  );
```

Parameters

This

The **EFI_MM_COMMUNICATION_PROTOCOL** instance.

CommBuffer

Pointer to the buffer to convey into MMRAM.

CommSize

The size of the data buffer being passed in. On exit, the size of data being returned. Zero if the handler does not wish to reply with any data. This parameter is optional and may be **NULL**.

Description

This function provides a service to send and receive messages from a registered UEFI service. The **EFI_MM_COMMUNICATION_PROTOCOL** driver is responsible for doing any of the copies such that the data lives in boot-service-accessible RAM.

A given implementation of the **EFI_MM_COMMUNICATION_PROTOCOL** may choose to use the **EFI_MM_CONTROL_PROTOCOL** for effecting the mode transition, or it may use some other method.

The agent invoking the communication interface at runtime may be virtually mapped. The MM infrastructure code and handlers, on the other hand, execute in physical mode. As a result, the non-MM agent, which may be executing in the virtual-mode OS context (as a result of an OS invocation of the UEFI **SetVirtualAddressMap()** service), should use a contiguous memory buffer with a physical address before invoking this service. If the virtual address of the buffer is used, the MM Driver may not know how to do the appropriate virtual-to-physical conversion.

To avoid confusion in interpreting frames, the *CommunicateBuffer* parameter should always begin with **EFI_MM_COMMUNICATE_HEADER**, which is defined in “Related Definitions” below. The header data is mandatory for messages sent **into** the MM agent.

If the *CommSize* parameter is omitted the *MessageLength* field in the **EFI_MM_COMMUNICATE_HEADER**, in conjunction with the size of the header itself, can be used to ascertain the total size of the communication payload.

If the *MessageLength* is zero, or too large for the MM implementation to manage, the MM implementation must update the *MessageLength* to reflect the size of the *Data* buffer that it can tolerate.

If the *CommSize* parameter is passed into the call, but the integer it points to, has a value of 0, then this must be updated to reflect the maximum size of the *CommBuffer* that the implementation can tolerate.

Once inside of MM, the MM infrastructure will call all registered handlers with the same *HandlerType* as the GUID specified by *HeaderGuid* and the *CommBuffer* pointing to *Data*.

This function is not reentrant.

The standard header is used at the beginning of the **EFI_MM_INITIALIATION_HEADER** structure during MM initialization. See "Related Definitions" below for more information.

Related Definitions

```
typedef struct {
  EFI_GUID                          HeaderGuid;
  UINTN                             MessageLength;
  UINT8                             Data[ANYSIZE_ARRAY];
} EFI_MM_COMMUNICATE_HEADER;
```

HeaderGuid

Allows for disambiguation of the message format. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

MessageLength

Describes the size of *Data* (in bytes) and does not include the size of the header..

Data

Designates an array of bytes that is *MessageLength* in size.

```
typedef  struct {
  EFI_MM_COMMUNICATE_HEADER Header;
  EFI_SYSTEM_TABLE          *SystemTable;
} EFI_MM_INITIALIZATION_HEADER;

#define EFI_MM_INITIALIZATION_GUID \
    0x99be0d8f, 0x3548, 0x48aa, \
    {0xb5, 0x77, 0xfc, 0xfb, 0xa5, 0x6a, 0x67, 0xf7}}
```

Header

A standard MM communication buffer header, where *HeaderGuid* is set to **EFI_MM_INITIALIZATION_GUID**.

SystemTable

A pointer to the UEFI System Table. As with DXE driver initialization, there is no guarantee that the entries in this structure which rely on architectural protocols are implemented at the time when this event is generated.

Status Codes Returned

EFI_SUCCESS	The message was successfully posted
EFI_INVALID_PARAMETER	The buffer was **NULL**.
EFI_BAD_BUFFER_SIZE	The buffer is too large for the MM implementation. If this error is returned, the *MessageLength* field in the *CommBuffer* header or the integer pointed by *CommSize*, are updated to reflect the maximum payload size the implementation can accommodate. See the function description above for more details.
EFI_ACCESS_DENIED	The *CommunicateBuffer* parameter or *CommSize* parameter, if not omitted, are in address range that cannot be accessed by the MM environment.

6 PI PEI PPIs

6.1 MM Access PPI

EFI_PEI_MM_ACCESS_PPI

Summary

This PPI is used to control the visibility of the MMRAM on the platform.

GUID

```
#define EFI_PEI_MM_ACCESS_PROTOCOL_GUID { \
  0x268f33a9, 0xcccd, 0x48be, { 0x88, 0x17, 0x86, 0x5, 0x3a,
0xc3, 0x2e, 0xd6 } \
 }
```

PPI Structure

```
typedef struct _EFI_PEI_MM_ACCESS_PPI {
 EFI_PEI_MM_OPEN             Open;
 EFI_PEI_MM_CLOSE            Close;
 EFI_PEI_MM_LOCK             Lock;
 EFI_PEI_MM_CAPABILITIES     GetCapabilities;
 BOOLEAN                     LockState;
 BOOLEAN                     OpenState;
} EFI_PEI_MM_ACCESS_PPI;
```

Parameters

Open

Opens the MMRAM. See the **Open()** function description.

Close

Closes th MMRAM. See the **Close()** function description.

Lock

Locks the MMRAM. See the **Lock()** function description.

GetCapabilities

Gets information about all MMRAM regions. See the **GetCapabilities()** function description.

LockState

Indicates the current state of the MMRAM. Set to **TRUE** if MMRAM is locked.

OpenState

Indicates the current state of the MMRAM. Set to **TRUE** if MMRAM is open.

Description

The **EFI_PEI_MM_ACCESS_PPI** abstracts the location and characteristics of MMRAM. The principal functionality found in the memory controller includes the following:

- Exposing the MMRAM to all non-MM agents, or the "open" state
- Shrouding the MMRAM to all but the MM agents, or the "closed" state
- Preserving the system integrity, or "locking" the MMRAM, such that the settings cannot be perturbed by either boot service or runtime agents

EFI_PEI_MM_ACCESS_PPI.Open()

Summary

Opens the MMRAM area to be accessible by a PEIM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_OPEN) (
 IN EFI_PEI_SERVICES                   **PeiServices,
 IN PEI_MM_ACCESS_PPI                  *This,
 IN UINTN                              DescriptorIndex
 );
```

Parameters

PeiServices

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

This

The **EFI_PEI_MM_ACCESS_PPI** instance.

DescriptorIndex

The region of MMRAM to Open.

Description

This function "opens" MMRAM so that it is visible while not inside of MM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of MMRAM. The function should return **EFI_DEVICE_ERROR** if the MMRAM configuration is locked.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of MMRAM.
EFI_DEVICE_ERROR	MMRAM cannot be opened, perhaps because it is locked.

EFI_PEI_MM_ACCESS_PPI.Close()

Summary

Inhibits access to the MMRAM.

Prototype

```
typedef
EFI_STATUS
 (EFIAPI EFI_PEI_MM_CLOSE)(
 IN EFI_PEI_SERVICES                  **PeiServices,
 IN EFI_PEI_MM_ACCESS_PPI             *This
 IN UINTN                             DescriptorIndex
 );
```

Parameters

PeiServices

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

This

The **EFI_PEI_MM_ACCESS_PPI** instance.

DescriptorIndex

The region of MMRAM to Open.

Description

This function “closes” MMRAM so that it is not visible while outside of MM. The function should return **EFI_UNSUPPORTED** if the hardware does not support hiding of MMRAM.

Status Codes Returned

EFI_SUCCESS	The operation was successful.
EFI_UNSUPPORTED	The system does not support opening and closing of MMRAM.
EFI_DEVICE_ERROR	MMRAM cannot be closed.

EFI_PEI_MM_ACCESS_PPI.Lock()

Summary

This function prohibits access to the MMRAM region. This function is usually implemented such that it is a write-once operation.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_LOCK) (
 IN EFI_PEI_SERVICES                  **PeiServices,
 IN EFI_PEI_MM_ACCESS_PPI             *This
 IN UINTN                             DescriptorIndex
 );
```

Parameters

PeiServices

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

This

The **EFI_PEI_MM_ACCESS_PPI** instance.

DescriptorIndex

The region of MMRAM to Lock.

Description

Inhibits access to the MMRAM.

Status Codes Returned

EFI_SUCCESS	The device was successfully locked.
EFI_UNSUPPORTED	The system does not support locking of MMRAM.

EFI_PEI_MM_ACCESS_PPI.GetCapabilities()

Summary

Queries the memory controller for the regions that will support MMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_CAPABILITIES) (
 IN EFI_PEI_SERVICES                    **PeiServices,
 IN CONST EFI_PEI_MM_ACCESS_PPI         *This,
 IN OUT UINTN                           *MmramMapSize,
 IN OUT EFI_MMRAM_DESCRIPTOR            *MmramMap
 );
```

Parameters

PeiServices

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

This

The **EFI_PEI_MM_ACCESS_PPI** instance.

MmramMapSize

A pointer to the size, in bytes, of the *MmramMemoryMap* buffer. On input, this value is the size of the buffer that is allocated by the caller. On output, it is the size of the buffer that was returned by the firmware if the buffer was large enough, or, if the buffer was too small, the size of the buffer that is needed to contain the map.

MmramMap

A pointer to the buffer in which firmware places the current memory map. The map is an array of **EFI_MMRAM_DESCRIPTORs**

Description

This function describes the MMRAM regions.

This data structure forms the contract between the **MM_ACCESS** and **MM_IPL** drivers. There is an ambiguity when any MMRAM region is remapped. For example, on some chipsets, some MMRAM regions can be initialized at one physical address but is later accessed at another processor address. There is currently no way for the MM IPL driver to know that it must use two different addresses depending on what it is trying to do. As a result, initial configuration and loading can use the physical address *PhysicalStart* while MMRAM is open. However, once the region has been closed and needs to be accessed by agents in MM, the *CpuStart* address must be used.

This PPI publishes the available memory that the chipset can shroud for the use of installing code.

These regions serve the dual purpose of describing which regions have been open, closed, or locked. In addition, these regions may include overlapping memory ranges, depending on the chipset implementation. The latter might include a chipset that supports T-SEG, where memory near the top of the physical DRAM can be allocated for MMRAM too.

The key thing to note is that the regions that are described by the PPI are a subset of the capabilities of the hardware.

Status Codes Returned

EFI_SUCCESS	The chipset supported the given resource.
EFI_BUFFER_TOO_SMALL	The *MmramMap* parameter was too small. The current buffer size needed to hold the memory map is returned in *MmramMapSize*.

6.2 MM Control PPI

EFI_PEI_MM_CONTROL_PPI

Summary

This PPI is used initiate synchronous MMI activations. This PPI could be published by a processor driver to abstract the MMI IPI or a driver which abstracts the ASIC that is supporting the APM port.

Because of the possibility of performing MMI IPI transactions, the ability to generate this event from a platform chipset agent is an optional capability for both IA-32 and x64-based systems.

GUID

```
#define EFI_PEI_MM_CONTROL_PPI_GUID { \
 0x61c68702, 0x4d7e, 0x4f43, { 0x8d, 0xef, 0xa7, 0x43, 0x5, 0xce,
0x74, 0xc5 } \
}
```

PPI Structure

```
typedef struct _EFI_PEI_MM_CONTROL_PPI {
 EFI_PEI_MM_ACTIVATE   Trigger;
 EFI_PEI_MM_DEACTIVATE Clear;
} EFI_PEI_MM_CONTROL_PPI;
```

Parameters

Trigger

Initiates the MMI activation. See the **Trigger()** function description.

Clear

Quiesces the MMI activation. See the **Clear()** function description.

Description

The **EFI_PEI_MM_CONTROL_PPI** is produced by a PEIM. It provides an abstraction of the platform hardware that generates an MMI. There are often I/O ports that, when accessed, will generate the MMI. Also, the hardware optionally supports the periodic generation of these signals.

EFI_PEI_MM_CONTROL_PPI.Trigger()

Summary

Invokes PPI activation from the PI PEI environment.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_ACTIVATE) (
  IN EFI_PEI_SERVICES                 **PeiServices,
  IN CONST EFI_PEI_MM_CONTROL_PPI     *This,
  IN OUT INT8                         *ArgumentBuffer OPTIONAL,
  IN OUT UINTN                        *ArgumentBufferSize OPTIONAL,
  IN BOOLEAN                          Periodic OPTIONAL,
  IN UINTN                            ActivationInterval OPTIONAL
 );
```

Parameters

PeiServices

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

This

The **EFI_PEI_MM_CONTROL_PPI** instance.

ArgumentBuffer

The value passed to the MMI handler. This value corresponds to the *SwMmiInputValue* in the *RegisterContext* parameter for the **Register()** function in the **EFI_MM_SW_DISPATCH_PROTOCOL** and in the Context parameter in the call to the **DispatchFunction**, see section 6.2.

ArgumentBufferSize

The size of the data passed in *ArgumentBuffer* or **NULL** if *ArgumentBuffer* is **NULL**.

Periodic

Optional mechanism to engender a periodic stream.

ActivationInterval

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function generates an MMI.

Status Codes Returned

EFI_SUCCESS	The MMI has been engendered.

EFI_DEVICE_ERROR	The timing is unsupported.
EFI_INVALID_PARAMETER	The activation period is unsupported.
EFI_INVALID_PARAMETER	The last periodic activation has not been cleared.
EFI_NOT_STARTED	The MM base service has not been initialized.

EFI_PEI_MM_CONTROL_PPI.Clear()

Summary

Clears any system state that was created in response to the **Trigger()** call.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_DEACTIVATE) (
IN EFI_PEI_SERVICES                     **PeiServices,
IN CONST EFI_PEI_MM_CONTROL_PPI    *This,
 IN BOOLEAN                          Periodic OPTIONAL
 );
```

Parameters

This

The **EFI_PEI_MM_CONTROL_PPI** instance.

Periodic

Optional parameter to repeat at this period one time or, if the *Periodic* Boolean is set, periodically.

Description

This function acknowledges and causes the deassertion of the MMI activation source. that was initiated by a preceding *Trigger* invocation. The results of this function update the software state of the communication infrastructure in the PEIM code, but it is ignorable from the perspective of the hardware state, though. This distinction stems from the fact that many implementations clear the hardware acknowledge in the MM-resident infrastructure itself and may also have other actions using that same activation hardware generated by MM drivers. This clear-in-MM distinction also avoids having the possible pathology of an asynchronous MMI being received in the time window between the RSM instruction (or other means of exiting MM) followed the flows engendered by the Trigger and the subsequent non-MM resident PEIM code invocation of the *Clear*.

Status Codes Returned

EFI_SUCCESS	The MMI has been engendered.
EFI_DEVICE_ERROR	The source could not be cleared.
EFI_INVALID_PARAMETER	The service did not support the *Periodic* input argument.

6.3 EFI DELAYED DISPATCH PPI (Required)

EFI_DELAYED_DISPATCH_PPI (Required)

Summary

Provide timed event service in PEI

GUID

```
#define EFI_DELAYED_DISPATCH_PPI_GUID \
  { \
    0x869c711d, 0x649c, 0x44fe, { 0x8b, 0x9e, 0x2c, 0xbb, 0x29,
0x11, 0xc3, 0xe6} } \
  }
```

PPI Interface Structure

```
struct _EFI_DELAYED_DISPATCH_PPI {
  EFI_DELAYED_DISPATCH_REGISTER      Register;
};
```

Parameters

Register

Register a callback to be called after a minimum delay has occurred.

Description

Define a time based event mechanism for PEI (Pei Delayed Dispatch):

Eliminate spin loops for timing. Instead of blocking for 50 or 100 ms, the driver would request callbacks after 50ms (or 100ms) and allow other PEIM’s to dispatch. Each loop through the PEI dispatcher will attempt to dispatch each of the queued delayed dispatch requests.

Method to schedule a callback after a minimum delay, not a real time callback. Maintains a 64 bit "State" for use by the callback.

PPI has only one function - Register, to register a function to be called after the required delay with the current value of "State".

EFI_DELAYED_DISPATCH_PPI.Register()

Summary

Register a callback to be called after a minimum delay has occurred.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_DELAYED_DISPATCH_REGISTER)(
  IN  EFI_DELAYED_DISPATCH_PPI      *This,
  IN  EFI_DELAYED_DISPATCH_FUNCTION  Function,
  IN  UINT64                        Context,
  OUT UINT32                        Delay
  );
```

Parameters

This

Pointer to the EFI_DELAYED_DISPATCH_PPI instance.

Function

Function to call back.

Context

Context data.

Delay

Delay interval.

Description

This function is invoked by a PEIM to have a handler returned.

Status Codes Returned

EFI_SUCCESS	Function successfully invoked
EFI_INVALID_PARAMETER	One of the arguments is not supported
EFI_OUT_OF_RESOURCES	No more entries

6.4 MM Configuration PPI

EFI_PEI_MM_CONFIGURATION_PPI

Summary

Reports the portions of MMRAM regions which cannot be used for the MMRAM heap.

GUID

```
#define EFI_PEI_MM_CONFIGURATION_PPI_GUID { \
 0xc109319, 0xc149, 0x450e, 0xa3, 0xe3, 0xb9, 0xba, 0xdd, 0x9d,
0xc3, 0xa4 \
}
```

PPI Structure

```
typedef struct _EFI_PEI_MM_CONFIGURATION_PPI {
  EFI_MM_RESERVED_MMRAM_REGION *MmramReservedRegions;
  EFI_PEI_MM_REGISTER_MM_ENTRY RegisterMmEntry;
} EFI_PEI_MM_CONFIGURATION_PPI;
```

Members

MmramReservedRegions

A pointer to an array MMRAM ranges used by the initial MM entry code.

RegisterMmEntry

A function to register the MM Foundation entry point.

Description

This PPI is a PPI published by a CPU PEIM to indicate which areas within MMRAM are reserved for use by the CPU for any purpose, such as stack, save state or MM entry point. If a platform chooses to let a CPU PEIM do MMRAM relocation, this PPI must be produced by this CPU PEIM.

The *MmramReservedRegions* points to an array of one or more **EFI_MM_RESERVED_MMRAM_REGION** structures, with the last structure having the *MmramReservedSize* set to 0. An empty array would contain only the last structure.

The **RegisterMmEntry()** function allows the MM IPL PEIM to register the MM Foundation entry point with the MM entry vector code.

EFI_PEI_MM_CONFIGURATION_PPI.RegisterMmEntry()

Summary

Register the MM Foundation entry point.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_REGISTER_MM_ENTRY) (
 IN CONST EFI_PEI_MM_CONFIGURATION_PPI *This,
 IN EFI_MM_ENTRY_POINT                 MmEntryPoint
 )
```

Parameters

This

The **EFI_PEI_MM_CONFIGURATION_PPI** instance.

MmEntryPoint

MM Foundation entry point.

Description

This function registers the MM Foundation entry point with the processor code. This entry point will be invoked by the MM Processor entry code as defined in section 2.5.

Status Codes Returned

EFI_SUCCESS	The entry-point was successfully registered.

6.5 MM Communication PPI

EFI_PEI_MM_COMMUNICATION_PPI

Summary

This PPI provides a means of communicating between drivers outside of MM and MMI handlers inside of MM in PEI phase.

GUID

```
#define EFI_PEI_MM_COMMUNICATION_PPI_GUID { \
 0xae933e1c, 0xcc47, 0x4e38, \
 { 0x8f, 0xe, 0xe2, 0xf6, 0x1d, 0x26, 0x5, 0xdf } \
}
```

PPI Structure

```
typedef struct _EFI_PEI_MM_COMMUNICATION_PPI {
 EFI_PEI_MM_COMMUNICATE Communicate;
} EFI_PEI_MM_COMMUNICATION_PPI;
```

Members

Communicate

Sends/receives a message for a registered handler. See the **Communicate()** function description.

Description

This PPI provides services for communicating between PEIM and a registered MMI handler.

EFI_PEI_MM_COMMUNICATION_PPI.Communicate()

Summary

Communicates with a registered handler.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_MM_COMMUNICATE) (
 IN CONST EFI_PEI_MM_COMMUNICATION_PPI  *This,
 IN OUT VOID                            *CommBuffer,
 IN OUT UINTN                           *CommSize
 );
```

Parameters

This

The **EFI_PEI_MM_COMMUNICATION_PPI** instance.

CommBuffer

Pointer to the buffer to convey into MMRAM.

CommSize

The size of the data buffer being passed in. On exit, the size of data being returned. Zero if the handler does not wish to reply with any data.

Description

This function provides a service to send and receive messages from a registered PEI service. The **EFI_PEI_MM_COMMUNICATION_PPI** driver is responsible for doing any of the copies such that the data lives in PEI-service-accessible RAM.

A given implementation of the **EFI_PEI_MM_COMMUNICATION_PPI** may choose to use the **EFI_MM_CONTROL_PPI** for effecting the mode transition, or it may use some other method.

The agent invoking the communication interface must be physical/virtually 1:1 mapped.

To avoid confusion in interpreting frames, the *CommBuffer* parameter should always begin with **EFI_MM_COMMUNICATE_HEADER**. The header data is mandatory for messages sent **into** the MM agent.

Once inside of MM, the MM infrastructure will call all registered handlers with the same *HandlerType* as the GUID specified by *HeaderGuid* and the *CommBuffer* pointing to *Data*.

This function is not reentrant.

Status Codes Returned

EFI_SUCCESS	The message was successfully posted
EFI_INVALID_PARAMETER	The buffer was **NULL**

7 MM Child Dispatch Protocols

7.1 Introduction

The services described in this chapter describe a series of protocols that abstract installation of handlers for a chipset-specific MM design. These services are all scoped to be usable only from within MMRAM.

The following protocols are defined in this chapter:

- **EFI_MM_SW_DISPATCH_PROTOCOL**
- **EFI_MM_SX_DISPATCH_PROTOCOL**
- **EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL**
- **EFI_MM_USB_DISPATCH_PROTOCOL**
- **EFI_MM_GPI_DISPATCH_PROTOCOL**
- **EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL**
- **EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL**
- **EFI_MM_IO_TRAP_DISPATCH_PROTOCOL**

MM Drivers which create instances of these protocols should install an instance of the **EFI_DEVICE_PATH_PROTOCOL** on the same handle. This allows other MM Drivers to distinguish between multiple instances of the same child dispatch protocol

7.2 MM Software Dispatch Protocol

EFI_MM_SW_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for a given MMI source generator.

GUID

```
#define EFI_MM_SW_DISPATCH_PROTOCOL_GUID    \
{ 0x18a3c6dc, 0x5eea, 0x48c8, \
0xa1, 0xc1, 0xb5, 0x33, 0x89, 0xf9, 0x89, 0x99}
```

Protocol Interface Structure

```
typedef struct _EFI_MM_SW_DISPATCH_PROTOCOL {
  EFI_MM_SW_REGISTER      Register;
  EFI_MM_SW_UNREGISTER    UnRegister;
  UINTN                   MaximumSwiValue;
} EFI_MM_SW_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

MaximumSwiValue

A read-only field that describes the maximum value that can be used in the **EFI_MM_SW_DISPATCH_PROTOCOL.Register()** service.

Description

The **EFI_MM_SW_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given software. These handlers will respond to software-generated MMI,s, and the maximum software-generated MMI value in the **EFI_MM_SW_REGISTER_CONTEXT** is denoted by *MaximumSwiValue*.

EFI_MM_SW_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given MMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_SW_REGISTER) (
  IN  CONST EFI_MM_SW_DISPATCH_PROTOCOL  *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT          DispatchFunction,
  IN  EFI_MM_SW_REGISTER_CONTEXT          *RegisterContext,
  OUT EFI_HANDLE                          *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_SW_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified software MMI is generated. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()**.

RegisterContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the software MMI input value for which the dispatch function should be invoked. Type **EFI_MM_SW_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the software MMI source specified by *RegisterContext->SwMmiCpuIndex* is detected. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using UnRegister().

If *SwMmiInputValue* is set to **(UINTN) -1** then a unique value will be assigned and returned in the structure. If no unique value can be assigned then **EFI_OUT_OF_RESOURCES** will be returned.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* (and *CommBufferSize*) pointing to an instance of **EFI_MM_SW_CONTEXT** indicating the index of the CPU which generated the software MMI.

Related Definitions

```
//******************************************************
// EFI_MM_SW_CONTEXT
//******************************************************
typedef struct {
 UINTN      SwMmiCpuIndex;
 UINT8      CommandPort;
 UINT8      DataPort;
} EFI_MM_SW_CONTEXT;
```

SwMmiCpuIndex

The 0-based index of the CPU which generated the software MMI.

CommandPort

This value corresponds directly to the *CommandPort* parameter used in the call to **Trigger()**, see section 5.4.

DataPort

This value corresponds directly to the *DataPort* parameter used in the call to **Trigger()**, see section 5.4.

```
//******************************************************
// EFI_MM_SW_REGISTER_CONTEXT
//******************************************************
typedef struct {
  UINTN     SwMmiInputValue;
} EFI_MM_SW_REGISTER_CONTEXT;
```

SwMmiInputValue

A number that is used during the registration process to tell the dispatcher which software input value to use to invoke the given handler.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The SW MMI input value is not within a valid range or is already in use.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.
EFI_OUT_OF_RESOURCES	A unique software MMI value could not be assigned for this dispatch.

EFI_MM_SW_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a software service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_SW_UNREGISTER) (
  IN  CONST EFI_MM_SW_DISPATCH_PROTOCOL  *This,
  IN  EFI_HANDLE                         DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_SW_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called in response to a software MMI.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.3 MM Sx Dispatch Protocol

EFI_MM_SX_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for a given Sx-state source generator.

GUID

```
#define EFI_MM_SX_DISPATCH_PROTOCOL_GUID    \
{ 0x456d2859, 0xa84b, 0x4e47, \
0xa2, 0xee, 0x32, 0x76, 0xd8, 0x86, 0x99, 0x7d }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_SX_DISPATCH_PROTOCOL {
  EFI_MM_SX_REGISTER    Register;
  EFI_MM_SX_UNREGISTER  UnRegister;
} EFI_MM_SX_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_MM_SX_DISPATCH_PROTOCOL** provides the ability to install child handlers to respond to sleep state related events.

EFI_MM_SX_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given Sx source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_SX_REGISTER) (
  IN  CONST EFI_MM_SX_DISPATCH_PROTOCOL    *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT           DispatchFunction,
  IN  CONST EFI_MM_SX_REGISTER_CONTEXT     *RegisterContext,
  OUT EFI_HANDLE                           *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_SX_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified sleep state event occurs. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in the MMST.

RegisterContext

Pointer to the dispatch function's context. The caller in fills this context before calling the **Register()** function to indicate to the **Register()** function on which Sx state type and phase the caller wishes to be called back. For this interface, the Sx driver will call the registered handlers for all Sx type and phases, so the Sx state handler(s) must check the *Type* and *Phase* field of **EFI_MM_SX_REGISTER_CONTEXT** and act accordingly.

DispatchHandle

Handle of the dispatch function, for when interfacing with the parent Sx state MM Driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the sleep state event specified by *RegisterContext* is detected. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to NULL and 0 respectively.

Related Definitions

```
//*******************************************************
// EFI_MM_SX_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  EFI_SLEEP_TYPE    Type;
  EFI_SLEEP_PHASE   Phase;
} EFI_MM_SX_REGISTER_CONTEXT;

//*******************************************************
// EFI_SLEEP_TYPE
//*******************************************************
typedef enum {
  SxS0,
  SxS1,
  SxS2,
  SxS3,
  SxS4,
  SxS5,
  EfiMaximumSleepType
} EFI_SLEEP_TYPE;

//*******************************************************
// EFI_SLEEP_PHASE
//*******************************************************
typedef enum {
  SxEntry,
  SxExit,
  EfiMaximumPhase
} EFI_SLEEP_PHASE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_UNSUPPORTED	The Sx driver or hardware does not support that Sx *Type/Phase.*
EFI_DEVICE_ERROR	The Sx driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The ICHN input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.

EFI_MM_SX_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters an Sx-state service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_SX_UNREGISTER) (
  IN  CONST EFI_MM_SX_DISPATCH_PROTOCOL  *This,
  IN  EFI_HANDLE                         DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_SX_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called in response to sleep event.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.4 MM Periodic Timer Dispatch Protocol

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the periodical timer MMI source generator.

GUID

```
#define EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL_GUID    \
  { 0x4cec368e, 0x8e8e, 0x4d71, \
  0x8b, 0xe1, 0x95, 0x8c, 0x45, 0xfc, 0x8a, 0x53}
```

Protocol Interface Structure

```
typedef struct _EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL {
  EFI_MM_PERIODIC_TIMER_REGISTER     Register;
  EFI_MM_PERIODIC_TIMER_UNREGISTER   UnRegister;
  EFI_MM_PERIODIC_TIMER_INTERVAL     GetNextShorterInterval;
} EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

GetNextShorterInterval

Returns the next MMI tick period that is supported by the chipset. See the **GetNextShorterInterval()** function description.

Description

The **EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given MMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_PERIODIC_TIMER_REGISTER) (
  IN  CONST EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT                DispatchFunction,
  IN  CONST EFI_MM_PERIODIC_TIMER_REGISTER_CONTEXT
*RegisterContext,
  OUT EFI_HANDLE                                   *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when at least the specified amount of time has elapsed. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in theMMST.

RegisterContext

Pointer to the dispatch function's context. The caller fills this context in before calling the **Register()** function to indicate to the **Register()** function the period at which the dispatch function should be invoked. Type **EFI_MM_PERIODIC_TIMER_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when at least the amount of time specified by *RegisterContext* has elapsed. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* pointing to an instance of **EFI_MM_PERIODIC_TIMER_CONTEXT** and *CommBufferSize* pointing to its size.

Related Definitions

```
//******************************************************
// EFI_MM_PERIODIC_TIMER_REGISTER_CONTEXT
//******************************************************

typedef struct {
  UINT64   Period;
  UINT64   MmiTickInterval;
} EFI_MM_PERIODIC_TIMER_REGISTER_CONTEXT;
```

Period

The minimum period of time in 100 nanosecond units that the child gets called. The child will be called back after a time greater than the time *Period*.

MmiTickInterval

The period of time interval between MMIs. Children of this interface should use this field when registering for periodic timer intervals when a finer granularity periodic MMI is desired.

Example: A chipset supports periodic MMIs on every 64 ms or 2 seconds. A child wishes to schedule a periodic MMI to fire on a period of 3 seconds. There are several ways to approach the problem:

The child may accept a 4 second periodic rate, in which case it registers with the following:

```
Period = 40000
MmiTickInterval = 20000
```

The resulting MMI will occur every 2 seconds with the child called back on every second MMI.

Note: *The same result would occur if the child set* **MmiTickInterval = 0**.

The child may choose the finer granularity MMI (64 ms):

```
Period = 30000
MmiTickInterval = 640
```

The resulting MMI will occur every 64 ms with the child called back on every 47th MMI.

Note: *The child driver should be aware that this will result in more MMIs occurring during system runtime, which can negatively impact system performance.*

```
typedef struct _EFI_MM_PERIODIC_TIMER_CONTEXT {
 UINT64              ElapsedTime;
} EFI_MM_PERIODIC_TIMER_CONTEXT;
```

ElapsedTime

The actual time in 100 nanosecond units elapsed since last called. A value of 0 indicates an unknown amount of time.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The ICHN input value is not within a valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a periodic timer service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_PERIODIC_TIMER_UNREGISTER) (
  IN CONST EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL  *This,
  IN EFI_HANDLE                                     DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the time has elapsed.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL. GetNextShorterInterval()

Summary

Returns the next MMI tick period that is supported by the chipset.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_PERIODIC_TIMER_INTERVAL) (
  IN     CONST EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL  *This,
  IN OUT UINT64                                        **MmiTickInterval
  );
```

Parameters

This

Pointer to the **EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL** instance.

MmiTickInterval

Pointer to pointer of the next shorter MMI interval period that is supported by the child. This parameter works as a get-first, get-next field. The first time that this function is called, **MmiTickInterval* should be set to **NULL** to get the longest MMI interval. The returned **MmiTickInterval* should be passed in on subsequent calls to get the next shorter interval period until **MmiTickInterval* **= NULL**.

Description

This service returns the next MMI tick period that is supported by the device. The order returned is from longest to shortest interval period.

Status Codes Returned

EFI_SUCCESS	The service returned successfully.

7.5 MM USB Dispatch Protocol

EFI_MM_USB_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the USB MMI source generator.

GUID

```
#define EFI_MM_USB_DISPATCH_PROTOCOL_GUID    \
  { 0xee9b8d90, 0xc5a6, 0x40a2, \
  0xbd, 0xe2, 0x52, 0x55, 0x8d, 0x33, 0xcc, 0xa1 }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_USB_DISPATCH_PROTOCOL {
  EFI_MM_USB_REGISTER     Register;
  EFI_MM_USB_UNREGISTER   UnRegister;
} EFI_MM_USB_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_MM_USB_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_MM_USB_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for the USB MMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_USB_REGISTER) (
  IN  CONST EFI_MM_USB_DISPATCH_PROTOCOL   *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT           DispatchFunction,
  IN  CONST EFI_MM_USB_REGISTER_CONTEXT    *RegisterContext,
  OUT EFI_HANDLE                           *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_USB_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when a USB-related MMI occurs. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in the MMST.

RegisterContext

Pointer to the dispatch function's context. The caller fills this context in before calling the **Register()** function to indicate to the **Register()** function the USB MMI source for which the dispatch function should be invoked. Type **EFI_MM_USB_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFISpecification.*

Description

This service registers a function (*DispatchFunction*) which will be called when the USB-related MMI specified by *RegisterContext* has occurred. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* containing NULL and *CommBufferSize* containing zero.

Related Definitions

```
//*******************************************************
// EFI_MM_USB_REGISTER_CONTEXT
//*******************************************************

typedef struct {
  EFI_USB_MMI_TYPE            Type;
  EFI_DEVICE_PATH_PROTOCOL    *Device;
} EFI_MM_USB_REGISTER_CONTEXT;
```

Type

Describes whether this child handler will be invoked in response to a USB legacy emulation event, such as port-trap on the PS/2* keyboard control registers, or to a USB wake event, such as resumption from a sleep state. Type **EFI_USB_MMI_TYPE** is defined below.

Device

The device path is part of the context structure and describes the location of the particular USB host controller in the system for which this register event will occur. This location is important because of the possible integration of several USB host controllers in a system. Type **EFI_DEVICE_PATH** is defined in the *UEFI Specification.*

```
//*******************************************************
// EFI_USB_MMI_TYPE
//*******************************************************
typedef enum {
  UsbLegacy,
  UsbWake
} EFI_USB_MMI_TYPE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The ICHN input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or MM) to manage this child.

EFI_MM_USB_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a USB service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_USB_UNREGISTER) (
  IN  CONST EFI_MM_USB_DISPATCH_PROTOCOL     *This,
  IN EFI_HANDLE                              DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_USB_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the USB event occurs.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully unregistered and the MMI source has been disabled, if there are no other registered child dispatch functions for this MMI source.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.6 MM General Purpose Input (GPI) Dispatch Protocol

EFI_MM_GPI_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the General Purpose Input (GPI) MMI source generator.

GUID

```
#define EFI_MM_GPI_DISPATCH_PROTOCOL_GUID    \
{ 0x25566b03, 0xb577, 0x4cbf, \
0x95, 0x8c, 0xed, 0x66, 0x3e, 0xa2, 0x43, 0x80 }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_GPI_DISPATCH_PROTOCOL {
  EFI_MM_GPI_REGISTER     Register;
  EFI_MM_GPI_UNREGISTER   UnRegister;
  UINTN                    NumSupportedGpis;
} EFI_MM_GPI_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

NumSupportedGpis

Denotes the maximum value of inputs that can have handlers attached.

Description

The **EFI_MM_GPI_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types. Several inputs can be enabled. This purpose of this interface is to generate an MMI in response to any of these inputs having a true value provided.

EFI_MM_GPI_DISPATCH_PROTOCOL.Register()

Summary

Registers a child MMI source dispatch function with a parent MM driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_GPI_REGISTER) (
  IN  CONST EFI_MM_GPI_DISPATCH_PROTOCOL   *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT           DispatchFunction,
  IN  CONST EFI_MM_GPI_REGISTER_CONTEXT    *RegisterContext,
  OUT EFI_HANDLE                           *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_GPI_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the specified GPI causes an MMI. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in the MMST.

RegisterContext

Pointer to the dispatch function's context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the GPI MMI source for which the dispatch function should be invoked. Type **EFI_MM_GPI_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an MMI is generated because of one or more of the GPIs specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* pointing to another instance of **EFI_MM_GPI_REGISTER_CONTEXT** describing the GPIs which actually caused the MMI and *CommBufferSize* pointing to the size of the structure.

Related Definitions

```
//*********************************************
// EFI_MM_GPI_REGISTER_CONTEXT
//*********************************************

typedef struct {
  UINT64     GpiNum;
} EFI_MM_GPI_REGISTER_CONTEXT;
```

GpiNum

A number from one of 2^64 possible GPIs that can generate an MMI. A 0 corresponds to logical GPI[0]; 1 corresponds to logical GPI[1]; and *GpiNum* of N corresponds to GPI[N], where N can span from 0 to 2^64-1.

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The GPI input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.

EFI_MM_GPI_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a General Purpose Input (GPI) service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_GPI_UNREGISTER) (
  IN  CONST EFI_MM_GPI_DISPATCH_PROTOCOL     *This,
  IN EFI_HANDLE                               DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_GPI_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the GPI triggers an MMI.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.7 MM Standby Button Dispatch Protocol

EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the standby button MMI source generator.

GUID

```
#define EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL_GUID    \
  { 0x7300c4a1, 0x43f2, 0x4017, \
  0xa5, 0x1b, 0xc8, 0x1a, 0x7f, 0x40, 0x58, 0x5b }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL {
  EFI_MM_STANDBY_BUTTON_REGISTER    Register;
  EFI_MM_STANDBY_BUTTON_UNREGISTER  UnRegister;
} EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL.Register()

Summary

Provides the parent dispatch service for a given MMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_STANDBY_BUTTON_REGISTER) (
  IN  CONST EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL  *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT                     DispatchFunction,
  IN  EFI_MM_STANDBY_BUTTON_REGISTER_CONTEXT         *RegisterContext,
  OUT EFI_HANDLE                                     *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when the standby button is pressed or released. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in the MMST.

RegisterContext

Pointer to the dispatch function’s context. The caller fills in this context before calling the register function to indicate to the register function the standby button MMI source for which the dispatch function should be invoked. Type **EFI_MM_STANDBY_BUTTON_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an MMI is generated because the standby button was pressed or released, as specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to **NULL**.

Related Definitions

```
//*******************************************************
// EFI_MM_STANDBY_BUTTON_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  EFI_STANDBY_BUTTON_PHASE  Phase;
} EFI_MM_STANDBY_BUTTON_REGISTER_CONTEXT;
```

Phase

Describes whether the child handler should be invoked upon the entry to the button activation or upon exit (i.e., upon receipt of the button press event or upon release of the event). This differentiation allows for workarounds or maintenance in each of these execution regimes. Type **EFI_STANDBY_BUTTON_PHASE** is defined below.

```
//*******************************************************
// EFI_STANDBY_BUTTON_PHASE;
//*******************************************************
typedef enum {
  EfiStandbyButtonEntry,
  EfiStandbyButtonExit,
  EfiStandbyButtonMax
} EFI_STANDBY_BUTTON_PHASE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The standby button input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.

EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a child MMI source dispatch function with a parent MM Driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_STANDBY_BUTTON_UNREGISTER) (
  IN CONST EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL *This,
  IN EFI_HANDLE                                  *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the standby button is pressed or released.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.8 MM Power Button Dispatch Protocol

EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL

Summary

Provides the parent dispatch service for the power button MMI source generator.

GUID

```
#define EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL_GUID    \
  { 0x1b1183fa, 0x1823, 0x46a7, \
  0x88, 0x72, 0x9c, 0x57, 0x87, 0x55, 0x40, 0x9d }
```

Protocol Interface Structure

```
typedef struct _EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL {
  EFI_MM_POWER_BUTTON_REGISTER    Register;
  EFI_MM_POWER_BUTTON_UNREGISTER  UnRegister;
} EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched by this protocol. See the **Register()** function description.

UnRegister

Removes a child service that was dispatched by this protocol. See the **UnRegister()** function description.

Description

The **EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL** provides the ability to install child handlers for the given event types.

EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL. Register()

Summary

Provides the parent dispatch service for a given MMI source generator.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_POWER_BUTTON_REGISTER) (
  IN  CONST EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL  *This,
  IN  EFI_MM_HANDLER_ENTRY_POINT                  DispatchFunction,
  IN  EFI_MM_POWER_BUTTON_REGISTER_CONTEXT        *RegisterContext,
  OUT EFI_HANDLE                                   *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when power button is pressed or released. Type **EFI_MM_HANDLER_ENTRY_POINT** is defined in "Related Definitions" in **MmiHandlerRegister()** in theMMST.

RegisterContext

Pointer to the dispatch function’s context. The caller fills in this context before calling the **Register()** function to indicate to the **Register()** function the power button MMI phase for which the dispatch function should be invoked. Type **EFI_MM_POWER_BUTTON_REGISTER_CONTEXT** is defined in "Related Definitions" below.

DispatchHandle

Handle generated by the dispatcher to track the function instance. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service registers a function (*DispatchFunction*) which will be called when an MMI is generated because the power button was pressed or released, as specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**.

The *DispatchFunction* will be called with *Context* set to the same value as was passed into this function in *RegisterContext* and with *CommBuffer* and *CommBufferSize* set to **NULL**.

Related Definitions

```
//*******************************************************
// EFI_MM_POWER_BUTTON_REGISTER_CONTEXT
//*******************************************************
typedef struct {
  EFI_POWER_BUTTON_PHASE  Phase;
} EFI_MM_POWER_BUTTON_REGISTER_CONTEXT;
```

Phase

Designates whether this handler should be invoked upon entry or exit. Type **EFI_POWER_BUTTON_PHASE** is defined in "Related Definitions" below.

```
//*******************************************************
// EFI_POWER_BUTTON_PHASE
//*******************************************************
typedef enum {
  EfiPowerButtonEntry,
  EfiPowerButtonExit,
  EfiPowerButtonMax
} EFI_POWER_BUTTON_PHASE;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered and the MMI source has been enabled.
EFI_DEVICE_ERROR	The driver was unable to enable the MMI source.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The power button input value is not within valid range.
EFI_OUT_OF_RESOURCES	There is not enough memory (system or SM) to manage this child.

EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL.UnRegister()

Summary

Unregisters a power-button service.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_POWER_BUTTON_UNREGISTER) (
  IN  CONST EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL  *This,
  IN  EFI_HANDLE                                  DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification.*

Description

This service removes the handler associated with *DispatchHandle* so that it will no longer be called when the standby button is pressed or released.

Status Codes Returned

EFI_SUCCESS	The service has been successfully removed.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.9 MM IO Trap Dispatch Protocol

EFI_MM_IO_TRAP_DISPATCH_PROTOCOL

Summary

This protocol provides a parent dispatch service for IO trap MMI sources.

GUID

```
#define EFI_MM_IO_TRAP_DISPATCH_PROTOCOL_GUID \
  { 0x58dc368d, 0x7bfa, 0x4e77, \
  0xab, 0xbc, 0xe, 0x29, 0x41, 0x8d, 0xf9, 0x30 }
```

Protocol Interface Structure

```
struct _EFI_MM_IO_TRAP_DISPATCH_PROTOCOL {
  EFI_MM_IO_TRAP_DISPATCH_REGISTER              Register;
  EFI_MM_IO_TRAP_DISPATCH_UNREGISTER            UnRegister;
} EFI_MM_IO_TRAP_DISPATCH_PROTOCOL;
```

Parameters

Register

Installs a child service to be dispatched when the requested IO trap MMI occurs. See the **Register()** function description.

UnRegister

Removes a previously registered child service. See the *Register()* and **UnRegister()** function descriptions.

Description

This protocol provides the ability to install child handlers for IO trap MMI. These handlers will be invoked to respond to specific IO trap MMI. IO trap MMI would typically be generated on reads or writes to specific processor IO space addresses or ranges. This protocol will typically abstract a limited hardware resource, so callers should handle errors gracefully.

EFI_MM_IO_TRAP_DISPATCH_PROTOCOL.Register ()

Summary

Register an IO trap MMI child handler for a specified MMI.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_MM_IO_TRAP_DISPATCH_REGISTER) (
  IN     CONST EFI_MM_IO_TRAP_DISPATCH_PROTOCOL      *This,
  IN     EFI_MM_HANDLER_ENTRY_POINT              DispatchFunction,
  IN OUT EFI_MM_IO_TRAP_REGISTER_CONTEXT         *RegisterContext,
  OUT EFI_HANDLE                                 *DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_IO_TRAP_DISPATCH_PROTOCOL** instance.

DispatchFunction

Function to register for handler when I/O trap location is accessed. Type **EFI_MM_HANDLER_ENTRY_POINT**is defined in "Related Definitions" in **MmiHandlerRegister()** in the MMST.

RegisterContext

Pointer to the dispatch function's context. The caller fills this context in before calling the register function to indicate to the register function the IO trap MMI source for which the dispatch function should be invoked.

DispatchHandle

Handle of the dispatch function, for when interfacing with the parent MM Driver. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI Specification*.

Description

This service registers a function (*DispatchFunction*) which will be called when an MMI is generated because of an access to an I/O port specified by *RegisterContext*. On return, *DispatchHandle* contains a unique handle which may be used later to unregister the function using **UnRegister()**. If the base of the I/O range specified is zero, then an I/O range with the specified length and characteristics will be allocated and the Address field in *RegisterContext* updated. If no range could be allocated, then **EFI_OUT_OF_RESOURCES** will be returned.

The service will not perform GCD allocation if the base address is non-zero or **EFI_MM_READY_TO_LOCK** has been installed. In this case, the caller is responsible for the existence and allocation of the specific IO range.

An error may be returned if some or all of the requested resources conflict with an existing IO trap child handler.

It is not required that implementations will allow multiple children for a single IO trap MMI source. Some implementations may support multiple children.

The *DispatchFunction* will be called with *Context* updated to contain information concerning the I/O action that actually happened and is passed in *RegisterContext,* with *CommBuffer* pointing to the data actually written and *CommBufferSize* pointing to the size of the data in *CommBuffer*.

Related Definitions

```
//
// IO Trap valid types
//
typedef enum {
  WriteTrap,
  ReadTrap,
  ReadWriteTrap,
  IoTrapTypeMaximum
} EFI_MM_IO_TRAP_DISPATCH_TYPE;

//
// IO Trap context structure containing information about the
// IO trap event that should invoke the handler
//
typedef struct {
  UINT16                                  Address;
  UINT16                                  Length;
  EFI_MM_IO_TRAP_DISPATCH_TYPE           Type;
} EFI_MM_IO_TRAP_REGISTER_CONTEXT;

//
// IO Trap context structure containing information about the IO
trap that occurred
//
typedef struct {
  UINT32                                  WriteData;
} EFI_MM_IO_TRAP_CONTEXT;
```

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully registered.
EFI_DEVICE_ERROR	The driver was unable to complete due to hardware error.
EFI_OUT_OF_RESOURCES	Insufficient resources are available to fulfill the IO trap range request.
EFI_INVALID_PARAMETER	*RegisterContext* is invalid. The input value is not within a valid range.

EFI_MM_IO_TRAP_DISPATCH_PROTOCOL.UnRegister ()

Summary

Unregister a child MMI source dispatch function with a parent MM Driver.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_MM_IO_TRAP_DISPATCH_UNREGISTER) (
  IN CONST EFI_MM_IO_TRAP_DISPATCH_PROTOCOL     *This,
  IN EFI_HANDLE                                  DispatchHandle
  );
```

Parameters

This

Pointer to the **EFI_MM_IO_TRAP_DISPATCH_PROTOCOL** instance.

DispatchHandle

Handle of the child service to remove. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

Description

This service removes a previously installed child dispatch handler. This does not guarantee that the system resources will be freed from the GCD.

Related Definitions

None

Status Codes Returned

EFI_SUCCESS	The dispatch function has been successfully unregistered.
EFI_INVALID_PARAMETER	The *DispatchHandle* was not valid.

7.10 HOBs

EFI_PEI_MM_CORE_GUID

Summary

A GUIDed HOB that indicates whether the MM Core has been loaded.

GUID

```
#define EFI_PEI_MM_CORE_GUID \
    {0x8d1b3618, 0x111b, 0x4cba, \
    {0xb7, 0x9a, 0x55, 0xb3, 0x2f, 0x60, 0xf0, 0x29} }
```

HOB Structure

```
typedef struct _EFI_PEI_MM_CORE_HOB {
  EFI_HOB_GENERIC_HEADER Header;
  EFI_GUID               Name;
  UINT32                 Flags;
} EFI_PEI_MM_CORE_HOB;
```

Members

Header

The HOB generic header with **Header.HobType** set to **EFI_HOB_TYPE_GUID_EXTENSION**.

Name

The GUID that specifies this particular HOB structure. Set to **EFI_PEI_MM_CORE_GUID**.

Flags

Bitmask that specifies which MM features have been initialized in SEC.

All other bits must be set to 0.

```
#define EFI_PEI_MM_CORE_LOADED 0x00000001
# MM Core Loaded
```

Description

This HOB is consumed by the MM IPL driver to understand which portions of MM initialization have been completed. For example the DXE MM IPL driver can determine whether MMRAM has been initialized and the MM Core loaded.

8 Interactions with PEI, DXE, and BDS

8.1 Introduction

This chapter describes issues related to image verification and interactions between SM and other PI Architecture phases.

8.2 MM and DXE

8.2.1 Software MMI Communication Interface (Method #1)

During the boot service phase of DXE/UEFI, there will be a messaging mechanism between MM and DXE drivers. This mechanism will allow a gradual state evolution of the SM handlers during the boot phase.

The purpose of the DXE/UEFI communication is to allow interfaces from either runtime or boot services to be proxied into SM. For example, a vendor may choose to implement their UEFI Variable Services in SM. The motivation to do so would include a design in which the SM code performed error logging by writing data to an UEFI variable in flash. The error generation would be asynchronous with respect to the foreground operating system (OS). A problem is that the OS could be writing an UEFI variable when the error condition, such as a Single-Bit Error (SBE) that was generated from main memory, occurred. To avoid two agents—SM and UEFI Runtime—both trying to write to flash at the same time, the runtime implementation of the **SetVariable()** UEFI call would simply be an invocation of the **EFI_MM_COMMUNICATION_PROTOCOL.Communicate()** interface. Then, the SM code would internally serialize the error logging flash write request and the OS **SetVariable()** request.

See the **EFI_MM_COMMUNICATION_PROTOCOL.Communicate()** service for more information on this interface.

8.2.2 Software MMI Communication Interface (Method #2)

This section describes an alternative mechanism that can be used to initiate inter-mode communication. This mechanism can be used in the OS present environment by non-firmware agents. Inter-mode communication can be initiated using special software MMI.

Details regarding the MMI are described in the SM Communication ACPI Table. This table is described in Appendix O of the *UEFI Specification*.

Firmware processes this software MMI in the same manner it processes direct invocation of the **Communicate()** function.

8.3 MM and PEI

8.3.1 Software MMI Communication Interface (Method #1)

During the PI PEI, there will be a messaging mechanism between MM and PEI drivers. This mechanism will allow a gradual state evolution of the MM Handlers during the PI PEI phase.

The purpose of the PEI communication is to allow interfaces from PEI services to be proxied into MM. For example, a vendor may choose to implement the LockBox Services in MM. The motivation to do so would include a design in which the MM code performed secure storage to save data for S3 resume. PEI phase LockBox service would simply be an invocation of the **EFI_PEI_MM_COMMUNICATION_PPI.Communicate()** interface. Then, the MM code would perform LockBox request.

See the **EFI_PEI_MM_COMMUNICATION_PPI.Communicate()** service for more information on this interface.

9 Other Related Notes For Support Of MM Drivers

9.1 File Types

The following new file types are added:

```
#define EFI_FV_FILETYPE_MM  0x0A
#define EFI_FV_FILETYPE_COMBINED_MM_DXE  0x0C
#define EFI_FV_FILETYPE_MM_STANDALONE 0x0E
```

9.1.1 File Type EFI_FV_FILETYPE_MM

The file type **EFI_FV_FILETYPE_MM** denotes a file that contains a PE32+ image that will be loaded into MMRAM in MM Tradition Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_MM_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

9.1.2 File Type EFI_FV_FILETYPE_COMBINED_MM_DXE

The file type **EFI_FV_FILETYPE_COMBINED_MM_DXE** denotes a file that contains a PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into MMRAM in MM Tradition Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_DXE_DEPEX** section. This section is ignored when the file is loaded into MMRAM.
- The file must contain no more than one **EFI_SECTION_MM_DEPEX** section. This section is ignored when the file is dispatched by the DXE Dispatcher.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the DXE driver that will be dispatched is defined by the **LoadImage()** boot service, which is used by the DXE Dispatcher. See the *Platform Initialization Specification, Volume 2* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

9.2 File Type EFI_FV_FILETYPE_MM_STANDALONE

The file type **EFI_FV_FILETYPE_MM_STANDALONE** denotes a file that contains a PE32+ image that will be loaded into MMRAM in MM Standalone Mode.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_MM_DEPEX** section.

There are no restrictions on the encapsulation of the leaf sections. In the event that more than one **EFI_SECTION_PE32** section is present in the file, the selection algorithm for choosing which one represents the MM driver that will be dispatched is defined by MM Foundation Dispatcher. See the *Platform Initialization Specification, Volume 4* for details. The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

9.3 File Section Types

The following new section type must be added:

```
#define EFI_SECTION_MM_DEPEX 0x1c
```

9.3.1 File Section Type EFI_SECTION_MM_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for an MM Driver.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_MM_DEPEX_SECTION;
```

Description

The *MM dependency expression section* is a leaf section that contains a dependency expression that is used to determine the dispatch order for MM Drivers. Before the MMRAM invocation of the MM Driver's entry point, this dependency expression must evaluate to TRUE. See the *Platform Initialization Specification, Volume 2* for details regarding the format of the dependency expression.

The dependency expression may refer to protocols installed in either the UEFI or the MM protocol database.

10 MCA/INIT/PMI Protocol

This document defines the basic plumbing required to run the MCA, PMI & INIT in a generic framework. They have been group together since MCA and INIT follows a very similar flow and all three have access to the min-state as defined by PAL.

It makes an attempt to bind the platform knowledge by the way of generic abstraction to the SAL MCA, PMI & INIT code. We have tried to create a private & public data structures for each CPU. For example, any CPU knowledge that should remain within the context of that CPU should be private. Any CPU knowledge that may be accessed by another CPU should be a Global Structure that can be accessed by any CPU for that domain. There are some flags that may be required globally (Sal Proc, Runtime Services, PMI, INIT, MCA) are made accessible through a protocol pointer that is described in section 5.

10.1 Machine Check and INIT

This section describes how Machine Check Abort Interrupt and INIT are handled in a UEFI 2.0 compliant system.

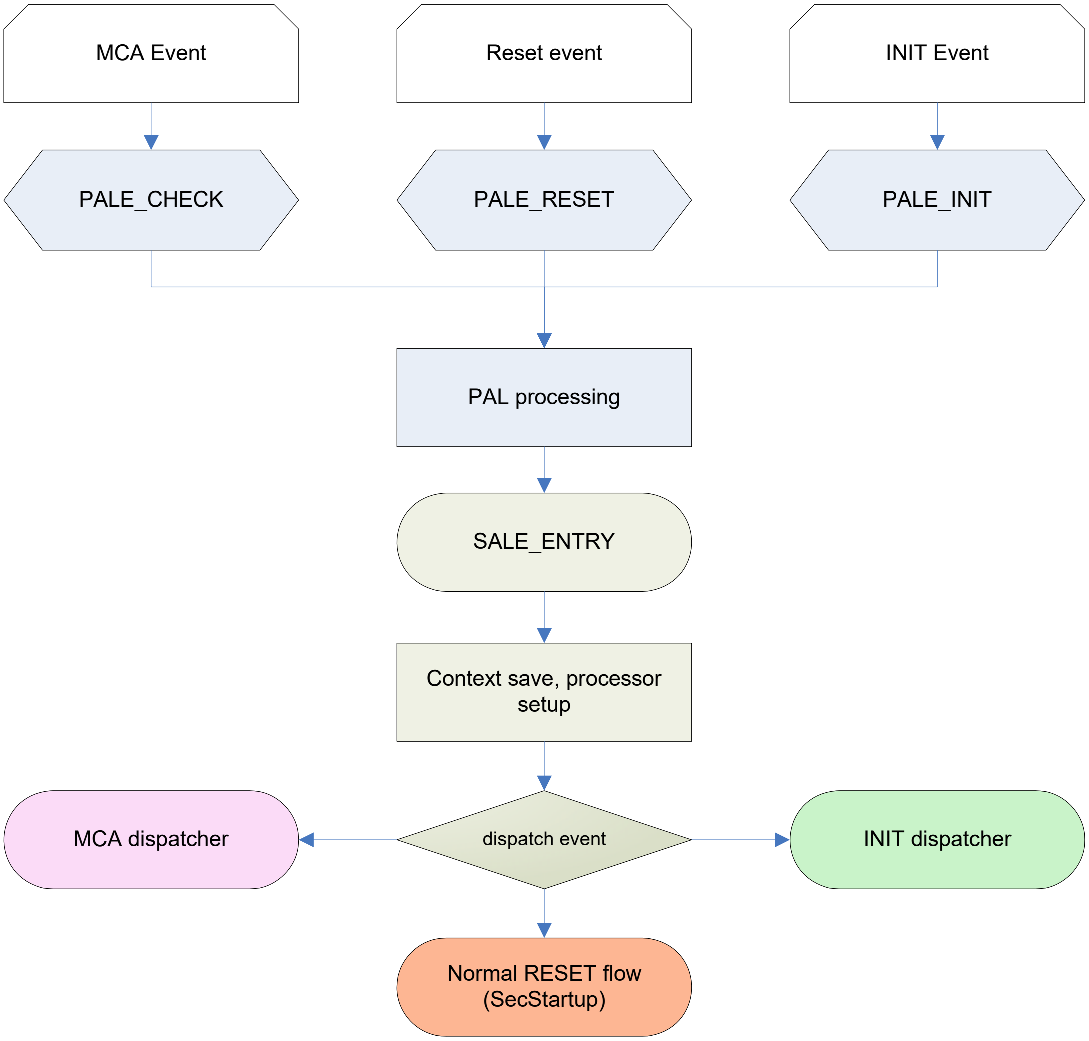


Figure 5. Early Reset, MCA and INIT flow

As shown in Figure 5 resets, MCA and INIT follow a near identical early flow. For all three events, PAL first processes the event, save some states if needed in the min-state before jumping to SAL through the common SALE_ENTRY entry point. SAL performs some early processor initialization, save some extra states to set up an environment in which the event can be handled and then branch to the appropriate event dispatcher (normal reset flow, MCA, INIT).

MCA/INIT handling per say consists of a generic dispatcher and one or more platform specific handlers. The dispatcher is responsible for handling tasks specified in SAL specification, such as performing rendezvous, before calling the event handlers in a fixed order. The handlers are responsible for error logging, error correction and any other platform specific task required to properly handle a MCA or INIT event.

10.2 MCA Handling

The machine check (MCA) code path in a typical machine based on IPF architecture is shown in the diagram below (see Figure 6).

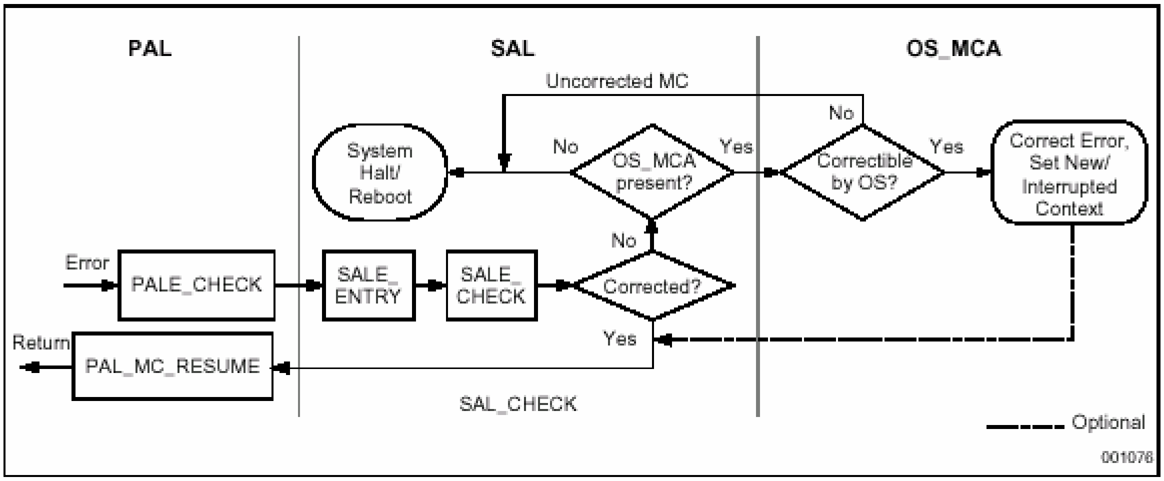


Figure 6. Basic MCA processing flow

MCA processing starts in PAL, running in physical mode. Control is then pass to SAL through the SALE_ENTRY entry point which in turn, after processing the MCA, pass control to the OS MCA handler.

In the PI architecture, OEMs have the choice to process MCA events in either entirely in ROM code, entirely in the RAM code or partly in ROM and partly in RAM. The early part of the MCA flow follow the SEC->PEI boot flow, with SALE_ENTRY residing in SEC while the MCA dispatcher is a PEIM dispatcher (see Figure 7). From that point on the rest of the code can reside in ROM or RAM.

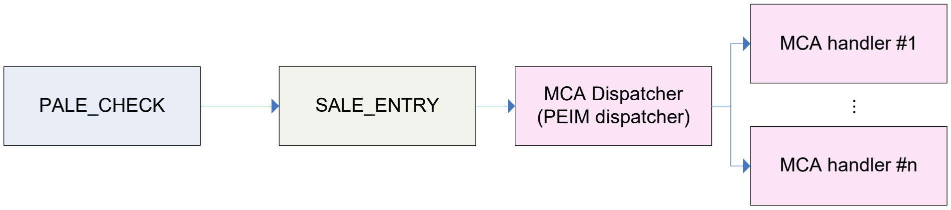


Figure 7. PI MCA processing flow

When PAL hands off control to SALE_ENTRY, it will supply unique hand off state in the processor registers as well as the minimum state saved buffer area pointer called “min-state pointer”. The min-state pointer is the only context available to SALE_ENTRY. This buffer is a unique per processor save area registered to each processor during normal OS boot path.

A sample implementation is described below to clarify some of the finer points of MCA/INIT/PMI. Actual implementations may vary.

Usually, we can anchor some extra data (the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure) required by the PEIM dispatcher and the MCA and INIT dispatchers to the min-state (see Figure 8).

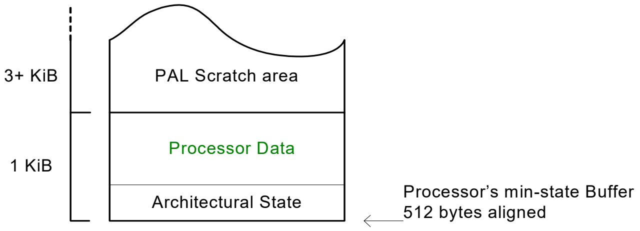


Figure 8. PI architectural data in the min-state

The software component (a PEIM or a DXE module) that includes the MCA and INIT dispatchers is responsible for registering the min-state on all processors and initializing **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structures. Only then can MCA be properly handled by the platform. To guarantee proper MCA and INIT handling, at least one handler is required to be registered with the MCA dispatcher. OEM might decide to use a monolithic handler or use multiple handlers.

The register state at the MCA dispatcher entry point is the same as the PALE_CHECK exit state with the following exceptions -

- GR1 contains GP for the *McaDispatcherProc*.
- PAL saves b0 in the min-state and can be used as scratch. b0 contains the address of the *McaDispatcherProc*.
- PAL saves static registers to the min-state. All static registers in both banks except GR16-GR20 in bank 0 can be used as scratch registers. SALE_ENTRY may freely modify these registers.

The MCA dispatcher is responsible for setting up a stack and backing store based on the values in the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure. The OS stack and backing store cannot be used since they might point to virtual addresses. The MCA dispatcher is also responsible for saving any registers not saved in the min-state that may be used by the MCA handling code in the PI per processor data. Since we want to use high-level language such as C, floating point registers f2 to f31 as well as branch registers b6 and b7 must be saved. Code used during MCA handling must be compiled with /f32 option to prevent the use of registers f33-f127. Otherwise, such code is responsible for saving and restoring floating point registers f33-f127 as well as any other registers not saved in the min-state or the PI per processor data.

Note that nested MCA recovery is not supported by the Itanium architecture as PAL uses the same min-state for every MCA and INIT event. As a result, the same context within the min-state is used by PI every time the MCA dispatcher is entered.

All the MCA handles are presented in a form of an Ordered List. The head of the Ordered List is a member of the Private Data Structure. In order to reach the MCA handle Ordered List the following steps are used:

1. PerCpuInfoPointer = MinStatePointer (From SALE_CHECK) + 4K
2. ThisCpuMcaPrivateData = PerCpuInfoPointer->Private
3. McaHandleListHead = ThisCpuMcaPrivateData->McaList

Or **((EFI_MCA_SAVE_DATA*)(((UINT8*) MinStatePointer) + 4*1024))->Private-> McaList**

On reaching the Ordered List from the private data we can obtain Plabel & MCA Handle Context. Using that we can execute each handle as they appear in the ordered list.

Once the last handler has completed execution, the MCA dispatcher is responsible for deciding whether to resume execution, halt the platform or reset the platform. This is based on the OS request and platform policies. Resuming to the interrupted context is accomplished by calling **PAL_MC_RESUME**.

As shown in Figure 6, the MCA handling flow requires access to certain shared hardware and software resources to support things such as error logging, error handling/correction and processor rendezvous. In addition, since MCAs are asynchronous, they might happen while other parts of the system are using those shared resources or while accessing those resources (for example during the execution of a SAL_PROC like PCI config write). We thus need a mechanism to allow shared access to two isolated model which are not aware of each others.

This is handled through the use of common code (libraries) and semaphores. The SAL PROCs and the MCAA/INIT code use the same libraries to implement any functionality shared between them such as platform reset, stall, PCI read/write. Semaphores are used to gate access to critical portion of the code and prevent multiple accesses to the same HW resource at the same time. To prevent deadlocks and guarantee proper OS handling of an MCA it might be necessary for the MCA/INIT handler to break semaphore or gets priority access to protected resources.

In addition to the previously mentioned semaphores used for gating access to HW resource, the multithreaded/MP MCA model may require an MCA specific semaphore to support things like monarch processor selection and log access. This semaphore should be visible from all processors. In addition some global are required for MCA processing to indicate a processor status (entering MCA, in MCA rendezvous, ready to enter OS MCA) with regards to the current MCA. This flags need to have a global scope since the MCA monarch may need to access them to make sure all processor are where they are supposed to be.

10.3 INIT Handling

Most of what have been defined for the MCA handling and dispatcher applies to the INIT code path. The early part of the INIT code path, up to the INIT dispatcher is identical to the MCA code path while some of the INIT handler code, like logging, can be shared with the MCA handler.

The INIT code path in a typical machine based on IPF architecture is shown in the diagram below.

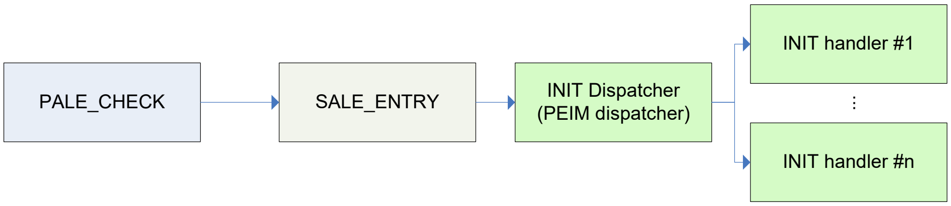


Figure 9. PI INIT processing flow

Like MCA, INIT processing starts in the PAL code in physical mode and then flows into PI code (OEM firmware code) through SALE_ENTRY. The INIT dispatcher is responsible for setting up a stack and backing store, saving the floating point registers before calling any code that may be written in higher level languages. At that point the dispatcher is ready to call the INIT handlers. As with MCA only one handler is required to exist but OEMs are free to implement a monolithic handler or use multiple handlers. Once the last handler has been executed, the dispatcher will resume to the interrupted context or reset the platform based on the OS request.

The MCA handler limitations regarding access to shared HW and SW resources applies to the INIT handler, as such library code and common semaphores should be used.

INIT events are always local to each processor. As a result we do not need INIT specific flags or semaphore in the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structures.

10.4 PMI

This section describes how PMI, platform management interrupts, are handled in EFI 2.0 compliant system. PMIs provide an operating system-independent interrupt mechanism to support OEM and vendor specific hardware event.

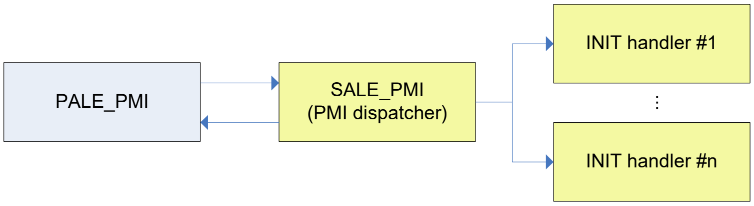


Figure 10. PMI handling flow

As shown in Figure 10, PMI handling is pretty similar to MCA and INIT handling in such that it consists of a generic dispatcher and one of more platform specific handlers. The dispatchers is the SAL PMI entry point (SALE_PMI) and is responsible for saving state and setting up the environment for the handler to execute. Contrary to MCA and INIT, PAL does not save any context in the min-state and it is the responsibility of the PMI dispatcher to save state. Since the min-state is available during PMI handling (PAL provides its address to the SAL PMI handler) the

MCA_INIT_PMI_PER_PROCESSOR_DATA data structure present in the min-state can be used. However an MCA/INIT event occurring while PMI is being would preclude the system from resuming from the PMI event. To alleviate this, a platform may decide to implement a separate copy of the **MCA_INIT_PMI_PER_PROCESSOR_DATA** data structure out side of the min-state, to be used for PMI state saving.

Once the state is saved, the platform specific PMI handlers are found using the order handler list provided in the private data structure. The mechanism used is the same one used in MCA and INIT handling.

10.5 Event Handlers

The events handlers are called by the various dispatchers.

10.5.1 MCA Handlers

MCA Handler

```
typedef
EFI_STATUS
SAL_RUNTIMESERVICE
(EFIAPI *EFI_SAL_MCA_HANDLER) (
  IN  VOID                          *ModuleGlobal,
  IN  UINT64                        ProcessorStateParameters,
  IN  EFI_PHYSICAL_ADDRESS          MinstateBase,
  IN  UINT64                        RendezvouseStateInformation,
  IN  UINT64                        CpuIndex,
  IN  SAL_MCA_COUNT_STRUCTURE       *McaCountStructure,
  IN OUT BOOLEAN                    *CorrectedMachineCheck
  );
```

Parameters

ModuleGlobal

The context of MCA Handler.

ProcessorStateParameters

The processor state parameters (PSP),

MinstateBase

Base address of the min-state.

RendezvousStateInformation

Rendezvous state information to be passed to the OS on OS MCA entry. Refer to the *Sal Specification 3.0 , section 4.8* for more information.

CpuIndex

Index of the logical processor

McaCountStructure

Pointer to the MCA records structure

CorrectedMachineCheck

This flag is set to **TRUE** is the MCA has been corrected by the handler or by a previous handler.

```
#pragma pack(1)
//
// MCA Records Structure
//
typedef struct {
  UINT64  First : 1;
  UINT64  Last : 1;
  UINT64  EntryCount : 16;
  UINT64  DispatchedCount : 16;
  UINT64  Reserved : 30;
} SAL_MCA_COUNT_STRUCTURE;

#pragma pack()
```

10.5.2 INIT Handlers

INIT Handler

```
typedef
EFI_STATUS
SAL_RUNTIMESERVICE
(EFIAPI *EFI_SAL_INIT_HANDLER) (
  IN  VOID                      *ModuleGlobal,
  IN  UINT64                    ProcessorStateParameters,
  IN  EFI_PHYSICAL_ADDRESS      MinstateBase,
  IN  BOOLEAN                   McaInProgress,
  IN  UINT64                    CpuIndex,
  IN  SAL_MCA_COUNT_STRUCTURE   *McaCountStructure,
  OUT BOOLEAN                 *DumpSwitchPressed
  );
```

Parameters

ModuleGlobal

The context of MCA Handler.

ProcessorStateParameters

The processor state parameters (PSP),

MinstateBase

Base address of the min-state.

McaInProgress

This flag indicates if an MCA is in progress.

CpuIndex

Index of the logical processor

McaCountStructure

Pointer to the MCA records structure

DumpSwitchPressed

This flag indicates the crash dump switch has been pressed.

10.5.3 PMI Handlers

PMI Handler

```
typedef
EFI_STATUS
(EFIAPI *SAL_PMI_HANDLER) (
  IN  VOID                          *ModuleGlobal,
  IN  UINT64                        CpuIndex,
  IN  UINT64                        PmiVector
);
```

Description

ModuleGlobal

The context of MCA Handler.

CpuIndex

Index of the logical processor

PmiVector

The PMI vector number as received from the PALE_PMI exit state (GR24).

10.6 MCA PMI INIT Protocol

Summary

This protocol is used to register MCA, INIT and PMI handlers with their respective dispatcher.

GUID

```
#define EFI_SAL_MCA_INIT_PMI_PROTOCOL_GUID \
     {
0xb60dc6e8,0x3b6f,0x11d5,0xaf,0x9,0x0,0xa0,0xc9,0x44,0xa0,0x5b }
```

Protocol Interface Structure

```
typedef struct {
  EFI_SAL_REGISTER_MCA_HANDLER  RegisterMcaHandler;
  EFI_SAL_REGISTER_INIT_HANDLER RegisterInitHandler;
  EFI_SAL_REGISTER_PMI_HANDLER  RegisterPmiHandler;
  BOOLEAN                       McaInProgress;
  BOOLEAN                       InitInProgress;
  BOOLEAN                       PmiInProgress;
} EFI_SAL_MCA_INIT_PMI_PROTOCOL;
```

Parameters

RegisterMcaHandler

Function to register a MCA handler.

RegisterInitHandler

Function to register an INIT handler.

RegisterPmiHandler

Function to register a PMI hander.

McaInProgress

Whether MCA handler is in progress

InitInProgress

Whether Init handler is in progress

PmiInProgress

Whether Pmi handler is in progress

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterMcaHandler ()

Summary

Register a MCA handler with the MCA dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_MCA_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL   *This,
  IN  EFI_SAL_MCA_HANDLER                     McaHandler,
  IN  VOID                                    ModuleGlobal
  IN  BOOLEAN                                 MakeFirst,
  IN  BOOLEAN                                 MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

McaHandler

The MCA handler to register as defined in section 10.5.1.

ModuleGlobal

The context of the MCA Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	MCA Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register an MCA handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterInitHandler ()

Summary

Register an INIT handler with the INIT dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_INIT_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL   *This,
  IN  EFI_SAL_INIT_HANDLER                    InitHandler,
  IN  VOID                                    ModuleGlobal
  IN  BOOLEAN                                 MakeFirst,
  IN  BOOLEAN                                 MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

InitHandlerT

The INIT handler to register as defined in section 10.5.2

ModuleGlobal

The context of the INIT Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	INIT Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register an INIT handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

EFI_SAL_MCA_INIT_PMI_PROTOCOL. RegisterPmiHandler ()

Summary

Register a PMI handler with the PMI dispatcher.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SAL_REGISTER_PMI_HANDLER) (
  IN  struct _EFI_SAL_MCA_INIT_PMI_PROTOCOL    *This,
  IN  EFI_SAL_PMI_HANDLER                      PmiHandler,
  IN  VOID                                     ModuleGlobal
  IN  BOOLEAN                                  MakeFirst,
  IN  BOOLEAN                                  MakeLast
  );
```

Parameters

This

The **EFI_SAL_MCA_INIT_PMI_PROTOCOL** instance.

PmiHandler

The PMI handler to register as defined in section 10.5.3.

ModuleGlobal

The context of the PMI Handler.

MakeFirst

This flag specifies the handler should be made first in the list.

MakeLast

This flag specifies the handler should be made last in the list.

Status Codes Returned

EFI_SUCCESS	INIT Handle was registered
EFI_OUT_OF_RESOURCES	No more resources to register a PMI handler
EFI_INVALID_PARAMETER	Invalid parameters were passed.

11 Extended SAL Services

This document describes the Extended SAL support for the EDK II. The Extended SAL uses a calling convention that is very similar to the SAL calling convention. This includes the ability to call Extended SAL Procedures in physical mode prior to **`SetVirtualAddressMap()`**, and the ability to call Extended SAL Procedures in physical mode or virtual mode after **`SetVirtualAddressMap()`**.

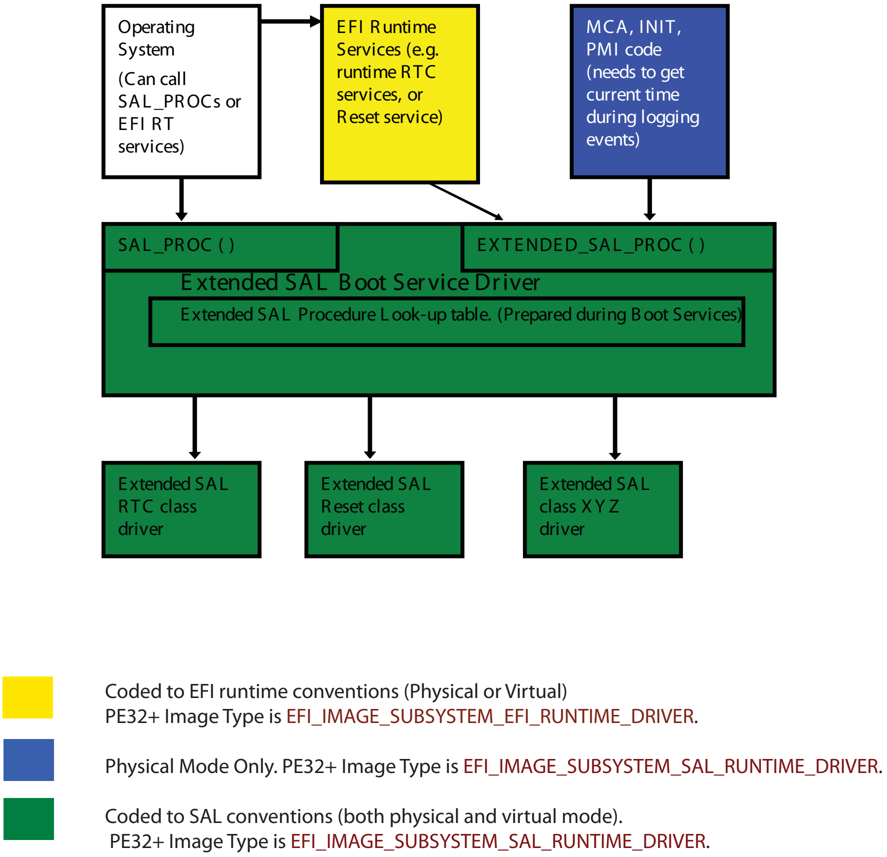
11.1 SAL Overview

The Extended SAL can be used to implement the following services:

- SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification.*
- EFI Runtime Services required by the *UEFI 2.0 Specification*, that may also be required by SAL Procedures, other Extended SAL Procedures, or MCA, INIT, and PMI flows.
- Services required to abstract hardware accesses from SAL Procedures and Extended SAL Procedures. This includes I/O port accesses, MMIO accesses, PCI Configuration Cycles, and access to non-volatile storage for logging purposes.
- Services required during the MCA, INIT, and PMI flows.

Note: *Arguments to SAL procedures are formatted the same as arguments and paramenters in this document. Example "`address` parameter to . . ."*

The Extended SAL support includes a DXE Protocol that supports the publishing of the SAL System Table along with services to register and call Extended SAL Procedures. It also includes a number of standard Extended SAL Service Classes that are required to implement EFI Runtime Services, the minimum set of required SAL Procedures, services to abstract hardware accesses, and services to support the MSA, INIT, and PMI flows. Platform developer may define addition Extended SAL Service Classes to provide platform specific functionality that requires the Extended SAL calling conventions.The SAL calling convention requires operation in both physical and virtual mode. Standard EFI runtime services work in either physical mode or virtual mode at a time. Therefore, the EFI code can call the SAL code, but not vice versa. To reduce code duplication resulting out of multiple operating modes, additional procedures called Extended SAL Procedures are implemented. Architected SAL procedures are a subset of the Extended SAL procedures. The individual Extended SAL procedures can be called through the entry point **`ExtendedSalProc()`** in the **`EXTENDED_SAL_BOOT_SERVICE_PROTOCOL`**. The cost of writing dual mode code is that one must strictly follow the SAL runtime coding rules. Experience on prior IPF platform shows us that the benefits outweigh the cost.


Operating System
(Can call SAL_PROCs or EFI RT services)
EFI Runtime Services (e.g. runtime RTC services, or Reset service)
MCA, INIT, PMI code (needs to get current time during logging events)
SAL_PROC ( )
EXTENDED_SAL_PROC ( )
Extended SAL Boot Service Driver
Extended SAL Procedure Look-up table. (Prepared during Boot Services)
Extended SAL RTC class driver
Extended SAL Reset class driver
Extended SAL class XYZ driver

🟨 Coded to EFI runtime conventions (Physical or Virtual)
PE32+ Image Type is EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER.

🟦 Physical Mode Only. PE32+ Image Type is EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER.

🟩 Coded to SAL conventions (both physical and virtual mode).
PE32+ Image Type is EFI_IMAGE_SUBSYSTEM_SAL_RUNTIME_DRIVER.

Figure 11. SAL Calling Diagram

Note: *In the figure above, arrows indicate the direction of calling. For example, OS code may call EFI runtime services or* **`SAL_PROCs`***. Extended SAL functions are divided in several classes based on their functionality, with no defined hierarchy. It is legal for an EFI Boot Service Code to call* **`ExtendedSalProc()`***. It is also legal for an Extended SAL procedure to call another Extended SAL Procedure via* **`ExtendedSalProc()`***. These details are not shown in the figure in order to maintain clarity.*

A driver with a module type of **`DXE_SAL_DRIVER`** is required to produce the **`EXTENDED_SAL_BOOT_SERVICE_PROTOCOL`**. This driver contains the entry point of the Extended SAL Procedures and dispatches previously registered procedures. It also provides services to register Extended SAL Procedures and functions to help construct the SAL System Table.

Drivers with a module type of **`DXE_SAL_DRIVER`** are required to produce the various Extended SAL Service Classes. It is expected that a single driver will supply all the Extended SAL Procedures that belong to a single Extended SAL Service Class. As each Extended SAL Service Class is registered, the GUID associated with that class is also installed into the EFI Handle Database. This allows other DXE drivers to use the Extended SAL Service Class GUIDs in their dependency expressions, so they only execute once their dependent Extended SAL Service Classes are available.

Drivers register the set of Extended SAL Procedures they produce with the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL**. Once this registration step is complete, the Extended SAL Procedure are available for use by other drivers.

11.2 Extended SAL Boot Service Protocol

This protocol supports the creation of the SAL System Table, and provides services to register and call Extended SAL Procedures. The driver that produces this protocol is required to allocate and initialize the SAL System Table. The SAL System Table must also be registered in the list of EFI System Configuration tables. The driver that produces this protocol must be of type **DXE_SAL_DRIVER**. This is required because the entry point to the **ExtendedSalProc()** function is always available, even after the OS assumes control of the platform at **ExitBootServices()**.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL

Summary

This section provides a detailed description of the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL**.

GUID

```
#define EXTENDED_SAL_BOOT_SERVICE_PROTOCOL_GUID \
  {0xde0ee9a4,0x3c7a,0x44f2, \
  {0xb7,0x8b,0xe3,0xcc,0xd6,0x9c,0x3a,0xf7}}
```

Protocol Interface Structure

```
typedef struct _EXTENDED_SAL_BOOT_SERVICE_PROTOCOL {
  EXTENDED_SAL_ADD_SST_INFO            AddSalSystemTableInfo;
  EXTENDED_SAL_ADD_SST_ENTRY           AddSalSystemTableEntry;
  EXTENDED_SAL_REGISTER_INTERNAL_PROC  RegisterExtendedSalProc;
  EXTENDED_SAL_PROC                    ExtendedSalProc;
} EXTENDED_SAL_BOOT_SERVICE_PROTOCOL;
```

Parameters

AddSalSystemTableInfo

Adds platform specific information to the to the header of the SAL System Table. Only available prior to **ExitBootServices()**.

AddSalSystemTableEntry

Add an entry into the SAL System Table. Only available prior to **ExitBootServices()**.

RegisterExtendedSalProc

Registers an Extended SAL Procedure. Extended SAL Procedures are named by a (GUID, FunctionID) pair. Extended SAL Procedures are divided into classes based on the functionality they provide. Extended SAL Procedures are callable only in physical mode prior to **SetVirtualAddressMap()**, and are callable in both virtual and physical mode after **SetVirtualAddressMap()**. Only available prior to **ExitBootServices()**.

ExtendedSalProc

Entry point for all extended SAL procedures. This entry point is always available.

Description

The **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** provides a mechanisms for platform specific drivers to update the SAL System Table and register Extended SAL Procedures that are callable in physical or virtual mode using the SAL calling convention. The services exported by the SAL System Table are typically implemented as Extended SAL Procedures. Services required by MCA, INIT, and PMI flows that are also required in the implementation of EFI Runtime Services are also typically implemented as Extended SAL Procedures. Extended SAL Procedures are named by a (GUID, FunctionID) pair. A standard set of these (GUID, FunctionID) pairs are defined in this specification. Platforms that require additional functionality from their Extended SAL Procedures may define additional (GUID, FunctionID) pairs.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableInfo()

Summary

Adds platform specific information to the to the header of the SAL System Table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_ADD_SST_INFO) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL  *This,
  IN UINT16                              SalAVersion,
  IN UINT16                              SalBVersion,
  IN CHAR8                               *OemId,
  IN CHAR8                               *ProductId
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

SalAVersion

Version of recovery SAL PEIM(s) in BCD format. Higher byte contains the major revision and the lower byte contains the minor revision.

SalBVersion

Version of DXE SAL Driver in BCD format. Higher byte contains the major revision and the lower byte contains the minor revision.

OemId

A pointer to a Null-terminated ASCII string that contains OEM unique string. The string cannot be longer than 32 bytes in total length.

ProductId

A pointer to a Null-terminated ASCII string that uniquely identifies a family of compatible products. The string cannot be longer than 32 bytes in total length.

Description

This function updates the platform specific information in the SAL System Table header. The **SAL_A_VERSION** field of the SAL System Table is set to the value specified by *SalAVersion*. The **SAL_B_VERSION** field of the SAL System Table is set to the value specified by *SalBVersion*. The **OEM_ID** field of the SAL System Table is filled in with the contents of the Null-terminated ASCII string specified by *OemId*. If *OemId* is **NULL** or the length of *OemId* is greater than 32 characters, then **EFI_INVALID_PARAMETER** is returned. The **PRODUCT_ID** field of the SAL System Table is filled in with the contents of the Null-terminated ASCII string specified by *ProductId*. If *ProductId* is **NULL** or the length of *ProductId* is greater than 32 characters, then **EFI_INVALID_PARAMETER** is returned. This function is also responsible for re-computing the **CHECKSUM** field of the SAL System Table after the **SAL_A_REVISION**, **SAL_B_REVISION**, **OEM_ID**, and **PRODUCT_ID** fields have been filled in. Once the **CHEKSUM** field has been updated, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SAL System Table header was updated successfully.
EFI_INVALID_PARAMETER	OemId is **NULL**.
EFI_INVALID_PARAMETER	ProductId is **NULL**.
EFI_INVALID_PARAMETER	The length of *OemId* is greater than 32 characters.
EFI_INVALID_PARAMETER	The length of *ProductId* is greater than 32 characters.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddSalSystemTableEntry()

Summary

Adds an entry to the SAL System Table.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_ADD_SST_ENTRY) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL   *This,
  IN UINT8                                *TableEntry,
  IN UINTN                                EntrySize
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

TableEntry

Pointer to a buffer containing a SAL System Table entry that is *EntrySize* bytes in length. The first byte of the *TableEntry* describes the type of entry. See the *Intel Itanium Processor Family System Abstraction Layer Specification* for more details.

EntrySize

The size, in bytes, of *TableEntry*.

Description

This function adds the SAL System Table Entry specified by *TableEntry* and *EntrySize* to the SAL System Table. If *TableEntry* is **NULL**, then **EFI_INVALID_PARAMETER** is returned. If the entry type specified in *TableEntry* is invalid, then **EFI_INVALID_PARAMETER** is returned. If the length of the *TableEntry* is not valid for the entry type specified in *TableEntry*, then **EFI_INVALID_PARAMETER** is returned. Otherwise, *TableEntry* is added to the SAL System Table. This function is also responsible for re-computing the **CHECKSUM** field of the SAL System Table. Once the **CHEKSUM** field has been updated, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The SAL System Table was updated successfully
EFI_INVALID_PARAMETER	*TableEntry* is NULL.
EFI_INVALID_PARAMETER	*TableEntry* specifies an invalid entry type.
EFI_INVALID_PARAMETER	*EntrySize* is not valid for this type of entry.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.AddExtendedSalProc( )

Summary

Registers an Extended SAL Procedure.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EXTENDED_SAL_REGISTER_INTERNAL_PROC) (
  IN EXTENDED_SAL_BOOT_SERVICE_PROTOCOL       *This,
  IN UINT64                                   ClassGuidLo,
  IN UINT64                                   ClassGuidHi,
  IN UINT64                                   FunctionId,
  IN SAL_INTERNAL_EXTENDED_SAL_PROC           InternalSalProc,
  IN VOID \
    *PhysicalModuleGlobal OPTIONAL
  );
```

Parameters

This

A pointer to the **EXTENDED_SAL_BOOT_SERVICE_PROTOCOL** instance.

ClassGuidLo

The lower 64-bits of the class GUID for the Extended SAL Procedure being added. Each class GUID contains one or more functions specified by a Function ID.

ClassGuidHi

The upper 64-bits of the class GUID for the Extended SAL Procedure being added. Each class GUID contains one or more functions specified by a Function ID.

FunctionId

The Function ID for the Extended SAL Procedure that is being added. This Function ID is a member of the Extended SAL Procedure class specified by *ClassGuidLo* and *ClassGuidHi*.

InternalSalProc

A pointer to the Extended SAL Procedure being added. The Extended SAL Procedure is named by the GUID and Function ID specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*.

PhysicalModuleGlobal

Pointer to a module global structure. This is a physical mode pointer. This pointer is passed to the Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, *FunctionId*, and *InternalSalProc*. If the system is in physical mode, then this pointer is passed unmodified to *InternalSalProc*. If the system is in virtual mode, then the virtual address associated with this pointer is passed to *InternalSalProc*. This parameter is optional and may be **NULL**. If it is **NULL**, then **NULL** is always passed to *InternalSalProc*.

Related Definitions

```
typedef
SAL_RETURN_REGS
(EFIAPI *SAL_INTERNAL_EXTENDED_SAL_PROC) (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

FunctionId

The Function ID associated with this Extended SAL Procedure.

Arg2

Second argument to the Extended SAL procedure.

Arg3

Third argument to the Extended SAL procedure.

Arg4

Fourth argument to the Extended SAL procedure.

Arg5

Fifth argument to the Extended SAL procedure.

Arg6

Sixth argument to the Extended SAL procedure.

Arg7

Seventh argument to the Extended SAL procedure.

Arg8

Eighth argument to the Extended SAL procedure.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure.

Description

The Extended SAL Procedure *specified by InternalSalProc and named by ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is added to the set of available Extended SAL Procedures. Each Extended SAL Procedure is allowed one module global to record any state information required during the execution of the Extended SAL Procedure. This module global is specified by *PhysicalModuleGlobal*.

If there are not enough resource available to add the Extended SAL Procedure, then **EFI_OUT_OF_RESOURCES** is returned.

If the Extended SAL Procedure specified by *InternalSalProc* and named by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* was not previously registered, then the Extended SAL Procedure along with its module global specified by *PhysicalModuleGlobal* is added to the set of Extended SAL Procedures, and **EFI_SUCCESS** is returned.

If the Extended SAL Procedure specified by *InternalSalProc* and named by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* was previously registered, then the module global is replaced with *PhysicalModuleGlobal*, and *EFI_SUCCESS* is returned.

Status Codes Returned

EFI_SUCCESS	The Extended SAL Procedure was added.
EFI_OUT_OF_RESOURCES	There are not enough resources available to add the Extended SAL Procedure.

EXTENDED_SAL_BOOT_SERVICE_PROTOCOL.ExtendedSalProc()

Summary

Calls a previously registered Extended SAL Procedure.

Prototype

```
typedef
SAL_RETURN_REGS
(EFIAPI *EXTENDED_SAL_PROC) (
  IN UINT64    ClassGuidLo,
  IN UINT64    ClassGuidHi,
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8
  );
```

Parameters

ClassGuidLo

The lower 64-bits of the class GUID for the Extended SAL Procedure that is being called.

ClassGuidHi

The upper 64-bits of the class GUID for the Extended SAL Procedure that is being called.

FunctionId

Function ID for the Extended SAL Procedure being called.

Arg2

Second argument to the Extended SAL procedure.

Arg3

Third argument to the Extended SAL procedure.

Arg4

Fourth argument to the Extended SAL procedure.

Arg5

Fifth argument to the Extended SAL procedure.

Arg6

Sixth argument to the Extended SAL procedure.

Arg7

Seventh argument to the Extended SAL procedure.

Arg8

Eighth argument to the Extended SAL procedure.

Description

This function calls the Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*. The set of previously registered Extended SAL Procedures is searched for a matching *ClassGuidLo*, *ClassGuidHi*, and *FunctionId*. If a match is not found, then **EFI_SAL_NOT_IMPLEMENTED** is returned. The module global associated with *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is retrieved. If that module global is not **NULL** and the system is in virtual mode, and the virtual address of the module global is not available, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the Extended SAL Procedure associated with *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* is called. The arguments specified by *FunctionId*, *Arg2*, *Arg3*, *Arg4*, *Arg5*, *Arg6*, *Arg7*, and *Arg8* are passed into the Extended SAL Procedure along with the *VirtrualMode* flag and *ModuleGlobal* pointer.

If the system is in physical mode, then the *ModuleGlobal* that was originally registered with **AddExtendedSalProc()** is passed into the Extended SAL Procedure. If the system is in virtual mode, then the virtual address associated with *ModuleGlobal* is passed to the Extended SAL Procedure. The EFI Runtime Service **ConvertPointer()** is used to convert the physical address of *ModuleGlobal* to a virtual address. If *ModuleGlobal* was registered as **NULL**, then **NULL** is always passed into the Extended SAL Procedure.

The return status from this Extended SAL Procedure is returned.

Status Codes Returned

EFI_SAL_NOT_IMPLEMENTED	The Extended SAL Procedure specified by *ClassGuidLo*, *ClassGuidHi*, and *FunctionId* has not been registered.
EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	The result returned from the specified Extended SAL Procedure

11.3 Extended SAL Service Classes

This chapter contains the standard set of Extended SAL service classes. These include EFI Runtime Services in the *UEFI 2.0 Specification*, SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification* , services required to abstract access to hardware devices, and services required in the handling of MCA, INIT, and PMI flows. Extended SAL Service Classes behave like PPIs and Protocols. They are named by GUID and contain a set of services for each GUID. This also allows platform developers to add new Extended SAL service classes over time to implement platform specific features that require the Extended SAL capabilities.

The following tables list the Extended SAL Service Classes defined by this specification. The following sections contain detailed descriptions of the functions in each of the classes.

Table 1. Extended SAL Service Classes – EFI Runtime Services

Name	Description
Real Time Clock Services Class	The Extended SAL Real Time Clock Services Class provides functions to access the real time clock.
Reset Services Class	The Extended SAL Reset Services Class provides platform reset services.
Status Code Services Class	The Extended SAL Status Code Services Class provides services to report status code information.
Monotonic Counter Services Class	The Extended SAL Monotonic Counter Services Class provides functions to access the monotonic counter.
Variable Services Class	The Extended SAL Variable Services Class provides functions to access EFI variables.

Table 2. Extended SAL Service Classes – SAL Procedures

Name	Description
Base Services Class	The Extended SAL Base Services Class provides base services that do not have any hardware dependencies including a number of SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification.*
Cache Services Class	The Extended SAL Cache Services Class provides services to initialize and flush the caches.
PAL Services Class	The Extended SAL PAL Services Class provides services to make PAL calls.
PCI Services Class	The Extended SAL PCI Services Class provides services to perform PCI configuration cycles.
MCA Log Services Class	The Extended SAL MCA Log Services Class provides logging services for MCA events.

Table 3. Extended SAL Service Classes – Hardware Abstractions

Name	Description
Base I/O Services Class	The Extended SAL Base I/O Services Class provides the basic abstractions for accessing I/O ports and MMIO.
Stall Services Class	The Extended SAL Stall Services Class provides functions to perform calibrated delays.
Firmware Volume Block Services Class	The Extended SAL Firmware Volume Block Services Class provides services that are equivalent to the Firmware Volume Block Protocol in the *Platform Initialization Specification*.

Table 4. Extended SAL Service Classes – Other

Name	Description
MP Services Class	The Extended SAL MP Services Class provides services for managing multiple CPUs.

MCA Services Class	TBD

11.3.1 Extended SAL Base I/O Services Class

Summary

The Extended SAL Base I/O Services Class provides the basic abstractions for accessing I/O ports and MMIO.

GUID

```
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID_LO \
  0x451531e15aea42b5
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID_HI \
  0xa6657525d5b831bc
#define EFI_EXTENDED_SAL_BASE_IO_SERVICES_PROTOCOL_GUID \
  {0x5aea42b5,0x31e1,0x4515,
  {0xbc,0x31,0xb8,0xd5,0x25,0x75,0x65,0xa6}}
```

Related Definitions

```
typedef enum {
  IoReadFunctionId,
  IoWriteFunctionId,
  MemReadFunctionId,
  MemWriteFunctionId,
} EFI_EXTENDED_SAL_BASE_IO_SERVICES_FUNC_ID;
```

Description

Table 5. Extended SAL Base I/O Services Class

Name	Description
ExtendedSalIoRead	This function is equivalent in functionality to the **Io.Read()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Read()** service is shown in Related Definitions.
ExtendedSalIoWrite	This function is equivalent in functionality to the **Io.Write()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Write()** service is shown in Related Definitions.
ExtendedSalMemRead	This function is equivalent in functionality to the **Mem.Read()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Read()** service is shown in Related Definitions.
ExtendedSalMemWrite	This function is equivalent in functionality to the **Mem.Write()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Write()** service is shown in Related Definitions.

ExtendedSalIoRead

Summary

This function is equivalent in functionality to the **Io.Read()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalIoRead (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalIoReadFunctionId**.

Arg2

Signifies the width of the I/O read operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the I/O read operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of I/O read operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The destination buffer to store the results. This argument is interpreted as a **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Io.Read()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Io.Read()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Io.Read() function in the CPU I/O PPI.

ExtendedSalIoWrite

Summary

This function is equivalent in functionality to the **Io.Write()** function of the CPU I/O PPI. See *Volume1:Platform Initialization Specification* Section 7.2. The function prototype for the **Io.Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalIoWrite (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalIoWriteFunctionId**.

Arg2

Signifies the width of the I/O write operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the I/O write operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of I/O write operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The source buffer of the value to write. This argument is interpreted as a **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Io.Write()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Io.Write()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Io.Write() function in the CPU I/O PPI.

ExtendedSalMemRead

Summary

This function is equivalent in functionality to the **Mem.Read()** function of the CPU I/O PPI. See *Volume 1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMemRead (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
 );
```

Parameters

FunctionId

Must be **EsalMemReadFunctionId**.

Arg2

Signifies the width of the MMIO read operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the MMIO read operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of MMIO read operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The destination buffer to store the results. This argument is interpreted as a **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Mem.Read()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Mem.Read()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Mem.Read() function in the CPU I/O PPI.

ExtendedSalMemWrite

Summary

This function is equivalent in functionality to the **Mem.Write()** function of the CPU I/O PPI. See *Volume 1:Platform Initialization Specification* Section 7.2. The function prototype for the **Mem.Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMemWrite (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMemWriteFunctionId**.

Arg2

Signifies the width of the MMIO write operation. This argument is interpreted as type **EFI_PEI_CPU_IO_PPI_WIDTH**. See the *Width* parameter in Related Definitions.

Arg3

The base address of the MMIO write operation. This argument is interpreted as a **UINT64**. See the *Address* parameter in Related Definitions.

Arg4

The number of MMIO write operations to perform. This argument is interpreted as a **UINTN**. See the *Count* parameter in Related Definitions.

Arg5

The source buffer of the value to write. This argument is interpreted as a **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_CPU_IO_PPI_IO_MEM) (
  IN  EFI_PEI_SERVICES          **PeiServices,
  IN  EFI_PEI_CPU_IO_PPI        *This,
  IN  EFI_PEI_CPU_IO_PPI_WIDTH  Width,
  IN  UINT64                    Address,
  IN  UINTN                     Count,
  IN  OUT VOID                  *Buffer
  );
```

Description

This function performs the equivalent operation as the **Mem.Write()** function in the CPU I/O PPI. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the status from performing the **Mem.Write()** function of the CPU I/O PPI is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Mem.Write() function in the CPU I/O PPI.

11.4 Extended SAL Stall Services Class

Summary

The Extended SAL Stall Services Class provides functions to perform calibrated delays.

GUID

```
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID_LO \
  0x4d8cac2753a58d06
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID_HI \
  0x704165808af0e9b5
#define EFI_EXTENDED_SAL_STALL_SERVICES_PROTOCOL_GUID \
  {0x53a58d06,0xac27,0x4d8c,\
  {0xb5,0xe9,0xf0,0x8a,0x80,0x65,0x41,0x70}}
```

Related Definitions

```
typedef enum {
  StallFunctionId,
} EFI_EXTENDED_SAL_STALL_FUNC_ID;
```

Description

Table 6. Extended SAL Stall Services Class

Name	Description
ExtendedSalStall	This function is equivalent in functionality to the EFI Boot Service **Stall()**. See *UEFI 2.0 Specification* Section 6.5. The function prototype for the **Stall()** service is shown in Related Definitions.

ExtendedSalStall

Summary

This function is equivalent in functionality to the EFI Boot Service **Stall()**. See *UEFI 2.0 Specification* Section 6.5. The function prototype for the **Stall()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalStall (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalStallFunctionId**.

Arg2

Specifies the delay in microseconds. This argument is interpreted as type **UINTN**. See *Microseconds* in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_STALL) (
  IN  UINTN  Microseconds
  );
```

Description

This function performs the equivalent operation as the **Stall()** function in the EFI Boot Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Stall()** function of the EFI Boot Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Stall() function in the EFI Boot Services Table.

11.4.1 Extended SAL Real Time Clock Services Class

Summary

The Extended SAL Real Time Clock Services Class provides functions to access the real time clock.

GUID

```
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID_LO \
  0x4d02efdb7e97a470
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID_HI \
  0x96a27bd29061ce8f
#define EFI_EXTENDED_SAL_RTC_SERVICES_PROTOCOL_GUID  \
  {0x7e97a470,0xefdb,0x4d02, \
  {0x8f,0xce,0x61,0x90,0xd2,0x7b,0xa2,0x96}
```

Related Definitions

```
typedef enum {
  GetTimeFunctionId,
  SetTimeFunctionId,
  GetWakeupTimeFunctionId,
  SetWakeupTimeFunctionId,
  GetRtcClassMaxFunctionId
  InitializeThresholdFunctionId,
  BumpThresholdCountFunctionId,
  GetThresholdCountFunctionId
} EFI_EXTENDED_SAL_RTC_SERVICES_FUNC_ID;
```

Description

Table 7. Extended SAL Real Time Clock Services Class

Name	Description
ExtendedSalGetTime	This function is equivalent in functionality to the EFI Boot Service **GetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetTime()** service is shown in Related Definitions.
ExtendedSalSetTime	This function is equivalent in functionality to the EFI Runtime Service **SetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetTime()** service is shown in Related Definitions.
ExtendedSalGetWakeupTime	This function is equivalent in functionality to the EFI Runtime Service **GetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetWakeupTime()** service is shown in Related Definitions.
ExtendedSalSetWakeupTime	This function is equivalent in functionality to the EFI Runtime Service **SetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetWakeupTime()** service is shown in Related Definitions.

ExtendedSalGetTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_TIME_CAPABILITIES** structure. See *Capabilities* in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_TIME) (
  OUT EFI_TIME                *Time,
  OUT EFI_TIME_CAPABILITIES   *Capabilities   OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **GetTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetTime() function in the EFI Runtime Services Table.

ExtendedSalSetTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetTime (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_TIME) (
  IN EFI_TIME   *Time
  );
```

Description

This function performs the equivalent operation as the **SetTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetTime() function in the EFI Runtime Services Table.

ExtendedSalGetWakeupTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **GetWakeupTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetWakeupTime (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetWakeupTimeFunctionId**.

Arg2

This argument is interpreted as a pointer to a **BOOLEAN** value. See *Enabled* in Related Definitions.

Arg3

This argument is interpreted as a pointer to a **BOOLEAN** value. See *Pending* in Related Definitions.

Arg4

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_WAKEUP_TIME) (
  OUT BOOLEAN    *Enabled,
  OUT BOOLEAN    *Pending,
  OUT EFI_TIME   *Time
  );
```

Description

This function performs the equivalent operation as the **GetWakeupTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetWakeupTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetWakeupTime() function in the EFI Runtime Services Table.

ExtendedSalSetWakeupTime

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetWakeupTime()**. See *UEFI 2.0 Specification* Section 7.2. The function prototype for the **SetWakeupTime()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetWakeupTime (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetWakeupTimeFunctionId**.

Arg2

This argument is interpreted as a **BOOLEAN** value. See *Enable* in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_TIME** structure. See *Time* in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_WAKEUP_TIME) (
  IN BOOLEAN   Enable,
  IN EFI_TIME  *Time   OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **SetWakeupTime()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetWakeupTime()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetWakeupTime() function in the EFI Runtime Services Table.

11.4.2 Extended SAL Reset Services Class

Summary

The Extended SAL Reset Services Class provides platform reset services.

GUID

```
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID_LO \
  0x46f58ce17d019990
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID_HI \
  0xa06a6798513c76a7
#define EFI_EXTENDED_SAL_RESET_SERVICES_PROTOCOL_GUID \
  {0x7d019990,0x8ce1,0x46f5,
 {0xa7,0x76,0x3c,0x51,0x98,0x67,0x6a,0xa0}}
```

Related Definitions

```
typedef enum {
  ResetSystemFunctionId,
} EFI_EXTENDED_SAL_RESET_FUNC_ID;
```

Description

Table 8. Extended SAL Reset Services Class

Name	Description
ExtendedSalResetSystem	This function is equivalent in functionality to the EFI Runtime Service **ResetSystem()**. See *UEFI 2.0 Specification* Section 7.4.1. The function prototype for the **ResetSystem()** service is shown in Related Definitions.

ExtendedSalResetSystem

Summary

This function is equivalent in functionality to the EFI Runtime Service **ResetSystem()**. See *UEFI 2.0 Specification* Section 7.4.1. The function prototype for the **ResetSystem()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalResetSystem (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalResetSystemFunctionId**.

Arg2

This argument is interpreted as a **EFI_RESET_TYPE** value. See *ResetType* in Related Definitions.

Arg3

This argument is interpreted as **EFI_STATUS** value. See *ResetStatus* in Related Definitions.

Arg4

This argument is interpreted as **UINTN** value. See *DataSize* in Related Definitions.

Arg5

This argument is interpreted a pointer to a Unicode string. See *ResetData* in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
VOID
(EFIAPI *EFI_RESET_SYSTEM) (
  IN EFI_RESET_TYPE   ResetType,
  IN EFI_STATUS       ResetStatus,
  IN UINTN            DataSize,
  IN CHAR16           *ResetData   OPTIONAL
  );
```

Description

This function performs the equivalent operation as the **ResetSystem()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **ResetSystem()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the ResetSystem() function in the EFI Runtime Services Table.

11.4.3 Extended SAL PCI Services Class

Summary

The Extended SAL PCI Services Class provides services to perform PCI configuration cycles.

GUID

```
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID_LO \
  0x4905ad66a46b1a31
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID_HI \
  0x6330dc59462bf692
#define EFI_EXTENDED_SAL_PCI_SERVICES_PROTOCOL_GUID \
  {0xa46b1a31,0xad66,0x4905,
  {0x92,0xf6,0x2b,0x46,0x59,0xdc,0x30,0x63}}
```

Related Definitions

```
typedef enum {
  SalPciConfigReadFunctionId,
  SalPciConfigWriteFunctionId,
} EFI_EXTENDED_SAL_PCI_SERVICES_FUNC_ID;
```

Description

Table 9. Extended SAL PCI Services Class

Name	Description
ExtendedSalPciRead	This function is equivalent in functionality to the SAL Procedure `SAL_PCI_CONFIG_READ`. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalPciWrite	This function is equivalent in functionality to the SAL Procedure `SAL_PCI_CONFIG_WRITE`. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalPciRead

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_READ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalPciRead (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPciReadFunctionId**.

Arg2

address parameter to **SAL_PCI_CONFIG_WRITE**.

Arg3

size parameter to **SAL_PCI_CONFIG_WRITE**.

Arg4

address_type parameter to **SAL_PCI_CONFIG_WRITE**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalPciWrite

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_PCI_CONFIG_WRITE**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalPciWrite (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPciWriteFunctionId**.

Arg2

address parameter to **SAL_PCI_CONFIG_WRITE**.

Arg3

size parameter to **SAL_PCI_CONFIG_WRITE**.

Arg4

value parameter to **SAL_PCI_CONFIG_WRITE**.

Arg5

address_type parameter to **SAL_PCI_CONFIG_WRITE**.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

11.4.4 Extended SAL Cache Services Class

Summary

The Extended SAL Cache Services Class provides services to initialize and flush the caches.

GUID

```
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID_LO \
  0x4ba52743edc9494
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID_HI \
  0x88f11352ef0a1888
#define EFI_EXTENDED_SAL_CACHE_SERVICES_PROTOCOL_GUID \
  {0xedc9494,0x2743,0x4ba5,\
 {0x88,0x18,0x0a,0xef,0x52,0x13,0xf1,0x88}}
```

Related Definitions

```
typedef enum {
  SalCacheInitFunctionId,
  SalCacheFlushFunctionId,
  SalCacheClassMaxFunctionId
} EFI_EXTENDED_SAL_CACHE_SERVICES_FUNC_ID;
```

Description

Table 10. Extended SAL Cache Services Class

Name	Description
ExtendedSalCacheInit	This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_INIT**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalCacheFlush	This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_FLUSH**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalCacheInit

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_INIT**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalCacheInit (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalCacheInitFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalCacheFlush

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CACHE_FLUSH**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalCacheFlush (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalCacheFlushFunctionId**.

Arg2

i_or_d parameter in **SAL_CACHE_FLUSH**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

11.4.5 Extended SAL PAL Services Class

Summary

The Extended SAL PAL Services Class provides services to make PAL calls.

GUID

```
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID_LO \
  0x438d0fc2e1cd9d21
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID_HI \
  0x571e966de6040397
#define EFI_EXTENDED_SAL_PAL_SERVICES_PROTOCOL_GUID \
  {0xe1cd9d21,0x0fc2,0x438d, \
 {0x97,0x03,0x04,0xe6,0x6d,0x96,0x1e,0x57}}
```

Related Definitions

```
typedef enum {
  PalProcFunctionId,
  SetNewPalEntryFunctionId,
  GetNewPalEntryFunctionId,
  EsalUpdatePalFunctionId,
} EFI_EXTENDED_SAL_PAL_SERVICES_FUNC_ID;
```

Description

Table 11. Extended SAL PAL Services Class

Name	Description
ExtendedSalPalProc	This function provides a C wrapper for making PAL Procedure calls. See the *Intel Itanium Architecture Software Developers Manual Volume2: System Architecture* Section 11.10 for details on the PAL calling conventions and the set of PAL Procedures.
ExtendedSalSetNewPalEntry	This function records the physical or virtual PAL entry point.
ExtendedSalSetNewPalEntry	This function retrieves the physical or virtual PAL entry point.

ExtendedSalPalProc

Summary

This function provides a C wrapper for making PAL Procedure calls. See the *Intel Itanium Architecture Software Developers Manual Volume2: System Architecture* Section 11.10 for details on the PAL calling conventions and the set of PAL Procedures.

Prototype

```
PAL_PROC_RETURN
EFIAPI
ExtendedSalPalProc (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPalProcFunctionId**.

Arg2

PAL_PROC Function ID.

Arg3

Arg2of the **PAL_PROC**.

Arg4

Arg3 of the **PAL_PROC**.

Arg5

Arg4 of the **PAL_PROC**.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function provide a C wrapper for making PAL Procedure calls. The **PAL_PROC** Function ID in Arg2 is used to determine if the **PAL_PROC** is stacked or static. If the PAL has been shadowed, then the memory copy of the PAL is called. Otherwise, the ROM version of the PAL is called. The caller does not need to worry whether or not the PAL has been shadowed or not (except for the fact that some of the PAL calls don't work until PAL has been shadowed). If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the return status from the **PAL_PROC** is returned.

ExtendedSalSetNewPalEntry

Summary

This function records the physical or virtual PAL entry point.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetNewPalEntry (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetNewPalEntryFunctionId**.

Arg2

This parameter is interpreted as a **BOOLEAN**. If it is **TRUE**, then PAL Entry Point specified by *Arg3* is a physical address. If it is **FALSE**, then the Pal Entry Point specified by *Arg3* is a virtual address.

Arg3

The PAL Entry Point that is being set.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function records the PAL Entry Point specified by *Arg3*, so **PAL_PROC** calls can be made with the **EsalPalProcFunctionId** Function ID. If *Arg2* is **TRUE**, then *Arg3* is the physical address of the PAL Entry Point. If *Arg2* is **FALSE**, then *Arg3* is the virtual address of the PAL Entry Point. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The PAL Entry Point was set
EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.

ExtendedSalGetNewPalEntry

Summary

This function retrieves the physical or virtual PAL entry point.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNewPalEntry (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNewPalEntryFunctionId**.

Arg2

This parameter is interpreted as a **BOOLEAN.** If it is **TRUE**, then physical address of the PAL Entry Point is retrieved. If it is **FALSE**, then the virtual address of the Pal Entry Point is retrieved.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the PAL Entry Point that as previously set with **EsalSetNewPalEntryFunctionId**. If *Arg2* is **TRUE**, then the physical address of the PAL Entry Point is returned in **SAL_RETURN_REGS**.*r9* and **EFI_SAL_SUCCESS** is returned. If *Arg2* is **FALSE** and a virtual mapping for the PAL Entry Point is not available, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. If *Arg2* is **FALSE** and a virtual mapping for the PAL Entry Point is available, then the virtual address of the PAL Entry Point is returned in **SAL_RETURN_REGS**.*r9* and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The PAL Entry Point was retrieved and returned in SAL_RETURN_REGS.r9.
EFI_SAL_VIRTUAL_ADDRESS_ERROR	A request for the virtual mapping of the PAL Entry Point was requested, and a virtual mapping is not currently available.

ExtendedSalUpdatePal

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_UPDATE_PAL**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalUpdatePal (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalUpdatePal**.

Arg2

param_buf parameter to **SAL_UPDATE_PAL**.

Arg3

scratch_buf parameter to **SAL_UPDATE_PAL**.

Arg4

scratch_buf_size parameter to **SAL_UPDATE_PAL**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

11.4.6 Extended SAL Status Code Services Class

Summary

The Extended SAL Status Code Services Class provides services to report status code information.

GUID

```
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID_LO \
  0x420f55e9dbd91d
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID_HI \
  0x4fb437849f5e3996
#define EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_PROTOCOL_GUID  \
  {0xdbd91d,0x55e9,0x420f,
  {0x96,0x39,0x5e,0x9f,0x84,0x37,0xb4,0x4f}}
```

Related Definitions

```
typedef enum {
  ReportStatusCodeServiceFunctionId,
} EFI_EXTENDED_SAL_STATUS_CODE_SERVICES_FUNC_ID;
```

Description

Table 12. Extended SAL Status Code Services Class

Name	Description
ExtendedSalReportStatusCode	This function is equivalent in functionality to the **ReportStatusCode()** service of the Status Code Runtime Protocol. See Section 12.2 of the *Volume 2:Platform Initialization Specification, Driver Execution Environment, Core Interface*. The function prototype for the **ReportStatusCode()** service is shown in Related Definitions.

ExtendedSalReportStatusCode

Summary

This function is equivalent in functionality to the **ReportStatusCode()** service of the Status Code Runtime Protocol. See Section 12.2 of the *Volume 2:Platform Initialization Specification, Driver Execution Environment, Core Interface*. The function prototype for the **ReportStatusCode()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalReportStatusCode (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalReportStatusCodeFunctionId**.

Arg2

This argument is interpreted as type **EFI_STATUS_CODE_TYPE**. See the *Type* parameter in Related Definitions.

Arg3 *T*

his argument is interpreted as type **EFI_STATUS_CODE_VALUE**. See the *Value* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINT32**. See the *Instance* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **CONST EFI_GUID**. See the *CallerId* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to type **CONST EFI_STATUS_CODE_DATA**. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_REPORT_STATUS_CODE) (
  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
  );
```

Description

This function performs the equivalent operation as the **ReportStatusCode** function of the Status Code Runtime Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **ReportStatusCode()** function of the Status Code Runtime Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the ReportStatusCode() function in the Status Code Runtime Protocol.

11.4.7 Extended SAL Monotonic Counter Services Class

Summary

The Extended SAL Monotonic Counter Services Class provides functions to access the monotonic counter.

GUID

```
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID_LO \
  0x408b75e8899afd18
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID_HI \
  0x54f4cd7e2e6e1aa4
#define EFI_EXTENDED_SAL_MTC_SERVICES_PROTOCOL_GUID \
  {0x899afd18,0x75e8,0x408b,\
  {0xa4,0x1a,0x6e,0x2e,0x7e,0xcd,0xf4,0x54}}
```

Related Definitions

```
typedef enum {
  GetNextHighMotonicCountFunctionId,
} EFI_EXTENDED_SAL_MTC_SERVICES_FUNC_ID;
```

Description

Table 13. Extended SAL Monotonic Counter Services Class

Name	Description
ExtendedSalGetNextHighMtc	This function is equivalent in functionality to the EFI Runtime Service **GetNextHighMonotonicCount()**. See *UEFI 2.0 Specification* Section 7.4.2. The function prototype for the **GetNextHighMonotonicCount()** service is shown in Related Definitions.

ExtendedSalGetNextHighMtc

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetNextHighMonotonicCount()**. See *UEFI 2.0 Specification* Section 7.4.2. The function prototype for the **GetNextHighMonotonicCount()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNextHighMtc (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNextHighMtcFunctionId**.

Arg2

This argument is interpreted as a pointer to a **UINT32**. See the *HighCount* parameter in Related Definitions.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_NEXT_HIGH_MONO_COUNT) (
  OUT UINT32   *HighCount
  );
```

Description

This function performs the equivalent operation as the **GetNextHighMonotoicCount()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetNextHighMonotonicCount()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetNextHighMonotonicCount() function in the EFI Runtime Services Table.

11.4.8 Extended SAL Variable Services Class

Summary

The Extended SAL Variable Services Class provides functions to access EFI variables.

GUID

```
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID_LO \
  0x4370c6414ecb6c53
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID_HI \
  0x78836e490e3bb28c
#define EFI_EXTENDED_SAL_VARIABLE_SERVICES_PROTOCOL_GUID \
  {0x4ecb6c53,0xc641,0x4370, \
  {0x8c,0xb2,0x3b,0x0e,0x49,0x6e,0x83,0x78}}
```

Related Definitions

```
typedef enum {
  EsalGetVariableFunctionId,
  EsalGetNextVariableNameFunctionId,
  EsalSetVariableFunctionId,
  EsalQueryVariableInfoFunctionId,
} EFI_EXTENDED_SAL_VARIABLE_SERVICES_FUNC_ID;
```

Description

Table 14. Extended SAL Variable Services Class

Name	Description
ExtendedSalGetVariable	This function is equivalent in functionality to the EFI Runtime Service **GetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetVariable()** service is shown in Related Definitions.
ExtendedSalGetNextVariableName	This function is equivalent in functionality to the EFI Runtime Service **GetNextVariableName()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetNextVariableName()** service is shown in Related Definitions.
ExtendedSalSetVariable	This function is equivalent in functionality to the EFI Runtime Service **SetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **SetVariable()** service is shown in Related Definitions.
ExtendedSalQueryVariableInfo	This function is equivalent in functionality to the EFI Runtime Service **QueryVariableInfo()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **QueryVariableInfo()** service is shown in Related Definitions.

ExtendedSalGetVariable

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetVariable()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetVariable (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetVariableFunctionId**.

Arg2

This argument is interpreted as a pointer to a Unicode string. See the *VariableName* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINTN**. See the *DataSize* parameter in Related Definitions.

Arg6

This argument is interpreted as a pointer to a buffer with type **VOID ***. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_VARIABLE) (
  IN      CHAR16     *VariableName,
  IN      EFI_GUID   *VendorGuid,
  OUT     UINT32     *Attributes,    OPTIONAL
  IN OUT  UINTN      *DataSize,
  OUT     VOID       *Data
  );
```

Description

This function performs the equivalent operation as the **GetVariable()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetVariable()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetVariable() function in the EFI Runtime Services Table.

ExtendedSalGetNextVariableName

Summary

This function is equivalent in functionality to the EFI Runtime Service **GetNextVariableName()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **GetNextVariableName()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetNextVariableName (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetNextVariableNameFunctionId**.

Arg2

This argument is interpreted as a pointer to value of type **UINTN**. See the *VariableNameSize* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to a Unicode string. See the *VendorName* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_NEXT_VARIABLE_NAME) (
  IN OUT UINTN      *VariableNameSize,
  IN OUT CHAR16     *VariableName,
  IN OUT EFI_GUID   *VendorGuid
  );
```

Description

This function performs the equivalent operation as the **GetNextVariableName()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetNextVariableName()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetNextVariableName() function in the EFI Runtime Services Table.

ExtendedSalSetVariable

Summary

This function is equivalent in functionality to the EFI Runtime Service **SetVariable()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **SetVariable()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetVariable (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetVariableFunctionId**.

Arg2

This argument is interpreted as a pointer to a Unicode string. See the *VariableName* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to an **EFI_GUID**. See the *VendorGuid* parameter in Related Definitions.

Arg4

This argument is interpreted as a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg5

This argument is interpreted as a value of type **UINTN**. See the *DataSize* parameter in Related Definitions.

Arg6

This argument is interpreted as a pointer to a buffer with type **VOID ***. See the *Data* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_SET_VARIABLE) (
  IN  CHAR16     *VariableName,
  IN  EFI_GUID   *VendorGuid,
  IN  UINT32     Attributes,
  IN  UINTN      DataSize,
  IN  VOID       *Data
  );
```

Description

This function performs the equivalent operation as the **SetVariable()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetVariable()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetVariable() function in the EFI Runtime Services Table.

ExtendedSalQueryVariableInfo

Summary

This function is equivalent in functionality to the EFI Runtime Service **QueryVariableInfo()**. See *UEFI 2.0 Specification* Section 7.1. The function prototype for the **QueryVariableInfo()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalQueryVariableInfo (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalQueryVariableInfoFunctionId**.

Arg2

This argument is interpreted as a value of type **UINT32**. See the *Attributes* parameter in Related Definitions.

Arg3

This argument is interpreted as a pointer to a value of type **UINT64**. See the *MaximumVariableStorageSize* parameter in Related Definitions.

Arg4

This argument is interpreted as a pointer to a value of type **UINT64**. See the *RemainingVariableStorageSize* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINT64**. See the *MaximumVariableSize* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_QUERY_VARIABLE_INFO) (
  IN  UINT32    Attributes,
  OUT UINT64    *MaximumVariableStorageSize,
  OUT UINT64    *RemainingVariableStorageSize,
  OUT UINT64    *MaximumVariableSize
  );
```

Description

This function performs the equivalent operation as the **QueryVariableInfo()** function in the EFI Runtime Services Table. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **QueryVariableInfo()** function of the EFI Runtime Services Table is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the QueryVariableInfo() function in the EFI Runtime Services Table.

11.4.9 Extended SAL Firmware Volume Block Services Class

Summary

The Extended SAL Firmware Volume Block Services Class provides services that are equivalent to the Firmware Volume Block Protocol in the *Platform Initialization Specification*.

GUID

```
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID_LO \
  0x4f1dbcbba2271df1
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID_HI \
  0x1a072f17bc06a998
#define EFI_EXTENDED_SAL_FVB_SERVICES_PROTOCOL_GUID \
  {0xa2271df1,0xbcbb,0x4f1d,\
  {0x98,0xa9,0x06,0xbc,0x17,0x2f,0x07,0x1a}}
```

Related Definitions

```
typedef enum {
  ReadFunctionId,
  WriteFunctionId,
  EraseBlockFunctionId,
  GetVolumeAttributesFunctionId,
  SetVolumeAttributesFunctionId,
  GetPhysicalAddressFunctionId,
  GetBlockSizeFunctionId,
} EFI_EXTENDED_SAL_FV_BLOCK_SERVICES_FUNC_ID;
```

Description

Table 15. Extended SAL Variable Services Class

Name	Description
ExtendedSalRead	This function is equivalent in functionality to the **Read()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **Read()** service is shown in Related Definitions.
ExtendedSalWrite	This function is equivalent in functionality to the **Write()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **Write()** service is shown in Related Definitions.
ExtendedSalEraseBlock	This function is equivalent in functionality to the **EraseBlocks()** service of the EFI Firmware Volume Block Protocol except this function can only erase one block per request. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **EraseBlock()** service is shown in Related Definitions.
ExtendedSalGetAttributes	This function is equivalent in functionality to the **GetAttributes()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **GetAttributes()** service is shown in Related Definitions.

ExtendedSalSetAttributes	This function is equivalent in functionality to the **`SetAttributes()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`SetAttributes()`** service is shown in Related Definitions.
ExtendedSalGetPhysicalAddress	This function is equivalent in functionality to the **`GetPhysicalAddress()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements.* The function prototype for the **`GetPhysicalAddress()`** service is shown in Related Definitions.
ExtendedSalGetBlockSize	This function is equivalent in functionality to the **`GetBlockSize()`** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`GetBlockSize()`** service is shown in Related Definitions.
ExtendedSalEraseCustomBlockRange	This function is similar in functionality to the **`EraseBlocks()`** service of the EFI Firmware Volume Block Protocol except this function can specify a range of blocks with offsets into the starting and ending block. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **`EraseBlock()`** service is shown in Related Definitions.

ExtendedSalRead

Summary

This function is equivalent in functionality to the **Read()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **Read()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRead (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbReadFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. See the *Lba* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINTN**. See the *Offset* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **UINTN**. See the *NumBytes* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to a buffer of type **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_READ) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  IN EFI_LBA                              Lba,
  IN UINTN                                Offset,
  IN OUT UINTN                            *NumBytes,
  OUT UINT8                               *Buffer
  );
```

Description

This function performs the equivalent operation as the **Read()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Read()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Read() function in the EFI Firmware Volume Block Protocol.

ExtendedSalWrite

Summary

This function is equivalent in functionality to the **Write()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **Write()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalWrite (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbWriteFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. See the *Lba* parameter in Related Definitions.

Arg4

This argument is interpreted as type **UINTN**. See the *Offset* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to type **UINTN**. See the *NumBytes* parameter in Related Definitions.

Arg6

This argument is interpreted as pointer to a buffer of type **VOID ***. See the *Buffer* parameter in Related Definitions.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_WRITE) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  IN EFI_LBA                             Lba,
  IN UINTN                               Offset,
  IN OUT UINTN                           *NumBytes,
  IN UINT8                               *Buffer
  );
```

Description

This function performs the equivalent operation as the **Write()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **Write()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the Write() function in the EFI Firmware Volume Block Protocol.

ExtendedSalEraseBlock

Summary

This function is equivalent in functionality to the **EraseBlocks()** service of the EFI Firmware Volume Block Protocol except this function can only erase one block per request. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **EraseBlock()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalEraseBlock (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbEraseBlockFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. This is the logical block address in the firmware volume to erase. Only a single block can be specified with this Extended SAL Procedure. The **EraseBlocks()** function in the EFI Firmware Volume Block Protocol supports a variable number of arguments that allow one or more block ranges to be specified.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_ERASE_BLOCKS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  ...
  );
```

Description

This function performs the equivalent operation as the **EraseBlock()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **EraseBlock()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the EraseBlock() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetAttributes

Summary

This function is equivalent in functionality to the **GetAttributes()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetAttributes()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetAttributes (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetAttributesFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_FVB_ATTRIBUTES**. See the *Attributes* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
EFI_STATUS
(EFIAPI *EFI_FVB_GET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  OUT EFI_FVB_ATTRIBUTES                  *Attributes
  );
```

Description

This function performs the equivalent operation as the **GetAttributes()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetAttributes()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetAttributes() function in the EFI Firmware Volume Block Protocol.

ExtendedSalSetAttributes

Summary

This function is equivalent in functionality to the **SetAttributes()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **SetAttributes()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetAttributes (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbSetAttributesFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_FVB_ATTRIBUTES**. See the *Attributes* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_SET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  IN OUT EFI_FVB_ATTRIBUTES              *Attributes
  );
```

Description

This function performs the equivalent operation as the **SetAttributes()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **SetAttributes()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the SetAttributes() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetPhysicalAddress

Summary

This function is equivalent in functionality to the **GetPhysicalAddress()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetPhysicalAddress()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetPhysicalAddress (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetPhysicalAddressFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as pointer to a value of type **EFI_PHYSICAL_ADDRESS**. See the *Address* parameter in Related Definitions.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_GET_PHYSICAL_ADDRESS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  OUT EFI_PHYSICAL_ADDRESS               *Address
  );
```

Description

This function performs the equivalent operation as the **GetPhysicalAddress()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetPhysicalAddress()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetPhysicalAddress() function in the EFI Firmware Volume Block Protocol.

ExtendedSalGetBlockSize

Summary

This function is equivalent in functionality to the **GetBlockSize()** service of the EFI Firmware Volume Block Protocol. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **GetBlockSize()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetBlockSize (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbGetBlockSizeFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**.

Arg3

This argument is interpreted as type **EFI_LBA**. See *Lba* parameter in Related Definitions.

Arg4 T

his argument is interpreted as a pointer to a value of type **UINTN**. See *BlockSize* parameter in Related Definitions.

Arg5

This argument is interpreted as a pointer to a value of type **UINTN**. See *NumberOfBlocks* parameter in Related Definitions.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_GET_BLOCK_SIZE) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  IN EFI_LBA                             Lba,
  OUT UINTN                              *BlockSize,
  OUT UINTN                              *NumberOfBlocks
  );
```

Description

This function performs the equivalent operation as the **GetBlockSize()** function of the EFI Firmware Volume Block Protocol. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **GetBlockSize()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the GetBlockSize() function in the EFI Firmware Volume Block Protocol.

ExtendedSalEraseCustomBlockRange

Summary

This function is similar in functionality to the **EraseBlocks()** service of the EFI Firmware Volume Block Protocol except this function can specify a range of blocks with offsets into the starting and ending block. See Section 2.4 of the *Volume 3:Platform Initialization Specification, Shared Architectural Elements*. The function prototype for the **EraseBlock()** service is shown in Related Definitions.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalEraseCustomBlockRange (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalFvbEraseCustomBlockRangeFunctionId**.

Arg2

This argument is interpreted as type **UINTN** that represents the Firmware Volume Block instance. This instance value is used to lookup a **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**. See the *This* parameter in Related Definitions.

Arg3

This argument is interpreted as type **EFI_LBA**. This is the starting logical block address in the firmware volume to erase.

Arg4

This argument is interpreted as type **UINTN**. This is the offset into the starting logical block to erase.

Arg5

This argument is interpreted as type **EFI_LBA**. This is the ending logical block address in the firmware volume to erase.

Arg6

This argument is interpreted as type **UINTN**. This is the offset into the ending logical block to erase.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Related Definitions

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_ERASE_BLOCKS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  ...
  );
```

Description

This function performs a similar operation as the **EraseBlock()** function of the EFI Firmware Volume Block Protocol. The main difference is that this function can perform a partial erase of the starting and ending blocks. The start of the erase operation is specified by *Arg3* and *Arg4*. The end of the erase operation is specified by *Arg5* and *Arg6*. If this function is called in virtual mode before any required mapping have been converted to virtual addresses, then **EFI_SAL_VIRTUAL_ADDRESS_ERROR** is returned. Otherwise, the one of the status codes defined in the **EraseBlock()** function of the EFI Firmware Volume Block Protocol is returned.

Status Codes Returned

EFI_SAL_VIRTUAL_ADDRESS_ERROR	This function was called in virtual mode before virtual mappings for the specified Extended SAL Procedure are available.
Other	See the return status codes for the EraseBlock() function in the EFI Firmware Volume Block Protocol.

11.4.10 Extended SAL MCA Log Services Class

Summary

The Extended SAL MCA Log Services Class provides logging services for MCA events.

GUID

```
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID_LO \
  0x4c0338a3cb3fd86e
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID_HI \
  0x7aaba2a3cf905c9a
#define EFI_EXTENDED_SAL_MCA_LOG_SERVICES_PROTOCOL_GUID \
  {0xcb3fd86e,0x38a3,0x4c03,\
  {0x9a,0x5c,0x90,0xcf,0xa3,0xa2,0xab,0x7a}}
```

Related Definitions

```
typedef enum {
  SalGetStateInfoFunctionId,
  SalGetStateInfoSizeFunctionId,
  SalClearStateInfoFunctionId,
  SalGetStateBufferFunctionId,
  SalSaveStateBufferFunctionId,
} EFI_EXTENDED_SAL_MCA_LOG_SERVICES_FUNC_ID;
```

ExtendedSalGetStateInfo

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_GET_STATE_INFO**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateInfo (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoFunctionId**.

Arg2

type parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

memaddr parameter to **SAL_GET_STATE_INFO**.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetStateInfoSize

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_GET_STATE_INFO_SIZE**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateInfoSize (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoSizeFunctionId**.

Arg2

type parameter to **SAL_GET_STATE_INFO_SIZE**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalClearStateInfo

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_CLEAR_STATE_INFO**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalClearStateInfo (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateInfoFunctionId**.

Arg2

type parameter to **SAL_CLEAR_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetStateBuffer

Summary

Returns a memory buffer to store error records.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetStateBuffer (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetStateBufferFunctionId**.

Arg2

Same as *type* parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns a memory buffer to store error records. The base address of the buffer is returned in **SAL_RETURN_REGS.***r9*, and the size of the buffer, in bytes, is returned in **SAL_RETURN_REGS.***r10*. If a buffer is not available, then **EFI_OUT_OF_RESOURCES** is returned. Otherwise, **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The memory buffer to store error records was returned in r9 and r10.
EFI_OUT_OF_RESOURCES	A memory buffer for string error records in not available.

ExtendedSalSaveStateBuffer

Summary

Saves a memory buffer containing an error records to nonvolatile storage.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSaveStateBuffer (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSaveStateBufferFunctionId**.

Arg2

Same as *type* parameter to **SAL_GET_STATE_INFO**.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function saved a memory buffer containing an error record to nonvolatile storage.

Status Codes Returned

EFI_SUCCESS	The memory buffer containing the error record was written to nonvolatile storage.
TBD	

11.4.11 Extended SAL Base Services Class

Summary

The Extended SAL Base Services Class provides base services that do not have any hardware dependencies including a number of SAL Procedures required by the *Intel Itanium Processor Family System Abstraction Layer Specification*.

GUID

```
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID_LO \
  0x41c30fe0d9e9fa06
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID_HI \
  0xf894335a4283fb96
#define EFI_EXTENDED_SAL_BASE_SERVICES_PROTOCOL_GUID \
  {0xd9e9fa06,0x0fe0,0x41c3,\
  {0x96,0xfb,0x83,0x42,0x5a,0x33,0x94,0xf8}}
```

Related Definitions

```
typedef enum {
  SalSetVectorsFunctionId,
  SalMcRendezFunctionId,
  SalMcSetParamsFunctionId,
  EsalGetVectorsFunctionId,
  EsalMcGetParamsFunctionId,
  EsalMcGetMcParamsFunctionId,
  EsalGetMcCheckinFlagsFunctionId,
  EsalGetPlatformBaseFreqFunctionId,
  EsalRegisterPhysicalAddrFunctionId,
  EsalBaseClassMaxFunctionId
} EFI_EXTENDED_SAL_BASE_SERVICES_FUNC_ID;
```

Description

Table 16. Extended SAL MP Services Class

Name	Description
ExtendedSalSetVectors	This function is equivalent in functionality to the SAL Procedure **SAL_SET_VECTORS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalMcRendez	This function is equivalent in functionality to the SAL Procedure **SAL_MC_RENDEZ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalMcSetParams	This function is equivalent in functionality to the SAL Procedure **SAL_MC_SET_PARAMS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalGetVectors	Retrieves information that was previously registered with the SAL Procedure **SAL_SET_VECTORS**.
ExtendedSalMcGetParams	Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.
ExtendedSalMcGetMcParams	Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.
ExtendedSalGetMcCheckinFlags	Used to determine if a specific CPU has called the SAL Procedure **SAL_MC_RENDEZ**.

ExtendedSalGetPlatformBaseFreq	This function is equivalent in functionality to the SAL Procedure **SAL_FREQ_BASE** with a clock_type of 0. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.
ExtendedSalRegisterPhysicalAddr	This function is equivalent in functionality to the SAL Procedure **SAL_REGISTER_PHYSICAL_ADDR**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

ExtendedSalSetVectors

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_SET_VECTORS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetVectors (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetVectorsFunctionId**.

Arg2

vector_type parameter to **SAL_SET_VECTORS**.

Arg3

phys_addr_1 parameter to **SAL_SET_VECTORS**.

Arg4

gp_1 parameter to **SAL_SET_VECTORS**.

Arg5

length_cs_1 parameter to **SAL_SET_VECTORS**.

Arg6

phys_addr_2 parameter to **SAL_SET_VECTORS**.

Arg7

gp_2 parameter to **SAL_SET_VECTORS**.

Arg8

length_cs_2 parameter to **SAL_SET_VECTORS**.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalMcRendez

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_MC_RENDEZ**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcRendez (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcRendezFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalMcSetParams

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_MC_SET_PARAMS**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcSetParams (
  IN UINT64    FunctionId,
  IN UINT64    Arg2,
  IN UINT64    Arg3,
  IN UINT64    Arg4,
  IN UINT64    Arg5,
  IN UINT64    Arg6,
  IN UINT64    Arg7,
  IN UINT64    Arg8,
  IN BOOLEAN   VirtualMode,
  IN VOID      *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcSetParamsFunctionId**.

Arg2

param_type parameter to **SAL_MC_SET_PARAMS**.

Arg3

i_or_m parameter to **SAL_MC_SET_PARAMS**.

Arg4

i_or_m_val parameter to **SAL_MC_SET_PARAMS**.

Arg5

time_out parameter to **SAL_MC_SET_PARAMS**.

Arg6

mca_opt parameter to **SAL_MC_SET_PARAMS**.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalGetVectors

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_SET_VECTORS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetVectors (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetVectorsFunctionId**.

Arg2

The vector type to retrieve. 0 – MCA, 1-BSP INIT, 2 – BOOT_RENDEZ, 3 – AP INIT.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns the vector information for the vector specified by *Arg2*. If the specified vector was not previously registered with the SAL Procedure **SAL_SET_VECTORS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the physical address of the requested vector is returned in **SAL_RETURN_REGS**.*r9*, the global pointer(GP) value is returned in **SAL_RETURN_REGS**.*r10*, the length and checksum information is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalMcGetParams

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetParams (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetParamsFunctionId**.

Arg2

The parameter type to retrieve. 1 – rendezvous interrupt, 2 – wake up, 3 – Corrected Platform Error Vector.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns information for the parameter type specified by *Arg2* that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**. If the parameter type specified by *Arg2* was not previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the **i_or_m** value is returned in **SAL_RETURN_REGS**.*r9*, the **i_or_m_val** value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SUCCESS** is returned.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalMcGetMcParams

Summary

Retrieves information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetMcParams (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetMcParamsFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns information that was previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**. If the information was not previously registered with the SAL Procedure **SAL_MC_SET_PARAMS**, then **SAL_NO_INFORMATION_AVAILABLE** is returned. Otherwise, the **rz_always** value is returned in **SAL_RETURN_REGS**.*r9*, **time_out** value is returned in **SAL_RETURN_REGS**.*r10*, **binit_escalate** value is returned in **SAL_RETURN_REGS.***r11*.

Status Codes Returned

EFI_SUCCESS	The information for the requested vector was returned,
SAL_NO_INFORMATION_AVAILABLE	The requested vector has not been registered with the SAL Procedure SAL_SET_VECTORS.

ExtendedSalGetMcCheckinFlags

Summary

Used to determine if a specific CPU has called the SAL Procedure **SAL_MC_RENDEZ**.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetMcCheckinFlags (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetMcChckinFlagsFunctionId**.

Arg2

The index of the CPU in the set of enabled CPUs to check.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function check to see if the CPU index specified by *Arg2* has called the SAL Procedure **SAL_MC_RENDEZ**. The CPU index values are assigned by the Extended SAL MP Services Class. If the CPU specified by *Arg2* has called the SAL Procedure **SAL_MC_RENDEZ**, then 1 is returned in **SAL_RETURN_REGS**.*r9*. Otherwise, **SAL_RETURN_REGS**.*r9* is set to 0. **EFI_SAL_SUCCESS** is always returned.

Status Codes Returned

EFI_SAL_SUCCESS	The checkin status of the requested CPU was returned.

ExtendedSalGetPlatformBaseFreq

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_FREQ_BASE** with a clock_type of 0. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcGetPlatformBaseFreq (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalMcGetPlatformBaseFreqFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8 *Reserved. Must be zero.*

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended

SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

ExtendedSalRegisterPhysicalAddr

Summary

This function is equivalent in functionality to the SAL Procedure **SAL_REGISTER_PHYSICAL_ADDR**. See the *Intel Itanium Processor Family System Abstraction Layer Specification* Chapter 9.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRegisterPhysicalAddr (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalRegisterPhysicalAddrFunctionId**.

Arg2

phys_entity parameter to **SAL_REGISTER_PHYSICAL_ADDRESS**.

Arg3

paddr parameter to **SAL_REGISTER_PHYSICAL_ADDRESS**.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

11.4.12 Extended SAL MP Services Class

Summary

The Extended SAL MP Services Class provides services for managing multiple CPUs.

GUID

```
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID_LO \
  0x4dc0cf18697d81a2
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID_HI \
  0x3f8a613b11060d9e
#define EFI_EXTENDED_SAL_MP_SERVICES_PROTOCOL_GUID \
  {0x697d81a2,0xcf18,0x4dc0,\
  {0x9e,0x0d,0x06,0x11,0x3b,0x61,0x8a,0x3f}}
```

Related Definitions

```
typedef enum {
  AddCpuDataFunctionId,
  RemoveCpuDataFunctionId,
  ModifyCpuDataFunctionId,
  GetCpuDataByIdFunctionId,
  GetCpuDataByIndexFunctionId,
  SendIpiFunctionId,
  CurrentProcInfoFunctionId,
  NumProcessorsFunctionId,
  SetMinStateFunctionId,
  GetMinStateFunctionId,
  EsalPhysicalIdInfo,
} EFI_EXTENDED_SAL_MP_SERVICES_FUNC_ID;
```

Description

Table 17. Extended SAL MP Services Class

Name	Description
ExtendedSalAddCpuData	Add a CPU to the database of CPUs.
ExtendedSalRemoveCpuData	Add a CPU to the database of CPUs.
ExtendedSalModifyCpuData	Updates the data for a CPU that is already in the database of CPUs.
ExtendedSalGetCpuDataById	Returns the information on a CPU specified by a Global ID.

ExtendedSalGetCpuDataByIndex	Returns information on a CPU specified by an index.
ExtendedSalWhoAmI	Returns the Global ID for the calling CPU.
ExtendedSalNumProcessors	Returns the number of currently enabled CPUs, the total number of CPUs, and the maximum number of CPUs that the platform supports.
ExtendedSalSetMinState	Sets the MINSTATE pointer for the CPU specified by a Global ID.
ExtendedSalGetMinState	Retrieves the MINSTATE pointer for the CPU specified by a Global ID.
ExtendedSalPhysicalIdInfo	Retrieves the Physical ID of a CPU in the platform.

ExtendedSalAddCpuData

Summary

Add a CPU to the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalAddCpuData (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalAddCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being added.

Arg3

The enable flag for the CPU being added. This value is interpreted as type **BOOLEAN**. **TRUE** means the CPU is enabled. **FALSE** means the CPU is disabled.

Arg4 *T*

he PAL Compatibility value for the CPU being added.

Arg5

The 16-bit Platform ID of the CPU being added.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function adds the CPU with a Global ID specified by *Arg2*, the enable flag specified by *Arg3*, and the PAL Compatibility value specified by *Arg4* to the database of CPUs in the platform. If there are not enough resource available to add the CPU, then **EFI_SAL_NOT_ENOUGH_SCRATCH** is returned. Otherwise, the CPU to added to the database, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU was added to the database.
EFI_SAL_NOT_ENOUGH_SCRATCH	There are not enough resource available to add the CPU.

ExtendedSalRemoveCpuData

Summary

Add a CPU to the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalRemoveCpuData (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalRemoveCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being added.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function removes the CPU with a Global ID specified by *Arg2* from the database of CPUs in the platform. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU specified by *Arg2* is removed from the database of CPUs, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU was removed from the database.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalModifyCpuData

Summary

Updates the data for a CPU that is already in the database of CPUs.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalModifyCpuData (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalModifyCpuDataFunctionId**.

Arg2

The 64-bit Global ID of the CPU being updated.

Arg3

The enable flag for the CPU being updated. This value is interpreted as type **BOOLEAN**. **TRUE** means the CPU is enabled. **FALSE** means the CPU is disabled.

Arg4

The PAL Compatibility value for the CPU being updated.

Arg5

The 16-bit Platform ID of the CPU being updated.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function updates the CPU with a Global ID specified by *Arg2*, the enable flag specified by *Arg3*, and the PAL Compatibility value specified by *Arg4* in the database of CPUs in the platform. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU specified by *Arg2* is updates with the enable flag specified by *Arg3* and the PAL Compatibility value specified by *Arg4*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU database was updated.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetCpuDataById

Summary

Returns the information on a CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetCpuDataById (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetCpuDataByIdFunctionId**.

Arg2

The 64-bit Global ID of the CPU to lookup.

Arg3 *T*

his parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in the set of enabled CPUs in the database is returned. If **FALSE**, then the index in the set of all CPUs in the database is returned.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the CPU specified by *Arg2* in the CPU database and returns the enable status and PAL Compatibility value. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the enable status is returned in **SAL_RETURN_REGS***.r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SAL_SUCCESS** is returned. If *Arg3* is **TRUE**, then the index of the CPU specified by *Arg2* in the set of enabled CPUs is returned in **SAL_RETURN_REGS**.*r11.* If *Arg3* is **FALSE**, then the index of the CPU specified by *Arg2* in the set of all CPUs is returned in **SAL_RETURN_REGS**.*r11*.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the specified CPU was returned.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetCpuDataByIndex

Summary

Returns information on a CPU specified by an index.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalGetCpuDataByIndex (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalGetCpuDataByIndexFunctionId**.

Arg2

The index of the CPU to lookup.

Arg3

This parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in *Arg2* is the index in the set of enabled CPUs. If **FALSE**, then the index in *Arg2* is the index in the set of all CPUs.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the CPU specified by *Arg2* in the CPU database and returns the enable status and PAL Compatibility value. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the enable status is returned in **SAL_RETURN_REGS**.*r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, the Global ID is returned in **SAL_RETURN_REGS**.*r11*, and **EFI_SAL_SUCCESS** is returned. If *Arg3* is **TRUE**, then *Arg2* is the index in the set of enabled CPUs. If *Arg3* is **FALSE**, then *Arg2* is the index in the set of all CPUs.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the specified CPU was returned.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalWhoiAmI

Summary

Returns the Global ID for the calling CPU.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalWhoAmI (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalWhoAmIFunctionId**.

Arg2 *T*

his parameter is interpreted as a **BOOLEAN** value. If **TRUE**, then the index in the set of enabled CPUs in the database is returned. If **FALSE**, then the index in the set of all CPUs in the database is returned.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the Global ID of the calling CPU. If the calling CPU is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the Global ID is returned in **SAL_RETURN_REGS**.*r9*, the PAL Compatibility value is returned in **SAL_RETURN_REGS**.*r10*, and **EFI_SAL_SUCCESS** is returned. If *Arg2* is **TRUE**, then the index of the calling CPU in the set of enabled CPUs is returned in **SAL_RETURN_REGS**.*r11*. If *Arg3* is **FALSE**, then the index of the calling CPU in the set of all CPUs is returned in **SAL_RETURN_REGS**.*r11*.

Status Codes Returned

EFI_SAL_SUCCESS	The Global ID for the calling CPU was returned.
EFI_SAL_NO_INFORMATION	The calling CPU is not in the database.

ExtendedSalNumProcessors

Summary

Returns the number of currently enabled CPUs, the total number of CPUs, and the maximum number of CPUs that the platform supports.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalNumProcessors (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalNumProcessorsFunctionId**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function returns the maximum number of CPUs that the platform supports in **SAL_RETURN_REGS**.*r9*, the total number of CPUs in **SAL_RETURN_REGS**.*r10*, and the number of enabled CPUs in **SAL_RETURN_REGS**.*r11*. **EFI_SAL_SUCCESS** is always returned.

Status Codes Returned

EFI_SAL_SUCCESS	The information on the number of CPUs in the platform was returned.

ExtendedSalSetMinState

Summary

Sets the MINSTATE pointer for the CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetMinState (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetMinStateFunctionId**.

Arg2

The 64-bit Global ID of the CPU to set the MINSTATE pointer.

Arg3

This parameter is interpreted as a pointer to the MINSTATE area for the CPU specified by Arg2.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function sets the MINSTATE pointer for the CPU specified by *Arg2* to the buffer specified by *Arg3*. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The MINSTATE pointer was set for the specified CPU.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalGetMinState

Summary

Retrieves the MINSTATE pointer for the CPU specified by a Global ID.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalSetMinState (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalSetMinStateFunctionId**.

Arg2

The 64-bit Global ID of the CPU to get the MINSTATE pointer.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the MINSTATE pointer for the CPU specified by *Arg2*. If the CPU specified by *Arg2* is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the MINSTATE pointer for the specified CPU is returned in **SAL_RETURN_REGS**.*r9*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The MINSTATE pointer for the specified CPU was retrieved.
EFI_SAL_NO_INFORMATION	The specified CPU is not in the database.

ExtendedSalPhysicalIdInfo

Summary

Returns the Physical ID for the calling CPU.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalPhysicalIdInfo (
  IN UINT64   FunctionId,
  IN UINT64   Arg2,
  IN UINT64   Arg3,
  IN UINT64   Arg4,
  IN UINT64   Arg5,
  IN UINT64   Arg6,
  IN UINT64   Arg7,
  IN UINT64   Arg8,
  IN BOOLEAN  VirtualMode,
  IN VOID     *ModuleGlobal  OPTIONAL
  );
```

Parameters

FunctionId

Must be **EsalPhysicalIdInfo**.

Arg2

Reserved. Must be zero.

Arg3

Reserved. Must be zero.

Arg4

Reserved. Must be zero.

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function looks up the Physical ID of the calling CPU. If the calling CPU is not present in the database, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the Physical ID is returned in **SAL_RETURN_REGS**.*r9*, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The Physical ID for the calling CPU was returned.
EFI_SAL_NO_INFORMATION	The calling CPU is not in the database.

11.4.13 Extended SAL MCA Services Class

Summary

The Extended SAL MCA Services Class provides services to

GUID

```
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID_LO \
  0x42b16cc72a591128
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID_HI \
  0xbb2d683b9358f08a
#define EFI_EXTENDED_SAL_MCA_SERVICES_PROTOCOL_GUID \
  {0x2a591128,0x6cc7,0x42b1,\
  {0x8a,0xf0,0x58,0x93,0x3b,0x68,0x2d,0xbb}}
```

Related Definitions

```
typedef enum {
  McaGetStateInfoFunctionId,
  McaRegisterCpuFunctionId,
} EFI_EXTENDED_SAL_MCA_SERVICES_FUNC_ID;
```

Description

Table 18. Extended SAL MCA Services Class

Name	Description
ExtendedSalMcaGetStateInfo	Obtain the buffer corresponding to the Machine Check Abort state information.
ExtendedSalMcaRegisterCpu	Register the CPU instance for the Machine Check Abort handling.

ExtendedSalMcaGetStateInfo

Summary

Obtain the buffer corresponding to the Machine Check Abort state information.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcaGetStateInfo (
  IN UINT64 FunctionId,
  IN UINT64 Arg2,
  IN UINT64 Arg3,
  IN UINT64 Arg4,
  IN UINT64 Arg5,
  IN UINT64 Arg6,
  IN UINT64 Arg7,
  IN UINT64 Arg8,
  IN BOOLEAN VirtualMode,
  IN VOID    *ModuleGlobal OPTIONAL
  );
```

Parameters

FunctionId

Must be EsalMcaGetStateInfoFunctionId.

Arg2

The 64-bit Global ID of the CPU to get the MINSTATE pointer.

Arg3

Pointer to the state buffer for output.

Arg4

Pointer to the required buffer size for output

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function retrieves the MINSTATE pointer specified by *Arg3* for the CpuId specified by *Arg2*, and calculates required size specified by *Arg4*. If the CPU specified by *Arg2* was not registered in system, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU state buffer related information will be returned, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	MINSTATE successfully got and size calculated.
EFI_SAL_NO_INFORMATION	The CPU was not registered in system.

ExtendedSalMcaRegisterCpu

Summary

Register the CPU instance for the Machine Check Abort handling.

Prototype

```
SAL_RETURN_REGS
EFIAPI
ExtendedSalMcaRegisterCpu (
IN UINT64 FunctionId,
IN UINT64 Arg2,
IN UINT64 Arg3,
IN UINT64 Arg4,
IN UINT64 Arg5,
IN UINT64 Arg6,
IN UINT64 Arg7,
IN UINT64 Arg8,
IN BOOLEAN VirtualMode,
IN VOID   *ModuleGlobal OPTIONAL
);
```

Parameters

FunctionId

Must be **EsalMcaRegisterCpuFunctionId**.

Arg2

The 64-bit Global ID of the CPU to register its MCA state buffer.

Arg3

The pointer of the CPU's state buffer.

Arg4

Reserved. Must be zero

Arg5

Reserved. Must be zero.

Arg6

Reserved. Must be zero.

Arg7

Reserved. Must be zero.

Arg8

Reserved. Must be zero.

VirtualMode

TRUE if the Extended SAL Procedure is being invoked in virtual mode. **FALSE** if the Extended SAL Procedure is being invoked in physical mode.

ModuleGlobal

A pointer to the global context associated with this Extended SAL Procedure. Implementation dependent.

Description

This function registers MCA state buffer specified by *Agr3* for CPU specified by *Arg2*. If the CPU specified by *Arg2* was not registered in system, then **EFI_SAL_NO_INFORMATION** is returned. Otherwise, the CPU state buffer is registered for MCA handing, and **EFI_SAL_SUCCESS** is returned.

Status Codes Returned

EFI_SAL_SUCCESS	The CPU state buffer is registered for MCA handing successfully.
EFI_SAL_NO_INFORMATION	The CPU was not registered in system.

12 SMM SPI Protocol Stack

12.1 Design

The design of the SPI protocol stack is almost identical between SMM and DXE. See the section on SPI Protocol Stack in Volume 5 for more details. The differences are described in this section.

SMM SPI support is primarily used to support SPI flash devices. Since SMM does not support device paths, there is no mechanism to identify a SPI controller and properly connect it to the corresponding bus in the board database. As such, only a single SPI host controller is allowed in SMM. The SMM version of the SPI bus driver connects to the first and only SPI host controller identified by gEfiSpiSmmHcProtocolGuid.

A separate SMM specific SPI configuration database is provided by the board layer. The SMM SPI bus driver connects to the SMM version of the SPI configuration database identified by gEfiSpiSmmConfigurationProtocolGuid. The SMM version of the SPI bus driver verifies that there is a single bus in the SMM version of the board database and connects this to the SPI controller.

The SPI protocol declarations are identical between SMM and DXE but SMM uses different GUIDs to identify the SPI protocols for SMM.

12.2 SMM SPI Protocols

EFI_LEGACY _SPI_SMM_FLASH_PROTOCOL GUID

```
// {Se3848d4-0db5-4fc0-9729-3f353d4f879f}
#define EFI_LEGACY_SPI_SMM_FLASH_PROTOCOL \
{ 0x5e3848d4, 0x0db5, 0x4fc0, { 0x97, 0x29, 0x3f, 0x35, \ 0x3d,
0x4f, 0x87, 0x9f }}
```

EFI_SPI_SMM_NOR_FLASH_PROTOCOL GUID

```
// {aab18f19-fe14-4666-8604-87ff6d662c9a}
#define EFI_SPI_SMM_NOR_FLASH_PROTOCOL \
{ 0xaab18f19, 0xfe14, 0x4666, { 0x86, 0x04, 0x87, 0xff, \ 0x6d,
0x66, 0x2c, 0x9a }}
```

SSM Flash Driver GUID

Use a pointer to gEfiSpiSmmNorFlashDriverGuid in the EFI_SPI_PERIPHERAL structure to connect a SPI NOR flash part to the SMM SPI flash driver.

EFI- SPI- SMM- CONFIGURATION PROTOCOL GUID

```
// {995c6eca-171b-45fd-a3aa-fd4c9c9def59}
#define EFI_SPI_SMM_CONFIGURATION_PROTOCOL \
{ 0x995c6eca, 0x171b, 0x45fd, { 0xa3, 0xaa, 0xfd, 0x4c, \ 0x9c,
0x9d, 0xef, 0x59 }}
```

EFI- SPI- SMM HC- PROTOCOL GUID

```
// {e9f02217-2093-4470-8a54-5c2cffe73ecb}
#define EFI_SPI_SMM_HC_PROTOCOL \
{ 0xe9f02217, 0x2093, 0x4470, { 0x8a, 0x54, 0x5c, 0x2c, \
0xff, 0xe7, 0x3e, 0xcb }}
```

EFI LEGACY- SPI- SMM- CONTROLLER- PROTOCOL GUID

```
// {62331b78-d8d0-4c8c-8ccb-d27dfe32db9b}
#define EFI_LEGACY_SPI_SMM_CONTROLLER_GUID \
{ 0x62331b78, 0xd8d0, 0x4c8c, { 0x8c, 0xcb, 0xd2, 0x7d, \ 0xfe,
0x32, 0xdb, 0x9b }}
```

Appendix A
Management Mode Backward Compatibility Types

In versions of the PI specification up to and including version 1.4, this volume described System Management Mode (SMM), and many of the types were named with this acronym as a part of their name. With later versions of the PI specification, these types and constants were renamed to follow the Management Mode (MM) nomenclature, to abstract the concepts from the x86 architecture System Management Mode.

In order to maintain continuity, this appendix details typedefs and #define statements that allow code developed with these earlier versions of the specification to compile unchanged.

```
typedef EFI_MM_ENTRY_POINT EFI_SMM_ENTRY_POINT;
typedef EFI_MM_ENTRY_CONTEXT EFI_SMM_ENTRY_CONTEXT;
typedef EFI_MM_STARTUP_THIS_AP EFI_SMM_STARTUP_THIS_AP;
#define EFI_SMM_SYSTEM_TABLE2_REVISION EFI_MM_SYSTEM_TABLE_REVISION
#define SMM_SMST_SIGNATURE MM_MMST_SIGNATURE
#define SMM_SPECIFICATION_MAJOR_REVISION
MM_SPECIFICATION_MAJOR_REVISION
#define SMM_SPECIFICATION_MINOR_REVISION
MM_SPECIFICATION_MINOR_REVISION
typedef EFI_MM_INSTALL_CONFIGURATION_TABLE
EFI_SMM_INSTALL_CONFIGURATION_TABLE2;
typedef EFI_MM_CPU_IO_PROTOCOL EFI_SMM_CPU_IO2_PROTOCOL;
typedef EFI_MM_REGISTER_PROTOCOL_NOTIFY
EFI_SMM_REGISTER_PROTOCOL_NOTIFY;
typedef EFI_MM_INTERRUPT_MANAGE EFI_SMM_INTERRUPT_MANAGE;
typedef EFI_MM_INTERRUPT_REGISTER EFI_SMM_INTERRUPT_REGISTER;
typedef EFI_MM_INTERRUPT_UNREGISTER EFI_SMM_INTERRUPT_UNREGISTER;
typedef EFI_MM_NOTIFY_FN EFI_SMM_NOTIFY_FN;
typedef EFI_MM_HANDLER_ENTRY_POINT EFI_SMM_HANDLER_ENTRY_POINT2;
typedef EFI_MM_STATUS_CODE_PROTOCOL EFI_SMM_STATUS_CODE_PROTOCOL;
#define EFI_SMM_STATUS_CODE_PROTOCOL_GUID
EFI_MM_STATUS_CODE_PROTOCOL_GUID
typedef EFI_MM_REPORT_STATUS_CODE EFI_SMM_REPORT_STATUS_CODE;
typedef EFI_MM_CPU_PROTOCOL EFI_SMM_CPU_PROTOCOL;
#define EFI_SMM_CPU_PROTOCOL_GUID EFI_MM_CPU_PROTOCOL_GUID
typedef EFI_MM_READ_SAVE_STATE EFI_SMM_READ_SAVE_STATE;
#define EFI_SMM_SAVE_STATE_REGISTER_GDTBASE
EFI_MM_SAVE_STATE_REGISTER_GDTBASE
#define EFI_SMM_SAVE_STATE_REGISTER_IDTBASE
EFI_MM_SAVE_STATE_REGISTER_IDTBASE
#define EFI_SMM_SAVE_STATE_REGISTER_LDTBASE
EFI_MM_SAVE_STATE_REGISTER_LDTBASE
#define EFI_SMM_SAVE_STATE_REGISTER_GDTLIMIT
EFI_MM_SAVE_STATE_REGISTER_GDTLIMIT
#define EFI_SMM_SAVE_STATE_REGISTER_IDTLIMIT
EFI_MM_SAVE_STATE_REGISTER_IDTLIMIT
#define EFI_SMM_SAVE_STATE_REGISTER_LDTLIMIT
EFI_MM_SAVE_STATE_REGISTER_LDTLIMIT
#define EFI_SMM_SAVE_STATE_REGISTER_LDTINFO
EFI_MM_SAVE_STATE_REGISTER_LDTINFO
#define EFI_SMM_SAVE_STATE_REGISTER_ES EFI_MM_SAVE_STATE_REGISTER_ES
#define EFI_SMM_SAVE_STATE_REGISTER_CS EFI_MM_SAVE_STATE_REGISTER_CS
#define EFI_SMM_SAVE_STATE_REGISTER_SS EFI_MM_SAVE_STATE_REGISTER_SS
#define EFI_SMM_SAVE_STATE_REGISTER_DS EFI_MM_SAVE_STATE_REGISTER_DS
#define EFI_SMM_SAVE_STATE_REGISTER_FS EFI_MM_SAVE_STATE_REGISTER_FS
#define EFI_SMM_SAVE_STATE_REGISTER_GS EFI_MM_SAVE_STATE_REGISTER_GS
#define EFI_SMM_SAVE_STATE_REGISTER_LDTR_SEL
EFI_MM_SAVE_STATE_REGISTER_LDTR_SEL
#define EFI_SMM_SAVE_STATE_REGISTER_TR_SEL
EFI_MM_SAVE_STATE_REGISTER_TR_SEL
#define EFI_SMM_SAVE_STATE_REGISTER_DR7 EFI_MM_SAVE_STATE_REGISTER_DR7
```

```
#define EFI_SMM_SAVE_STATE_REGISTER_DR6 EFI_MM_SAVE_STATE_REGISTER_DR6
#define EFI_SMM_SAVE_STATE_REGISTER_R8 EFI_MM_SAVE_STATE_REGISTER_R8
#define EFI_SMM_SAVE_STATE_REGISTER_R9 EFI_MM_SAVE_STATE_REGISTER_R9
#define EFI_SMM_SAVE_STATE_REGISTER_R10 EFI_MM_SAVE_STATE_REGISTER_R10
#define EFI_SMM_SAVE_STATE_REGISTER_R11 EFI_MM_SAVE_STATE_REGISTER_R11
#define EFI_SMM_SAVE_STATE_REGISTER_R12 EFI_MM_SAVE_STATE_REGISTER_R12
#define EFI_SMM_SAVE_STATE_REGISTER_R13 EFI_MM_SAVE_STATE_REGISTER_R13
#define EFI_SMM_SAVE_STATE_REGISTER_R14 EFI_MM_SAVE_STATE_REGISTER_R14
#define EFI_SMM_SAVE_STATE_REGISTER_R15 EFI_MM_SAVE_STATE_REGISTER_R15
#define EFI_SMM_SAVE_STATE_REGISTER_RAX EFI_MM_SAVE_STATE_REGISTER_RAX
#define EFI_SMM_SAVE_STATE_REGISTER_RBX EFI_MM_SAVE_STATE_REGISTER_RBX
#define EFI_SMM_SAVE_STATE_REGISTER_RCX EFI_MM_SAVE_STATE_REGISTER_RCX
#define EFI_SMM_SAVE_STATE_REGISTER_RDX EFI_MM_SAVE_STATE_REGISTER_RDX
#define EFI_SMM_SAVE_STATE_REGISTER_RSP EFI_MM_SAVE_STATE_REGISTER_RSP
#define EFI_SMM_SAVE_STATE_REGISTER_RBP EFI_MM_SAVE_STATE_REGISTER_RBP
#define EFI_SMM_SAVE_STATE_REGISTER_RSI EFI_MM_SAVE_STATE_REGISTER_RSI
#define EFI_SMM_SAVE_STATE_REGISTER_RDI EFI_MM_SAVE_STATE_REGISTER_RDI
#define EFI_SMM_SAVE_STATE_REGISTER_RIP EFI_MM_SAVE_STATE_REGISTER_RIP
#define EFI_SMM_SAVE_STATE_REGISTER_RFLAGS
EFI_MM_SAVE_STATE_REGISTER_RFLAGS
#define EFI_SMM_SAVE_STATE_REGISTER_CR0 EFI_MM_SAVE_STATE_REGISTER_CR0
#define EFI_SMM_SAVE_STATE_REGISTER_CR3 EFI_MM_SAVE_STATE_REGISTER_CR3
#define EFI_SMM_SAVE_STATE_REGISTER_CR4 EFI_MM_SAVE_STATE_REGISTER_CR4
#define EFI_SMM_SAVE_STATE_REGISTER_FCW EFI_MM_SAVE_STATE_REGISTER_FCW
#define EFI_SMM_SAVE_STATE_REGISTER_FSW EFI_MM_SAVE_STATE_REGISTER_FSW
#define EFI_SMM_SAVE_STATE_REGISTER_FTW EFI_MM_SAVE_STATE_REGISTER_FTW
#define EFI_SMM_SAVE_STATE_REGISTER_OPCODE
EFI_MM_SAVE_STATE_REGISTER_OPCODE
#define EFI_SMM_SAVE_STATE_REGISTER_FP_EIP
EFI_MM_SAVE_STATE_REGISTER_FP_EIP
#define EFI_SMM_SAVE_STATE_REGISTER_FP_CS
EFI_MM_SAVE_STATE_REGISTER_FP_CS
#define EFI_SMM_SAVE_STATE_REGISTER_DATAOFFSET
EFI_MM_SAVE_STATE_REGISTER_DATAOFFSET
#define EFI_SMM_SAVE_STATE_REGISTER_FP_DS
EFI_MM_SAVE_STATE_REGISTER_FP_DS
#define EFI_SMM_SAVE_STATE_REGISTER_MM0 EFI_MM_SAVE_STATE_REGISTER_MM0
#define EFI_SMM_SAVE_STATE_REGISTER_MM1 EFI_MM_SAVE_STATE_REGISTER_MM1
#define EFI_SMM_SAVE_STATE_REGISTER_MM2 EFI_MM_SAVE_STATE_REGISTER_MM2
#define EFI_SMM_SAVE_STATE_REGISTER_MM3 EFI_MM_SAVE_STATE_REGISTER_MM3
#define EFI_SMM_SAVE_STATE_REGISTER_MM4 EFI_MM_SAVE_STATE_REGISTER_MM4
#define EFI_SMM_SAVE_STATE_REGISTER_MM5 EFI_MM_SAVE_STATE_REGISTER_MM5
#define EFI_SMM_SAVE_STATE_REGISTER_MM6 EFI_MM_SAVE_STATE_REGISTER_MM6
#define EFI_SMM_SAVE_STATE_REGISTER_MM7 EFI_MM_SAVE_STATE_REGISTER_MM7
#define EFI_SMM_SAVE_STATE_REGISTER_XMM0
EFI_MM_SAVE_STATE_REGISTER_XMM0
#define EFI_SMM_SAVE_STATE_REGISTER_XMM1
EFI_MM_SAVE_STATE_REGISTER_XMM1
#define EFI_SMM_SAVE_STATE_REGISTER_XMM2
EFI_MM_SAVE_STATE_REGISTER_XMM2
#define EFI_SMM_SAVE_STATE_REGISTER_XMM3
```

```
EFI_MM_SAVE_STATE_REGISTER_XMM3
#define EFI_SMM_SAVE_STATE_REGISTER_XMM4
EFI_MM_SAVE_STATE_REGISTER_XMM4
#define EFI_SMM_SAVE_STATE_REGISTER_XMM5
EFI_MM_SAVE_STATE_REGISTER_XMM5
#define EFI_SMM_SAVE_STATE_REGISTER_XMM6
EFI_MM_SAVE_STATE_REGISTER_XMM6
#define EFI_SMM_SAVE_STATE_REGISTER_XMM7
EFI_MM_SAVE_STATE_REGISTER_XMM7
#define EFI_SMM_SAVE_STATE_REGISTER_XMM8
EFI_MM_SAVE_STATE_REGISTER_XMM8
#define EFI_SMM_SAVE_STATE_REGISTER_XMM9
EFI_MM_SAVE_STATE_REGISTER_XMM9
#define EFI_SMM_SAVE_STATE_REGISTER_XMM10
EFI_MM_SAVE_STATE_REGISTER_XMM10
#define EFI_SMM_SAVE_STATE_REGISTER_XMM11
EFI_MM_SAVE_STATE_REGISTER_XMM11
#define EFI_SMM_SAVE_STATE_REGISTER_XMM12
EFI_MM_SAVE_STATE_REGISTER_XMM12
#define EFI_SMM_SAVE_STATE_REGISTER_XMM13
EFI_MM_SAVE_STATE_REGISTER_XMM13
#define EFI_SMM_SAVE_STATE_REGISTER_XMM14
EFI_MM_SAVE_STATE_REGISTER_XMM14
#define EFI_SMM_SAVE_STATE_REGISTER_XMM15
EFI_MM_SAVE_STATE_REGISTER_XMM15
#define EFI_SMM_SAVE_STATE_REGISTER_IO EFI_MM_SAVE_STATE_REGISTER_IO
#define EFI_SMM_SAVE_STATE_REGISTER_LMA EFI_MM_SAVE_STATE_REGISTER_LMA
#define EFI_SMM_SAVE_STATE_REGISTER_PROCESSOR_ID
EFI_MM_SAVE_STATE_REGISTER_PROCESSOR_ID
#define EFI_SMM_SAVE_STATE_REGISTER EFI_MM_SAVE_STATE_REGISTER
#define EFI_SMM_SAVE_STATE_REGISTER_LMA_32BIT
EFI_MM_SAVE_STATE_REGISTER_LMA_32BIT
#define EFI_SMM_SAVE_STATE_REGISTER_LMA_64BIT
EFI_MM_SAVE_STATE_REGISTER_LMA_64BIT
#define EFI_SMM_WRITE_SAVE_STATE EFI_MM_WRITE_SAVE_STATE
#define EFI_SMM_SAVE_STATE_IO_INFO EFI_MM_SAVE_STATE_IO_INFO

typedef EFI_MM_CPU_IO_PROTOCOL EFI_SMM_CPU_IO2_PROTOCOL;
#define EFI_SMM_CPU_IO2_PROTOCOL_GUID EFI_MM_CPU_IO_PROTOCOL_GUID
typedef EFI_MM_IO_ACCESS EFI_SMM_IO_ACCESS2;
typedef EFI_MM_CPU_IO EFI_SMM_CPU_IO2;
typedef EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL
EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL;
#define EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID
EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL_GUID
typedef EFI_MM_READY_TO_LOCK_SMM_PROTOCOL
EFI_SMM_READY_TO_LOCK_SMM_PROTOCOL;
#define EFI_SMM_READY_TO_LOCK_PROTOCOL_GUID
EFI_MM_READY_TO_LOCK_PROTOCOL_GUID
typedef EFI_MM_END_OF_DXE_PROTOCOL EFI_SMM_END_OF_DXE_PROTOCOL;
#define EFI_SMM_END_OF_DXE_PROTOCOL_GUID
```

```
EFI_MM_END_OF_DXE_PROTOCOL_GUID
#define EFI_SMM_BASE2_PROTOCOL_GUID EFI_MM_BASE_PROTOCOL_GUID
typedef EFI_MM_GET_MMST_LOCATION EFI_SMM_GET_SMST_LOCATION2;
typedef EFI_MM_ACCESS_PROTOCOL EFI_SMM_ACCESS2_PROTOCOL;
#define EFI_SMM_ACCESS2_PROTOCOL_GUID EFI_MM_ACCESS_PROTOCOL_GUID
typedef EFI_MM_OPEN EFI_SMM_OPEN2;
typedef EFI_MM_CLOSE EFI_SMM_CLOSE2;
typedef EFI_MM_LOCK EFI_SMM_LOCK2;
typedef EFI_MM_CONTROL_PROTOCOL EFI_SMM_CONTROL2_PROTOCOL;
#define EFI_SMM_CONTROL2_PROTOCOL_GUID EFI_MM_CONTROL_PROTOCOL_GUID
typedef EFI_MM_PERIOD EFI_SMM_PERIOD;
typedef EFI_MM_ACTIVATE EFI_SMM_ACTIVATE2;
typedef EFI_MM_DEACTIVATE EFI_SMM_DEACTIVATE2;
#define EFI_SMM_CONFIGURATION_PROTOCOL_GUID
EFI_MM_CONFIGURATION_PROTOCOL_GUID
typedef EFI_MM_REGISTER_SMM_ENTRY EFI_SMM_REGISTER_SMM_ENTRY;
typedef EFI_MM_COMMUNICATION_PROTOCOL EFI_SMM_COMMUNICATION_PROTOCOL;
#define EFI_SMM_COMMUNICATION_PROTOCOL_GUID
EFI_MM_COMMUNICATION_PROTOCOL_GUID
typedef EFI_MM_COMMUNICATE EFI_SMM_COMMUNICATE2;
typedef EFI_MM_COMMUNICATE_HEADER EFI_SMM_COMMUNICATE_HEADER;
typedef EFI_MM_SW_DISPATCH_PROTOCOL EFI_SMM_SW_DISPATCH2_PROTOCOL;
#define EFI_SMM_SW_DISPATCH2_PROTOCOL_GUID
EFI_MM_SW_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_SW_REGISTER EFI_SMM_SW_REGISTER2;
typedef EFI_MM_SW_UNREGISTER EFI_SMM_SW_UNREGISTER2;
typedef EFI_MM_SX_DISPATCH_PROTOCOL EFI_SMM_SX_DISPATCH2_PROTOCOL;
typedef EFI_MM_SX_DISPATCH_PROTOCOL_GUID
EFI_SMM_SX_DISPATCH2_PROTOCOL_GUID;
typedef EFI_MM_SX_REGISTER EFI_SMM_SX_REGISTER2;
typedef EFI_MM_SX_REGISTER_CONTEXT EFI_SMM_SX_REGISTER_CONTEXT;
typedef EFI_MM_SX_UNREGISTER EFI_SMM_SX_UNREGISTER2;
typedef EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL
EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL;
#define EFI_SMM_PERIODIC_TIMER_DISPATCH2_PROTOCOL_GUID
EFI_MM_PERIODIC_TIMER_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_PERIODIC_TIMER_REGISTER
EFI_SMM_PERIODIC_TIMER_REGISTER2;
typedef EFI_MM_PERIODIC_TIMER_CONTEXT EFI_SMM_PERIODIC_TIMER_CONTEXT;
typedef EFI_MM_PERIODIC_TIMER_UNREGISTER
EFI_SMM_PERIODIC_TIMER_UNREGISTER2;
typedef EFI_MM_PERIODIC_TIMER_INTERVAL
EFI_SMM_PERIODIC_TIMER_INTERVAL2;
typedef EFI_MM_USB_DISPATCH_PROTOCOL EFI_SMM_USB_DISPATCH2_PROTOCOL;
#define EFI_SMM_USB_DISPATCH2_PROTOCOL_GUID
EFI_MM_USB_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_USB_REGISTER EFI_SMM_USB_REGISTER2;
typedef EFI_MM_USB_REGISTER_CONTEXT EFI_SMM_USB_REGISTER_CONTEXT;
typedef EFI_USB_MMI_TYPE EFI_USB_SMI_TYPE;
typedef EFI_MM_GPI_DISPATCH_PROTOCOL EFI_SMM_GPI_DISPATCH2_PROTOCOL;
#define EFI_SMM_GPI_DISPATCH2_PROTOCOL_GUID
```

```
EFI_MM_GPI_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_GPI_REGISTER EFI_SMM_GPI_REGISTER2;
typedef EFI_MM_GPI_REGISTER_CONTEXT EFI_SMM_GPI_REGISTER_CONTEXT;
typedef EFI_MM_GPI_UNREGISTER EFI_SMM_GPI_UNREGISTER2;
typedef EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL
EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL;
#define EFI_SMM_STANDBY_BUTTON_DISPATCH2_PROTOCOL_GUID
EFI_MM_STANDBY_BUTTON_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_STANDBY_BUTTON_REGISTER
EFI_SMM_STANDBY_BUTTON_REGISTER2;
typedef EFI_MM_STANDBY_BUTTON_REGISTER_CONTEXT
EFI_SMM_STANDBY_BUTTON_REGISTER_CONTEXT;
typedef EFI_MM_STANDBY_BUTTON_UNREGISTER
EFI_SMM_STANDBY_BUTTON_UNREGISTER2;
typedef EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL
EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL;
#define EFI_SMM_POWER_BUTTON_DISPATCH2_PROTOCOL_GUID
EFI_MM_POWER_BUTTON_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_POWER_BUTTON_REGISTER EFI_SMM_POWER_BUTTON_REGISTER2;
typedef EFI_MM_POWER_BUTTON_REGISTER_CONTEXT
EFI_SMM_POWER_BUTTON_REGISTER_CONTEXT;
typedef EFI_MM_POWER_BUTTON_UNREGISTER
EFI_SMM_POWER_BUTTON_UNREGISTER2;
typedef EFI_MM_IO_TRAP_DISPATCH_PROTOCOL
EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL;
#define EFI_SMM_IO_TRAP_DISPATCH2_PROTOCOL_GUID
EFI_MM_IO_TRAP_DISPATCH_PROTOCOL_GUID
typedef EFI_MM_IO_TRAP_DISPATCH_REGISTER
EFI_SMM_IO_TRAP_DISPATCH2_REGISTER;
typedef EFI_MM_IO_TRAP_DISPATCH_TYPE EFI_SMM_IO_TRAP_DISPATCH_TYPE;
typedef EFI_MM_IO_TRAP_REGISTER_CONTEXT
EFI_SMM_IO_TRAP_REGISTER_CONTEXT;
typedef EFI_SMM_IO_TRAP_CONTEXT EFI_MM_IO_TRAP_CONTEXT;
typedef EFI_MM_IO_TRAP_DISPATCH_UNREGISTER
EFI_SMM_IO_TRAP_DISPATCH2_UNREGISTER;
typedef EFI_MM_IO_TRAP_DISPATCH_REGISTER
EFI_SMM_IO_TRAP_DISPATCH2_REGISTER;
#define EFI_FV_FILETYPE_SMM EFI_FV_FILETYPE_MM
#define EFI_FV_FILETYPE_COMBINED_SMM_DXE EFI_FV_FILETYPE_MM_DXE
#define EFI_FV_FILETYPE_SMM_CORE EFI_FV_FILETYPE_MM_CORE
#define EFI_SECTION_SMM_DEPEX EFI_SECTION_MM_DEPEX
typedef EFI_MM_DEPEX_SECTION EFI_SMM_DEPEX_SECTION2;
typedef EFI_MM_DEPEX_SECTION EFI_SMM_DEPEX_SECTION;
```

A.1 EFI_SMM_BASE2_PROTOCOL

This structure is deprecated. It is identical in content to **EFI_MM_BASE_PROTOCOL**.

```
typedef struct _EFI_SMM_BASE2_PROTOCOL {
  EFI_SMM_INSIDE_OUT2 InSmm;
  EFI_SMM_GET_SMST_LOCATION2 GetSmstLocation;
} EFI_SMM_BASE2_PROTOCOL;
EFI_SMM_RESERVED_SMRAM_REGION
```

This structure is deprecated. It is identical in content to **EFI_MM_RESERVED_MMRAM_REGION**.

```
typedef struct _EFI_SMM_RESERVED_SMRAM_REGION {
  EFI_PHYSICAL_ADDRESS SmramReservedStart;
  UINT64 SmramReservedSize;
} EFI_SMM_RESERVED_SMRAM_REGION;
```

EFI_SMM_CONFIGURATION_PROTOCOL

This structure is deprecated. It is identical in content to **EFI_MM_CONFIGURATION_PROTOCOL**.

```
typedef struct _EFI_SMM_CONFIGURATION_PROTOCOL {
  EFI_SMM_RESERVED_SMRAM_REGION *SmramReservedRegions;
  EFI_SMM_REGISTER_SMM_ENTRY RegisterSmmEntry;
} EFI_SMM_CONFIGURATION_PROTOCOL;
```

EFI_SMM_CAPABILITIES2

This type is deprecated. It is identical in content to **EFI_MM_CAPABILITIES**.

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CAPABILITIES2) (
  IN CONST EFI_SMM_ACCESS2_PROTOCOL *This,
  IN OUT UINTN *SmramMapSize,
  IN OUT EFI_SMRAM_DESCRIPTOR *SmramMap
  );
```

EFI_SMM_INSIDE_OUT2

This type is deprecated. It is identical in content to **EFI_MM_INSIDE_OUT**.

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_INSIDE_OUT2) (
  IN CONST EFI_SMM_BASE2_PROTOCOL *This,
  OUT BOOLEAN *InSmram
  );
```

EFI_SMM_SW_CONTEXT

This structure is deprecated. It is identical in content to **EFI_MM_SW_CONTEXT**;

```
typedef struct {
  UINTN SwSmiCpuIndex;
  UINT8 CommandPort;
  UINT8 DataPort;
} EFI_SMM_SW_CONTEXT;
```

EFI_SMM_SW_REGISTER_CONTEXT

This structure is deprecated. It is identical in content to **EFI_MM_SW_REGISTER_CONTEXT**.

```
typedef struct {
  UINTN SwSmiInputValue;
} EFI_SMM_SW_REGISTER_CONTEXT;
```

EFI_SMM_PERIODIC_TIMER_REGISTER_CONTEXT

This structure is deprecated. It is identical in content to **EFI_MM_PERIODIC_TIMER_REGISTER_CONTEXT**.

```
typedef struct {
  UINT64 Period;
  UINT64 SmiTickInterval;
} EFI_SMM_PERIODIC_TIMER_REGISTER_CONTEXT;
```

EFI_SMM_SAVE_STATE_IO_WIDTH

This type is deprecated. It is identical in content to **EFI_MM_SAVE_STATE_IO_WIDTH**.

```
typedef enum {
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT8 = 0,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT16 = 1,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT32 = 2,
  EFI_SMM_SAVE_STATE_IO_WIDTH_UINT64 = 3
} EFI_SMM_SAVE_STATE_IO_WIDTH
```

EFI_SMM_SAVE_STATE_IO_TYPE

This type is deprecated. It is identical in content to **EFI_MM_SAVE_STATE_IO_TYPE**.

```
typedef enum {
  EFI_SMM_SAVE_STATE_IO_TYPE_INPUT = 1,
  EFI_SMM_SAVE_STATE_IO_TYPE_OUTPUT = 2,
  EFI_SMM_SAVE_STATE_IO_TYPE_STRING = 4,
  EFI_SMM_SAVE_STATE_IO_TYPE_REP_PREFIX = 8
} EFI_SMM_SAVE_STATE_IO_TYPE
```

EFI_SMM_IO_WIDTH

This type is deprecated. It is identical in content to **EFI_MM_IO_WIDTH**.

```
typedef enum {
  SMM_IO_UINT8 = 0,
  SMM_IO_UINT16 = 1,
  SMM_IO_UINT32 = 2,
  SMM_IO_UINT64 = 3
} EFI_SMM_IO_WIDTH;
EFI_SMM_SYSTEM_TABLE2
```

This structure must match the members of **EFI_MM_SYSTEM_TABLE** up to and including *MmiHandlerUnregister*.

```
typedef struct _EFI_SMM_SYSTEM_TABLE2 { EFI_TABLE_HEADERHdr;

CHAR16*SmmFirmwareVendor;
UINT32SmmFirmwareRevision;

EFI_SMM_INSTALL_CONFIGURATION_TABLE2 SmmInstallConfigurationTable;

  EFI_SMM_CPU_IO_PROTOCOL              SmmIo;

//
    // Runtime memory service
```

```
//
EFI_ALLOCATE_POOLSmmAllocatePool;
EFI_FREE_POOLSmmFreePool;
EFI_ALLOCATE_PAGESSmmAllocatePages;
EFI_FREE_PAGESSmmFreePages;

//
// MP service
//
EFI_SMM_STARTUP_THIS_APSmmStartupThisAp;

//
// CPU information records
//
UINTNCurrentlyExecutingCpu;
UINTNNumberOfCpus;
  UINTN*CpuSaveStateSize;
  VOID**CpuSaveState;

//
// Extensibility table
//
UINTNNumberOfTableEntries;
EFI_CONFIGURATION_TABLE*SmmConfigurationTable;

  //
  // Protocol services
  //
EFI_INSTALL_PROTOCOL_INTERFACESmmInstallProtocolInterface;
EFI_UNINSTALL_PROTOCOL_INTERFACESmmUninstallProtocolInterface;
EFI_HANDLE_PROTOCOLSmmHandleProtocol; EFI_SMM_REGISTER_PROTOCOL_NOTIFY
SmmRegisterProtocolNotify;
  EFI_LOCATE_HANDLESmmLocateHandle;
  EFI_LOCATE_PROTOCOLSmmLocateProtocol;

  //
  // MMI management functions
  //
  EFI_SMM_INTERRUPT_MANAGESmiManage;
  EFI_SMM_INTERRUPT_REGISTERSmiHandlerRegister;
```

```
      EFI_SMM_INTERRUPT_UNREGISTERSmiHandlerUnRegister;
    } EFI_SMM_SYSTEM_TABLE2;
```

Platform Initialization (PI) Specification

Volume 5:
Standards

Version 1.7

January 2019

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

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

Revision History

Revision	Mantis ID / Description	Date
1.7	• 1848 PEI Core PEIM Migration Support Change • 1856 SecCore/PeiCore BFV Requirement Change • 1860 MM MP Protocol issues • 1885 Add extended data for EFI_SW_DXE_BS_EC_BOOT_OPTION_LOAD_ERROR • 1889 New Status Codes • 1891 PEI delayed dispatch • 1892 new pi spec revision • 1955 set (*Attributes) when EFI_PEI_GET_VARIABLE2 returns EFI_BUFFER_TOO_SMALL and Attributes is non-NULL • 1967 New architectural PPI for PI PEI Core FV Location • 1969 Incompatible Pci Ignore Option Rom	January 2019
1.6 Errata A	• 1828 Add decorator 'OPTIONAL' for Attributes parameter of EFI_PEI_GET_VARIABLE2. • 1849 Issues in PI Spec Vol. 5 Ch. 18 • 1857 Specifies how notifications are passed from SEC to PEI. • 1884 Variadic API issue in S3 save API • 1907 Clarification of the EFI_MM_COMMUNICATION_PROTOCOL • 1940 Clarify EFI_MM_CONFIGURATION_PROTOCOL_GUID	December 2018
1.6	• 1567 Layered SPI bus • 1648 PI Binding for RISC-V • 1746 Add an FV Extended Header entry that contains the used size of the FV • 1763 MM Handler state notification protocol • 1764 Add additional alignment • 1768 Update the PI Spec to 1.6 • 1777 Update Revision History • 1778 Update front matter	April 2017
1.5 Errata A	• 1587 pre permanent memory page allocation • 1665 Incorrect status code for an AP calling EFI_MP_SERVICES_PROTOCOL.SwitchBSP() • 1734 Outdated EFI spec reference in Save State Write • 1735 Several copy & paste errors in Save State Write • 1747 Clarify that MM_ACCESS_PROTOCOL should cover all MMRAM region used by the platform	April 2017

Revision	Mantis ID / Description	Date
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 • 1593 coalesce language enhancements • 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 • 1607 Update MM guid def'n to match edkII impl • 1626 Add new Status Code for BDS Attempting UEFI BootOrder entries • 1628 Minor feedback for PI 1.5 Vol 4 SMM Draft • 1666 Graphics Device Info Hob	4/26/16
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

Revision	Mantis ID / Description	Date
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.	2/20/15
1.3 Errata A	• 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
1.3	• 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

Revision	Mantis ID / Description	Date
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 • 937SMM report protocol notify issue errata • 951 Root Handler Processing by SmiManage • 958 • 969Vol 1 errata: TE Header parameters	10/26/12
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
1.2.1	• 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

Revision	Mantis ID / Description	Date
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 • 737 Remove SMM Communication ACPI Table definition . • 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 • 753 SIO PEI and UEFI-Driver Model Architecture • 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

Revision	Mantis ID / Description	Date
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 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	• 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

Revision	Mantis ID / Description	Date
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	• 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

Revision	Mantis ID / Description	Date
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
1.1 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

Revision	Mantis ID / Description	Date
1.1 Errata	• 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	Revises typographical errors and minor omissions--see Errata for details	04/25/08
1.1 correction	Restore (missing) MP protocol	03/12/08
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

Revision	Mantis ID / Description	Date
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
1.0	Initial public release.	8/21/06

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.

Table of Contents

Revision History iii
Table of Contents xiii
List of Tables xxi
List of Figures xxii

1 Introduction 1
1.1 Overview 1
1.2 Terms Used in this Document 1
1.3 Conventions Used in this Document 5
1.3.1 Data Structure Descriptions 6
1.3.2 Protocol Descriptions 6
1.3.3 Procedure Descriptions 6
1.3.4 Pseudo-Code Conventions 7
1.3.5 Typographic Conventions 7
1.4 Requirements 8

2 SMBus Host Controller Design Discussion 10
2.1 SMBus Host Controller Overview 10
2.2 Related Information 10
2.3 SMBus Host Controller Protocol Terms 10
2.4 SMBus Host Controller Protocol Overview 11

3 SMBus Host Controller Code Definitions 12
3.1 Introduction 12
3.2 SMBus Host Controller Protocol 12
EFI_SMBUS_HC_PROTOCOL 12
EFI_SMBUS_HC_PROTOCOL.Execute() 14
EFI_SMBUS_HC_PROTOCOL.ArpDevice() 16
EFI_SMBUS_HC_PROTOCOL.GetArpMap() 18
EFI_SMBUS_HC_PROTOCOL.Notify() 19

4 SMBus Design Discussion 21
4.1 Introduction 21
4.2 Target Audience 21
4.3 Related Information 21
4.4 PEI SMBus PPI Overview 21

5 SMBus PPI Code Definitions 23
5.1 Introduction 23
5.2 PEI SMBus PPI 24
EFI_PEI_SMBUS2_PPI 24
EFI_PEI_SMBUS2_PPI.Execute() 26
EFI_PEI_SMBUS2_PPI.ArpDevice() 29
EFI_PEI_SMBUS2_PPI.GetArpMap() 32

EFI_PEI_SMBUS2_PPI.Notify() 34

6 SMBIOS Protocol 36

EFI_SMBIOS_PROTOCOL 36

EFI_SMBIOS_PROTOCOL.Add() 38

EFI_SMBIOS_PROTOCOL.UpdateString() 41

EFI_SMBIOS_PROTOCOL.Remove() 42

EFI_SMBIOS_PROTOCOL.GetNext() 43

7 IDE Controller 45

7.1 IDE Controller Overview 45

7.2 Design Discussion 45

7.2.1 IDE Controller Initialization Protocol Overview 45

7.2.2 IDE Controller Initialization Protocol References 46

7.2.3 Background 47

7.2.4 Simplifying the Design of IDE Drivers 48

7.2.5 Configuring Devices on the IDE Bus 48

7.2.6 Sample Implementation for a Simple PCI IDE Controller 50

7.3 Code Definitions 51

EFI_IDE_CONTROLLER_INIT_PROTOCOL 51

EFI_IDE_CONTROLLER_INIT_PROTOCOL 52

EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo() 54

EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase() 56

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData() 59

EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode() 66

EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode() 70

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming() 72

7.3.1 IDE Disk Information Protocol 73

EFI_DISK_INFO_PROTOCOL 73

EFI_DISK_INFO_PROTOCOL.Interface 75

EFI_DISK_INFO_PROTOCOL.Inquiry() 77

EFI_DISK_INFO_PROTOCOL.Identify() 78

EFI_DISK_INFO_PROTOCOL.SenseData() 79

EFI_DISK_INFO_PROTOCOL.WhichIde() 80

8 S3 Resume 81

8.1 S3 Overview 81

8.2 Goals 81

8.3 Requirements 81

8.4 Assumptions 81

8.4.1 Multiple Phases of Platform Initialization 81

8.4.2 Process of Platform Initialization 82

8.5 Restoring the Platform 82

8.5.1 Phases in the S3 Resume Boot Path 83

8.6 PEI Boot Script Executer PPI 86

EFI_PEI_S3_RESUME2_PPI 87

EFI_PEI_S3_RESUME_PPI. S3RestoreConfig() 88

8.7 S3 Save State Protocol 89

EFI_S3_SAVE_STATE_PROTOCOL 89
8.7.1 Save State Write 90
EFI_S3_SAVE_STATE_PROTOCOL.Write() 91
EFI_BOOT_SCRIPT_IO_WRITE_OPCODE 93
EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE 95
EFI_BOOT_SCRIPT_IO_POLL_OPCODE 97
EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE 99
EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE 101
EFI_BOOT_SCRIPT_MEM_POLL_OPCODE 102
EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE 104
EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE 106
EFI_BOOT_SCRIPT_PCI_CONFIG_POLL_OPCODE 108
EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE 110
EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE 112
EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE 114
EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE 116
EFI_BOOT_SCRIPT_STALL_OPCODE 118
EFI_BOOT_SCRIPT_DISPATCH_OPCODE 119
EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE 120
EFI_BOOT_SCRIPT_INFORMATION_OPCODE 121
8.7.2 Save State Insert 121
EFI_S3_SAVE_STATE_PROTOCOL.Insert() 122
8.7.3 Save State Label 123
EFI_S3_SAVE_STATE_PROTOCOL.Label() 124
8.7.4 Save State Compare 125
EFI_S3_SAVE_STATE_PROTOCOL.Compare() 126
8.8 S3 SMM Save State Protocol 126
EFI_S3_SMM_SAVE_STATE_PROTOCOL 127

9 ACPI System Description Table Protocol 128

9.1 EFI_ACPI_SDT_PROTOCOL 128
EFI_ACPI_SDT_PROTOCOL.GetAcpiTable() 130
EFI_ACPI_SDT_PROTOCOL.RegisterNotify() 132
EFI_ACPI_SDT_PROTOCOL.Open() 134
EFI_ACPI_SDT_PROTOCOL.OpenSdt() 135
EFI_ACPI_SDT_PROTOCOL.Close() 136
EFI_ACPI_SDT_PROTOCOL.GetChild() 137
EFI_ACPI_SDT_PROTOCOL.GetOption() 138
EFI_ACPI_SDT_PROTOCOL.SetOption() 144
EFI_ACPI_SDT_PROTOCOL.FindPath() 145

10 PCI Host Bridge 146

10.1 PCI Host Bridge Overview 146
10.2 PCI Host Bridge Design Discussion 146
10.3 PCI Host Bridge Resource Allocation Protocol 147
10.3.1 PCI Host Bridge Resource Allocation Protocol Overview 147
10.3.2 Host Bus Controllers 147
10.3.3 Producing the PCI Host Bridge Resource Allocation Protocol 148

10.3.4 Required PCI Protocols 149
10.3.5 Relationship with EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL 149
10.4 Sample PCI Architectures 150
10.4.1 Sample PCI Architectures Overview 150
10.4.2 Desktop System with 1 PCI Root Bridge 150
10.4.3 Server System with 4 PCI Root Bridges 151
10.4.4 Server System with 2 PCI Segments 152
10.4.5 Server System with 2 PCI Host Buses 152
10.5 ISA Aliasing Considerations 153
10.6 Programming of Standard PCI Configuration Registers 154
10.7 Sample Implementation 155
10.7.1 PCI enumeration process 158
10.7.2 Sample Enumeration Implementation 160
10.8 PCI HostBridge Code Definitions 161
10.8.1 Introduction 161
10.8.2 PCI Host Bridge Resource Allocation Protocol 161
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL 161
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase() 167
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetNextRootBridge() 171
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes() 173
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration() 175
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers() 177
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources() 180
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetProposedResources() 182
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController() 185
10.9 End of PCI Enumeration Overview 188
10.9.1 End of PCI Enumeration Protocol 188
11 PCI Platform 189
11.1 Introduction 189
11.2 PCI Platform Overview 189
11.3 PCI Platform Support Related Information 190
11.3.1 Industry Specifications 190
11.3.2 PCI Specifications 190
11.4 PCI Platform Protocol 190
11.4.1 PCI Platform Protocol Overview 190
11.5 Incompatible PCI Device Support Protocol 191
11.5.1 Incompatible PCI Device Support Protocol Overview 191
11.5.2 Usage Model for the Incompatible PCI Device Support Protocol 191

11.6 PCI Code Definitions 192
11.6.1 PCI Platform Protocol 192
EFI_PCI_PLATFORM_PROTOCOL 192
EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify() 194
EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController() 196
EFI_PCI_PLATFORM_PROTOCOL.GetPlatformPolicy() 198
EFI_PCI_PLATFORM_PROTOCOL.GetPciRom() 200
11.6.2 PCI Override Protocol 201
EFI_PCI_OVERRIDE_PROTOCOL 201
11.6.3 Incompatible PCI Device Support Protocol 202
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL 202
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice() 204

12 Hot Plug PCI 207
12.1 HOT PLUG PCI Overview 207
12.2 Hot Plug PCI Initialization Protocol Introduction 207
12.3 Hot Plug PCI Initialization Protocol Related Information 207
12.4 Requirements 208
12.5 Sample Implementation for a Platform Containing PCI Hot Plug* Slots 209
12.6 PCI Hot Plug PCI Initialization Protocol 210
EFI_PCI_HOT_PLUG_INIT_PROTOCOL 210
EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList() 213
EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc() 215
EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding() 219
12.7 PCI Hot Plug Request Protocol 221
EFI_PCI_HOTPLUG_REQUEST_PROTOCOL.Notify() 223
12.8 Sample Implementation for a Platform Containing PCI Hot Plug* Slots 225

13 Super I/O Protocol 227
13.1 Super I/O Protocol 227
EFI_SIO_PROTOCOL 227
EFI_SIO_PROTOCOL.RegisterAccess() 229
EFI_SIO_PROTOCOL.GetResources() 231
EFI_SIO_PROTOCOL.SetResources() 233
EFI_SIO_PROTOCOL.PossibleResources() 234
EFI_SIO_PROTOCOL.Modify() 235

14 Super I/O and ISA Host Controller Interactions 237
14.1 Design Descriptions 237
14.1.1 Super I/O 238
14.1.2 ISA Bus 240
14.1.3 ISA Host Controller 241
14.1.4 Logical Devices 241
14.2 Code Definitions 242
14.2.1 EFI_SIO_PPI 242
14.2.2 EFI_ISA_HC_PPI 248
14.2.3 EFI_ISA_HC_PROTOCOL 250

14.2.4 EFI_ISA_HC_SERVICE_BINDING_PROTOCOL 253
14.2.5 EFI_SIO_CONTROL_PROTOCOL 253

15 CPU I/O Protocol 256
15.1 CPU I/O Protocol Terms 256
15.2 CPU I/O Protocol2 Description 256
15.2.1 EFI CPU I/O Overview 256
15.3 Code Definitions 257
15.3.1 CPU I/O Protocol 257
EFI_CPU_IO2_PROTOCOL 257
EFI_CPU_IO2_PROTOCOL.Mem.Read() and Mem.Write() 260
EFI_CPU_IO2_PROTOCOL.Io.Read() and Io.Write() 262

16 Legacy Region Protocol 264
16.1 Legacy Region Protocol 264
16.2 Code Definitions 264
16.2.1 Legacy Region Protocol 264
EFI_LEGACY_REGION2_PROTOCOL 264
EFI_LEGACY_REGION2_PROTOCOL.Decode() 266
EFI_LEGACY_REGION2_PROTOCOL.Lock() 267
EFI_LEGACY_REGION2_PROTOCOL.BootLock() 268
EFI_LEGACY_REGION2_PROTOCOL.UnLock() 269
EFI_LEGACY_REGION2_PROTOCOL.GetInfo() 270

17 I2C Protocol Stack 273
17.1 Design Discussion 273
17.1.1 I2C Bus Overview 273
17.1.2 2C Protocol Stack Overview 275
17.1.3 PCI Comparison 283
17.1.4 Hot Plug Support 285
17.2 DXE Code definitions 285
17.2.1 I2C Master Protocol 286
EFI_I2C_MASTER_PROTOCOL 286
EFI_I2C_MASTER_PROTOCOL.SetBusFrequency() 294
EFI_I2C_MASTER_PROTOCOL.Reset() 295
EFI_I2C_MASTER_PROTOCOL.StartRequest() 296
17.2.2 I2C Host Protocol 297
EFI_I2C_HOST_PROTOCOL 297
EFI_I2C_HOST_PROTOCOL.QueueRequest() 299
17.2.3 I2C I/O Protocol 301
EFI_I2C_IO_PROTOCOL 301
EFI_I2C_IO_PROTOCOL.QueueRequest() 304
17.2.4 I2C Bus Configuration Management Protocol 305
EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL 305
EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL.
EnableI2cBusConfiguration() 308
17.2.5 I2C Enumerate Protocol 309
EFI_I2C_ENUMERATE_PROTOCOL 309

EFI_I2C_ENUMERATE_PROTOCOL.Enumerate() 311
EFI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency() 312
17.3 PEI Code definitions 312
17.3.1 I2C Master PPI 313
EFI_PEI_I2C_MASTER 313
EFI_PEI_I2C_MASTER_PPI.SetBusFrequency() 314
EFI_PEI_I2C_MASTER_PPI.Reset() 315
EFI_PEI_I2C_MASTER_PPI.StartRequest() 316
17.3.2 I2C Host PPI 317
EFI_PEI_I2C_HOST 317
EFI_PEI_I2C_HOST.StartRequest() 319
17.3.3 I2C I/O PPI 320
EFI_PEI_I2C_IO 320
EFI_I2C_IO_PROTOCOL.GetDeviceInfo() 323
EFI_I2C_IO_PROTOCOL.GetDeviceInfoIdList() 324
EFI_PEI_I2C_IO.StartRequest() 325
17.3.4 I2C Bus Configuration Management PPI 326
EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT 326
EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT.
EnableI2cBusConfiguration() 328
EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT.I2cDeviceReset () .. 330
17.3.5 I2C Enumerate PPI 331
EFI_PEI_I2C_ENUMERATE 331
EFI_PEI_I2C_ENUMERATE_PROTOCOL.Enumerate() 332
EFI_PEI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency() 333

18 SPI Protocol Stack 334

18.1 Design Discussion 334
18.1.1 SPI Bus Overview 334
18.1.2 SPI Protocol Stack Overview 334
18.1.3 Application Layer 336
18.1.4 SPI Peripheral Layer 337
18.1.5 SPI I/O Interface 337
18.1.6 SPI Bus Layer 338
18.1.7 SPI Host Controller Layer 339
18.2 DXE Code Definitions 341
EFI_SPI_CONFIGURATION_PROTOCOL 341
EFI_SPI_CHIP_SELECT 342
EFI_SPI_PART 343
EFI_SPI_PERIPHERAL 344
EFI_SPI_CLOCK 345
EFI_SPI_BUS 346
EFI_SPI_NOR_FLASH_PROTOCOL 347
EFI_SPI_NOR_FLASH_PROTOCOL.GetFlashId() 349
EFI_SPI_NOR_FLASH_PROTOCOL.ReadData() 349
EFI_SPI_NOR_FLASH_PROTOCOL.LfReadData() 351
EFIEFI_SPI_NOR_FLASH_PROTOCOL.ReadStatus() 352

EFI_SPI_NOR_FLASH_PROTOCOL.WriteStatus() 353
EFI_SPI_NOR_FLASH_PROTOCOL.WriteData() 354
EFI_SPI_NOR_FLASH_PROTOCOL.Erase() 355
EFI_LEGACY_SPI_FLASH_PROTOCOL 356
EFI_LEGACY _SPI_FLASH_PROTOCOL.BiosBaseAddress() 356
EFI_LEGACY _SPI_FLASH_PROTOCOL.ClearSpiProtect() 357
EFI_LEGACY _SPI_FLASH_PROTOCOL.IsRangeProtected() 358
EFI_LEGACY_SPI_FLASH_PROTOCOL.ProtectNextRange() 358
EFI_LEGACY _SPI_FLASH_PROTOCOL.LockController() 359
EFI_SPI_IO_PROTOCOL 360
EFI_SPI_BUS_ TRANSACTION 361
EFI_SPI_IO_PROTOCOL.Transaction() 362
EFI_SPI_IO_PROTOCOL.UpdateSpiPeripheral() 365
EFI_SPI_HC_PROTOCOL 366
EFI_SPI_HC_PROTOCOL.ChipSelect() 367
EFI_SPI_HC_PROTOCOL.Clock() 368
EFI_SPI_HC_PROTOCOL.Transaction() 368
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL 369
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.EraseBlockOpcode() 370
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.WriteStatusPrefix() 371
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.BiosBaseAddress() 372
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.ClearSpiProtect() 372
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.IsRangeProtected() 373
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.ProtectNextRange() 374
EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.LockController() 375

Appendix A Error Codes 376

List of Tables

Table 1. Drivers Involved in Configuring IDE Devices 49
Table 2. Field descriptiond for EFI_IDE_CONTROLLER_ENUM_PHASE.......................... 57
Table 3. **EFI_ATAPI_IDENTIFY_DATA** Definition 61
Table 4. EFI_ATA_EXT_TRANSFER_PROTOCOL field descriptions 69
Table 5. AML terms and supported options 140
Table 6. Standard PCI Devices – Header Type 0... 154
Table 7. PCI-to-PCI Bridge – Header Type 1 155
Table 8. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage................................ 165
Table 9. ACPI 2.0 & 3.0 End Tag Usage 165
Table 10. I/O Resource Flag (Resource Type = 1) Usage... 166
Table 11. Memory Resource Flag (Resource Type = 0) Usage 166
Table 12. Enumeration Descriptions.. 169
Table 13. EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES field descriptions 174
Table 14. ACPI 2.0 & 3.0 Resource Descriptor Field Values for StartBusEnumeration()... 176
Table 15. ACPI 2.0 & 3.0 Resource Descriptor Field Values for SetBusNumbers()........... 178
Table 16. ACPI 2.0& 3.0 Resource Descriptor Field Values for SubmitResources().......... 181
Table 17. ACPI 2.0 & 3.0 GetProposedResources() Resource Descriptor Field Values ... 183
Table 18. EFI_RESOURCE_ALLOCATION_STATUS field descriptions 184
Table 19. EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE field descriptions. 187
Table 20. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage............................... 206
Table 21. ACPI 2.0 & 3.0 End Tag Usage 206
Table 22. Description of possible states for EFI_HPC_STATE 217
Table 23. EFI_HPC_PADDING_ATTRIBUTES field descriptions 221
Table 24. Functions in Legacy Region Protocol 264

List of Figures

Figure 1. PI Architechture S3 Resume Boot Path 83
Figure 2. PEI Phase in S3 Resume Boot Path 84
Figure 3. Configuration Save for PEI Phase 85
Figure 4. Host Bus Controllers 148
Figure 5. Producing the PCI Host Bridge Resource Allocation Protocol 149
Figure 6. Desktop System with 1 PCI Root Bridge 151
Figure 7. Server System with 4 PCI Root Bridges 151
Figure 8. Server System with 2 PCI Segments 152
Figure 9. Server System with 2 PCI Host Buses 153
Figure 10. Super I/O and ISA Host Controller Interactions 237
Figure 11. EFI CPU I/O2 Protocol 257
Figure 12. Simple 12C Bus 273
Figure 13. Multiple I2C Bus Frequencies 274
Figure 14. Limited address Space 274
Figure 15. I2C Protocol Stack 277
Figure 16. SPI Bus 334
Figure 17. SPI Layers 335

1 Introduction

1.1 Overview

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

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

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

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

This specification does the following:

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

1.2 Terms Used in this Document

16-bit PC Card

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

CardBay PC Card

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

CardBus bridge

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

CardBus PC Card

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

HB

Host bridge. See PCI host bridge.

HPB

Hot Plug Bus.

HPC

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

HPRT

Hot Plug Resource Table.

incompatible PCI device

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

JEITA

Japan Electronics and Information Technology Association.

legacy PHPC

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

MWI

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

PC Card

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

PC Card Standard

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

PCI bus

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

PCI bus driver

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

PCI configuration space

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

PCI controller

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

PCI device

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

PCI enumeration

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

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

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

PCI function

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

PCI host bridge

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

PCI root bridge

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

PCI segment

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

PEC

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

PCI-to-CardBus bridges

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

Pre-EFI Initialization.

PEIM

Pre-EFI Initialization Module.

greater than 256 PCI buses.

PERR

Parity Error.

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

PHPC

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

PPI

PEIM-to-PEIM Interface.

RB

Root bridge. See PCI root bridge.

resource padding

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

root HPC

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

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

root PCI bus

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

SERR

System error.

SHPC

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

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

SMBus

System Management Bus.

SMBus host controller

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

SMBus master device

Any device that initiates SMBus transactions and drives the clock.

SMBus PPI

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

SMBus slave device

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

UDID

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

1.3 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.3.1 Data Structure Descriptions

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

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

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

Structure Name: The formal name of the data structure.

Summary:A brief description of the data structure.

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

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

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

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

1.3.2 Protocol Descriptions

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

Protocol Name: The formal name of the protocol interface.

Summary:A brief description of the protocol interface.

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

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

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

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

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

1.3.3 Procedure Descriptions

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

ProcedureName(): The formal name of the procedure.

Summary:A brief description of the procedure.

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

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

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

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

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

1.3.4 Pseudo-Code Conventions

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

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

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

1.3.5 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain textThe 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.

BoldIn text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.

*Italic*In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

BOLD Monospace Computer code, example code segments, and all prototype code segments use a **BOLD Monospace** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

Bold Monospace In the online help version of this specification, words in a **Bold Monospace** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **Bold Monospace** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.

*Italic Monospace*In code or in text, words in *Italic Monospace* indicate placeholder names for variable information that must be supplied (i.e. arguments).

`Plain Monospace`In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

1.4 Requirements

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

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

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

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

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

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

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

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

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

2 SMBus Host Controller Design Discussion

2.1 SMBus Host Controller Overview

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

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

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

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

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

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

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

2.2 Related Information

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

Industry Specifications

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

2.3 SMBus Host Controller Protocol Terms

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

PEC

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

SMBus

System Management Bus.

SMBus host controller

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

SMBus master device

Any device that initiates SMBus transactions and drives the clock.

SMBus slave device

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

UDID

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

2.4 SMBus Host Controller Protocol Overview

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

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

See “SMBus Host Controller Protocol” on page 12 for the definition of this protocol.

3 SMBus Host Controller Code Definitions

3.1 Introduction

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

- **EFI_SMBUS_HC_PROTOCOL**

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

- **EFI_SMBUS_NOTIFY_FUNCTION**

3.2 SMBus Host Controller Protocol

EFI_SMBUS_HC_PROTOCOL

Summary

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

GUID

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

Protocol Interface Structure

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

Parameters

Execute

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

ArpDevice

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

GetArpMap

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

Notify

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

Description

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

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

EFI_SMBUS_HC_PROTOCOL.Execute()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

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

Command

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

Operation

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

PecCheck

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

Length

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_SMBUS_HC_PROTOCOL.ArpDevice()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

ArpAll

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

SmbusUdid

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

SlaveAddress

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

Description

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

Status Codes Returned

EFI_SUCCESS	The last data that was returned from the access matched the poll exit criteria.
EFI_CRC_ERROR	Checksum is not correct (PEC is incorrect).
EFI_TIMEOUT	*Timeout* expired before the operation was completed. *Timeout* is determined by the SMBus host controller device.

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

EFI_SMBUS_HC_PROTOCOL.GetArpMap()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

Length

Size of the buffer that contains the SMBus device map.

SmbusDeviceMap

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

Description

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

Status Codes Returned

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

EFI_SMBUS_HC_PROTOCOL.Notify()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_SMBUS_HC_PROTOCOL** instance.

SlaveAddress

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

Data

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

NotifyFunction

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

Description

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

Related Definitions

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

SlaveAddress

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

Data

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

Status Codes Returned

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

4 SMBus Design Discussion

4.1 Introduction

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

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

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

4.2 Target Audience

This document is intended for the following readers:

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

4.3 Related Information

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

Industry Specifications

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

4.4 PEI SMBus PPI Overview

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

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

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

See **"EFI_PEI_SMBUS2_PPI" on page 24** for the definition of **EFI_PEI_SMBUS2_PPI**. This PPI is produced by each of the SMBus host controllers in the system.

5 SMBus PPI Code Definitions

5.1 Introduction

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

- **EFI_PEI_SMBUS2_PPI**

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

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

5.2 PEI SMBus PPI

EFI_PEI_SMBUS2_PPI

Summary

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

GUID

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

PPI Interface Structure

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

Parameters

Execute

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

ArpDevice

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

GetArpMap

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

Notify

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

Identifier

Identifier which uniquely identifies this SMBus controller in a system.

Description

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

EFI_PEI_SMBUS2_PPI.Execute()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

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

Command

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

Operation

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

PecCheck

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

Length

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

Buffer

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

Description

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

Related Definitions

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

SmbusDeviceAddress

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

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


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

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

Status Codes Returned

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

EFI_PEI_SMBUS2_PPI.ArpDevice()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

ArpAll

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

SmbusUdid

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

SlaveAddress

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

Description

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

Related Definitions

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

VendorSpecificId

A unique number per device.

SubsystemDeviceId

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

SubsystemVendorId

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

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

Interface

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

DeviceId

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

VendorId

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

VendorRevision

UDID version number and a silicon revision identification.

DeviceCapabilities

Describes the device's capabilities.

Status Codes Returned

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

EFI_PEI_SMBUS2_PPI.GetArpMap()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

Length

Size of the buffer that contains the SMBus device map.

SmbusDeviceMap

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

Description

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

Related Definitions

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

SmbusDeviceAddress

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

SmbusDeviceUdid

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

Status Codes Returned

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

EFI_PEI_SMBUS2_PPI.Notify()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

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

Data

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

NotifyFunction

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

Description

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

Related Definitions

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

SmbusPpi

A pointer to the **EFI_PEI_SMBUS2_PPI** instance.

SlaveAddress

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

Data

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

Status Codes Returned

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

6 SMBIOS Protocol

EFI_SMBIOS_PROTOCOL

Summary

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

GUID

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

Protocol Interface Structure

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

Member Description

Add

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

UpdateString

Update a string in the SMBIOS record.

Remove

Remove an SMBIOS record.

GetNext

Discover all SMBIOS records.

MajorVersion

The major revision of the SMBIOS specification supported.

MinorVersion

The minor revision of the SMBIOS specification supported.

Description

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

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

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

EFI_SMBIOS_PROTOCOL.Add()

Summary

Add an SMBIOS record.

Prototype

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

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

ProducerHandle

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

SmbiosHandle

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

Record

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

Description

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

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

Related Definitions

```
typedef UINT8  EFI_SMBIOS_TYPE;
typedef UINT16 EFI_SMBIOS_HANDLE;

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

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

```
#define EFI_SMBIOS_TYPE_MEMORY_CHANNEL                       37
#define EFI_SMBIOS_TYPE_IPMI_DEVICE_INFORMATION              38
#define EFI_SMBIOS_TYPE_SYSTEM_POWER_SUPPLY                  39

#define EFI_SMBIOS_TYPE_ADDITIONAL_INFORMATION               40
#define EFI_SMBIOS_TYPE_ONBOARD_DEVICES_EXTENDED_INFORMATION 41
#define EFI_SMBIOS_TYPE_MANAGEMENT_CONTROLLER_HOST_INTERFACE 42

#define EFI_SMBIOS_TYPE_INACTIVE                            126
#define EFI_SMBIOS_TYPE_END_OF_TABLE                        127
#define EFI_SMBIOS_OEM_BEGIN                                128
#define EFI_SMBIOS_OEM_END                                  255

typedef UINT8 EFI_SMBIOS_STRING;
```

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

Status Codes Returned

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

EFI_SMBIOS_PROTOCOL.UpdateString()

Summary

Update the string associated with an existing SMBIOS record.

Prototype

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

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

SMBIOS Handle of structure that will have its string updated.

StringNumber

The non-zero string number of the string to update

String

Update the *StringNumber* string with *String*.

Description

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

Status Codes Returned

EFI_SUCCESS	*SmbiosHandle* had its *StringNumber String* updated.
EFI_INVALID_PARAMETER	*SmbiosHandle* does not exist.
EFI_UNSUPPORTED	String was not added because it is longer than the SMBIOS Table supports.
EFI_NOT_FOUND	The *StringNumber*.is not valid for this SMBIOS record.

EFI_SMBIOS_PROTOCOL.Remove()

Summary

Remove an SMBIOS record.

Prototype

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

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

The handle of the SMBIOS record to remove.

Description

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

Status Codes Returned

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

EFI_SMBIOS_PROTOCOL.GetNext()

Summary

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

Prototype

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

Parameters

This

The **EFI_SMBIOS_PROTOCOL** instance.

SmbiosHandle

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

Type

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

Record

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

ProducerHandle

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

Description

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

Status Codes Returned.

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

7 IDE Controller

7.1 IDE Controller Overview

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

This specification does the following:

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

7.2 Design Discussion

7.2.1 IDE Controller Initialization Protocol Overview

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

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

7.2.1.1 IDE Controller Terms

The following terms are used throughout this document.

AHCI

Advanced Host Controller Interface.

ATAPI

AT Attachment Packet Interface

enumeration group

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

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

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

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

IDE controller

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

PATA

Parallel ATA.

PATA controller

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

SATA

Serial ATA.

SATA controller

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

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

7.2.2 IDE Controller Initialization Protocol References

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

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

7.2.3 Background

7.2.3.1 IDE Requirements

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

This protocol is designed with the following requirements in mind:

1. The timing registers in a PATA IDE controller are vendor specific. (See *ATA Host Adapter Standards*, Working Draft Version 0f, for more information.) The programming of these registers needs to be abstracted from the driver entity.
2. The IDE Controller Initialization Protocol should also support a case where a specific channel is disabled and/or it should not be scanned. This protocol also needs a mechanism to address individual devices in various SATA and PATA configurations. This protocol needs to support the following:
    - "A variable number of channels per controller
    - "A variable number of devices per channel

7.2.3.1.1 PATA Controllers

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

7.2.3.1.2 SATA Contollers

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

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

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

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

7.2.3.1.3 Bus Neutral

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

7.2.3.2 PCI IDE controller

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

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

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

7.2.4 Simplifying the Design of IDE Drivers

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

- "The driver entity is the only driver that will send commands to the ATA devices. No device-specific drivers are needed for IDE devices because all the devices belong to the same class (i.e., storage) and the driver entity can have inherent knowledge of these commands. IDE bus equivalents of **EFI_PCI_IO_PROTOCOL** and **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** for accessing IDE devices are not required. It is possible to further simplify the design of the driver entity if it does not have to deal with the ATAPI devices. It can enumerate the ATA and ATAPI devices and install the **EFI_SCSI_PASSTHRU_PROTOCOL** on ATAPI device handles. Either way, IDE-bus-specific I/O protocols are not needed. See the *UEFI Specification* for the definitions of the EFI PCI I/O Protocol, PCI Root Bridge I/O Protocol, and the SCSI Pass Thru Protocol.
- "IDE devices are accessed and configured through a set of standard registers in the IDE controller. The ATA committee is standardizing the layout of these registers. (See *ATA Host Adapter Standards*, Working Draft Version 0f, for more information.) For Serial ATA (SATA) controllers, the *Serial ATA Advanced Host Controller Interface (AHCI) Specification* defines a standard register interface. Although the layout is dependent on the bus on which the controller is located, the layout for a particular bus is fixed. As a result, the driver entity can be required to know about the register layout for buses that it chooses to support. For example, for a PCI IDE controller, the IDE driver can access the base of the command block register for channel 0 using the following steps:

1. Check bit 0 of register 0x9 (Programming Interface Code) in the PCI configuration space of the controller to determine whether it is operating in compatibility mode or native PCI mode. For this example, we will assume that the controller is operating in native mode.
2. Read register 0x10 (Base Address Register [BAR] 0) of the controller. Clear bit 0 of the value that was read to get the command block base

7.2.5 Configuring Devices on the IDE Bus

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

Table 1. Drivers Involved in Configuring IDE Devices

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

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

7.2.6 Sample Implementation for a Simple PCI IDE Controller

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

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

1. The IDE controller driver as well as the driver entity follow the EFI Driver Model. They are loaded and both install (at least) one instance of the **EFI_DRIVER_BINDING_PROTOCOL** on their image handle. An ATA hard drive behind a PCI IDE controller is one of the boot devices.
2. The PCI bus driver enumerates the PCI bus, finds the PCI IDE controller, creates a handle for it, and installs an instance of **EFI_PCI_IO_PROTOCOL** and **EFI_DEVICE_PATH_PROTOCOL** on that handle.
3. The Boot Device Selection (BDS) phase searches for an appropriate driver to own the IDE controller device and finds the IDE controller driver. It then connects the IDE controller device and the IDE controller driver. The IDE controller driver opens the **EFI_PCI_IO_PROTOCOL BY_DRIVER**. It may perform some other preprogramming at this point.
4. BDS searches for a driver to own the IDE device and finds the driver entity. The driver entity's Supported() function checks for the presence of **EFI_IDE_CONTROLLER_INIT_PROTOCOL** on the parent of the IDE device (i.e., the IDE controller).
5. The EFI Boot Services function **ConnectController()** calls the **Start()** function of the driver entity, which starts the IDE bus enumeration. The following steps are performed by the **Start()** function.
    - The driver entity locates the **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. It opens the **EFI_IDE_CONTROLLER_INIT_PROTOCOL BY_DRIVER**. If it needs to open **EFI_PCI_IO_PROTOCOL**, it may open it by **GET_PROTOCOL**. The driver entity reads the *EnumAll* and *ChannelCount* fields in **EFI_IDE_CONTROLLER_INIT_PROTOCOL**. In this case, *EnumAll* is **FALSE**. The driver entity also obtains the channel number from **Start().RemainingDevicePath**.
    - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeChannelEnumeration*, *Channel*).
    - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo** (*This*, *Channel*, **Enabled*, **MaxDevices*) to find out the number of devices on this channel. If **Enabled* **= FALSE,** it exits with an error code. If the device number of the device to be connected is too large, it exits with an error code.
    - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeChannelReset*, *Channel*).
    - The driver entity resets the channel.
    - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterChannelReset*, *Channel*).
    - The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeBeforeDevicePresenceDetection*, *Channel*). The IDE controller driver may insert a predelay here or may ensure that various IDE bus signals are at desired levels.
    - The driver entity attempts to detect devices on the channel. Note than there can be no more than MaxDevices on the channel.

- The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterDevicePresenceDetection*, *Channel*).
- The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeResetMode*, *Channel*). The IDE controller sets up the controller with the default timings.

6. For all the devices on this channel:
   - The driver entity gathers **EFI_IDENTIFY_DATA** for the device and submits it to the IDE controller driver using **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()**. Submit **NULL** data for devices that do not exist.
   - The driver entity may call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()** to disqualify modes that it does not support.
7. For all the detected devices on this channel:
   - Call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()** to get the optimum mode settings. The IDE controller driver uses controller-specific algorithms and platform information to calculate the best modes.
   - The driver entity enables the appropriate modes by sending an ATA **SET_FEATURES** command to the device. It the device returns an error, it disqualifies that mode for that device and goes back to step 7. This time step 7 (first bullet) will not consider the failed mode. The implementation then returns here to step 7 (second bullet) with new (less optimum) modes.
8. For all the detected devices on this channel, call **EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()**to program the timings. Note that we reset the mode settings in step 5(last bullet), so the settings for nonexistent devices will remain at their default levels.
9. The driver entity calls **EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase** (*This*, *EfiIdeAfterChannelEnumeration*, *Channel*).
10. Install **EFI_BLOCK_IO_PROTOCOL** on that device handle.

7.3 Code Definitions

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

following protocol is defined in this section:

EFI_IDE_CONTROLLER_INIT_PROTOCOL

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

```
EFI_IDE_CONTROLLER_ENUM_PHASE
EFI_IDENTIFY_DATA
EFI_ATA_IDENTIFY_DATA
EFI_ATAPI_IDENTIFY_DATA
EFI_ATA_COLLECTIVE_MODE
EFI_ATA_MODE
EFI_ATA_EXTENDED_MODE
EFI_ATA_EXT_TRANSFER_PROTOCOL
```

EFI_IDE_CONTROLLER_INIT_PROTOCOL

Summary

Provides the basic interfaces to abstract an IDE controller.

GUID

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

Protocol Interface Structure

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

Parameters

GetChannelInfo

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

NotifyPhase

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

SubmitData

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

DisqualifyMode

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

CalculateMode

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

SetTiming

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

EnumAll

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

ChannelCount

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

Description

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

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

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.GetChannelInfo()

Summary

Returns the information about the specified IDE channel.

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Enabled

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

MaxDevices

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

Description

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

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

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

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

Status Codes Returned

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.NotifyPhase()

Summary

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

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Phase

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

Channel

Zero-based channel number.

Description

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

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

Related Definitions

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

Table 2. Field descriptiond for EFI_IDE_CONTROLLER_ENUM_PHASE

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

Status Codes Returned

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

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SubmitData()

Summary

Submits the device information to the IDE controller driver.

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

IdentifyData

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

Related Definitions

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

#define   EFI_ATAPI_DEVICE_IDENTIFY_DATA  0x8000
```

AtaData

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

AtapiData

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

Table 3. **EFI_ATAPI_IDENTIFY_DATA** Definition .

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

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

#pragma pack(1)
///
/// EFI_ATA_IDENTIFY_DATA is strictly complied with ATA/ATAPI-8
Spec
///
typedef struct {
  UINT16  config;                              ///< General
Configuration
  UINT16  obsolete_1;
  UINT16  specific_config;                     ///< Specific
Configuration
  UINT16  obsolete_3;
  UINT16  retired_4_5[2];
  UINT16  obsolete_6;
  UINT16  cfa_reserved_7_8[2];
  UINT16  retired_9;
  CHAR8   SerialNo[20];                        ///< word 10~19
  UINT16  retired_20_21[2];
  UINT16  obsolete_22;
  CHAR8   FirmwareVer[8];                      ///< word 23~26
  CHAR8   ModelName[40];                       ///< word 27~46
  UINT16  multi_sector_cmd_max_sct_cnt;
  UINT16  trusted_computing_support;
  UINT16  capabilities_49;
  UINT16  capabilities_50;
  UINT16  obsolete_51_52[2];
  UINT16  field_validity;
  UINT16  obsolete_54_58[5];
  UINT16  multi_sector_setting;
  UINT16  user_addressable_sectors_lo;
  UINT16  user_addressable_sectors_hi;
  UINT16  obsolete_62;
  UINT16  multi_word_dma_mode;
  UINT16  advanced_pio_modes;
  UINT16  min_multi_word_dma_cycle_time;
  UINT16  rec_multi_word_dma_cycle_time;
  UINT16  min_pio_cycle_time_without_flow_control;
  UINT16  min_pio_cycle_time_with_flow_control;
  UINT16  reserved_69_74[6];
  UINT16  queue_depth;
  UINT16  reserved_76_79[4];                  ///< reserved for
Serial ATA
  UINT16  major_version_no;
  UINT16  minor_version_no;
  UINT16  command_set_supported_82;           ///< word 82
  UINT16  command_set_supported_83;           ///< word 83
  UINT16  command_set_feature_extn;           ///< word 84
  UINT16  command_set_feature_enb_85;         ///< word 85
  UINT16  command_set_feature_enb_86;         ///< word 86
  UINT16  command_set_feature_default;        ///< word 87
  UINT16  ultra_dma_mode;                     ///< word 88
  UINT16  time_for_security_erase_unit;
  UINT16  time_for_enhanced_security_erase_unit;
  UINT16  advanced_power_management_level;
  UINT16  master_password_identifier;
  UINT16  hardware_configuration_test_result;
  UINT16  acoustic_management_value;
  UINT16  stream_minimum_request_size;
  UINT16  streaming_transfer_time_for_dma;
  UINT16  streaming_access_latency_for_dma_and_pio;
  UINT16  streaming_performance_granularity[2];///< word 98~99
  UINT16  maximum_lba_for_48bit_addressing[4]; ///< word 100~103
  UINT16  streaming_transfer_time_for_pio;
  UINT16  reserved_105;
  UINT16  phy_logic_sector_support;            ///< word 106
  UINT16  interseek_delay_for_iso7779;
  UINT16  world_wide_name[4];                  ///< word 108~111
  UINT16  reserved_for_128bit_wwn_112_115[4];
  UINT16  reserved_for_technical_report;
  UINT16  logic_sector_size_lo;                ///< word 117
  UINT16  logic_sector_size_hi;                ///< word 118
  UINT16  features_and_command_sets_supported_ext; ///< word 119
  UINT16  features_and_command_sets_enabled_ext;   ///< word 120
  UINT16  reserved_121_126[8];
  UINT16  obsolete_127;
  UINT16  security_status;                         ///< word 128
  UINT16  vendor_specific_129_159[31];
  UINT16  cfa_power_mode;                          ///< word 160
  UINT16  reserved_for_compactflash_161_175[15];
  CHAR8   media_serial_number[60];                 ///< word
176~205
  UINT16  sct_command_transport;                   ///< word 206
  UINT16  reserved_207_208[2];
  UINT16  alignment_logic_in_phy_blocks;           ///< word 209
  UINT16  write_read_verify_sector_count_mode3[2]; ///< word
210~211
  UINT16  verify_sector_count_mode2[2];
  UINT16  nv_cache_capabilities;
  UINT16  nv_cache_size_in_logical_block_lsw;       ///< word 215
  UINT16  nv_cache_size_in_logical_block_msw;       ///< word 216
  UINT16  nv_cache_read_speed;
  UINT16  nv_cache_write_speed;
  UINT16  nv_cache_options;                        ///< word 219
  UINT16  write_read_verify_mode;                  ///< word 220
  UINT16  reserved_221;
  UINT16  transport_major_revision_number;
  UINT16  transport_minor_revision_number;
  UINT16  reserved_224_233[10];
  UINT16  min_number_per_download_microcode_mode3; ///< word 234
  UINT16  max_number_per_download_microcode_mode3; ///< word 235
  UINT16  reserved_236_254[19];
  UINT16  integrity_word;
} EFI_ATA_IDENTIFY_DATA;
#pragma pack()

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

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

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

Description

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

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

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

Status Codes Returned

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.DisqualifyMode()

Summary

Disqualifies specific modes for an IDE device.

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the Channel.

BadModes

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

Description

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

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

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

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

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

Related Definitions

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

PioMode

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

SingleWordDmaMode

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

MultiWordDmaMode

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

UdmaMode

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

ExtModeCount

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

ExtMode

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

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

Valid

TRUE if *Mode* is valid.

Mode

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

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

TransferProtocol

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

Mode

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

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

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

Table 4. **EFI_ATA_EXT_TRANSFER_PROTOCOL** field descriptions

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

Status Codes Returned

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.CalculateMode()

Summary

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

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

SupportedModes

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

Description

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

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

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

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

The driver entity may collect timing information for various devices in any order. The driver entity is responsible for making sure that all the dependencies are satisfied; for example, the *SupportedModes* information for device A that was previously returned may become stale after a call to **DisqualifyMode()** for device B.

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

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

Status Codes Returned

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

EFI_IDE_CONTROLLER_INIT_PROTOCOL.SetTiming()

Summary

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

Prototype

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

Parameters

This

Pointer to the **EFI_IDE_CONTROLLER_INIT_PROTOCOL** instance.

Channel

Zero-based channel number.

Device

Zero-based device number on the *Channel*.

Modes

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

Description

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

Status Codes Returned

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

7.3.1 IDE Disk Information Protocol

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

EFI_DISK_INFO_PROTOCOL

Summary

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

GUID

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

Protocol Interface Structure

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

Parameters

Interface

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

Inquiry

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

Identify

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

SenseData

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

WhichIde

Specific controller.

Description

The **EFI_DISK_INFO_PROTOCOL** provides controller specific information.

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

EFI_DISK_INFO_PROTOCOL.Interface

Summary

GUID of the type of interfaces

Related Definitions

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

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

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

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

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

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

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

Description

The type of interface being described.

EFI_DISK_INFO_PROTOCOL.Inquiry()

Summary

Provides inquiry information for the controller type.

Prototype

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

Parameters

This

Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

InquiryData

Pointer to a buffer for the inquiry data.

InquiryDataSize

Pointer to the value for the inquiry data size.

Description

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

Status Codes Returned

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

EFI_DISK_INFO_PROTOCOL.Identify()

Summary

Provides identify information for the controller type.

Prototype

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

Parameters

This

Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

IdentifyData

Pointer to a buffer for the identify data.

IdentifyDataSize

Pointer to the value for the identify data size.

Description

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

Status Codes Returned

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

EFI_DISK_INFO_PROTOCOL.SenseData()

Summary

Provides sense data information for the controller type.

Prototype

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

Parameters

This

Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

SenseData

Pointer to the *SenseData*.

SenseDataSize

Size of *SenseData* in bytes.

SenseDataNumber

Pointer to the value for the sense data size.

Description

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

Status Codes Returned

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

EFI_DISK_INFO_PROTOCOL.WhichIde()

Summary

Provides IDE channel and device information for the interface

Prototype

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

Parameters

This

Pointer to the **EFI_DISK_INFO_PROTOCOL** instance.

IdeChannel

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

IdeDevice

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

Description

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

Status Codes Returned

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

8 S3 Resume

8.1 S3 Overview

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

This specification does the following:

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

8.2 Goals

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

Extensibility:

The PI S3 resume boot path should easily adapt to different platforms, such as Itanium®-based platforms those based on 32-bit Intel® architecture (IA-32), and x64 platforms by replacing only a few platform-specific modules.

High performance:

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

8.3 Requirements

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

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

8.4 Assumptions

8.4.1 Multiple Phases of Platform Initialization

The PI Architecture consists of multiple phases. For example:

- Pre-EFI Initialization (PEI)
- Driver Execution Environment (DXE)
- SMM (System Management Mode)

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

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

8.4.2 Process of Platform Initialization

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

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

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

8.5 Restoring the Platform

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

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

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

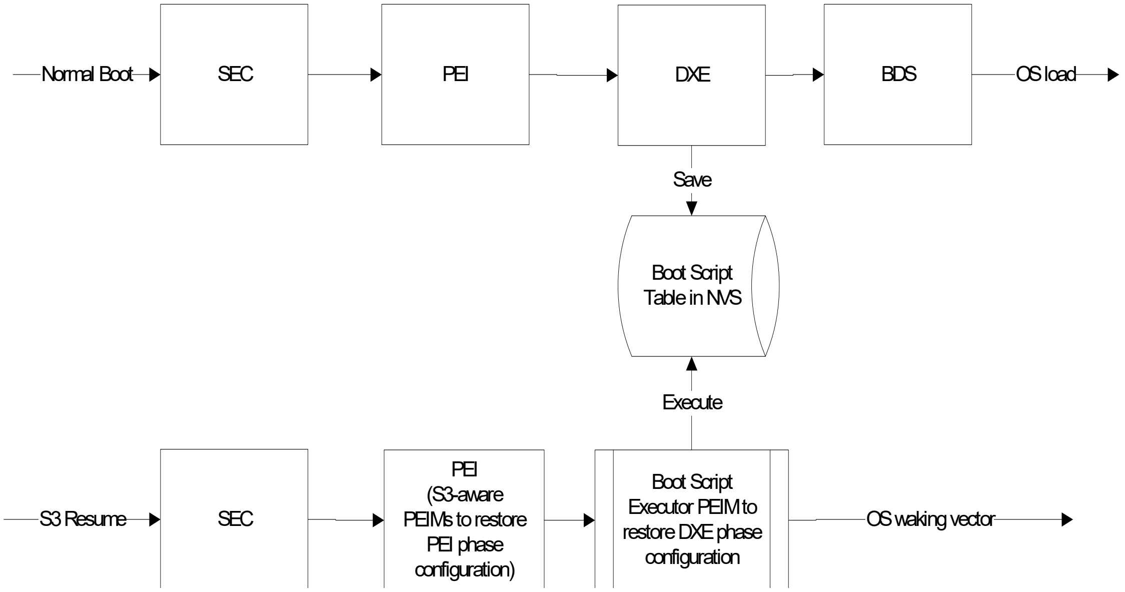

Normal Boot
SEC
PEI
DXE
BDS
OS load
Save
Boot Script Table in NVS
Execute
S3 Resume
SEC
PEI (S3-aware PEIMs to restore PEI phase configuration)
Boot Script Executor PEIM to restore DXE phase configuration
OS waking vector

Figure 1. PI Architechture S3 Resume Boot Path

8.5.1 Phases in the S3 Resume Boot Path

8.5.1.1 SEC and the S3 Resume Boot Path

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

8.5.1.2 PEI

8.5.1.2.1 PEI and the S3 Resume Boot Path

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

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

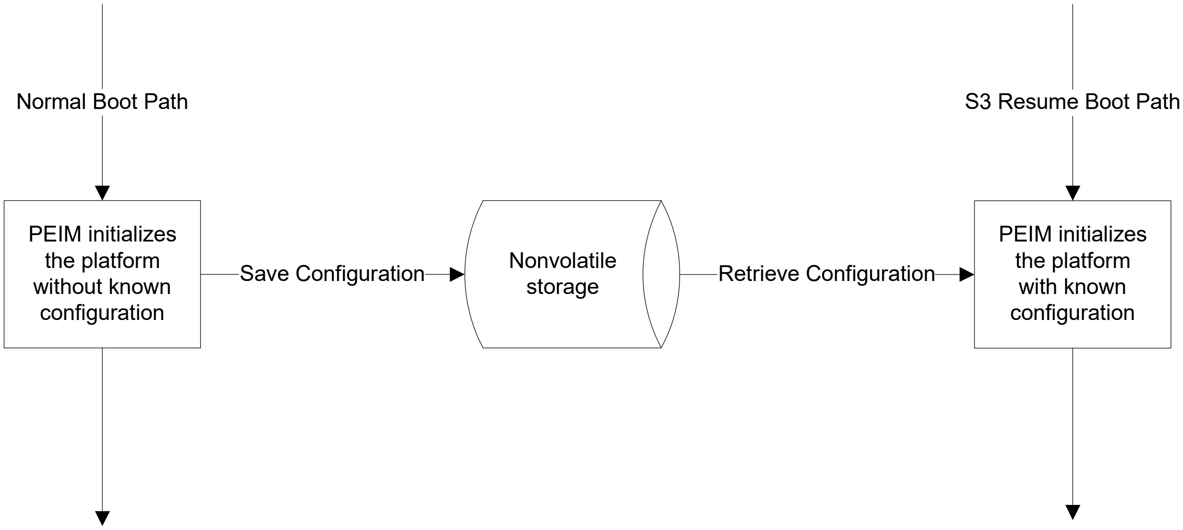

Normal Boot Path
S3 Resume Boot Path
PEIM initializes the platform without known configuration
Save Configuration
Nonvolatile storage
Retrieve Configuration
PEIM initializes the platform with known configuration

Figure 2. PEI Phase in S3 Resume Boot Path

8.5.1.2.2 Saving Configuration Data in PEI

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

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

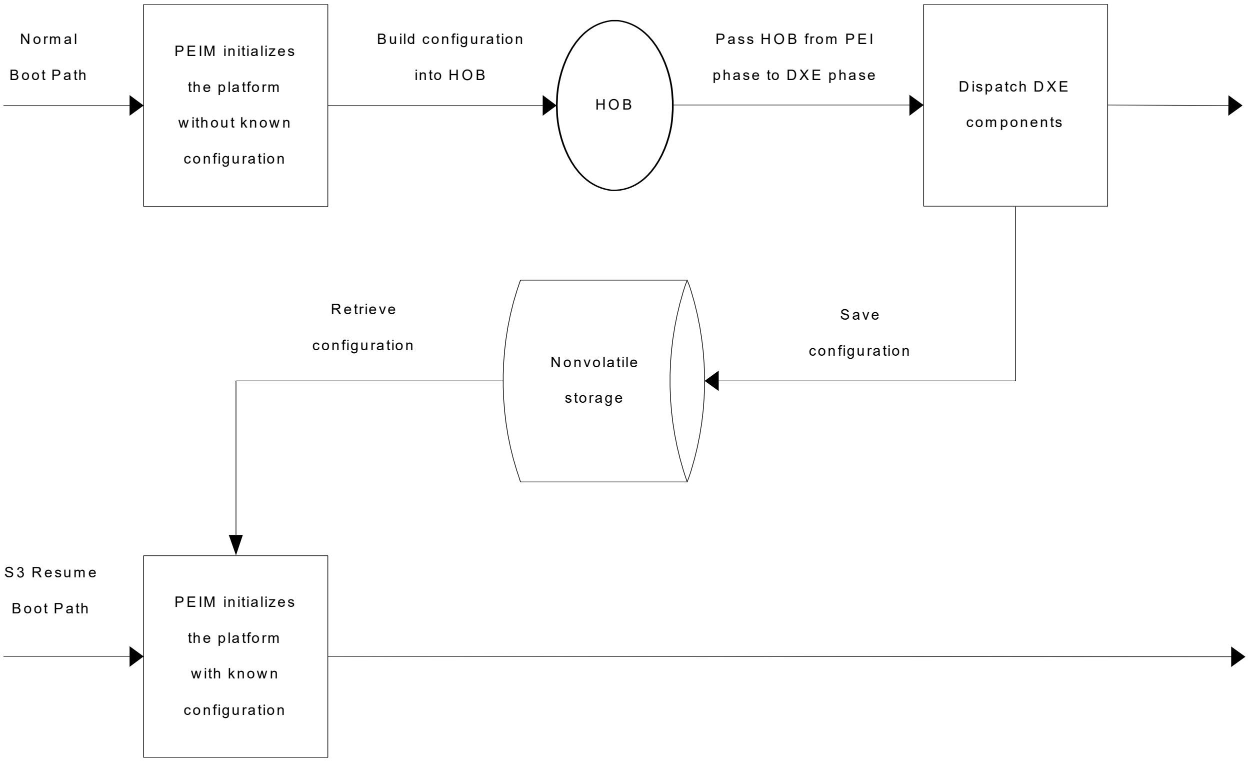


Figure 3. Configuration Save for PEI Phase

8.5.1.3 DXE

8.5.1.3.1 DXE and the S3 Resume Boot Path

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

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

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

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

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

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

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

8.5.1.3.2 S3 Resume PPI and DXE IPL PPI

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

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

8.5.1.4 SMM

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

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

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

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

8.6 PEI Boot Script Executer PPI

EFI_PEI_S3_RESUME2_PPI

Summary

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

GUID

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

PPI Interface Structure

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

Parameters

S3RestoreConfig2

Perform S3 resume operation.

Description

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

EFI_PEI_S3_RESUME_PPI. S3RestoreConfig()

Summary

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

Prototype

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

Parameters

This

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

Description

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

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

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

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

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

Status Codes Returned

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

8.7 S3 Save State Protocol

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

EFI_S3_SAVE_STATE_PROTOCOL

Summary

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

GUID

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

Protocol Interface Structure

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

Parameters

Write

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

Insert

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

Label

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

Compare

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

Description

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

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

8.7.1 Save State Write

EFI_S3_SAVE_STATE_PROTOCOL.Write()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

…

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

Description

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

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

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

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

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

Related Definitions

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

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

```
//*******************************************
// EFI_BOOT_SCRIPT_WIDTH
//*******************************************

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

Status Codes Returned

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

8.7.1.1 Opcodes for Write()

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

EFI_BOOT_SCRIPT_IO_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The base address of the I/O operations.

Count

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_IO_READ_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The base address of the I/O operations.

Data

A pointer to the data to be OR-ed.

DataMask

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_IO_POLL_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The base address of the I/O operations.

Data

The comparison value used for the polling exit criteria.

DataMask

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

Delay

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_MEM_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

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

Count

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_MEM_READ_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

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

Data

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

DataMask

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_MEM_POLL_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the EFI_S3_SAVE_STATE_PROTOCOL instance.

OpCode

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

Width

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

Address

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

Data

The comparison value used for the polling exit criteria.

DataMask

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

Delay

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG_WRITE_OPCODE

Summary

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

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Count

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG_READ_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Data

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

DataMask

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG_POLL_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Data

The comparison value used for the polling exit criteria.

DataMask

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

Delay

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG2_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Segment

The PCI segment number for Address.

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Count

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG2_READ_WRITE_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Width

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

Segment

The PCI segment number for *Address*.

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Data

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

DataMask

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE**. Value **EFI_BOOT_SCRIPT_PCI_CONFIG2_POLL_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

Width

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

Segment

The PCI segment number for *Address*.

Address

The address within the PCI configuration space. For address format details, see the "PCI Configuration Address" in the UEFI Specification, under EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Read() and EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Pci.Write().

Data

The comparison value used for the polling exit criteria.

DataMask

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

Delay

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_SMBUS_EXECUTE_OPCODE

Summary

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

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_BOOT_SCRIPT_WRITE) (
  IN CONST _EFI_S3_SAVE_STATE_PROTOCOL        *This,
  IN UINT16                                   OpCode,
  IN EFI_SMBUS_DEVICE_ADDRESS                 SlaveAddress,
  IN EFI_SMBUS_DEVICE_COMMAND                 Command,
  IN EFI_SMBUS_OPERATION                      Operation,
  IN BOOLEAN                                  PecCheck,
  IN UINTN                                    *Length,
  IN VOID                                     *Buffer
  );
```

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

SlaveAddress

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

Command

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

Operation

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

PecCheck

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

Length

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

Buffer

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

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_STALL_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

Duration

Duration in microseconds of the stall.

Description

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_DISPATCH_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

EntryPoint

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

Description

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

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

Must be set to **EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE**. Value **EFI_BOOT_SCRIPT_DISPATCH_2_OPCODE** is defined in "Related Definitions" in **EFI_S3_SAVE_STATE_PROTOCOL.Write()**.

EntryPoint

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

Context

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

Description

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

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

Status Codes Returned

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

EFI_BOOT_SCRIPT_INFORMATION_OPCODE

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

OpCode

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

InformationLenght

Length of the data in bytes.

Information

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

Description

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

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

Status Codes Returned

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

8.7.2 Save State Insert

EFI_S3_SAVE_STATE_PROTOCOL.Insert()

Summary

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

Prototype

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

Parameters

This

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

BeforeOrAfter

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

Position

On entry, specifies the position in the boot script table where the opcode will be inserted, either before or after, depending on *BeforeOrAfter*. On exit, if not NULL, specifies the position of the inserted opcode in the boot script table.

OpCode

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

…

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

Description

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

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

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

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

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

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

Related Definitions

```
typedef VOID *EFI_S3_BOOT_SCRIPT_POSITION;
```

Status Codes Returned

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

8.7.3 Save State Label

EFI_S3_SAVE_STATE_PROTOCOL.Label()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

BeforeOrAfter

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

CreateIfNotFound

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

Position

On entry, specifies the position in the boot script table where the label will be inserted, either before or after, depending on *BeforeOrAfter*. On exit, if not NULL, specifies the position of the inserted label in the boot script table.

Label

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

Description

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

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

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

Status Codes Returned

EFI_SUCCESS	The label already exists or was inserted.
EFI_NOT_FOUND	The label did not already exist and *CreateifNotFound* was **FALSE**.
EFI_INVALID_PARAMETER	The *Label* is *NULL* or points to an empty string.
EFI_INVALID_PARAMETER	The *Position* is not a valid position in the boot script table.
EFI_OUT_OF_RESOURCES	There is insufficient memory to store the boot script.

8.7.4 Save State Compare

EFI_S3_SAVE_STATE_PROTOCOL.Compare()

Summary

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

Prototype

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

Parameters

This

A pointer to the **EFI_S3_SAVE_STATE_PROTOCOL** instance.

Position1, Position2

The positions in the boot script table to compare.

RelativePosition

On return, points to the result of the comparison.

Description

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

Status Codes Returned

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

8.8 S3 SMM Save State Protocol

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

The S3 SMM Save State Protocol shares the interface definition with the S3 Save State Protocol but it has a different GUID. It is an SMM protocol. Having separate protocols for SMM and DXE makes it easier to accommodate the differences in the operating environment between SMM and DXE.

EFI_S3_SMM_SAVE_STATE_PROTOCOL

Summary

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

GUID

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

Protocol Interface Structure

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

Parameters

Write

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

Insert

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

Label

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

Compare

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

Description

The **EFI_S3_SMM_SAVE_STATE_PROTOCOL** provides the PI SMMboot script abstraction.

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

The order of replay is the order either of the S3 Save State Protocol or S3 SMM Save State Protocol **Write()** functions were called during the boot process.

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

9 ACPI System Description Table Protocol

9.1 EFI_ACPI_SDT_PROTOCOL

Summary

Provides services for creating ACPI system description tables.

GUID

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

Protocol Interface Structure

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

Related Definitions

```
#define UINT32 EFI_ACPI_TABLE_VERSION;

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

Members

AcpiVersion

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

GetTable

Enumerate the ACPI tables.

`RegisterNotify`

Register a notification when a table is installed.

`Open`

Create a handle from an ACPI opcode.

`OpenSdt`

Create a handle from an ACPI table.

`Close`

Close an ACPI handle.

`GetChild`

Cycle through the child objects of an ACPI handle.

`GetOption`

Return one of the optional pieces of the opcode.

`SetOption`

Change one of the optional pieces of the opcode.

`FindPath`

Given an ACPI path, return an ACPI handle.

EFI_ACPI_SDT_PROTOCOL.GetAcpiTable()

Summary

Returns a requested ACPI table.

Prototype

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

Parameters

Index

The zero-based index of the table to retrieve.

Table

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

Version

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

TableKey

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

Description

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

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

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

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

Related Definitions

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

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

Status Codes Returned

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

EFI_ACPI_SDT_PROTOCOL.RegisterNotify()

Summary

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

Prototype

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

Parameters

Register

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

Notification

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

Description

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

Status Codes Returned

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

Related Definitions

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

Table

A pointer to the ACPI table header.

Version

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

TableKey

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

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

EFI_ACPI_SDT_PROTOCOL.Open()

Summary

Create a handle from an ACPI opcode

Prototype

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

Parameters

Buffer

Points to the ACPI opcode.

Handle

Upon return, holds the handle.

Related Definitions

```
typedef VOID *EFI_ACPI_HANDLE;
```

Description

Creates a handle from a single ACPI opcode.

Status Code Values

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

EFI_ACPI_SDT_PROTOCOL.OpenSdt()

Summary

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

Prototype

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

Parameters

TableKey

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

Handle

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

Description

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

Related Definitions

```
typedef VOID *EFI_ACPI_HANDLE;
```

Status Codes Returned

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

EFI_ACPI_SDT_PROTOCOL.Close()

Summary

Close an ACPI handle.

Prototype

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

Parameters

Handle

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

Description

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

Status Code Values

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

EFI_ACPI_SDT_PROTOCOL.GetChild()

Summary

Return the child ACPI objects.

Prototype

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

Parameters

ParentHandle

Parent handle.

Handle

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

Description

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

Status Code Values

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

EFI_ACPI_SDT_PROTOCOL.GetOption()

Summary

Retrieve information about an ACPI object.

Prototype

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

Parameters

Handle

ACPI object handle.

Index

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

DataType

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

Data

Upon return, points to the pointer to the data.

DataSize

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

Related Definitions

```
typedef UINT32 EFI_ACPI_DATA_TYPE;

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

Description

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

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

Table 5. AML terms and supported options

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

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

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

Status Code Returns

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

EFI_ACPI_SDT_PROTOCOL.SetOption()

Summary

Change information about an ACPI object.

Prototype

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

Parameters

Handle

ACPI object handle.

Index

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

Data

Points to the data.

DataSize

The size of the *Data*.

Description

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

Status Code Returns

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

EFI_ACPI_SDT_PROTOCOL.FindPath()

Summary

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

Prototype

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

Parameters

HandleIn

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

AcpiPath

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

HandleOut

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

Description

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

Status Code Returns

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

10 PCI Host Bridge

10.1 PCI Host Bridge Overview

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

This specification does the following:

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

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** does not describe platform policies. The platform policies are described by the **EFI_PCI_PLATFORM_PROTOCOL**, which is desribed in section 11.6.1. Silicon-related policies are described by the **EFI_PCI_OVERRIDE_PROTOCOL**, which is described in section 11.6.2.

10.2 PCI Host Bridge Design Discussion

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

This chapter discusses the following:

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

10.3 PCI Host Bridge Resource Allocation Protocol

10.3.1 PCI Host Bridge Resource Allocation Protocol Overview

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

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

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

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

10.3.2 Host Bus Controllers

A platform can be viewed as the following:

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

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

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

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

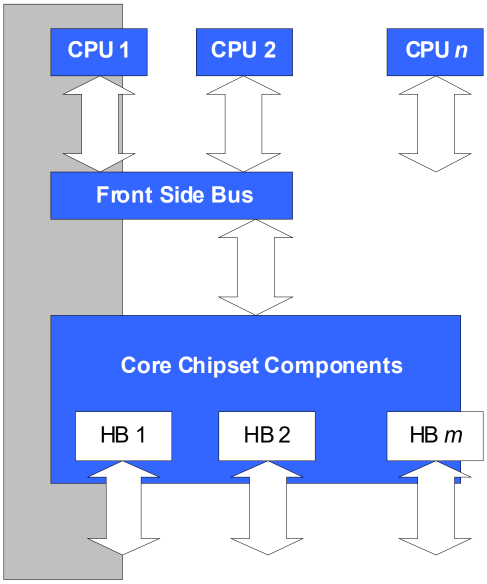


Figure 4. Host Bus Controllers

10.3.3 Producing the PCI Host Bridge Resource Allocation Protocol

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

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

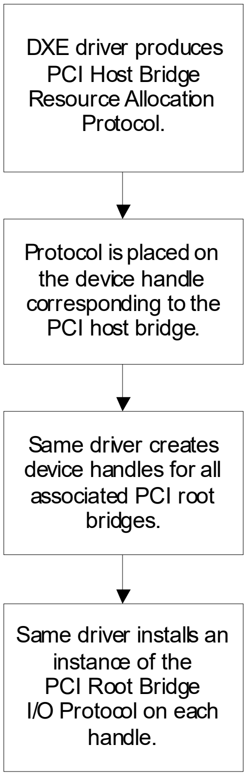

DXE driver produces PCI Host Bridge Resource Allocation Protocol.
Protocol is placed on the device handle corresponding to the PCI host bridge.
Same driver creates device handles for all associated PCI root bridges.
Same driver installs an instance of the PCI Root Bridge I/O Protocol on each handle.

Figure 5. Producing the PCI Host Bridge Resource Allocation Protocol

10.3.4 Required PCI Protocols

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

- **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_**
- **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**

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

10.3.5 Relationship with EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL

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

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

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

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

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

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

10.4 Sample PCI Architectures

10.4.1 Sample PCI Architectures Overview

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

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

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

10.4.2 Desktop System with 1 PCI Root Bridge

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

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

The firmware for this platform would produce the following:

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

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

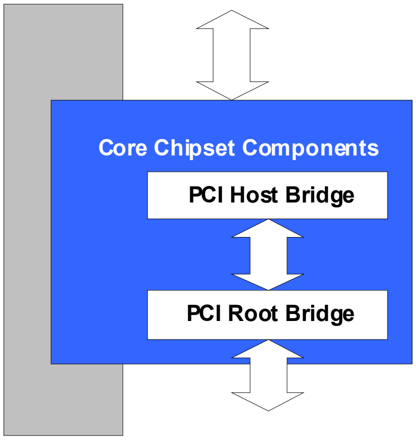


Figure 6. Desktop System with 1 PCI Root Bridge

10.4.3 Server System with 4 PCI Root Bridges

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

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

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

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

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

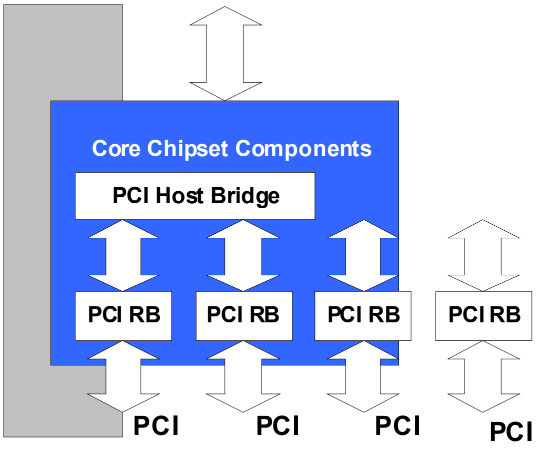


Figure 7. Server System with 4 PCI Root Bridges

10.4.4 Server System with 2 PCI Segments

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

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

The firmware for this platform would produce the following:

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

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

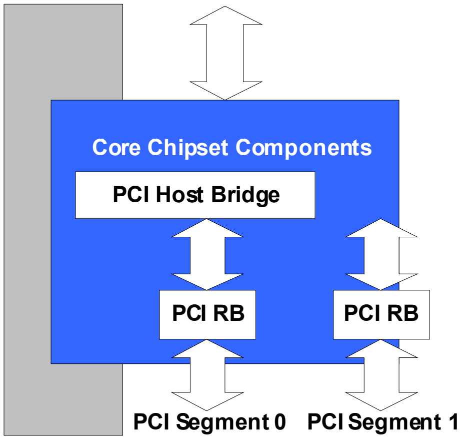


Figure 8. Server System with 2 PCI Segments

10.4.5 Server System with 2 PCI Host Buses

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

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

The firmware for this platform would produce the following:

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

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

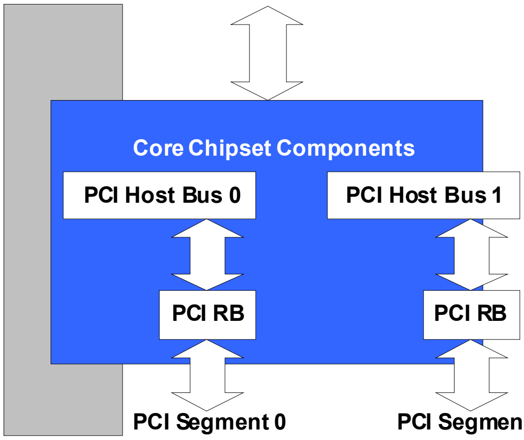


Figure 9. Server System with 2 PCI Host Buses

10.5 ISA Aliasing Considerations

The PCI host bridge driver will handle the ISA alias addresses based on the platform policy. The platform communicates the policy to the PCI host bridge driver using the **EFI_PCI_PLATFORM_PROTOCOL**. If the PCI host bridge driver cannot locate an instance of **EFI_PCI_PLATFORM_PROTOCOL**, it will not reserve the ISA alias addresses. The PCI bus driver is not aware of this policy and probes devices to gather resource requirements regardless of this policy. The **EFI_PCI_PLATFORM_PROTOCOL** is defined in section 11.6.1.

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

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

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

The PCI host bridge driver sets the _RNG bit to communicate the availability of the ISA alias range to the PCI bus driver. If the _RNG flag is set, the PCI bus enumerator is not allowed to allocate the ISA alias addresses to any PCI device. See Table 10 in the "Description" section of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** for the definition of the _RNG flag. In this case, a PCI device’s request to turn on aliasing will succeed because one or more PCI devices may be occupying aliased addresses. The _RNG flag is the only aspect of the protocol interface structure that is affected by ISA aliasing.

10.6 Programming of Standard PCI Configuration Registers

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

Table 6. Standard PCI Devices – Header Type 0

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

Table 7. PCI-to-PCI Bridge – Header Type 1

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

10.7 Sample Implementation

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

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

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

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

...Other initialization activities take place.

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

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

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

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

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

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

10.7.1 PCI enumeration process

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

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

10.7.1.1 Other initialization activities take place.

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

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

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

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

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

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

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

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

15. Notify the drivers that resource allocation is complete by using **EfiPciHostBridgeEndResourceAllocation**.

16. Notify the drivers that bus enumeration is complete by calling **EfiPciHostBridgeEndEnumeration**.

17. Deallocate any temporary buffers.

18. Install the **EFI_PCI_ENUMERATION_COMPLETE_GUID** protocol.

10.7.1.3 Sample PCI Device Set Up Implementation

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

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

10.7.2 Sample Enumeration Implementation

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

10.7.2.1 PCI Enumeration Phases

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

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

10.7.2.2 Additional locations to preprocess PCI devices

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

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

10.8 PCI HostBridge Code Definitions

10.8.1 Introduction

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

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

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

10.8.2 PCI Host Bridge Resource Allocation Protocol

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL

Summary

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

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

GUID

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

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL
{
  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE
                                         NotifyPhase;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_B
RIDGE
                                         GetNextRootBridge;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_ATTRIBUTES
                                         GetAllocAttributes;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUME
RATION
                                         StartBusEnumeration;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS
                                         SetBusNumbers;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCE
S
                                         SubmitResources;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RE
SOURCES
                                         GetProposedResources;

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONT
ROLLER
                                         PreprocessController;
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL;
```

Parameters

NotifyPhase

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

GetNextRootBridge

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

GetAllocAttributes

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

StartBusEnumeration

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

SetBusNumbers

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

SubmitResources

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

GetProposedResources

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

PreprocessController

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

Description

The **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** provides the basic resource allocation services to the PCI bus driver. There is one **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance for each PCI host bridge in a system. The following will typically have only one PCI host bridge:

- Embedded systems
- Desktops
- Workstations
- Most servers

High-end servers may have multiple PCI host bridges. A PCI bus driver that wishes to manage a PCI bus in a system will have to retrieve the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance that is associated with the PCI bus to be managed. A device handle for a PCI host bridge will not contain an

EFI_DEVICE_PATH_PROTOCOL instance because the PCI host bridge is a software abstraction and has no equivalent in the ACPI name space.

All applicable member functions use ACPI 2.0 or ACPI 3.0 resource descriptors to describe resources. Using ACPI resource descriptors does the following:

- Allows other types of resources to be described in the future because they are very generic in nature.
- Avoids multiple structure definitions for describing resources.
- Maintains compatibility with the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** definition.

Only the following two resource descriptor types from the *ACPI Specification* may be used to describe the current resources that are allocated to a PCI root bridge:

- QWORD Address Space Descriptor (*ACPI 3.0*)
- End Tag (*ACPI 3.0)*

The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors, followed by an End Tag. Table 8 and Table 9 below contain these two descriptor types. Table 10 and Table 11 define how resource-specific flags are used. See the *ACPI Specification* for details on the field values.

Table 8. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes, not including the first two fields.
0x03	0x01		Resource type: 0: Memory range 1: I/O range 2: Bus number range
0x04	0x01		General flags. Flags that are common to all resource types: **Bits[7:4]:** Reserved (must be 0) **Bit[3] _MAF:** Always returned as 1 while returning allocated requests to indicate that the specified max address is fixed. **Bit[2] _MIF:** Always returned as 1 while returning allocated requests to indicate that the specified min address is fixed. **Bit[1] _DEC:** Ignored. **Bit[0]:** Ignored.
0x05	0x01		Type-specific flags. Ignored.
0x06	0x08		Address Space Granularity. Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. Ignored for I/O and bus resource requests. Ignored during **GetProposedResources()**.
0x0E	0x08		Address Range Minimum. Set to the base of the allocated address range (bus, I/O, memory) during **GetProposedResources()**. Ignored during **SubmitResources()**.
0x16	0x08		Address Range Maximum. Used to indicate the BAR Index from 0 to 6, where 6 is the Option Rom BAR. Specially, (UINT64)-1 in this field means all the PCI BARs on the device except the Option Rom BAR.
0x1E	0x08		Address Translation Offset. Used to indicate alignment requirement during **SubmitResources()** and ignored during **GetProposedResources()**. This value must be 2^n-1. The address base must be a multiple of the granularity field. That is, if this field is 4KiB-1, the allocated address must be a multiple of 4 KiB. **Note:** The interpretation of this field for Platform Initialization is different from the *ACPI Specification* and PCI Root Bridge I/O Protocol.
0x26	0x08		Address Range Length. This field specifies the amount of resources that are requested or allocated in number of bytes.

Table 9. ACPI 2.0 & 3.0 End Tag Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag.
0x01	0x01	0x00	Checksum. Set to 0 to indicate that checksum is to be ignored.

Table 10. I/O Resource Flag (Resource Type = 1) Usage

Bits	Meaning
Bits[7:1]	Ignored.
Bit[0]	_RNG. Ignored during an allocation request. Setting this bit while returning allocated resources means that the I/O allocation must be limited to the ISA I/O ranges. In that case, the PCI bus driver must allocate I/O addresses out of the ISA I/O ranges. The following are the SA I/O ranges: n100–n3FF n500–n7FF n900–nBFF nD00–nFFF See ISA Aliasing Considerations for more details.

Table 11. Memory Resource Flag (Resource Type = 0) Usage

Bits	Meaning
Bits[7:3]	Ignored.
Bit[2:1]	_MEM. Memory attributes. **Value and Meaning:** 0 The memory is nonprefetchable. 1 Invalid. 2 Invalid. 3 The memory is prefetchable. **Note:** The interpretation of these bits is somewhat different from the *ACPI Specification.* According to the *ACPI Specification,* a value of 0 implies noncacheable memory and the value of 3 indicates prefetchable and cacheable memory.
Bit[0]	Ignored.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase()

Summary

These are the notifications from the PCI bus driver that it is about to enter a certain phase of the PCI enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_NOTIFY_PHASE)
(
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL   *This,
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE       Phase
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

Phase

The phase during enumeration. Type **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This member function can be used to notify the host bridge driver to perform specific actions, including any chipset-specific initialization, so that the chipset is ready to enter the next phase. Nine notification points are defined at this time. See "Related Definitions" below for definitions of various notification points and section 10.7 for usage.

More synchronization points may be added as required in the future.

Related Definitions

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
  EfiPciHostBridgeBeginEnumeration,
  EfiPciHostBridgeBeginBusAllocation,
  EfiPciHostBridgeEndBusAllocation,
  EfiPciHostBridgeBeginResourceAllocation,
  EfiPciHostBridgeAllocateResources,
  EfiPciHostBridgeSetResources,
  EfiPciHostBridgeFreeResources,
  EfiPciHostBridgeEndResourceAllocation,
  EfiPciHostBridgeEndEnumeration,
  EfiMaxPciHostBridgeEnumeratonPhase
} EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE;
```

Table 12 provides a description of the fields in the above enumeration:

Table 12. Enumeration Descriptions

Enumeration	Description
`EfiPciHostBridgeBeginEnumeration`	Resets the host bridge PCI apertures and internal data structures. The PCI enumerator should issue this notification before starting a fresh enumeration process. Enumeration cannot be restarted after sending any other notification such as `EfiPciHostBridgeBeginBusAllocation`.
`EfiPciHostBridgeBeginBusAllocation`	The bus allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.
`EfiPciHostBridgeEndBusAllocation`	The bus allocation and bus programming phase is complete. No specific action is required here. This notification can be used to perform any chipset-specific programming.
`EfiPciHostBridgeBeginResourceAllocation`	The resource allocation phase is about to begin. No specific action is required here. This notification can be used to perform any chipset-specific programming.
`EfiPciHostBridgeAllocateResources`	Allocates resources per previously submitted requests for all the PCI root bridges. These resource settings are returned on the next call to `GetProposedResources()`. Before calling `NotifyPhase()` with a *Phase* of `EfiPciHostBridgeAllocateResource`, the PCI bus enumerator is responsible for gathering I/O and memory requests for all the PCI root bridges and submitting these requests using `SubmitResources()`. This function pads the resource amount to suit the root bridge hardware, takes care of dependencies between the PCI root bridges, and calls the Global Coherency Domain (GCD) with the allocation request. In the case of padding, the allocated range could be bigger than what was requested. Note that the size of the allocated range could be smaller than what was requested. This scenario could happen due to an allocation failure, a host bridge hardware limitation, or any other reason. In that case, the call will return an `EFI_OUT_OF_RESOURCES` error. If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. The PCI bus driver can call `GetProposedResouces()` to find out which of the resource types were partially allocated and the difference between the amount that was requested and the amount that was allocated. The PCI bus enumerator should readjust the requested sizes (by dropping certain PCI devices or PCI buses) to obtain a best fit. The PCI bus driver can call `NotifyPhase (EfiPciHostBridgeFreeResources)` to free up the original assignments and resubmit the adjusted resource requests with `SubmitResources()`.

Enumeration	Description
EfiPciHostBridgeSetResources	Programs the host bridge hardware to decode previously allocated resources (proposed resources) for all the PCI root bridges. After the hardware is programmed, reassigning resources will not be supported. The bus settings are not affected.
EfiPciHostBridgeFreeResources	Deallocates resources that were previously allocated for all the PCI root bridges and resets the I/O and memory apertures to their initial state. The bus settings are not affected. If the request to allocate resources fails, the PCI enumerator can use this notification to deallocate previous resources, adjust the requests, and retry allocation.
EfiPciHostBridgeEndResourceAllocation	The resource allocation phase is completed. No specific action is required here. This notification can be used to perform any chipset-specific programming.
EfiPciHostBridgeEndBusEnumeration	The bus enumeration phase is completed. No specific action is required here. This notification can be used to perform any chipset-specific programming.

Status Codes Returned

EFI_SUCCESS	The notification was accepted without any errors.
EFI_INVALID_PARAMETER	The *Phase* is invalid.
EFI_NOT_READY	This phase cannot be entered at this time. For example, this error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if **SubmitResources()** has not been called for one or more PCI root bridges before this call.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. This error is valid for a *Phase* of **EfiPciHostBridgeSetResources**.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources. This error is valid for a *Phase* of **EfiPciHostBridgeAllocateResources** if the previously submitted resource requests cannot be fulfilled or were only partially fulfilled.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetNextRootBridge()

Summary

Returns the device handle of the next PCI root bridge that is associated with this host bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_NEXT_ROOT_
BRIDGE) (
  IN     EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN OUT EFI_HANDLE                                      *RootBridgeHandle
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

Returns the device handle of the next PCI root bridge. On input, it holds the *RootBridgeHandle* that was returned by the most recent call to **GetNextRootBridge()**. If *RootBridgeHandle* is **NULL** on input, the handle for the first PCI root bridge is returned. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Description

This function is called multiple times to retrieve the device handles of all the PCI root bridges that are associated with this PCI host bridge. Each PCI host bridge is associated with one or more PCI root bridges. On each call, the handle that was returned by the previous call is passed into the interface, and on output the interface returns the device handle of the next PCI root bridge. The caller can use the handle to obtain the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** for that root bridge. When there are no more PCI root bridges to report, the interface returns **EFI_NOT_FOUND**. A PCI enumerator must enumerate the PCI root bridges in the order that they are returned by this function.

The search is initiated by passing in a **NULL** device handle as input. Some of the member functions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** operate on a PCI root bridge and expect the *RootBridgeHandle* as an input.

There is no requirement that this function return the root bridges in any specific relation with the EFI device paths of the root bridges.

This function can also be used to determine the number of PCI root bridges that were produced by this PCI host bridge. The host bridge hardware may provide mechanisms to change the number of root bridges that it produces, but such changes must be completed before the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not an **EFI_HANDLE** that was returned on a previous call to **GetNextRootBridge()**.
EFI_NOT_FOUND	There are no more PCI root bridge device handles.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()

Summary

Returns the allocation attributes of a PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_GET_ATTRIBUTES) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT UINT64                                            *Attributes
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The device handle of the PCI root bridge in which the caller is interested. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Attributes

The pointer to attributes of the PCI root bridge. The permitted attribute values are defined in "Related Definitions" below.

Description

The function returns the allocation attributes of a specific PCI root bridge. The attributes can vary from one PCI root bridge to another. These attributes are different from the decode-related attributes that are returned by the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()** member function. The *RootBridgeHandle* parameter is used to specify the instance of the PCI root bridge. The device handles of all the root bridges that are associated with this host bridge must be obtained by calling **GetNextRootBridge()**. The attributes are static in the sense that they do not change during or after the enumeration process. The hardware may provide mechanisms to change the attributes on the fly, but such changes must be completed before **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** is installed. The permitted values of **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES** are defined in "Related Definitions" below. The caller uses these attributes to combine multiple resource requests. For example, if the flag **EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM** is set, the PCI bus enumerator needs to include requests for the prefetchable memory in the nonprefetchable memory pool and not request any prefetchable memory.

Related Definitions

```
//*******************************************************
// EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES
//*******************************************************

#define EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM       1
#define EFI_PCI_HOST_BRIDGE_MEM64_DECODE           2
```

Following is a description of the fields in the above definition:

Table 13. EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_ATTRIBUTES field descriptions

EFI_PCI_HOST_BRIDGE_COMBINE_MEM_PMEM	If this bit is set, then the PCI root bridge does not support separate windows for nonprefetchable and prefetchable memory. A PCI bus driver needs to include requests for prefetchable memory in the nonprefetchable memory pool.
EFI_PCI_HOST_BRIDGE_MEM64_DECODE	If this bit is set, then the PCI root bridge supports 64-bit memory windows. If this bit is not set, the PCI bus driver needs to include requests for a 64-bit memory address in the corresponding 32-bit memory pool.

Status Codes Returned

EFI_SUCCESS	The requested attribute information was returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Attributes* is **NULL**.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration()

Summary

Sets up the specified PCI root bridge for the bus enumeration process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_START_BUS_ENUMERAT
ION) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                  RootBridgeHandle,
  OUT VOID                                        **Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge to be set up. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

Pointer to the pointer to the PCI bus resource descriptor.

Description

This member function sets up the root bridge for bus enumeration and returns the PCI bus range over which the search should be performed in ACPI (2.0 & 3.0) resource descriptor format. The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **StartBusEnumeration()**.

Table 14. ACPI 2.0 & 3.0 Resource Descriptor Field Values for StartBusEnumeration()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be scanned.
Address Range Length	Set to the number of PCI buses that may be scanned. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

This function cannot return resource descriptors for anything other than bus resources. This function can be used to prevent a PCI bus driver from scanning certain PCI buses to work around a chipset limitation. Because the size of ACPI resource descriptors is not fixed, **StartBusEnumeration()** is responsible for allocating memory for the buffer *Configuration*.

The PCI segment is implicit and is identified by the *SegmentNumber* field in the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the PCI root bridge handle *RootBridgeHandle*.

Status Codes Returned

EFI_SUCCESS	The PCI root bridge was set up and the bus range was returned in *Configuration*.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()

Summary

Programs the PCI root bridge hardware so that it decodes the specified PCI bus range.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SET_BUS_NUMBERS) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN EFI_HANDLE                                    RootBridgeHandle,
  IN VOID                                         *Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose bus range is to be programmed. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the PCI bus resource descriptor.

Description

This member function programs the specified PCI root bridge to decode the bus range that is specified by the input parameter *Configuration*.

The bus range information is specified in terms of the ACPI (2.0 & 3.0) resource descriptor format. The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **SetBusNumbers()**.

Table 15. ACPI 2.0 & 3.0 Resource Descriptor Field Values for SetBusNumbers()

Field	Setting
Address Range Minimum	Set to the lowest bus number to be decoded.
Address Range Length	Set to the number of PCI buses that should be decoded. The highest bus number is computed by adding the length to the lowest bus number and subtracting 1.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

This call will return **EFI_INVALID_PARAMETER** without programming the hardware if either of the following are specified:

- Any descriptors other than bus type descriptors
- Any invalid descriptors

The bus range is typically a subset of what was returned during **StartBusEnumeration()**. If **SetBusNumbers()** is called with incorrect (but valid) parameters, it may cause system failure.

The PCI segment is implicit and is identified by the *SegmentNumber* field in the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the PCI root bridge handle *RootBridgeHandle*. This call cannot alter the following:

- The *SegmentNumber* field in the corresponding instances of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
- The segment number settings in the hardware

The caller is responsible for allocating and deallocating a buffer to hold *Configuration*. If the call returns **EFI_DEVICE_ERROR**, the PCI bus enumerator can optionally attempt another bus setting.

Status Codes Returned

EFI_SUCCESS	The bus range for the PCI root bridge was programmed.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Configuration* is **NULL**.
EFI_INVALID_PARAMETER	*Configuration* does not point to a valid ACPI (2.0 & 3.0) resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* does not include a valid ACPI 2.0 bus resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* includes valid ACPI (2.0 & 3.0) resource descriptors other than bus descriptors.
EFI_INVALID_PARAMETER	*Configuration* contains one or more invalid ACPI resource descriptors.
EFI_INVALID_PARAMETER	"Address Range Minimum" is invalid for this root bridge.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this root bridge.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()

Summary

Submits the I/O and memory resource requirements for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_SUBMIT_RESOURCES) (
  IN EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN EFI_HANDLE                                        RootBridgeHandle,
  IN  VOID                                             *Configuration
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge whose I/O and memory resource requirements are being submitted. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the PCI I/O and PCI memory resource descriptor.

Description

This function is used to submit all the I/O and memory resources that are required by the specified PCI root bridge. The input parameter *Configuration* is used to specify the following:

- The various types of resources that are required
- The associated lengths in terms of ACPI (2.0 & 3.0) resource descriptor format

The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **SubmitResources()**.

Table 16. ACPI 2.0& 3.0 Resource Descriptor Field Values for SubmitResources()

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Used to differentiate between a 32-bit memory request and a 64-bit memory request. For a 32-bit memory request, this field should be set to 32. For a 64-bit memory request, this field should be set to 64. All other values result in this function returning the error code of **EFI_INVALID_PARAMETER**.
Address Range Maximum	Used to specify the alignment requirement. If "Address Range Maximum" is of the form 2^n-1, this member function returns the error code **EFI_INVALID_PARAMETER**. The address base must be a multiple of the granularity field. That is, if this field is 4 KiB-1, the allocated address must be a multiple of 4 KiB.
Address Range Minimum	Ignored.
Address Translation Offset	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

The caller must ask for appropriate alignment using the "Address Range Maximum" field. The caller is responsible for allocating and deallocating a buffer to hold *Configuration*.

It is considered an error if no resource requests are submitted for a PCI root bridge. If a PCI root bridge does not require any resources, a zero-length resource request must explicitly be submitted.

If the *Configuration* includes one or more invalid resource descriptors, all the resource descriptors are ignored and the function returns **EFI_INVALID_PARAMETER**.

Status Codes Returned

EFI_SUCCESS	The I/O and memory resource requests for a PCI root bridge were accepted.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Configuration* is **NULL**.
EFI_INVALID_PARAMETER	*Configuration* does not point to a valid ACPI (2.0 & 3.0) resource descriptor.
EFI_INVALID_PARAMETER	*Configuration* includes requests for one or more resource types that are not supported by this PCI root bridge. This error will happen if the caller did not combine resources according to *Attributes* that were returned by **GetAllocAttributes()**.
EFI_INVALID_PARAMETER	"Address Range Maximum" is invalid.
EFI_INVALID_PARAMETER	"Address Range Length" is invalid for this PCI root bridge.
EFI_INVALID_PARAMETER	"Address Space Granularity" is invalid for this PCI root bridge.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetProposedResources()

Summary

Returns the proposed resource settings for the specified PCI root bridge.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_GET_PROPOSED_RESOUR
CES) (
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL  *This,
  IN  EFI_HANDLE                                        RootBridgeHandle,
  OUT VOID                                              **Configuration
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The PCI root bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

Configuration

The pointer to the pointer to the PCI I/O and memory resource descriptor.

Description

This member function returns the proposed resource settings for the specified PCI root bridge. The proposed resource settings are prepared when **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeAllocateResources.** The output parameter *Configuration* specifies the following:

- The various types of resources, excluding bus resources, that are allocated
- The associated lengths in terms of ACPI (2.0 & 3.0) resource descriptor format

The following table lists the fields in the ACPI (2.0 & 3.0) resource descriptor that are set for **GetProposedResources()**.

Table 17. ACPI 2.0 & 3.0 GetProposedResources() Resource Descriptor Field Values

Field	Setting
Address Range Length	Set to the size of the aperture that is requested.
Address Space Granularity	Ignored.
Address Range Minimum	Indicates the starting address of the allocated ranges.
Address Translation Offset	Indicates the allocation status. Allocation status is defined in "Related Definitions" below.
Address Range Maximum	Ignored.
All other fields	Ignored.

Note: *See the "Description" section of the PCI Host Bridge Resource Allocation Protocol definition for a description of these ACPI resource descriptor fields.*

The callee is responsible for allocating a buffer to hold *`Configuration`* because the caller does not know the number of descriptors that are required. The caller is also responsible for deallocating the buffer.

If **`NotifyPhase()`** is called with a *`Phase`* of **`EfiPciHostBridgeAllocateResources`** and returns **`EFI_OUT_OF_RESOURCES`**, the PCI bus enumerator may use **`GetProposedResources()`** to retrieve the proposed settings. The **`EFI_OUT_OF_RESOURCES`** error status indicates that one or more requests could not be fulfilled or were partially fulfilled. Additional details of the allocation status for each type of resource can be retrieved from the "Address Translation Offset" field in the resource descriptor that was returned by this function; also see "Related Definitions" below for defined allocation status values. This error could happen for the following reasons:

- Allocation failure
- A limitation in the host bridge hardware
- Any other reason

If the allocated windows are smaller than what was requested, the PCI bus enumerator may not be able to fit all the devices within the range. In that case, the PCI bus enumerator may choose to readjust the requested sizes (by dropping certain devices or PCI buses) to obtain a best fit. The PCI bus driver calls **`NotifyPhase()`** with a *`Phase`* of **`EfiPciHostBridgeFreeResources`** to free the original assignments.

If this member function is able to only partially fulfill the requests for one or more resource types, the root bridges that are first in the list will get resources first. The ordering of the root bridges is determined by the output of **`GetNextRootBridge()`**. The handle to the first root bridge is obtained by calling **`GetNextRootBridge()`** with an input handle of **`NULL`**.

In the case of I/O resources, the PCI bus enumerator must check the _RNG flag. If this flag is set, the I/O ranges that are allocated to the devices must come from the non-ISA I/O subset.

For example, if this flag is set, the "Address Range Minimum" is 0x1000, and the "Address Range Length" is 0x1000, then the following I/O ranges can be allocated to PCI devices:

- 0x1000–0x10FF
- 0x1400–0x14FF

- 0x1800–0x18FF
- 0x1C00–0x1CFF

This call is made before **NotifyPhase()** is called with a *Phase* of **EfiPciHostBridgeSetResources**. After that time, the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.Configuration()** member function should be used to obtain the resources that were consumed by a particular PCI root bridge.

Related Definitions

```
//
****************************************************************
// EFI_RESOURCE_ALLOCATION_STATUS
//
****************************************************************
typedef  UINT64    EFI_RESOURCE_ALLOCATION_STATUS;

#define EFI_RESOURCE_SATISFIED                      0
#define EFI_RESOURCE_NOT_SATISFIED                  (UINT64) -1
```

Following is a description of the fields in the above definition. All other values indicate that the request of this resource type could be partially fulfilled. The exact value indicates how much more space is still required to fulfill the requirement.

Table 18. EFI_RESOURCE_ALLOCATION_STATUS field descriptions

EFI_RESOURCE_SATISFIED	The request of this resource type could be fulfilled.
EFI_RESOURCE_NOT_SATISFIED	The request of this resource type could not be fulfilled for its absence in the host bridge resource pool.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_DEVICE_ERROR	Programming failed due to a hardware error.
EFI_OUT_OF_RESOURCES	The request could not be completed due to a lack of resources.

EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController()

Summary

Provides the hooks from the PCI bus driver to every PCI controller (device/function) at various stages of the PCI enumeration process that allow the host bridge driver to preinitialize individual PCI controllers before enumeration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *
EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL_PREPROCESS_CONT
ROLLER)(
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL *This,
  IN  EFI_HANDLE                                       RootBridgeHandle,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS       PciAddress,
  IN  EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE     Phase
 );
```

Parameters

This

Pointer to the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** instance.

RootBridgeHandle

The associated PCI root bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification.*

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** member functions to access the PCI configuration space of the device. See *UEFI 2.1 Specification* for the definition of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**.

Phase

The phase of the PCI device enumeration. Type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** is defined in "Related Definitions" below.

Description

This function is called during the PCI enumeration process. No specific action is expected from this member function. It allows the host bridge driver to preinitialize individual PCI controllers before enumeration.

The parameter *RootBridgeHandle* can be used to locate the instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** that is installed on the root bridge that is the parent of the specific PCI function. The parameter *PciAddress* can be passed to the **Pci.Read()** and

Pci.Write() functions of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** instance to access the PCI configuration space of the specific PCI function.

This member function is invoked during PCI enumeration and before the PCI enumerator has created a handle for the PCI function. As a result, the **EFI_PCI_IO_PROTOCOL** cannot be used at this point.

Two notification points are defined at this time. See type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** in "Related Definitions" below for definitions of these notification points and ISA Aliasing Considerations for usage. More synchronization points may be added as required in the future.

Related Definitions

```
//*******************************************************
// EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE
//*******************************************************
typedef enum {
 EfiPciBeforeChildBusEnumeration,
 EfiPciBeforeResourceCollection
} EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE;
```

Following is a description of the fields in the above enumeration:

Table 19. EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE field descriptions

EfiPciBeforeChildBusEnumeration	This notification is applicable only to PCI-to-PCI bridges and indicates that the PCI enumerator is about to begin enumerating the bus behind the PCI-to-PCI bridge. This notification is sent after the primary bus number, the secondary bus number, and the subordinate bus number registers in the PCI-to-PCI bridge are programmed to valid (but not necessary final) values. Programming of the bus number register allows the chipset code to scan devices on the bus that are immediately behind the PCI-to-PCI bridge. This notification can be used to reset the secondary PCI bus. Some PCI-to-PCI bridges can drive their secondary bus at various clock speeds (33 MHz or 66 MHz, for example) and support PCI-X* or conventional PCI mode. These bridges must be set up to operate at the correct speed and correct mode before the downstream devices and buses are enumerated. This notification can be used to perform that activity. The host bridge code cannot reprogram the bus numbers in the PCI-to-PCI bridge or reprogram any upstream devices during this notification. It can touch the downstream devices because the PCI enumerator has not found these devices. If there are multiple PCI-to-PCI bridges on the same PCI bus, the order in which the notification is sent to these bridges is implementation specific. On the other hand, it is guaranteed that a PCI-to-PCI bridge will see this notification before the downstream bridge receives this notification or its child devices receive the **EfiPciBeforeResourceCollection** notification.
EfiPciBeforeResourceCollection	This notification is sent before the PCI enumerator probes the Base Address Register (BAR) registers for every valid PCI function. This notification can be used to program the backside registers that determine the BAR size or any other programming such as the master latency timer, cache line size, and PERR and SERR control. This notification is sent regardless of whether the function implements BAR or not. In the case of a multifunction device, this notification is sent for every function of the device. The order within the functions is not specified. The order in which this notification is sent to various devices/functions on the same bus is implementation specific.

Status Codes Returned

EFI_SUCCESS	The requested parameters were returned.
EFI_INVALID_PARAMETER	*RootBridgeHandle* is not a valid root bridge handle.
EFI_INVALID_PARAMETER	*Phase* is not a valid phase that is defined in **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE**.
EFI_DEVICE_ERROR	Programming failed due to a hardware error. The PCI enumerator should not enumerate this device, including its child devices if it is a PCI-to-PCI bridge.

10.9 End of PCI Enumeration Overview

This specification defines the indicia to inform the platform when the PCI enumeration process has completed. This allows for some post enumeration finalization actions to occur, if necessary.

10.9.1 End of PCI Enumeration Protocol

The indicia for this finalization action is a protocol. The obligation of the platform that supports this capability is as follows. Once PCI enumeration is complete, the **EFI_PCI_ENUMERATION_PROTOCOL** shall be installed on the same handle as the host bridge protocol.

This protocol is always installed with a NULL pointer.

GUID

```
#define EFI_PCI_ENUMERATION_COMPLETE_GUID \
  {  \
   0x30cfe3e7, 0x3de1, 0x4586,
  { 0xbe, 0x20, 0xde, 0xab, 0xa1, 0xb3, 0xb7, 0x93  } \
  }
```

The protocol can be used as an indicia by other DXE agents that the process of PCI device enumeration has been completed.

11 PCI Platform

11.1 Introduction

This section contains the basic definitions of protocols that provide PCI platform support. The following protocols are defined in this section:

EFI_PCI_PLATFORM_PROTOCOL
EFI_PCI_OVERRIDE_PROTOCOL
EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

EFI_PCI EXECUTION_PHASE
EFI_PCI_PLATFORM_POLICY

11.2 PCI Platform Overview

This section defines the core code and services that are required for an implementation of the following protocols in this specification:

- PCI Platform Protocol
- PCI Override Protocol
- Incompatible PCI Device Support Protocol

The PCI Platform Protocol allows a PCI bus driver to obtain the platform policy and call a platform driver at various points in the enumeration phase. The Incompatible PCI Device Support Protocol allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification.*

This specification does the following:

- Describes the basic components of the PCI Platform Protocol
- Describes the basic components of the Incompatible PCI Device Support Protocol and how firmware configures incompatible PCI devices
- Provides code definitions for the PCI Platform Protocol, the Incompatible PCI Device Support Protocol, and their related type definitions that are architecturally required by this specification.

This document is intended for the following readers:

- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in IntelÆ architecture-based products.
- Operating system developers who will be adapting their shrink-wrapped operating system products to run on Intel architecture-based platforms.

Readers of this specification are assumed to have solid knowledge of the *UEFI 2.1 Specification.*

11.3 PCI Platform Support Related Information

The following publications and sources of information may be useful to you or are referred to by this specification.

11.3.1 Industry Specifications

- *Advanced Configuration and Power Interface Specification* (hereafter referred to as the *ACPI Specification*), version 3.0.

11.3.2 PCI Specifications

- *Conventional PCI Specification*, version 3.0: http://www.pcisig.com*
- *PCI-to-PCI Bridge Architecture Specification*, revision 1.2: http://www.pcisig.com*
- *PCI-to-PCI Bridges and CardBus Controllers on Windows 2000, Windows XP, and Windows Server 2003:* http://www.microsoft.com/whdc/system/bus/PCI/pcibridge-cardbus.mspx*

11.4 PCI Platform Protocol

11.4.1 PCI Platform Protocol Overview

"PCI Host Bridge Resource Allocation Protocol", Section 10.8.2 defines and describes the PCI Host Bridge Resource Allocation Protocol. The PCI Host Bridge Resource Allocation Protocol driver provides chipset-specific functionality that works across processor architectures and unique platform features. It does not address issues where an implementation varies across platforms.

In contrast, the PCI Override Protocol and PCI Platform Protocol provide interfaces allow a platform driver or codebase driver to perform platform-specific actions. For example:

- Allow a PCI bus driver to obtain platform policy. The platform can use this protocol to control whether the PCI bus driver reserves I/O ranges for ISA aliases and VGA aliases. The default policy for the PCI bus driver is to reserve I/O ranges for both ISA aliases and VGA aliases, which may result in a large amount of I/O space being unavailable for PCI devices. This protocol allows the platform driver to change this policy.
- Call a platform driver at various points in the enumeration phase. The platform driver can use these hooks to perform various platform-specific activities. Examples of such activities include but are not limited to the following:
- **PlatformPrepController()** can be used to program the PCI subsystem vendor ID and device ID into onboard and chipset devices.
- **PlatformPrepController()** and **PlatformNotify()** can be used for implementing hardware workarounds.
- **PlatformPrepController()** can be used for preprogramming any backside registers that control the Base Address Register (BAR) window sizes.
- **PlatformPrepController()** can be used to set PCI or PCI-X* bus speeds for PCI bridges that support multiple bus speeds.

- Allow PCI option ROMs to be stored in local storage. The platform can store PCI option ROMs in local storage (e.g., a firmware volume) and report their existence to the PCI bus driver using the **GetPciRom()** member function. Option ROMs for embedded PCI controllers are often stored in a platform-specific location. The same member function can be used to override the default PCI ROM on an add-in card with one from platform-specific storage.

A platform should implement this protocol if any of the functionality that is listed above is required.

See Code Definitions for the definition of **EFI_PCI_PLATFORM_PROTOCOL** and the member functions listed above. See Section 10.8.2 for additional PCI-related design discussion.

11.5 Incompatible PCI Device Support Protocol

11.5.1 Incompatible PCI Device Support Protocol Overview

Some PCI devices do not fully comply with the PCI Specification. For example, a PCI device may request that its I/O Base Address Register (BAR) be placed on a 0x200 boundary even though it is requesting an I/O with a length of 0x100. The Incompatible PCI Device Support Protocol allows a PCI bus driver to handle resource allocation for some PCI devices that do not comply with the *PCI Specification.*

In the PI Architecture, the platform-specific PCI host bridge driver works with the generic, standard PCI bus driver to configure the entire PCI subsystem. Even though the exact configuration is up to individual incompatible devices, it is a platform choice to support those incompatible PCI devices. For example, one platform may not want to support those incompatible devices while another platform appears more tolerant of those devices.

See Code Definitions for the definition of the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**.

11.5.2 Usage Model for the Incompatible PCI Device Support Protocol

The following describes the usage model for the Incompatible PCI Device Support Protocol:

1. The PCI bus driver locates **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**. If the PCI bus driver cannot find this protocol, simply follow the regular PCI enumeration path. Otherwise, go to step 2.
2. For each PCI device that was detected, the PCI bus driver begins collecting the required PCI resources by probing the Base Address Register (BAR) for each device.
3. For each device, call **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()** to check whether this PCI device is an incompatible device. If this device in not an incompatible device, go to step 5.
4. Use the *Configuration* that is returned by **CheckDevice()** to override or modify the original PCI resource requirements.
5. Follow the normal PCI enumeration process.

11.6 PCI Code Definitions

This section contains the basic definitions of protocols that provide PCI platform support. The following protocols are defined in this section:

- **EFI_PCI_PLATFORM_PROTOCOL**
- **EFI_PCI_OVERRIDE_PROTOCOL**
- **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent function definition:

- **EFI_PCI_CHIPSET_EXECUTION_PHASE**
- **EFI_PCI_PLATFORM_POLICY**

11.6.1 PCI Platform Protocol

EFI_PCI_PLATFORM_PROTOCOL

Summary

This protocol provides the interface between the PCI bus driver/PCI Host Bridge Resource Allocation driver and a platform-specific driver to describe the unique features of a platform. This protocol is optional.

GUID

```
#define EFI_PCI_PLATFORM_PROTOCOL_GUID \
  { 0x7d75280, 0x27d4, 0x4d69, 0x90, 0xd0, 0x56, 0x43, 0xe2, \
  0x38, 0xb3, 0x41)
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_PLATFORM_PROTOCOL {
  EFI_PCI_PLATFORM_PHASE_NOTIFY          PlatformNotify;
  EFI_PCI_PLATFORM_PREPROCESS_CONTROLLER PlatformPrepController;
  EFI_PCI_PLATFORM_GET_PLATFORM_POLICY   GetPlatformPolicy;
  EFI_PCI_PLATFORM_GET_PCI_ROM           GetPciRom;
} EFI_PCI_PLATFORM_PROTOCOL;
```

Parameters

PlatformNotify

The notification from the PCI bus enumerator to the platform that it is about to enter a certain phase during the enumeration process. See the **PlatformNotify()** function description.

PlatformPrepController

The notification from the PCI bus enumerator to the platform for each PCI controller at several predefined points during PCI controller initialization. See the **PlatformPrepController()** function description.

GetPlatformPolicy

Retrieves the platform policy regarding enumeration. See the **GetPlatformPolicy()** function description.

GetPciRom

Gets the PCI device's option ROM from a platform-specific location. See the **GetPciRom()** function description.

Description

The **EFI_PCI_PLATFORM_PROTOCOL** is published by a platform-aware driver. This protocol is optional; see PCI Platform Protocol Overview in Design Discussion for scenarios in which this protocol is required. There cannot be more than one instance of this protocol in the system.

If the PCI bus driver detects the presence of this protocol before enumeration, it will use the PCI Platform Protocol to obtain information about the platform policy. The PCI bus driver will use this protocol to get the PCI device's option ROM from a platform-specific location in storage. It will also call the various member functions of this protocol at predefined points during PCI bus enumeration. The member functions can be used for performing any platform-specific initialization that is appropriate during the particular phase.

EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PHASE_NOTIFY) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL               *This,
  IN  EFI_HANDLE                                    HostBridge,
  IN  EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE Phase,
  IN  EFI_PCI_EXECUTION_PHASE                       ExecPhase
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

HostBridge

The handle of the host bridge controller. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

Phase

The phase of the PCI bus enumeration. Type **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PHASE** is defined in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase()**.

ExecPhase

Defines the execution phase of the PCI chipset driver. Type **EFI_PCI_EXECUTION_PHASE** is defined in "Related Definitions" below.

Description

The **PlatformNotify()** function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Several notification points are defined at this time. More notification points may be added as required in the future. The function should return **EFI_UNSUPPORTED** for any value of Phase that that the function does not support.

The PCI bus driver calls this function twice for every Phase-once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.

This member function may not perform any error checking on the input parameters. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Related Definitions

```
//*******************************************************************
// EFI_PCI_EXECUTION_PHASE
//*******************************************************************
typedef enum {
  BeforePciHostBridge = 0,
  ChipsetEntry        = 0,
  AfterPciHostBridge  = 1,
  ChipsetExit         = 1,
  MaximumExecutionPhase
} EFI_PCI_EXECUTION_PHASE;

typedef EFI_PCI_EXECUTION_PHASE EFI_PCI_CHIPSET_EXECUTION_PHASE;
```

Note: **EFI_PCI_EXECUTION_PHASE** *is used to call a platform protocol and execute platform-specific code. Following is a description of the fields in the above enumeration.*

BeforePciHostBridge

The phase that indicates the entry point to the PCI Bus Notify phase. This platform hook is called before the PCI bus driver calls the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** driver.

AfterPciHostBridge

The phase that indicates the exit point to the PCI Bus Notify phase before returning to the PCI Bus Driver Notify phase. This platform hook is called after the PCI bus driver calls the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** driver.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_UNSUPPORTED	The function does not support the phase specified by *Phase*.

EFI_PCI_PLATFORM_PROTOCOL.PlatformPrepController()

Summary

The platform driver receives notifications from the PCI bus enumerator at various phases during PCI controller initialization, just like the PCI host bridge driver.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_PREPROCESS_CONTROLLER) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL                *This,
  IN  EFI_HANDLE                                     HostBridge,
  IN  EFI_HANDLE                                RootBridge,
  IN  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS    PciAddress,
  IN  EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE   Phase,
  IN  EFI_PCI_EXECUTION_PHASE                        ExecPhase
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

HostBridge

The associated PCI host bridge handle. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

RootBridge

The associated PCI root bridge handle.

PciAddress

The address of the PCI device on the PCI bus. This address can be passed to the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL** functions to access the PCI configuration space of the device. See the *UEFI 2.1 Specification* for the definition of **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**.

Phase

The phase of the PCI controller enumeration. Type **EFI_PCI_CONTROLLER_RESOURCE_ALLOCATION_PHASE** is defined in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.PreprocessController()**.

ExecPhase

Defines the execution phase of the PCI chipset driver. Type **EFI_PCI_CHIPSET_EXECUTION_PHASE** is defined in **EFI_PCI_PLATFORM_PROTOCOL.PlatformNotify()**.

Description

The **PlatformPrepController()** function can be used to notify the platform driver so that it can perform platform-specific actions. No specific actions are required.

Several notification points are defined at this time. More synchronization points may be added as required in the future. The function should return EFI_UNSUPPORTED for any value of Phase that that the function does not support.

The PCI bus driver calls the platform driver twice for every PCI controller—once before the PCI Host Bridge Resource Allocation Protocol driver is notified, and once after the PCI Host Bridge Resource Allocation Protocol driver has been notified.

This member function may not perform any error checking on the input parameters. It also does not return any error codes. If this member function detects any error condition, it needs to handle those errors on its own because there is no way to surface any errors to the caller.

Status Codes Returned

EFI_SUCCESS	The function completed successfully.

EFI_PCI_PLATFORM_PROTOCOL.GetPlatformPolicy()

Summary

The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve platform policies regarding PCI enumeration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_GET_PLATFORM_POLICY) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL            *This,
  OUT EFI_PCI_PLATFORM_POLICY                    *PciPolicy,
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

PciPolicy

The platform policy with respect to VGA and ISA aliasing. Type **EFI_PCI_PLATFORM_POLICY** is defined in "Related Definitions" below.

Description

The **GetPlatformPolicy()** function retrieves the platform policy regarding PCI enumeration. The PCI bus driver and the PCI Host Bridge Resource Allocation Protocol driver can call this member function to retrieve the policy.

The **EFI_PCI_IO_PROTOCOL.Attributes()** function allows a PCI device driver to ask for various legacy ranges. Because PCI device drivers run after PCI enumeration, a request for legacy allocation comes in after PCI enumeration. The only practical way to guarantee that such a request from a PCI device driver will be fulfilled is to preallocate these ranges during enumeration. The PCI bus enumerator does not know which legacy ranges may be requested and therefore must rely on **GetPlatformPolicy()**. The data that is returned by **GetPlatformPolicy()** determines the supported attributes that are returned by the **EFI_PCI_IO_PROTOCOL.Attributes()** function.

See "Related Definitions" below for a description of the output parameter *PciPolicy*. For example, the platform can decide if it wishes to support devices that require ISA aliases using this parameter. Note that the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL.GetAttributes()** function returns the attributes that the root bridge hardware supports and does not depend upon preallocations.

Related Definitions

```
typedef UINT32 EFI_PCI_PLATFORM_POLICY;
```

EFI_PCI_PLATYFORM_POLICY is a bitmask with the following legal combinations.

```
#define  EFI_RESERVE_NONE_IO_ALIAS        0x0000
#define  EFI_RESERVE_ISA_IO_ALIAS         0x0001
#define  EFI_RESERVE_ISA_IO_NO_ALIAS      0x0002
#define  EFI_RESERVE_VGA_IO_ALIAS         0x0004
#define  EFI_RESERVE_VGA_IO_NO_ALIAS      0x0008
```

Status Codes Returned

EFI_SUCCESS	The function completed successfully.
EFI_UNSUPPORTED	The function is not supported.
EFI_INVALID_PARAMETER	*PciPolicy* is **NULL**.

EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()

Summary

Gets the PCI device's option ROM from a platform-specific location.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_PCI_PLATFORM_GET_PCI_ROM) (
  IN  CONST EFI_PCI_PLATFORM_PROTOCOL        *This,
  IN  EFI_HANDLE                             PciHandle,
  OUT VOID                                   **RomImage,
  OUT UINTN                                  *RomSize
  );
```

Parameters

This

Pointer to the **EFI_PCI_PLATFORM_PROTOCOL** instance.

PciHandle

The handle of the PCI device. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *UEFI 2.1 Specification*.

RomImage

If the call succeeds, the pointer to the pointer to the option ROM image. Otherwise, this field is undefined. The memory for RomImage is allocated by **EFI_PCI_PLATFORM_PROTOCOL.GetPciRom()** using the UEFI Boot Service **AllocatePool()**. It is the caller's responsibility to free the memory using the UEFI Boot Service **FreePool()**, when the caller is done with the option ROM.

RomSize

If the call succeeds, a pointer to the size of the option ROM size. Otherwise, this field is undefined.

Description

The **GetPciRom()** function gets the PCI device's option ROM from a platform-specific location. The option ROM will be loaded into memory. This member function is used to return an image that is packaged as a PCI 2.2 option ROM. The image may contain both legacy and UEFI option ROMs. See the *UEFI 2.1 Specification* for details. This member function can be used to return option ROM images for embedded controllers. Option ROMs for embedded controllers are typically stored in platform-specific storage, and this member function can retrieve it from that storage and return it to the PCI bus driver. The PCI bus driver will call this member function before scanning the ROM that is attached to any controller, which allows a platform to specify a ROM image that is different from the ROM image on a PCI card.

Status Codes Returned

EFI_SUCCESS	The option ROM was available for this device and loaded into memory.
EFI_NOT_FOUND	No option ROM was available for this device.
EFI_OUT_OF_RESOURCES	No memory was available to load the option ROM.
EFI_DEVICE_ERROR	An error occurred in getting the option ROM.

11.6.2 PCI Override Protocol

EFI_PCI_OVERRIDE_PROTOCOL

Summary

This protocol provides the interface between the PCI bus driver/PCI Host Bridge Resource Allocation driver and an implementation's driver to describe the unique features of a platform. This protocol is optional.

GUID

```
#define EFI_PCI_OVERRIDE_GUID \
  { 0xb5b35764, 0x460c, 0x4a06, { 0x99, 0xfc, 0x77, 0xa1, \
  0x7c, 0x1b, 0x5c, 0xeb } }
```

Protocol Interface Structure

```
typedef EFI_PCI_PLATFORM_PROTOCOL EFI_PCI_OVERRIDE_PROTOCOL;
```

Description

The PCI Override Protocol is published by an implementation aware driver. This protocol is optional. But it must be called, if present, during PCI enumeration. There cannot be more than one instance of this protocol in the system.

If the PCI bus driver detects the presence of this protocol before bus enumeration, it will use the PCI Override Protocol to obtain information about the platform policy. If the PCI Platform Protocol does not exist or returns an error, then this protocol is called.

The PCI bus driver will use this protocol to get the PCI device's option ROM from an implementation-specific location in storage. If the PCI Platform Protocol does not exist or returns an error, then this function is called.

It will also call the various member functions of this protocol at predefined points during PCI bus enumeration. The member functions can be used for performing any implementation-specific initialization that is appropriate during the particular phase.

11.6.3 Incompatible PCI Device Support Protocol

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL

Summary

Allows the PCI bus driver to support resource allocation for some PCI devices that do not comply with the PCI Specification.

Note: *This protocol is optional. Only those platforms that implement this protocol will have the capability to support incompatible PCI devices. The absence of this protocol can cause the PCI bus driver to configure these incompatible PCI devices incorrectly. As a result, these devices may not work properly.*

GUID

```
#define EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL_GUID \
  {0xeb23f55a, 0x7863, 0x4ac2, 0x8d, 0x3d, 0x95, 0x65, 0x35, \
  0xde, 0x3, 0x75}
```

Protocol Interface Structure

```
typedef struct _EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL {
  EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_CHECK_DEVICE  CheckDevice;
} EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL;
```

Parameters

CheckDevice

Returns a list of ACPI resource descriptors that detail any special resource configuration requirements if the specified device is a recognized incompatible PCI device. See the **CheckDevice()** function description.

Description

The **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** is used by the PCI bus driver to support resource allocation for some PCI devices that do not comply with the *PCI Specification*. This protocol can find some incompatible PCI devices and report their special resource requirements to the PCI bus driver. The generic PCI bus driver does not have prior knowledge of any incompatible PCI devices. It interfaces with the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** to find out if a device is incompatible and to obtain the special configuration requirements for a specific incompatible PCI device.

This protocol is optional, and only one instance of this protocol can be present in the system. If a platform supports this protocol, this protocol is produced by a Driver Execution Environment (DXE) driver and must be made available before the Boot Device Selection (BDS) phase. The PCI bus driver will look for the presence of this protocol before it begins PCI enumeration.

If this protocol exists in a platform, it indicates that the platform has the capability to support those incompatible PCI devices. However, final support for incompatible PCI devices still depends on the implementation of the PCI bus driver. The PCI bus driver may fully, partially, or not even support these incompatible devices.

During PCI bus enumeration, the PCI bus driver will probe the PCI Base Address Registers (BARs) for each PCI device—regardless of whether the PCI device is incompatible or not—to determine the resource requirements so that the PCI bus driver can invoke the proper PCI resources for them. Generally, this resource information includes the following:

- Resource type
- Resource length
- Alignment

However, some incompatible PCI devices may have special requirements. As a result, the length or the alignment that is derived through BAR probing may not be exactly the same as the actual resource requirement of the device. For example, there are some devices that request I/O resources at a length of 0x100 from their I/O BAR, but these incompatible devices will never work correctly if an odd I/O base address, such as 0x100, 0x300, or 0x500, is assigned to the BAR. Instead, these devices request an even base address, such as 0x200 or 0x400. The Incompatible PCI Device Support Protocol can then be used to obtain these special resource requirements for these incompatible PCI devices. In this way, the PCI bus driver will take special consideration for these devices during PCI resource allocation to ensure that they can work correctly.

This protocol may support the following incompatible PCI BAR types:

- I/O or memory length that is different from what the BAR reports
- I/O or memory alignment that is different from what the BAR reports
- Fixed I/O or memory base address

See the *Conventional PCI Specification 3.0* for the details of how a PCI BAR reports the resource length and the alignment that it requires.

EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL.CheckDevice()

Summary

Returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_CHECK_DEVICE) (
  IN  EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL  *This,
  IN  UINTN                                          VendorId,
  IN  UINTN                                          DeviceId,
  IN  UINTN                                          RevisionId,
  IN  UINTN                                    SubsystemVendorId,
  IN  UINTN                                    SubsystemDeviceId,
  OUT VOID                                        **Configuration
  );
```

Parameters

This

Pointer to the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL** instance.

VendorID

A unique ID to identify the manufacturer of the PCI device. See the *Conventional PCI Specification 3.0* for details.

DeviceID

A unique ID to identify the particular PCI device. See the *Conventional PCI Specification 3.0* for details.

RevisionID

A PCI device-specific revision identifier. See the *Conventional PCI Specification 3.0* for details.

SubsystemVendorId

Specifies the subsystem vendor ID. See the *Conventional PCI Specification 3.0* for details.

SubsystemDeviceId

Specifies the subsystem device ID. See the *Conventional PCI Specification* 3.0 for details.

Configuration

A list of ACPI resource descriptors that detail the configuration requirement. See Table 20 in the "Description" subsection below for the definition.

Description

The **CheckDevice()** function returns a list of ACPI resource descriptors that detail the special resource configuration requirements for an incompatible PCI device.

Prior to bus enumeration, the PCI bus driver will look for the presence of the **EFI_INCOMPATIBLE_PCI_DEVICE_SUPPORT_PROTOCOL**. Only one instance of this protocol can be present in the system. For each PCI device that the PCI bus driver discovers, the PCI bus driver calls this function with the device's vendor ID, device ID, revision ID, subsystem vendor ID, and subsystem device ID. If the *VendorId*, *DeviceId*, *RevisionId*, *SubsystemVendorId*, or *SubsystemDeviceId* value is set to **(UINTN)-1**, that field will be ignored. The ID values that are not **(UINTN)-1** will be used to identify the current device.

This function will only return **EFI_SUCCESS**. However, if the device is an incompatible PCI device, a list of ACPI resource descriptors will be returned in *Configuration*. Otherwise, **NULL** will be returned in *Configuration* instead. The PCI bus driver does not need to allocate memory for *Configuration*. However, it is the PCI bus driver's responsibility to free it. The PCI bus driver then can configure this device with the information that is derived from this list of resource nodes, rather than the result of BAR probing.

Only the following two resource descriptor types from the *ACPI Specification* may be used to describe the incompatible PCI device resource requirements:

- QWORD Address Space Descriptor (ACPI 2.0, section 6.4.3.5.1; also ACPI 3.0)
- End Tag (ACPI 2.0, section 6.4.2.8; also ACPI 3.0)

The QWORD Address Space Descriptor can describe memory, I/O, and bus number ranges for dynamic or fixed resources. The configuration of a PCI root bridge is described with one or more QWORD Address Space Descriptors, followed by an End Tag. Table 20 and Table 21 below contain these two descriptor types. See the *ACPI Specification* for details on the field values.

Table 20. ACPI 2.0 & 3.0 QWORD Address Space Descriptor Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x8A	QWORD Address Space Descriptor
0x01	0x02	0x2B	Length of this descriptor in bytes, not including the first two fields.
0x03	0x01		Resource type: 0: Memory range 1: I/O range Other values will be ignored.
0x04	0x01		General flags. Ignored.
0x05	0x01		Type-specific flags. Only used when Address Translation Offset == 6. Value: 0: Skip device option ROM (do not probe option rom BAR). Other values will be ignored
0x06	0x08		Address Space Granularity. Ignored if the value is 0. Ignored if the PCI BAR is I/O. Ignored if the PCI BAR is 32-bt memory. If PCI BAR is 64-bit memory and this field is 32, then the PCI BAR resource is allocated below 4GB. If the PCI BAR is 64-bit memory and this field is 64, then the PCI BAR resource is allocated above 4GB.
0x16	0x08		Address Range Maximum. Used to indicate the BAR Index from 0 to 6, where 6 is the Option Rom BAR. Specially, **(UINT64)-1** in this field means all the PCI BARs on the device except the Option Rom BAR.
0x1E	0x08		Address Translation Offset. Used to indicate the BAR Index from 0 to 6, where 6 is the Option ROM BAR. Specially, **(UINT64)-1** in this field means all the PCI BARs are on the device, except the Option ROM BAR.
0x26	0x08		Address Range Length. Length of the requested resource. If the device has no special length request, it must be 0. Then the length that was obtained from BAR probing will be applied.

Table 21. ACPI 2.0 & 3.0 End Tag Usage

Byte Offset	Byte Length	Data	Description
0x00	0x01	0x79	End Tag.
0x01	0x01	0x00	Checksum. Set to 0 to indicate that checksum is to be ignored.

Status Codes Returned

EFI_SUCCESS	The function always returns **EFI_SUCCESS**.

12 Hot Plug PCI

12.1 HOT PLUG PCI Overview

This specification defines the core code and services that are required for an implementation of the Hot-Plug PCI Initialization Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. The same protocol may be used by other buses such as CardBus that support hot plugging. This specification does the following:

- Describes the basic components of the hot-plug PCI subsystem and the Hot-Plug PCI Initialization Protocol
- Provides code definitions for the Hot-Plug PCI Initialization Protocol and the hot-plug-PCI–related type definitions that are architecturally required.

12.2 Hot Plug PCI Initialization Protocol Introduction

This chapter describes the Hot-Plug PCI Initialization Protocol. A PCI bus driver, running in the EFI Boot Services environment, uses this protocol to initialize the hot-plug subsystem. This protocol is generic enough to include PCI-to-CardBus bridges and PCI Express* systems. This protocol abstracts the hot-plug controller initialization and resource padding. This protocol is required on platforms that support PCI Hot Plug* or PCI Express slots. For the purposes of initialization, a CardBus PC Card bus is treated in the same way. This protocol is not required on all other platforms.

This protocol is not intended to support hot plugging of PCI cards during the preboot stage. Separate components can be developed if such support is desired.

See Hot-Plug PCI Initialization Protocol in Code Definitions for the definition of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**.

12.3 Hot Plug PCI Initialization Protocol Related Information

The following resources are referenced throughout this specification or may be useful to you:

- *Conventional PCI Specification,* revision 3.0: http://www.pcisig.com/*
- *PC Card Standard,* volumes 1, 7, and 8: http://www.pcmcia.org/*
- *PCI Express Base Specification,* revision 1.0a: http://www.pcisig.com/*
- *PCI Hot-Plug Specification,* revision 1.1: http://www.pcisig.com/*
- *PCI Standard Hot-Plug Controller and Subsystem Specification,* revision 1.0: http://www.pcisig.com/*

12.4 Requirements

PI Architecture firmware must support platforms with PCI Hot Plug* slots and PCI Express* Hot Plug slots, as well as CardBus PC Card sockets. In both cases, the user is allowed to plug in new devices or remove existing devices during runtime. PCI Hot Plug slots are controlled by a PCI Hot Plug controller whereas CardBus sockets are controlled by a PCI-to-CardBus bridge. PCI Express Hot Plug slots are controlled by a PCI Express root port or a downstream port in a switch. The term "Hot Plug Controller" (HPC) in this document refers to all of these types of controllers. From the standpoint of initialization, all three are identical and have the same general requirements, as follows:

- The root HPCs may come up uninitialized after system reset. These HPCs must be initialized by the system firmware.
- Every HPC may require resource padding. The padding must be taken into account during PCI enumeration. This scenario is true for conventional PCI, PCI Express, and CardBus PC Cards because they all consume shared system resources (I/O, memory, and bus). These resources are produced by the root PCI bridge.

These general requirements place the following specific requirements on an implementation of the PI Architechture PCI hot plug support:

- PI Architecture firmware must handle root HPCs differently than other regular PCI devices. When a root HPC is initialized, the hot-plug slots or CardBus sockets are enabled and this process may uncover more PCI buses and devices. In that respect, root HPCs are somewhat like PCI bridges. The root HPC initialization process may involve detecting bus type and optimum bus speed. The initialization process may also detect faults and voltage mismatches. The initialization process may be specific to the controller and the platform. At the time of the root HPC initialization, the PCI bus may not be fully initialized and the standard PCI bus-specific protocols are not available. PI Architecture firmware must provide an alternate infrastructure for the initialization code. In other words, the HPC initialization code should not be required to understand the bus numbering scheme and other chipset details.
- PI Architecture firmware must support an unlimited number of HPCs in the system. PI Architecture firmware must support various types of HPCs as long as they follow industry standards or conventions. A mix of various types of HPCs is allowed.
- PI Architecture firmware must support legacy PCI Hot Plug Controllers (PHPCs; class code 0x6, subclass code 0x4) as well as Standard (PCI) Hot Plug Controllers (SHPCs). Other conventional PCI Hot Plug controllers are not supported.
- PI Architecture firmware must be capable of supporting a PHPC that is a child of another PHPC. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that the child PHPC must be initialized without firmware assistance because it is not a root PHPC.
- PI Architecture firmware must be capable of supporting SHPCs on an add-in card. In that case, the *PCI Standard Hot-Plug Controller and Subsystem Specification* requires that such an SHPC must be initialized without firmware assistance because it is not a root PHPC. PI Architecture firmware must also support plug-in CardBus bridges that follow the *CardBus Specification,* which is part of the *PC Card Standard.*

- As stated above, root HPCs may require firmware initialization. PI Architecture firmware must be capable of supporting root HPCs that are initialized by hardware and do not require any firmware initialization.
- A PI Architecture PCI bus enumerator must overallocate resources for PCI Hot Plug buses and CardBus sockets. The amount of overallocation may be platform specific.
- The root HPC initialization process may be time consuming. An SHPC can take as long as 15 seconds to enable power to a hot-plug bus without violating the PCI Special Interest Group (PCI-SIG*) requirements. PI Architecture firmware should be able to initialize multiple HPCs in parallel to reduce boot time. In contrast, CardBus initialization is quick.
- PI Architecture firmware should be able to handle when an HPC fails. PI Architecture firmware should be able to handle an HPC that has been disabled.
- The PCI bus driver in PI Architecture firmware is not required to assume anything that is not in one of the PCI-SIG specifications.
- It must be possible to produce legacy Hot Plug Resource Tables (HPRTs) if necessary. HPRTs are described in the *PCI Standard Hot-Plug Controller and Subsystem Specification.*

12.5 Sample Implementation for a Platform Containing PCI Hot Plug* Slots

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI bus enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. Actual implementations may vary although the relative ordering of events is critical. The activities related to PCI Hot Plug* are underlined. Please note that multiple passes of bus enumeration are required in a system containing PCI Hot Plug slots.

See section 10.3 for definitions of the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL** and its member functions.

If the platform supports PCI Hot Plug, an instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is installed.

The PCI enumeration process begins.

Look for instances of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. If it is not found, all the hot-plug subsystem initialization steps can be skipped. If one exists, create a list of root Hot Plug Controllers (HPCs) by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()**.

Notify the host bridge driver that bus enumeration is about to begin by calling **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.NotifyPhase** (EfiPciHostBridgeBeginBusAllocation).

For every PCI root bridge handle, do the following:

1. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration** (This,RootBridgeHandle).
2. Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. See the *UEFI 2.1 Specification* for the definition of the PCI Root Bridge I/O Protocol.
3. Allocate memory to hold resource requirements. These can be two resource descriptors, one to hold bus requirements and another to hold the I/O and memory requirements.

4. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.

   Scan all the devices in the specified bus range and the specified segment, one bus at a time. If the device is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If the device path of a device matches that of a root HPC and it is not a PCI-to-CardBus bridge, it must be initialized by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()** before the bus it controls can be fully enumerated. The PCI bus enumerator determines the PCI address of the PCI Hot Plug Controller (PHPC) and passes it as an input to **InitializeRootHpc()**.
5. Continue to scan devices on that root bridge and start the initialization of all root HPCs.
6. Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()** so that the HPCs under initialization are still accessible. SetBusNumbers() cannot affect the PCI addresses of the HPCs.

Wait until all the HPCs that were found on various root bridges in step 5 to complete initialization.

Go back to step 5 for another pass and rescan the PCI buses. For all the root HPCs and the nonroot HPCs, call **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()** to obtain the amount of overallocation and add that amount to the requests from the physical devices. Reprogram the bus numbers by taking into account the bus resource padding information. This action will require calling **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()**. The rescan is not required if there is only one root bridge in the system.

Once the memory resources are allocated and a PCI-to-CardBus bridge is part of the HpcList, it will be initialized.

12.6 PCI Hot Plug PCI Initialization Protocol

EFI_PCI_HOT_PLUG_INIT_PROTOCOL

Summary

This protocol provides the necessary functionality to initialize the Hot Plug Controllers (HPCs) and the buses that they control. This protocol also provides information regarding resource padding.

Note: *This protocol is required only on platforms that support one or more PCI Hot Plug* slots or CardBus sockets.*

GUID

```
#define EFI_PCI_HOT_PLUG_INIT_PROTOCOL_GUID \
  { 0xaa0e8bc1, 0xdabc, 0x46b0, 0xa8, 0x44, 0x37, 0xb8, 0x16,\
  0x9b, 0x2b, 0xea }
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOT_PLUG_INIT_PROTOCOL {
  EFI_GET_ROOT_HPC_LIST                    GetRootHpcList;
  EFI_INITIALIZE_ROOT_HPC                  InitializeRootHpc;
  EFI_GET_HOT_PLUG_PADDING                 GetResourcePadding;
} EFI_PCI_HOT_PLUG_INIT_PROTOCOL;
```

Parameters

GetRootHpcList

Returns a list of root HPCs and the buses that they control. See the **GetRootHpcList()** function description.

InitializeRootHpc

Initializes the specified root HPC. See the **InitializeRootHpc()** function description.

GetResourcePadding

Returns the resource padding that is required by the HPC. See the **GetResourcePadding()** function description.

Description

The **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** provides a mechanism for the PCI bus enumerator to properly initialize the HPCs and CardBus sockets that require initialization. The HPC initialization takes place before the PCI enumeration process is complete. There cannot be more than one instance of this protocol in a system. This protocol is installed on its own separate handle.

Because the system may include multiple HPCs, one instance of this protocol should represent all of them. The protocol functions use the device path of the HPC to identify the HPC. When the PCI bus enumerator finds a root HPC, it will call **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**. If InitializeRootHpc() is unable to initialize a root HPC, the PCI enumerator will ignore that root HPC and continue the enumeration process. If the HPC is not initialized, the devices that it controls may not be initialized, and no resource padding will be provided.

From the standpoint of the PCI bus enumerator, HPCs are divided into the following two classes:

Root HPC

These HPCs must be initialized by calling InitializeRootHpc() during the enumeration process. These HPCs will also require resource padding. The platform code must have *a priori* knowledge of these devices and must know how to initialize them. There may not be any way to access their PCI configuration space before the PCI enumerator programs all the upstream bridges and thus enables the path to these devices. The PCI bus enumerator is responsible for determining the PCI bus address of the HPC before it calls InitializeRootHpc().

Nonroot HPC

These HPCs will not need explicit initialization during enumeration process. These HPCs will require resource padding. The platform code does not have to have *a priori* knowledge of these devices.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()

Summary

Returns a list of root Hot Plug Controllers (HPCs) that require initialization during the boot process.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_ROOT_HPC_LIST) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL   *This,
  OUT UINTN                            *HpcCount,
  OUT EFI_HPC_LOCATION                 **HpcList
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcCount

The number of root HPCs that were returned.

HpcList

The list of root HPCs. HpcCount defines the number of elements in this list. Type **EFI_HPC_LOCATION** is defined in "Related Definitions" below.

Description

This procedure returns a list of root HPCs. The PCI bus driver must initialize these controllers during the boot process. The PCI bus driver may or may not be able to detect these HPCs. If the platform includes a PCI-to-CardBus bridge, it can be included in this list if it requires initialization. The HpcList must be self consistent. An HPC cannot control any of its parent buses. Only one HPC can control a PCI bus. Because this list includes only root HPCs, no HPC in the list can be a child of another HPC. This policy must be enforced by the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. The PCI bus driver may not check for such invalid conditions.

The callee allocates the buffer HpcList.

Related Definitions

```
//*******************************************************
// EFI_HPC_LOCATION
//*******************************************************
typedef struct {
  EFI_DEVICE_PATH_PROTOCOL   *HpcDevicePath;
  EFI_DEVICE_PATH_PROTOCOL   *HpbDevicePath;
} EFI_HPC_LOCATION;
```

HpcDevicePath

The device path to the root HPC. An HPC cannot control its parent buses. The PCI bus driver requires this information so that it can pass the correct HpcPciAddress to the **InitializeRootHpc()** and **GetResourcePadding()** functions. Type **EFI_DEVICE_PATH_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of the *UEFI 2.1 Specification.*

HpbDevicePath

The device path to the Hot Plug Bus (HPB) that is controlled by the root HPC. The PCI bus driver uses this information to check if a particular PCI bus has hot-plug slots. The device path of a PCI bus is the same as the device path of its parent. For Standard (PCI) Hot Plug Controllers (SHPCs) and PCI Express*, HpbDevicePath is the same as HpcDevicePath.

Status Codes Returned

EFI_SUCCESS	HpcList was returned.
EFI_OUT_OF_RESOURCES	HpcList was not returned due to insufficient resources.
EFI_INVALID_PARAMETER	HpcCount is NULL.
EFI_INVALID_PARAMETER	HpcList is NULL.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()

Summary

Initializes one root Hot Plug Controller (HPC). This process may causes initialization of its subordinate buses.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_INITIALIZE_ROOT_HPC) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL     *This,
  IN  EFI_DEVICE_PATH_PROTOCOL           *HpcDevicePath,
  IN  UINT64                             HpcPciAddress,
  IN  EFI_EVENT                          Event, OPTIONAL
  OUT EFI_HPC_STATE                      *HpcState
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC that is being initialized. Type **EFI_DEVICE_PATH_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of the *UEFI 2.1 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

Event

The event that should be signaled when the HPC initialization is complete. Set to **NULL** if the caller wants to wait until the entire initialization process is complete. The event must be of type **EFI_EVENT_NOTIFY_SIGNAL**. Type **EFI_EVENT** is defined in **CreateEvent()** in the *UEFI Specification.*

HpcState

The state of the HPC hardware. The type **EFI_HPC_STATE** is defined in "Related Definitions" below.

Description

This function initializes the specified HPC. At the end of initialization, the hot-plug slots or sockets (controlled by this HPC) are powered and are connected to the bus. All the necessary registers in the HPC are set up. For a Standard (PCI) Hot Plug Controller (SHPC), the registers that must be set up are defined in the *PCI Standard Hot Plug Controller and Subsystem Specification.* For others HPCs, they are specific to the HPC hardware. The initialization process may choose not to enable certain PCI Hot Plug* slots or sockets for any reason. The PCI Hot Plug slots or CardBus sockets that are left disabled at this stage are not available to the system. A PCI slot may be disabled due to a power fault, PCI bus type mismatch, or power budget constraints. The HPC initialization process can be time consuming. Powering up the slots that are controlled by SHPCs can take up to 15 seconds. In a system with multiple HPCs, it is desirable to perform these activities in parallel. Therefore, this procedure supports nonblocking execution mode.

If **InitializeRootHpc()** is called with a non-NULL event, HPC initialization is considered complete after the event is signaled. If **InitializeRootHpc()** is called with a non-NULL event, a return from **InitializeRootHpc()** with EFI_SUCCESS marks the completion of the HPC initialization.

The PCI bus enumerator will call this function for every root HPC that is returned by **GetRootHpcList()**.

The PCI bus enumerator must make sure that the registers that are required during HPC initialization are accessible before calling **InitializeRootHpc()**. The determination of whether the registers are accessible is based on the following rules:

- For HPCs (legacy HPCs, SHPCs inside a PCI-to-PCI bridge, and PCI Express* HPCs), the PCI configuration space of the HPC device must be accessible. In other words, all the upstream bridges including root bridges and special-purpose PCI-to-PCI bridges are programmed to forward PCI configuration cycles to the HPC.
- SHPCs inside a root bridge are accessible without any initialization of the PCI bus.
- PCI-to-CardBus bridges have their registers mapped into the memory space using a memory Base Address Register (BAR).

This function takes the device path of the HPC as an input. At the time of HPC initialization, the PCI bus enumeration is not complete. The PCI bus enumerator may not have created a handle for the HPC and the hot-plug initialization code cannot use the **EFI_PCI_IO_PROTOCOL** or **EFI_DEVICE_PATH_PROTOCOL** like other PCI device drivers. The device path uniquely identifies the HPC and also the PCI bus that it controls.

If the HPC is a PCI device, the hot-plug initialization code may need its address on the PCI bus (**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_PCI_ADDRESS**; see the *UEFI 2.1 Specification* for its definition) to access its registers. The PCI address of a regular PCI device is dynamic but is known to the PCI bus driver. Therefore, the PCI bus driver provides it through the input parameter HpcPciAddress to this function. Passing this address eliminates the need for **InitializeRootHpc()** to convert the device path into the PCI address. If the HPC is a function in a multifunction device, this address is the PCI address of that function. The HPC’s configuration space must be accessible at the specified HpcPciAddress until the HPC initialization is complete. In other words, the PCI bus driver cannot renumber PCI buses that are upstream to the HPC while it is being initialized.

This member function can use the **LocateDevicePath()** function to locate the appropriate instance of the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**.

If the *Event* is not NULL, this function will return control to the caller without completing the entire initialization. This function must perform some basic checks to make sure that it knows how to initialize the specified HPC before returning control. The *Event* is signaled when the initialization process completes, regardless of whether it results in a failure. The caller must check HpcState to get the initialization status after the event is signaled.

If *Event* is not NULL, it is possible that the *Event* may be signaled before this function returns. There are at least two cases where that may happen:

- A simple implementation of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** may force the caller to wait until the initialization is complete. In that case, the InitializeRootHpc() function may signal the event before it returns control back to the caller.
- The HPC may already have been initialized by the time **InitializeRootHpc()** is called. In that case, **InitializeRootHpc()** will signal *Event* and return control back to the caller.

HpcState returns the state of the HPC at the time when control returns. If Event is NULL, HpcState must indicate that the HPC has completed initialization. If Event is not NULL, HpcState can indicate that the HPC has not completed initialization when this function returns, but HpcState must be updated before Event is signaled.

The firmware may not wait until InitializeRootHpc() to start HPC initialization. The firmware may start the initialization earlier in the boot process and the initialization may be completely done by the time the PCI bus enumerator calls InitializeRootHpc(). An HPC can be initialized by hardware alone, and no firmware initialization may be needed. For such HPCs, this member function does not have to do any real work. In such cases, InitializeRootHpc() merely acts as a synchronization point.

Related Definitions

```
//*******************************************************
// EFI_HPC_STATE
//*******************************************************
// Describes current state of an HPC

typedef UINT16 EFI_HPC_STATE;

#define  EFI_HPC_STATE_INITIALIZED    0x01
#define  EFI_HPC_STATE_ENABLED        0x02
```

Following is a description of the possible states for **EFI_HPC_STATE**.

Table 22. Description of possible states for EFI_HPC_STATE

0	Not initialized
EFI_HPC_STATE_INITIALIZED	The HPC initialization function was called and the HPC completed initialization, but it was not enabled for some reason. The HPC may be disabled in hardware, or it may be disabled due to user preferences, hardware failure, or other reasons. No resource padding is required.
EFI_HPC_STATE_INITIALIZED | EFI_HPC_ENABLED	The HPC initialization function was called, the HPC completed initialization, and it was enabled. Resource padding is required.

Status Codes Returned

EFI_SUCCESS	If *Event* is NULL, the specific HPC was successfully initialized. If *Event* *i*s not NULL, Event will be signaled at a later time when initialization is complete.
EFI_UNSUPPORTED	This instance of **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC. If *Event* is not NULL, it will not be signaled.

EFI_OUT_OF_RESOURCES	Initialization failed due to insufficient resources. If `Event` is not NULL, it will not be signaled.
EFI_INVALID_PARAMETER	HpcState is NULL.

EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()

Summary

Returns the resource padding that is required by the PCI bus that is controlled by the specified Hot Plug Controller (HPC).

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_GET_HOT_PLUG_PADDING) (
  IN  EFI_PCI_HOT_PLUG_INIT_PROTOCOL    *This,
  IN  EFI_DEVICE_PATH_PROTOCOL          *HpcDevicePath,
  IN  UINT64                            HpcPciAddress,
  OUT EFI_HPC_STATE                     *HpcState,
  OUT VOID                              **Padding,
  OUT EFI_HPC_PADDING_ATTRIBUTES        *Attributes
  );
```

Parameters

This

Pointer to the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** instance.

HpcDevicePath

The device path to the HPC. Type **EFI_DEVICE_PATH_PROTOCOL** is defined in **LocateDevicePath()** in section 11.2 of the *UEFI 2.1 Specification.*

HpcPciAddress

The address of the HPC function on the PCI bus.

HpcState

The state of the HPC hardware. Type **EFI_HPC_STATE** is defined in **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**.

Padding

The amount of resource padding that is required by the PCI bus under the control of the specified HPC. Because the caller does not know the size of this buffer, this buffer is allocated by the callee and freed by the caller.

Attributes

Describes how padding is accounted for. The padding is returned in the form of ACPI (2.0 & 3.0) resource descriptors. The exact definition of each of the fields is the same as in **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()** in section 10.8.2**.** Type **EFI_HPC_PADDING_ATTRIBUTES** is defined in "Related Definitions" below.

Description

This function returns the resource padding that is required by the PCI bus that is controlled by the specified HPC. This member function is called for all the root HPCs and nonroot HPCs that are detected by the PCI bus enumerator. This function will be called before PCI resource allocation is completed. This function must be called after all the root HPCs, with the possible exception of a PCI-to-CardBus bridge, have completed initialization. Waiting until initialization is completed allows the HPC driver to optimize the padding requirement. The calculation may take into account the number of empty and/or populated PCI Hot Plug* slots, the number of PCI-to-PCI bridges among the populated slots, and other factors. This information is available only after initialization is complete. PCI-to-CardBus bridges require memory resources before the initialization is started and therefore are considered an exception. The padding requirements are relatively constant for PCI-to-CardBus bridges and an estimated value must be returned.

If **InitializeRootHpc()** is called with a non-NULL event, HPC initialization is considered complete after the event is signaled. If **InitializeRootHpc()** is called with a non-NULL event, a return from **InitializeRootHpc()** with **EFI_SUCCESS** marks the completion of HPC initialization.

The input parameters *HpcDevicePath*, *HpcPciAddress*, and *HpcState* are described in **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()**. The value of HpcPciAddress for the same root HPC may be different from what was passed to **InitializeRootHpc()**. The HPC’s configuration space must be accessible at the specified HpcPciAddress until this function returns control.

The padding is returned in the form of ACPI (2.0 & 3.0) resource descriptors. The exact definition of each of the fields is the same as in the **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SubmitResources()** function. See the section 10.8 for the definition of this function.

The PCI bus driver is responsible for adding this resource request to the resource requests by the physical PCI devices. If *Attributes* is EfiPaddingPciBus, the padding takes effect at the PCI bus level. If *Attributes* is EfiPaddingPciRootBridge, the required padding takes effect at the root bridge level. For details, see the definition of **EFI_HPC_PADDING_ATTRIBUTES** in "Related Definitions" below.

Note that the padding request cannot ask for specific legacy resources such as COM port addresses. Legacy PC Card devices may require such resources. Supporting these resource requirements is outside the scope of this specification.

Related Definitions

```
//*******************************************************
// EFI_HPC_PADDING_ATTRIBUTES
//*******************************************************
// Describes how resource padding should be applied

typedef enum {
  EfiPaddingPciBus,
  EfiPaddingPciRootBridge
} EFI_HPC_PADDING_ATTRIBUTES;
```

Following is a description of the fields in the above definition.

Table 23. EFI_HPC_PADDING_ATTRIBUTES field descriptions

EfiPaddingPciBus	Apply the padding at a PCI bus level. In other words, the resources that are allocated to the bus containing hot-plug slots are padded by the specified amount. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures will indicate the padding.
EfiPaddingPciRootBridge	Apply the padding at a PCI root bridge level. If a PCI root bridge includes more than one hot-plug bus, the resource padding requests for these buses are added together and the resources that are allocated to the root bridge are padded by the specified amount. This strategy may reduce the total amount of padding, but requires reprogramming of PCI-to-PCI bridges in a hot-add event. If the hot-plug bus is behind a PCI-to-PCI bridge, the PCI-to-PCI bridge apertures do not indicate the padding for that bus.

Status Codes Returned

EFI_SUCCESS	The resource padding was successfully returned.
EFI_UNSUPPORTED	This instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** does not support the specified HPC.
EFI_NOT_READY	This function was called before HPC initialization is complete.
EFI_INVALID_PARAMETER	HpcState is NULL.
EFI_INVALID_PARAMETER	Padding is NULL.
EFI_INVALID_PARAMETER	Attributes is NULL.
EFI_OUT_OF_RESOURCES	ACPI (2.0 & 3.0) resource descriptors for Padding cannot be allocated due to insufficient resources.

12.7 PCI Hot Plug Request Protocol

A hot-plug capable PCI bus driver should produce the EFI PCI Hot Plug Request protocol. When a PCI device or a PCI-like device (for example, 32-bit PC Card) is installed after PCI bus does the enumeration, the PCI bus driver can be notified through this protocol. For example, when a 32-bit

PC Card is inserted into the PC Card socket, the PC Card bus driver can call interface of this protocol to notify PCI bus driver to allocate resource and create handles for this PC Card.

Summary

Provides services to notify PCI bus driver that some events have happened in a hot-plug controller (for example, PC Card socket, or PHPC), and ask PCI bus driver to create or destroy handles for the PCI-like devices.

GUID

```
#define EFI_PCI_HOTPLUG_REQUEST_PROTOCOL_GUID \
  {0x19cb87ab,0x2cb9,0x4665,0x83,0x60,0xdd,0xcf,0x60,0x54,\
  0xf7,0x9d}
```

Protocol Interface Structure

```
typedef struct _EFI_PCI_HOTPLUG_REQUEST_PROTOCOL {
  EFI_PCI_HOTPLUG_REQUEST_NOTIFY     Notify;
} EFI_PCI_HOTPLUG_REQUEST_PROTOCOL;
```

Parameters

Notify

Notify the PCI bus driver that some events have happened in a hot-plug controller (for example, PC Card socket, or PHPC), and ask PCI bus driver to create or destroy handles for the PCI-like devices. See Section 0 for a detailed description.

Description

The **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** is installed by the PCI bus driver on a separate handle when PCI bus driver starts up. There is only one instance in the system. Any driver that wants to use this protocol must locate it globally.

The **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** allows the driver of hot-plug controller, for example, PC Card Bus driver, to notify PCI bus driver that an event has happened in the hot-plug controller, and the PCI bus driver is requested to create (add) or destroy (remove) handles for the specified PCI-like devices. For example, when a 32-bit PC Card is inserted, this protocol interface will be called with an add operation, and the PCI bus driver will enumerate and start the devices inserted; when a 32-bit PC Card is removed, this protocol interface will be called with a remove operation, and the PCI bus driver will stop the devices and destroy their handles.

The existence of this protocol represents the capability of the PCI bus driver. If this protocol exists in system, it means PCI bus driver is hot-plug capable, thus together with the effort of PC Card bus driver, hot-plug of PC Card can be supported. Otherwise, the hot-plug capability is not provided.

EFI_PCI_HOTPLUG_REQUEST_PROTOCOL.Notify()

Summary

This function is used to notify PCI bus driver that some events happened in a hot-plug controller, and the PCI bus driver is requested to start or stop specified PCI-like devices.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PCI_HOTPLUG_REQUEST_NOTIFY) (
  IN EFI_PCI_HOTPLUG_REQUEST_PROTOCOL *This,
  IN EFI_PCI_HOTPLUG_OPERATION        Operation,
  IN EFI_HANDLE                       Controller,
  IN EFI_DEVICE_PATH_PROTOCOL         *RemainingDevicePath
OPTIONAL,
  IN OUT UINT8                        NumberOfChildren,
  IN OUT EFI_HANDLE                   *ChildHandleBuffer
  );
```

Parameters

This

A pointer to the **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** instance. Type **EFI_PCI_HOTPLUG_REQUEST_PROTOCOL** is defined in Section 0.

Operation

The operation the PCI bus driver is requested to make. See "Related Definitions" for the list of legal values.

Controller

The handle of the hot-plug controller.

RemainingDevicePath

The remaining device path for the PCI-like hot-plug device. It only contains device path nodes behind the hot-plug controller. It is an optional parameter and only valid when the Operation is a add operation. If it is NULL, all devices behind the PC Card socket are started.

NumberOfChildren

The number of child handles. For a add operation, it is an output parameter. For a remove operation, it’s an input parameter. When it contains a non-zero value, children handles specified in *ChildHandleBuffer* are destroyed. Otherwise, PCI bus driver is notified to stop managing the controller handle.

ChildHandleBuffer

The buffer which contains the child handles. For a add operation, it is an output parameter and contains all newly created child handles. For a remove operation, it contains child handles to be destroyed when *NumberOfChildren* contains a non-zero value. It can be NULL when *NumberOfChildren* is 0. It's the caller's responsibility to allocate and free memory for this buffer.

Description

This function allows the PCI bus driver to be notified to act as requested when a hot-plug event has happened on the hot-plug controller. Currently, the operations include add operation and remove operation.

If it is a add operation, the PCI bus driver will enumerate, allocate resources for devices behind the hot-plug controller, and create handle for the device specified by *RemainingDevicePath*. The *RemainingDevicePath* is an optional parameter. If it is not NULL, only the specified device is started; if it is NULL, all devices behind the hot-plug controller are started. The newly created handles of PC Card functions are returned in the *ChildHandleBuffer*, together with the number of child handle in *NumberOfChildren*.

If it is a remove operation, when *NumberOfChildren* contains a non-zero value, child handles specified in *ChildHandleBuffer* are stopped and destroyed; otherwise, PCI bus driver is notified to stop managing the controller handle.

Related Definitions

```
//*******************************************************
// EFI PCI HOTPLUG NOTIFY OPERATION
//*******************************************************
typedef enum {
  EfiPciHotPlugRequestAdd,
  EfiPciHotplugRequestRemove
} EFI_PCI_HOTPLUG_OPERATION;
```

EfiPciHotplugRequestAdd

The PCI bus driver is requested to create handles for the specified devices. An array of **EFI_HANDLE** is returned, a NULL element marks the end of the array.

EfiPciHotplugRequestRemove

The PCI bus driver is requested to destroy handles for the specified devices.

Status Codes Returned

EFI_SUCCESS	The handles for the specified device have been created or destroyed as requested, and for an add operation, the new handles are returned in *ChildHandleBuffer*.
EFI_INVALID_PARAMETER	*Operation* is not a legal value.
EFI_INVALID_PARAMETER	*Controller* is NULL or not a valid handle.
EFI_INVALID_PARAMETER	*NumberOfChildren* is NULL.
EFI_INVALID_PARAMETER	*ChildHandleBuffer* is NULL while *Operation* is ìremoveî and *NumberOfChildren* contains a non-zero value.
EFI_INVALID_PARAMETER	*ChildHandleBuffer* is NULL while *Operation* is ìaddî.
EFI_OUT_OF_RESOURCES	There are no enough resources to start the devices.

12.8 Sample Implementation for a Platform Containing PCI Hot Plug* Slots

Typically, the PCI bus driver will enumerate and allocate resources to all devices for a PCI host bridge. A sample algorithm for PCI bus enumeration is described below to clarify some of the finer points of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. Actual implementations may vary although the relative ordering of events is critical. The activities related to PCI Hot Plug* are underlined. Please note that hot plug PCI devices may require that multiple passes of bus enumeration are required.

There are several phases during the PCI bus enumeration process when PCI hot plug slots are present. At each phase, the PlatformNotify function of the **EFI_PCI_PLATFORM_PROTOCOL** and **EFI_PCI_OVERRIDE_PROTOCOL** will be called with the execution phase BeforePciHostBridge. Then the PCI host bridge driver function NotifyPhase is called. Finally, the PlatformNotify functions are called again, but with the execution phase **AfterPciHostBridge**.

1. If the platform supports PCI Hot Plug, an instance of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL** is installed.
2. The PCI enumeration process begins.
3. Look for instances of the **EFI_PCI_HOT_PLUG_INIT_PROTOCOL**. If it is not found, all the hot-plug subsystem initialization steps can be skipped. If one exists, create a list of root Hot Plug Controllers (HPCs) by calling **EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetRootHpcList()**.
4. Notify the drivers using **EfiPciHostBridgeBeginBusAllocation**.
5. For every PCI root bridge handle, do the following:
   - Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.StartBusEnumeration** (*This, RootBridgeHandle*).
   - Make sure each PCI root bridge handle supports the **EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**. See the *UEFI 2.1`Specification* for the definition of the PCI Root Bridge I/O Protocol.
   - Allocate memory to hold resource requirements.
   - Call **EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.GetAllocAttributes()** to get the attributes of this PCI root bridge. This information is used to combine different types of memory resources in the next step.

   Scan all the devices in the specified bus range and the specified segment, one bus at a time. If the device is a PCI-to-PCI bridge, update the bus numbers and program the bus number registers in the PCI-to-PCI bridge hardware. If the device path of a device matches that of a root HPC and it is not a PCI-to-CardBus bridge, it must be initialized by calling **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL.InitializeRootHpc()`** before the bus it controls can be fully enumerated. The PCI bus enumerator determines the PCI address of the PCI Hot Plug Controller (PHPC) and passes it as an input to **`InitializeRootHpc()`**.

- Continue to scan devices on that root bridge and start the initialization of all root HPCs.
- Call **`EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()`** so that the HPCs under initialization are still accessible. SetBusNumbers() cannot affect the PCI addresses of the HPCs.

6. Wait until all the HPCs that were found on various root bridges in step 5 to complete initialization.
7. Go back to step 5 for another pass and rescan the PCI buses. For all the root HPCs and the nonroot HPCs, call **`EFI_PCI_HOT_PLUG_INIT_PROTOCOL.GetResourcePadding()`** to obtain the amount of overallocation and add that amount to the requests from the physical devices. Reprogram the bus numbers by taking into account the bus resource padding information. This action requires calling **`EFI_PCI_HOST_BRIDGE_RESOURCE_ALLOCATION_PROTOCOL.SetBusNumbers()`**. The rescan is not required if there is only one root bridge in the system.

Once the memory resources are allocated and a PCI-to-CardBus bridge is part of the HpcList, it will be initialized.

13 Super I/O Protocol

13.1 Super I/O Protocol

EFI_SIO_PROTOCOL

Summary

The Super I/O driver installs an instance of this protocol on the handle of every device within the Super I/O chip.

GUID

```
#define EFI_SIO_PROTOCOL_GUID  \
  { 0x215fdd18, 0xbd50, 0x4feb, { 0x89, 0xb, 0x58, 0xca, \
  0xb, 0x47, 0x39, 0xe9 } }
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_PROTOCOL {
    EFI_SIO_REGISTER_ACCESS    RegisterAccess;
    EFI_SIO_GET_RESOURCES      GetResources;
    EFI_SIO_SET_RESOURCES      SetResources;
    EFI_SIO_POSSIBLE_RESOURCES PossibleResources;
    EFI_SIO_MODIFY             Modify;
} EFI_SIO_PROTOCOL;
```

Parameters

RegisterAccess

Provides a low level access to the registers for the Super I/O.

GetResources

Provides a list of current resources consumed by the device in ACPI Resource Descriptor Format.

SetResources

Sets resources for a device.

PossibleResources

Provides a collection of possible resource descriptors for the device. Each resource descriptor in the collection defines a combination of resources that can potentially be used by the device.

Modify

Provides an interface for table based programming of the Super I/O registers.

Description

The Super I/O Protocol is installed by the Super I/O driver. The Super I/O driver is a UEFI driver model compliant driver. In the **Start()** routine of the Super I/O driver, a handle with an instance of **EFI_SIO_PROTOCOL** is created for each device within the Super I/O. The device within the Super I/O is powered up, enabled, and assigned with the default set of resources. In the **Stop()** routine of the Super I/O driver, the device is disabled and Super I/O protocol is uninstalled.

EFI_SIO_PROTOCOL.RegisterAccess()

Summary

Provides a low level access to the registers for the Super I/O.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_REGISTER_ACCESS) (
  IN CONST EFI_SIO_PROTOCOL *This,
  IN BOOLEAN                 Write,
  IN BOOLEAN                 ExitCfgMode,
  IN UINT8                   Register,
  IN OUT UINT8               *Value
  );
```

Parameters

This

Indicates a pointer to the calling context.

Write

Specifies the type of the register operation. If this parameter is **TRUE**, *Value* is interpreted as an input parameter and the operation is a register write. If this parameter is **FALSE**, *Value* is interpreted as an output parameter and the operation is a register read.

ExitCfgMode

Exit Configuration Mode Indicator. If this parameter is set to **TRUE**, the Super I/O driver will turn off configuration mode of the Super I/O prior to returning from this function. If this parameter is set to **FALSE**, the Super I/O driver will leave Super I/O in the configuration mode.

The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to register access.

Register

Register number.

Value

If *Write* is **TRUE**, *Value* is a pointer to the buffer containing the byte of data to be written to the Super I/O register. If *Write* is **FALSE**, *Value* is a pointer to the destination buffer for the byte of data to be read from the Super I/O register.

Description

The **RegisterAccess()** function provides low level interface to the registers in the Super I/O.

Note: *This function only provides access to the internal registers of the Super I/O chip. For example, on a typical desktop system, these are the registers accessed via the 0x2E/0x2F indexed port I/O.*

This function cannot be used to access I/O or memory locations assigned to individual logical devices.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	The *Value* is NULL
EFI_INVALID_PARAMETER	Invalid *Register* number

EFI_SIO_PROTOCOL.GetResources()

Summary

Provides an interface to get a list of the current resources consumed by the device in the ACPI Resource Descriptor format.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_GET_RESOURCES)(
  IN  CONST EFI_SIO_PROTOCOL    *This,
  OUT ACPI_RESOURCE_HEADER_PTR  *ResourceList
  );
```

Parameters

This

Indicates a pointer to the calling context.

ResourceList

A pointer to an ACPI resource descriptor list that defines the current resources used by the device. Type **ACPI_RESOURCE_HEADER_PTR** is defined in the "Related Definitions" below.

Description

GetResources() returns a list of resources currently consumed by the device. The *ResourceList* is a pointer to the buffer containing resource descriptors for the device. The descriptors are in the format of Small or Large ACPI resource descriptor as defined by ACPI specification (2.0 & 3.0). The buffer of resource descriptors is terminated with the 'End tag' resource descriptor.

Related Definitions

```
typedef union {
  UINT8 Byte;
  struct{
    UINT8 Length : 3;
    UINT8 Name   : 4;
    UINT8 Type   : 1;
  }Bits;
} ACPI_SMALL_RESOURCE_HEADER;

typedef struct {
  union {
    UINT8 Byte;
    struct{
      UINT8 Name   : 7;
      UINT8 Type   : 1;
    }Bits;
  } Header;
  UINT16 Length;
} ACPI_LARGE_RESOURCE_HEADER;

typedef union {
  ACPI_SMALL_RESOURCE_HEADER *SmallHeader;
  ACPI_LARGE_RESOURCE_HEADER *LargeHeader;
} ACPI_RESOURCE_HEADER_PTR;
```

Length

Length of the resource descriptor in bytes.

Name

Resource descriptor name. Possible values for this field are defined in the ACPI specification.

Type

Descriptor type.

0 – ACPI Small Resource Descriptor

1 – ACPI Large Resource Descriptor

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	ResourceList is NULL

EFI_SIO_PROTOCOL.SetResources()

Summary

Sets the resources for the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_SET_RESOURCES)(
  IN CONST EFI_SIO_PROTOCOL   *This,
  IN ACPI_RESOURCE_HEADER_PTR ResourceList
  );
```

Parameters

This

Indicates a pointer to the calling context.

ResourceList

Pointer to the ACPI resource descriptor list. Type **ACPI_RESOURCE_HEADER_PTR** is defined in the "Related Definitions" section of **EFI_SIO_PROTOCOL.GetResources()**.

Description

SetResources() sets the resources for the device. *ResourceList* is a pointer to the ACPI resource descriptor list containing requested resources for the device.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*ResourceList* is invalid
EFI_ACCESS_DENIED	Some of the resources in *ResourceList* are in use

EFI_SIO_PROTOCOL.PossibleResources()

Summary

Provides a collection of resource descriptor lists. Each resource descriptor list in the collection defines a combination of resources that can potentially be used by the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_POSSIBLE_RESOURCES)(
  IN  CONST EFI_SIO_PROTOCOL      *This,
  OUT ACPI_RESOURCE_HEADER_PTR    *ResourceCollection
  );
```

Parameters

This

Indicates a pointer to the calling context.

ResourceCollection

Collection of the resource descriptor lists. Type **ACPI_RESOURCE_HEADER_PTR** is defined in the “Related Definitions” section of **EFI_SIO_PROTOCOL.GetResources()**.

Description

PossibleResources() returns a collection of resource descriptor lists. Each resource descriptor list in the collection defines a combination of resources that can potentially be used by the device. The descriptors are in the format of Small or Large ACPI Resource Descriptor as defined by the *ACPI Specification* (2.0 & 3.0). The collection is terminated with the ‘End tag’ resource descriptor.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*ResourceCollection* is NULL

EFI_SIO_PROTOCOL.Modify()

Summary

Provides an interface for a table based programming of the Super I/O registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_MODIFY)(
  IN CONST EFI_SIO_PROTOCOL         *This,
  IN CONST EFI_SIO_REGISTER_MODIFY *Command,
  IN UINTN                          NumberOfCommands
  );
```

Parameters

This

Indicates a pointer to the calling context.

Command

A pointer to an array of *NumberOfCommands* **EFI_SIO_REGISTER_MODIFY** structures. Each structure specifies a single Super I/O register modify operation. Type **EFI_SIO_REGISTER_MODIFY** is defined in the "Related Definitions" below.

NumberOfCommands

Number of elements in the *Command* array.

Description

The **Modify()** function provides an interface for table based programming of the Super I/O registers. This function can be used to perform programming of multiple Super I/O registers with a single function call. For each table entry, the *Register* is read, its content is bitwise ANDed with *AndMask,* and then ORed with *OrMask* before being written back to the *Register*. The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to table processing. Once the table is processed, the Super I/O device has to be returned to the original state.

Note: *This function only provides access to the internal registers of the Super I/O chip. For example, on a typical desktop system, these are the registers accessed via the 0x2E/0x2F indexed port I/O.*

This function cannot be used to access I/O or memory locations assigned to individual logical devices.

Related Definitions

```
typedef struct {
  UINT8     Register;
  UINT8     AndMask;
  UINT8     OrMask;
} EFI_SIO_REGISTER_MODIFY;
```

Register

Register number.

AndMask

Bitwise AND mask.

OrMask

Bitwise OR mask.

Status Codes Returned

EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETER	*Command* is NULL

14 Super I/O and ISA Host Controller Interactions

14.1 Design Descriptions

The PI architecture provides a means to interact in a standard fashion with Super I/O devices. For the purposes of this specification, the Super I/O is a device residing on an ISA or LPC or similar bus that consumes I/O and/or memory resources and provides multiple standard logical devices, such as PC/AT compatible floppy, serial port, parallel port, keyboard or mouse. There may be more than one of these devices behind each of the ISA/LPC buses.

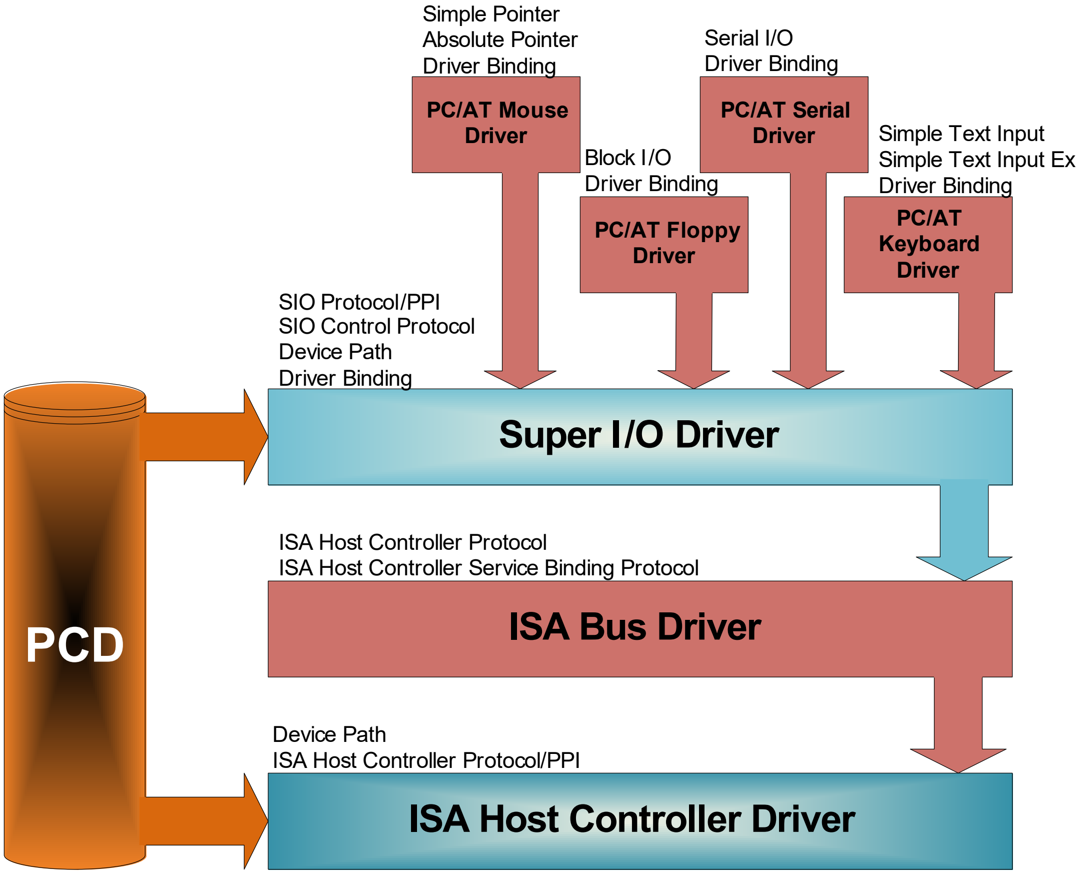


Figure 10. Super I/O and ISA Host Controller Interactions

Mouse, Floppy, Serial and keyboard Drivers

The Mouse, Floppy, Serial and Keyboard drivers are UEFI driver-model drivers that support devices produced by the Super I/O component. When started, they use the optional SIO Control protocol to enable the logical device, to produce the standard UEFI protocols used for console or booting, such as Serial I/O or Block I/O. They typically examine the device paths on the child handles created by the Super I/O drivers for the ACPI device path nodes that refer to their devices (e.g. PNP0501, PNP0303, etc.).

Super I/O Driver

The Super I/O driver consists of a UEFI driver-model driver (in DXE) and PEIM (in PEI) that supports a Super I/O component. The Super I/O components support multiple logical devices, such as the PS/2 keyboard controller, a floppy controller or serial/IrDa controller. When started, the Super I/O driver verifies it is present on the board and produces child handles for each of the logical devices that are enabled. On each child handle it installs an instance of the Device Path protocol, the SIO protocol and the SIO Control protocol.

ISA Bus Driver

The ISA Bus driver consists of a UEFI driver-model driver (DXE only) that produces the ISA Host Controller Service Binding protocol, which manages the many-to-one relationship between Super I/O drivers in the system and an ISA Host Controller.

ISA Host Controller Driver

The ISA Host Controller driver is a DXE driver that supports a PCI-ISA or PCI-LPC bridge component. It creates a child handle that represents the ISA Bus and installs the ISA Host Controller protocol and the Device Path protocol with an ACPI device path node (PNP0A05/PNP0A06).

PCD

The Platform Configuration Database (PCD) provides configuration information about the device configuration. Information concerning configured I/O addresses can be placed into the PCD by platform drivers and then used by the various silicon drivers, including SIO to find base addresses and logical device configuration.

14.1.1 Super I/O

The Super I/O DXE driver and PEIM encapsulate the functionality of the Super I/O component. They are both responsible to:

- Detect the presence of the component, using information from the PCD and the apertures opened by the ISA host controller.
- Configure the component and its logical devices using information from the PCD.
- Publish information about the component and the logical devices it supports using the SIO protocol/PPI.

14.1.1.1 DXE

The Super I/O DXE Driver is responsible for:

- Producing the Driver Binding protocol's **`Supported()`**, **`Start()`**and **`Stop()`**member functions on the driver image handle.
- Installing the same GUID as used for the SioGuid member of the Super I/O PPI on the image handle. This allows other drivers to detect which Super I/O is present in the system.
- Checking Super I/O controller presence. The **`Supported()`** function must check whether the Super I/O controller is present in the system and whether the handle has an instance of the ISA Host Controller Service Binding protocol installed on it. For more information, see "Working With The ISA Bus", below.

- Creatingchild handles for each logical device. The **Start()** function creates a child handle for each logical device using the ISA Host Controller Service Binding protocol and installs the SIO and SIO Control protocols on each one. For more information, see "Working With Logical Devices", below.

14.1.1.1.1 Working with the ISA Bus

The system may contain an ISA bus bridge and zero or more Extended I/O bus bridges. The Super I/O driver checks each of these to see whether it is present.

Supported()

The Super I/O DXE driver's Driver Binding protocol **Supported()** function typically performs the following steps:

1. Verifies that the controller handle has an installed instance of the ISA Host Controller Service Binding protocol.
2. Opens the apertures necessary to access the component's configuration I/O address (i.e. 0x2e/0x2f) using the ISA Host Controller protocol.
3. Verifies the device's signature to determine whether the component is actually present using these configuration I/O addresses. For example, it might read a device-specific register and check for a signature.
4. Closes the aperture and any opened protocols.

Start()

The Super I/O DXE driver's Driver Binding protocol **Start()**function typically performs the following steps:

1. Detects whether Super I/O DXE driver is already managing the device indicated by the configuration I/O address. One method of doing this is to create a Device Path with the configuration I/O address embedded in one device node, then use *LocateDevicePath* to determine whether a child handle with the ISA Host Controller protocol installed, exists.
2. Creates a child handle for the SIO using the ISA Host Controller Service Binding protocol.
   - Opens the I/O apertures used for the configuration I/O address.
   - Installs an instance of the Device Path and (optionally) the SIO and SIO Control protocol
3. Creates child handles for each logical device. Install an instance of the Device Path and SIO protocol and (optionally) the SIO Control protocol on each child handle.
4. Installs an instance of the Device Path, SIO and SIO Control protocols on each of the child handles.

Stop()

The Super I/O DXE driver's Driver Binding protocol **Stop()**function typically performs the following steps:

1. Uninstalls the instances of the Device Path, SIO and SIO control protocols from each of the child handles.
2. Destroys the Super I/O's own child handle using the ISA Host Controller Service Binding protocol.

SetResources()

The Super I/O DXE driver's SIO protocol **SetResources()**function typically calls the **OpenIoAperture()** and **CloseIoAperture()** member functions of the ISA Host Controller protocol for the I/O addresses related to the individual logical devices.

14.1.1.1.2 Working with Logical Devices

The Super I/O controller supports many different logical devices. Some of these devices, such as the floppy controller, keyboard controller, MIDI controller and serial port are standard PC/AT devices. These drivers produce interfaces based on these industry-standard interfaces. Also, the Super I/O component itself may act as a logical device.

For each logical device, the following steps are taken during **Start()**:

1. Create a child handle.
2. Install the **EFI_SIO_PROTOCOL** (with correct current resource settings) on the child handle.
3. Install the **EFI_SIO_CONTROL_PROTOCOL** on the same child handle. This protocol allows a standard drivers to correctly enable and disable their resources when the **Start()** and **Stop()** members of the Driver Binding protocol that they produce is called.
4. If the device implements one of the standard PC/AT devices, install the **EFI_DEVICE_PATH_PROTOCOL** by appending a device node containing the ACPI HID of the PC/AT device to the device path of the ISA bus on which it is installed..

For more information, see "Logical Devices"

14.1.1.2 PEI

The Super I/O PEIM is responsible to:

- Read its configuration information from the PCD.
- Detect if the Super I/O device is present in the system. If necessary, it should open the aperture required to access the configuration registers using the ISA Host Controller PPI. If the Super I/O device is not detected, the driver should close the aperture and exit immediately.
- Install the **EFI_SIO_PPI** for the Super I/O. The Identifier field allows consumers of the PPI to know which device's register set can be accessed by using the PPI's functions, in cases where multiple Super I/Os are supported on a platform.
- Allocate I/O and memory resources. All I/O and memory resources are allocated using the **EFI_ISA_HC_PPI**, which handles opening and closing bridge apertures.

The Super I/O PEIM should have the **EFI_ISA_HC_PPI** in its dependency expression.

14.1.2 ISA Bus

The ISA Bus is the logical device that manages the child devices attached to the ISA Host Controller.

It consumes the ISA Host Controller protocol produced by the ISA Host Controller and installs the ISA Host Controller Service Binding protocol on the ISA Host Controller's handle.

14.1.3 ISA Host Controller

The Host Controller is the device that translates the memory and I/O cycles from a parent device (such as a PCI bus) into memory and I/O cycles for the target devices.

14.1.3.1 DXE

The ISA Host Controller driver creates a child handle for the ISA Host Controller and installs an instance of the ISA Host Controller protocol and Device Path protocol on it. The Device Path instance for the child handle will have an extra ACPI device path node for either PNP0A05 (standard subtractive-decode ISA bus) or PNP0A06 (positive-decode extended I/O bus). If a bridge device can support more than one of these simultaneously, the _UID field of the device path node must contain a different value.

For PCI-ISA/LPC bridges, there are two classes of the ISA Host Controller Driver: generic and chipset-specific. The generic ISA Bus driver connects to any standard subtractive-decode PCI-ISA bridge device (class code:6, sub-class: 1, programming I/F 0).

Chipset-specific ISA Bus Drivers are used for PCI-ISA (or PCI-LPC) bridges that support positive decode. These bridges have device-specific mechanisms for opening and closing the I/O and memory apertures. These apertures determine which address ranges will be passed through to devices attached to the ISA/LPC side of the bridge. In this case, the registration process includes opening of apertures and guaranteeing that I/O access falls within the addresses that go the specified bus.

The ISA Host Controller is responsible for reporting the actual address and size of the apertures using the DXE GCD services.

14.1.3.2 PEI

The ISA Bus PEIM comes in two versions: generic and chipset-specific.

The generic version is used for subtractive-decode ISA (or LPC) buses. It implements the **EFI_ISA_HC_PPI** with a device identifier of all zeroes. All of the aperture functions report **EFI_UNSUPPORTED**.

The chipset-specific version implements the **EFI_ISA_HC_PPI**, which opens and close apertures for ISA/LPC buses that are positive decode. The device identifier is filled in with the PCI PFA of the PCI-ISA bridge device.

14.1.4 Logical Devices

Logical Device drivers are UEFI driver model drivers that support many of the standard PC/AT peripherals. They are designed to connect to the device paths produced by the Super I/O DXE driver. Each of these drivers produces the Driver Binding and related protocols used in implementing UEFI driver model drivers.

Each of these drivers supports more than one instance of a specific device can be in a system. Calls to **Stop()**and **Start()** will disable or enable the device and stop consumption of all system resources. This allows Super I/O drivers to be loaded and unloaded. These drivers can use the SIO Control protocol to enable consumption of system I/O and memory resources when they are started or stopped.

14.2 Code Definitions

14.2.1 EFI_SIO_PPI

Summary

Super I/O register access.

GUID

```
#define EFI_SIO_PPI_GUID \
  {0x23a464ad, 0xcb83, 0x48b8, \
  {0x94, 0xab, 0x1a, 0x6f, 0xef, 0xcf, 0xe5, 0x22}}
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_PPI {
  EFI_PEI_SIO_REGISTER_READ    Read;
  EFI_PEI_SIO_REGISTER_WRITE   Write;
  EFI_PEI_SIO_REGISTER_MODIFY  Modify;
  EFI_GUID                     SioGuid;
  PEFI_SIO_INFO                Info;
} EFI_SIO_PPI, *PEFI_SIO_PPI;
```

Members

Read

This function reads a register's value from the Super I/O controller.

Write

This function writes a value to a register in the Super I/O controller.

Modify

This function modifies zero or more registers in the Super I/O controller using a table.

SioGuid

This GUID uniquely identifies the Super I/O controller.

Info

This pointer is to an array which maps EISA identifiers to logical devices numbers.

Description

This PPI provides low-level access to Super I/O registers using **Read()** and **Write()**. It also uniquely identifies this Super I/O controller using a GUID and provides mappings between ACPI-style PNP IDs and the logical device numbers. There is one instance of this PPI per Super I/O device.

This PPI is produced by the Super I/O PEIM after the driver has determined that it is present in the system.

Related Definitions

```
typedef struct _EFI_SIO_INFO {
  EFI_ACPI_HID   Hid;
  EFI_ACPI_UID   Uid;
  UINT8          Ldn;
} EFI_SIO_INFO, *PEFI_SIO_INFO;
```

Hid

This is the EISA-style Plug-and-Play identifier for one of the devices on the super I/O controller. The standard values are:

EFI_ACPI_PNP_HID_KBC - 101/102-key Keyboard

EFI_ACPI_PNP_HID_LPT - Standard parallel port

EFI_ACPI_PNP_HID_COM - Standard serial port

EFI_ACPI_PNP_HID_FDC - Standard floppy controller

EFI_ACPI_PNP_HID_MIDI - Standard MIDI controller

EFI_ACPI_PNP_HID_GAME - Standard joystick controller

EFI_ACPI_PNP_HID_END - Specifies the end of the information list.

Uid

This is the unique zero-based instance number for a device on the super I/O. For example, if there are two serial ports, one of them would have a Uid of 0 and the other would have a Uid of 1.

Ldn

This is the Logical Device Number for this logical device in the Super I/O. This value can be used in the **Read()** and **Write()** functions. The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.

```
typedef UINT32 EFI_ACPI_HID;
typedef UINT32 EFI_ACPI_UID;

#define EFI_ACPI_PNP_HID_KBC  EFI_PNP_ID(0x0303)
#define EFI_ACPI_PNP_HID_LPT  EFI_PNP_ID(0x0400)
#define EFI_ACPI_PNP_HID_COM  EFI_PNP_ID(0x0500)
#define EFI_ACPI_PNP_HID_FDC  EFI_PNP_ID(0x0700)
#define EFI_ACPI_PNP_HID_MIDI EFI_PNP_ID(0xB006)
#define EFI_ACPI_PNP_HID_END  EFI_PNP_ID(0x0000)
#define EFI_ACPI_PNP_HID_GAME EFI_PNP_ID(0xB02F)
```

```
#pragma pack(1)
typedef struct _EFI_SIO_INFO {
  EFI_ACPI_HID      Hid;
  EFI_ACPI_UID      Uid;
  UINT8             Ldn;
} EFI_SIO_INFO, *PEFI_SIO_INFO;
#pragma pack()
```

14.2.1.1 EFI_SIO_PPI.Read()

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SIO_REGISTER_READ) (
  IN EFI_PEI_SERVICES     **PeiServices,
  IN CONST EFI_SIO_PPI    *This,
  IN BOOLEAN              ExitCfgMode,
  IN EFI_SIO_REGISTER     Register,
  OUT UINT8               *IoData
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services.

This

A pointer to this instance of the **EFI_SIO_PPI**.

ExitCfgMode

A boolean specifying whether the driver should turn on configuration mode (FALSE) or turn off configuration mode (TRUE) after completing the read operation. The driver must track the current state of the configuration mode (if any) and turn on configuration mode (if necessary) prior to register access.

Register

A value specifying the logical device number (bits 15:8) and the register to read (bits 7:0). The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.

IoData

A pointer to the returned register value.

Description

This function provides low-level read access to a device register. The register is specified as an 8-bit logical device number and an 8-bit register value. The logical device numbers for specific SIO devices can be determined using the Info member of the PPI structure.

If this function completes successfully, it will return **EFI_SUCCESS** and *IoData* will point to the returned Super I/O register value. If the register value was invalid for this device or *IoData* was NULL, then it will return **EFI_INVALID_PARAMETERS**. If the register could not be read within the correct amount of time, it will return **EFI_TIMEOUT**. If the device had some sort of fault or the device was not present, it will return **EFI_DEVICE_ERROR**.

Return Values

This function returns standard EFI status codes.

Status Code Value	Description
EFI_SUCCESS	Success.
EFI_TIMEOUT	The register could not be read in the a reasonable amount of time. The exact time is device-specific.
EFI_INVALID_PARAMETERS	Register was out of range for this device. *IoData* was NULL
EFI_DEVICE_ERROR	There was a device fault or the device was not present.

Related Definitions

```
typedef UINT16 EFI_SIO_REGISTER;

#define EFI_SIO_REG(ldn,reg) (EFI_SIO_REGISTER)(((ldn)<<8)|reg)

#define EFI_SIO_LDN_GLOBAL 0xFF
```

14.2.1.2 EFI_SIO_PPI.Write()

Write a Super I/O register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_SIO_REGISTER_WRITE) (
  IN EFI_PEI_SERVICES      **PeiServices,
  IN CONST EFI_SIO_PPI     *This,
  IN BOOLEAN               ExitCfgMode,
  IN EFI_SIO_REGISTER      Register,
  IN UINT8                 IoData
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services.

This

A pointer to this instance of the **EFI_SIO_PPI**.

ExitCfgMode

A boolean specifying whether the device should turn on configuration mode (FALSE) or turn off configuration mode (TRUE) after completing the write operation. The driver must track the current state of the configuration mode (if any) and turn on configuration mode (if necessary) prior to register access.

Register

A value specifying the logical device number and the register to read. The logical device number can be determined by using the Super I/O chip specification or by looking up the value in the Info field of the **EFI_SIO_PPI**. The logical device number of **EFI_SIO_LDN_GLOBAL** indicates that global registers will be used.

IoData

An 8-bit register value.

Status Code Return

Status Code Value	Description
EFI_SUCCESS	Success.
EFI_TIMEOUT	The register could not be read in the a reasonable amount of time. The exact time is device-specific.
EFI_INVALID_PARAMETERS	Register was out of range for this device. *IoData* was NULL
EFI_DEVICE_ERROR	There was a device fault or the device was not present.

Description

This function provides low-level write access to a Super I/O register.

The register is specified as an 8-bit logical device number and an 8-bit register value. The logical device numbers for specific SIO devices can be determined using the Info member of the PPI structure.

If this function completes successfully, it will return **EFI_SUCCESS** and *IoData* will point to the returned Super I/O register value. If the register value was invalid for this device or *IoData* was NULL, then it will return **EFI_INVALID_PARAMETERS**. If the register could not be read within the correct amount of time, it will return **EFI_TIMEOUT**. If the device had some sort of fault or the device was not present, it will return **EFI_DEVICE_ERROR**.

14.2.1.3 EFI_SIO_PPI.Modify()

Summary

Provides an interface for a table based programming of the Super I/O registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_MODIFY)(
  IN EFI_PEI_SERVICES                 **PeiServices,
  IN CONST EFI_SIO_PPI                *This,
  IN CONST EFI_SIO_REGISTER_MODIFY    *Command,
  IN UINTN                            NumberOfCommands
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services.

This

A pointer to this instance of the **EFI_SIO_PPI**.

Command

A pointer to an array of *NumberOfCommands* **EFI_SIO_REGISTER_MODIFY** structures. Each structure specifies a single Super I/O register modify operation. Type **EFI_SIO_REGISTER_MODIFY** is defined in **EFI_SIO_PROTOCOL.Modify()**.

NumberOfCommands

The number of elements in the Command array.

Description

The **Modify()** function provides an interface for table based programming of the Super I/O registers. This function can be used to perform programming of multiple Super I/O registers with a single function call. For each table entry, the *Register* is read, its content is bitwise ANDed with *AndMask*, and then ORed with *OrMask* before being written back to the *Register*. The Super I/O driver must track the current state of the Super I/O and enable the configuration mode of Super I/O if necessary prior to table processing. Once the table is processed, the Super I/O device must be returned to the original state.

Status Code Return

Status Code Value	Description
EFI_SUCCESS	The operation completed successfully
EFI_INVALID_PARAMETERS	*Command* is NULL

14.2.2 EFI_ISA_HC_PPI

GUID

```
#define EFI_ISA_HC_PPI_GUID \
  {0x8d48bd70, 0xc8a3, 0x4c06, \
  {0x90, 0x1b, 0x74, 0x79, 0x46, 0xaa, 0xc3, 0x58}}
```

PPI Structure

```
typedef struct _EFI_ISA_HC_PPI {
  UINT32 Version;

  UINT32 Address;
  EFI_PEI_ISA_HC_OPEN_IO       OpenIoAperture;
  EFI_PEI_ISA_HC_CLOSE_IO      CloseIoAperture;
} EFI_ISA_HC_PPI, *PEFI_ISA_HC_PPI;
```

Members

Version

An unsigned integer that specifies the version of the PPI structure. Initialized to zero.

PciAddress

The address of the ISA/LPC Bridge device. For PCI, this is the segment, bus, device and function of the a ISA/LPC Bridge device.

If bits 24-31 are 0, then the definition is:

Bits 0:2 – Function

Bits 3-7 – Device

Bits 8:15 – Bus

Bits 16-23 – Segment

Bits 24-31 – Bus Type

If bits 24-31 are 0xff, then the definition is platform-specific.

OpenIoAperture

Opens an aperture on a positive-decode ISA Host Controller.

CloseIoAperture

Closes an aperture on a positive-decode ISA Host Controller.

14.2.2.1 EFI_ISA_HC_PPI.OpenIoAperture()

Open I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ISA_HC_OPEN_IO) (
  IN CONST EFI_ISA_HC_PPI   *This,
  IN UINT16                 IoAddress,
  IN UINT16                 IoLength,
  OUT UINT64                *IoApertureHandle
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services Table.

This

A pointer to this instance of the **EFI_ISA_HC_PPI**.

IoAddress

An unsigned integer that specifies the first byte of the I/O space required.

IoLength

An unsigned integer that specifies the number of bytes of the I/O space required.

IoApertureHandle

A pointer to the returned I/O aperture handle. This value can be used on subsequent calls to **CloseIoAperture()**.

Description

This function opens an I/O aperture in a ISA Host Controller for the I/O addresses specified by *IoAddress* to *IoAddress* + *IoLength* - 1. It is possible that more than one caller may be assigned to the same aperture.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and doesa not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

If this function completes successfully, then it returns **EFI_SUCCESS**. If there is no available I/O aperture, then this function returns **EFI_OUT_OF_RESOURCES**.

14.2.2.2 EFI_ISA_HC_PPI.CloseIoAperture()

Close I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_ISA_HC_CLOSE_IO) (
  IN CONST EFI_ISA_HC_PPI     *This,
  IN UINT64                   IoApertureHandle
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services Table.

This

A pointer to this instance of the **EFI_ISA_HC_PPI**.

IoApertureHandle

The I/O aperture handle previously returned from a call to **OpenIoAperture()**.

Description

This function closes a previously opened I/O aperture handle. If there are no more I/O aperture handles that refer to the hardware I/O aperture resource, then the hardware I/O aperture is closed.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

14.2.3 EFI_ISA_HC_PROTOCOL

Summary

Provides registration and enumeration of ISA devices.

GUID

```
#define EFI_ISA_HC_PROTOCOL_GUID \
  {0xbcdaf080, 0x1bde, 0x4e22, \
  {0xae, 0x6a, 0x43, 0x54, 0x1e, 0x12, 0x8e, 0xc4}}
```

Protocol Interface Structure

```
typedef struct _EFI_ISA_HC_PROTOCOL {
  UINT32 Version;

  EFI_ISA_HC_OPEN_IO        OpenIoAperture;
  EFI_ISA_HC_CLOSE_IO       CloseIoAperture;
} EFI_ISA_HC_PROTOCOL, *PEFI_ISA_HC_PROTOCOL;
```

Members

Version

The version of this protocol. Higher version numbers are backward compatible with lower version numbers. The current version is 0.

OpenIoAperture

Open an I/O aperture.

CloseIoAperture

Close an I/O aperture.

Description

This protocol provides registration for ISA devices on a positive- or subtractive-decode ISA bus. It allows devices to be registered and also handles opening and closing the apertures which are positively-decoded.

14.2.3.1 EFI_ISA_HC_PROTOCOL.OpenIoAperture()

Open I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ISA_HC_OPEN_IO) (
  IN CONST EFI_ISA_HC_PROTOCOL  *This,
  IN UINT16                     IoAddress,
  IN UINT16                     IoLength,
  OUT UINT64                    *IoApertureHandle
  );
```

Parameters

This

A pointer to this instance of the **EFI_ISA_HC_PROTOCOL**.

IoAddress

An unsigned integer that specifies the first byte of the I/O space required.

IoLength

An unsigned integer that specifies the number of bytes of the I/O space required.

IoApertureHandle

A pointer to the returned I/O aperture handle. This value can be used on subsequent calls to **CloseIoAperture()**.

Description

This function opens an I/O aperture in a ISA Host Controller for the I/O addresses specified by *IoAddress* to *IoAddress* + *IoLength* - 1.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

If this function completes successfully, then it returns **EFI_SUCCESS**. If there is no available I/O aperture, then this function returns **EFI_OUT_OF_RESOURCES**.

14.2.3.2 EFI_ISA_HC_PROTOCOL.CloseIoAperture()

Close I/O aperture.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_ISA_HC_CLOSE_IO) (
  IN CONST EFI_ISA_HC_PROTOCOL *This,
  IN UINT64                    IoApertureHandle
  );
```

Parameters

PeiServices

A pointer to a pointer to the PEI Services Table.

This

A pointer to this instance of the **EFI_ISA_HC_PROTOCOL**.

IoApertureHandle

The I/O aperture handle previously returned from a call to **OpenIoAperture()**.

Description

This function closes a previously opened I/O aperture handle. If there are no more I/O aperture handles that refer to the hardware I/O aperture resource, then the hardware I/O aperture is closed.

It may be possible that a single hardware aperture may be used for more than one device. This function tracks the number of times that each aperture is referenced, and does not close the hardware aperture (via **CloseIoAperture()**) until there are no more references to it.

14.2.4 EFI_ISA_HC_SERVICE_BINDING_PROTOCOL

Summary

Manages child devices for an ISA Host Controller.

GUID

```
#define EFI_ISA_HC_SERVICE_BINDING_PROTOCOL_GUID \
  {0xfad7933a, 0x6c21, 0x4234, \
  {0xa4, 0x34, 0x0a, 0x8a, 0x0d, 0x2b, 0x07, 0x81}}
```

Protocol Interface Structure

The protocol interface structure is the same for all service binding protocols and can be found in Section 10.6 (“EFI Service Binding Protocol”).

Description

The ISA Host Controller Service Binding protocol permits multiple Super I/O devices to use the services provide by an ISA Host Controller. The function **CreateChild()** installs an instance of the ISA Host Controller protocol on each child handle created.

14.2.5 EFI_SIO_CONTROL_PROTOCOL

Summary

Provide low-level services for SIO devices that enable them to be used in the UEFI driver model.

GUID

```
#define EFI_SIO_CONTROL_PROTOCOL_GUID \
  {0xb91978df, 0x9fc1, 0x427d, \
  {0xbb, 0x5, 0x4c, 0x82, 0x84, 0x55, 0xca, 0x27}}
```

Protocol Interface Structure

```
typedef struct _EFI_SIO_CONTROL_PROTOCOL {
  UINT32 Version;

  EFI_SIO_CONTROL_ENABLE        EnableDevice;
  EFI_SIO_CONTROL_DISABLE       DisableDevice;
} EFI_SIO_CONTROL_PROTOCOL, PEFI_SIO_CONTROL_PROTOCOL;
```

Members

Version

The version of this protocol. Higher version numbers are backward compatible with lower version numbers. The current version is 0.

EnableDevice

Enable a device.

DisableDevice

Disable a device.

Description

The **EFI_SIO_CONTROL_PROTOCOL** provides control over the decoding of Super I/O and memory resources by a logical device within a Super I/O. While the logical devices often implement industry standard interfaces (such as PS/2 keyboard or serial port), these standard interfaces do not describe how to enable or disable the memory and I/O resources for those devices. Instead, this control is usually implemented within the Super I/O device itself through proprietary means. The industry standard drivers may utilize these functions in their implementations of the Driver Binding protocol's **Start()**and **Stop()**functions.

The Super I/O driver installs this protocol on the same child handle as the **EFI_SIO_PROTOCOL**.

14.2.5.1 EFI_SIO_CONTROL_PROTOCOL.Enable()

Summary

Enable an ISA-style device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_CONTROL_ENABLE)(
  IN CONST EFI_SIO_CONTROL_PROTOCOL *This
  );
```

Parameters

This

A pointer to this instance of the **EFI_SIO_CONTROL_PROTOCOL**.

Description

This function enables a logical ISA device and, if necessary, configures it to default settings, including memory, I/O, DMA and IRQ resources.

If the function completed successfully, then this function returns **EFI_SUCCESS**.

If the device could not be enabled because there were insufficient resources either for the device itself or for the records needed to track the device, then this function returns **EFI_OUT_OF_RESOURCES**.

If this device is already enabled, then this function returns **EFI_ALREADY_STARTED**. If this device cannot be enabled, then this function returns **EFI_UNSUPPORTED**.

14.2.5.2 EFI_SIO_CONTROL_PROTOCOL.Disable()

Summary

Disable a logical ISA device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIO_CONTROL_DISABLE)(
  IN CONST EFI_SIO_CONTROL_PROTOCOL *This
  );
```

Parameters

This

A pointer to this instance of the **EFI_SIO_CONTROL_PROTOCOL**.

Description

This function disables a logical ISA device so that it no longer consumes system resources, such as memory, I/O, DMA and IRQ resources. Enough information must be available so that subsequent **Enable()** calls would properly reconfigure the device.

If this function completed successfully, then it returns **EFI_SUCCESS**.

If the device could not be disabled because there were insufficient resources either for the device itself or for the records needed to track the device, then this function returns **EFI_OUT_OF_RESOURCES**.

If this device is already disabled, then this function returns **EFI_ALREADY_STARTED**. If this device cannot be disabled, then this function returns **EFI_UNSUPPORTED**.

15 CPU I/O Protocol

This document describes the CPU I/O Protocol. This protocol provides an I/O abstraction for a system processor. This protocol is used by a PCI root bridge I/O driver to perform memory-mapped I/O and I/O transactions. The I/O or memory primitives can be used by the consumer of the protocol to materialize bus-specific configuration cycles, such as the transitional configuration address and data ports for PCI. Only drivers that require direct access to the entire system should use this protocol. This is a boot-services only protocol.

15.1 CPU I/O Protocol Terms

The following are the terms that are used throughout this document to describe the CPU I/O Protocol.

coherency domain

The address resources of a system as seen by a processor. It consists of both system memory and I/O space.

CPU I/O Protocol

A software abstraction that provides access to the I/O and memory regions in a single coherency domain.

SMP

Symmetric multiprocessing. A collection of processors that share a common view of I/O and memory-mapped I/O.

15.2 CPU I/O Protocol2 Description

This section describes the CPU I/O Protocol. This protocol is used by code—typically PCI root bridge I/O drivers and drivers that need I/O prior to the loading of the PCI root bridge I/O driver—that is running in the EFI Boot Services environment to access memory and I/O.This protocol can be also used by non-PC-AT* systems to abstract the I/O mechanism published by the processor and/or integrated CPU-I/O complex.

See Code Definitions for the definition of **EFI_CPU_IO_PROTOCOL2**.

15.2.1 EFI CPU I/O Overview

The interfaces that are provided in the **EFI_CPU_IO2_PROTOCOL** are for performing basic operations to memory and I/O. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The **EFI_CPU_IO2_PROTOCOL** allows for future innovation of the platform. It abstracts processor-device-specific code from the system memory map. This abstraction allows system designers to make changes to the system memory map without impacting platform-independent code that is consuming basic system resources.

Systems with one to many processors in a symmetric multiprocessing (SMP) configuration will contain a single instance of the **EFI_CPU_IO2_PROTOCOL**. This protocol is an abstraction from a software point of view. This protocol is attached to the device handle of a processor driver. The CPU I/O Protocol is the parent to a set of PCI Root Bridge I/O Protocol instances that may contain many PCI segments. A CPU I/O Protocol instance might also be the parent of a series of protocols that abstract host-bus attached devices.

CPU I/O Protocol instances are either produced by the system firmware or an EFI driver. When a CPU I/O Protocol is produced, it is placed on a device handle without an EFI Device Path Protocol instance. The figure below shows a device handle that has the **EFI_CPU_IO2_PROTOCOL** installed on it.

Figure 11. EFI CPU I/O2 Protocol

Other characteristics of the CPU I/O Protocol include the following:

- The protocol uses re-entrancy to enable possible use by a debugger agent that is outside of the generic EFI Task Priority Level (TPL) priority mechanism.

See Code Definitions for the definition of **EFI_CPU_IO2_PROTOCOL**.

15.3 Code Definitions

This section contains the basic definitions of the CPU I/O Protocol (**EFI_CPU_IO2_PROTOCOL**).

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent protocol or function definition:

- **EFI_CPU_IO_PROTOCOL_ACCESS**
- **EFI_CPU_IO_PROTOCOL_WIDTH**

15.3.1 CPU I/O Protocol

EFI_CPU_IO2_PROTOCOL

Summary

Provides the basic memory and I/O interfaces that are used to abstract accesses to devices in a system.

GUID

```
#define EFI_CPU_IO2_PROTOCOL_GUID  \
  {0xad61f191, 0xae5f, 0x4c0e, 0xb9, 0xfa, 0xe8, 0x69, 0xd2, \
  0x88, 0xc6, 0x4f}
```

Protocol Interface Structure

```
typedef struct _EFI_CPU_IO2_PROTOCOL {
  EFI_CPU_IO_PROTOCOL_ACCESS       Mem;
  EFI_CPU_IO_PROTOCOL_ACCESS       Io;
} EFI_CPU_IO2_PROTOCOL;
```

Parameters

Mem.Read

Allows reads from memory-mapped I/O space. See the **Mem.Read()** function description. Type **EFI_CPU_IO_PROTOCOL_ACCESS** is defined in "Related Definitions" below.

Mem.Write

Allows writes to memory-mapped I/O space. See the **Mem.Write()** function description.

Io.Read

Allows reads from I/O space. See the **Io.Read()** function description. Type **EFI_CPU_IO_PROTOCOL_ACCESS** is defined in "Related Definitions" below.

Io.Write

Allows writes to I/O space. See the **Io.Write()** function description.

Description

The **EFI_CPU_IO2_PROTOCOL** provides the basic memory and I/O interfaces that are used to abstract accesses to platform hardware. This hardware can include PCI- or host-bus-attached peripherals and buses. There is one **EFI_CPU_IO2_PROTOCOL** instance for each PI System. Embedded systems, desktops, and workstations will typically have only one PI System. Non–symmetric multiprocessing (non-SMP), high-end servers may have multiple PI Systems. A device driver that wishes to make I/O transactions in a system will have to retrieve the **EFI_CPU_IO2_PROTOCOL** instance. A device handle for an PI System will minimally contain an **EFI_CPU_IO2_PROTOCOL** instance.

Related Definitions

```
//*******************************************************
// EFI_CPU_IO2_PROTOCOL_ACCESS
//*******************************************************
typedef struct {
  EFI_CPU_IO_PROTOCOL_IO_MEM  Read;
  EFI_CPU_IO_PROTOCOL_IO_MEM  Write;
} EFI_CPU_IO_PROTOCOL_ACCESS;
```

Read

This service provides the various modalities of memory and I/O read.

Write

This service provides the various modalities of memory and I/O write.

EFI_CPU_IO2_PROTOCOL.Mem.Read() and Mem.Write()

Summary

Enables a driver to access memory-mapped registers in the PI System memory space.

Prototype

```
typedef
EFI_STATUS

(EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN     EFI_CPU_IO2_PROTOCOL                *This,
  IN     EFI_CPU_IO_PROTOCOL_WIDTH           Width,
  IN     UINT64                              Address,
  IN     UINTN                               Count,
  IN OUT VOID                                *Buffer
  );
```

Parameters

This

A pointer to the **EFI_CPU_IO2_PROTOCOL** instance.

Width

Signifies the width of the memory operation. Type **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in "Related Definitions" below.

Address

The base address of the memory operation.

Count

The number of memory operations to perform. The number of bytes moved is *Width* size ** Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The **Mem.Read()** and **Mem.Write()** functions enable a driver to access memory-mapped registers in the PI System memory space.

The memory operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and memory width restrictions that a PI System on a platform might require. For example, on some platforms, width requests of **EfiCpuIoWidthUint64** do not work. Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**, or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16, EfiCpuIoWidthFillUint32, or EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Related Definitions

```
//*******************************************************
// EFI_CPU_IO_PROTOCOL_WIDTH
//*******************************************************
typedef enum {
  EfiCpuIoWidthUint8,
  EfiCpuIoWidthUint16,
  EfiCpuIoWidthUint32,
  EfiCpuIoWidthUint64,
  EfiCpuIoWidthFifoUint8,
  EfiCpuIoWidthFifoUint16,
  EfiCpuIoWidthFifoUint32,
  EfiCpuIoWidthFifoUint64,
  EfiCpuIoWidthFillUint8,
  EfiCpuIoWidthFillUint16,
  EfiCpuIoWidthFillUint32,
  EfiCpuIoWidthFillUint64,
  EfiCpuIoWidthMaximum
} EFI_CPU_IO_PROTOCOL_WIDTH;
```

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PI System.
EFI_INVALID_PARAMETER	*Width* is invalid for this PI System.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this PI System.

EFI_CPU_IO2_PROTOCOL.Io.Read() and Io.Write()

Summary

Enables a driver to access registers in the PI CPU I/O space.

Prototype

```
typedef
EFI_STATUS
 (EFIAPI *EFI_CPU_IO_PROTOCOL_IO_MEM) (
  IN     EFI_CPU_IO2_PROTOCOL              *This,
  IN     EFI_CPU_IO_PROTOCOL_WIDTH        Width,
  IN     UINT64                           Address,
  IN     UINTN                            Count,
  IN OUT VOID                             *Buffer
  );
```

Parameters

This

A pointer to the **EFI_CPU_IO2_PROTOCOL** instance.

Width

Signifies the width of the I/O operation. Type **EFI_CPU_IO_PROTOCOL_WIDTH** is defined in **EFI_CPU_IO2_PROTOCOL.Mem()**.

Address

The base address of the I/O operation. The caller is responsible for aligning the *Address* if required.

Count

The number of I/O operations to perform. The number of bytes moved is *Width* size *** *Count*, starting at *Address*.

Buffer

For read operations, the destination buffer to store the results. For write operations, the source buffer from which to write data.

Description

The *Io.Read()* and **Io.Write()** functions enable a driver to access PCI controller registers in the PI CPU I/O space.

The I/O operations are carried out exactly as requested. The caller is responsible for satisfying any alignment and I/O width restrictions that a PI System on a platform might require. For example on some platforms, width requests of **EfiCpuIoWidthUint64** do not work. Misaligned buffers, on the other hand, will be handled by the driver.

If *Width* is **EfiCpuIoWidthUint8**, **EfiCpuIoWidthUint16**, **EfiCpuIoWidthUint32**, or **EfiCpuIoWidthUint64**, then both *Address* and *Buffer* are incremented for each of the *Count* operations that is performed.

If *Width* is **EfiCpuIoWidthFifoUint8**, **EfiCpuIoWidthFifoUint16**, **EfiCpuIoWidthFifoUint32**, or **EfiCpuIoWidthFifoUint64**, then only *Buffer* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times on the same *Address*.

If *Width* is **EfiCpuIoWidthFillUint8**, **EfiCpuIoWidthFillUint16**, **EfiCpuIoWidthFillUint32**, or **EfiCpuIoWidthFillUint64**, then only *Address* is incremented for each of the *Count* operations that is performed. The read or write operation is performed *Count* times from the first element of *Buffer*.

Status Codes Returned

EFI_SUCCESS	The data was read from or written to the PI System.
EFI_INVALID_PARAMETER	*Width* is invalid for this PI System.
EFI_INVALID_PARAMETER	*Buffer* is **NULL**.
EFI_UNSUPPORTED	The *Buffer* is not aligned for the given *Width*.
EFI_UNSUPPORTED	The address range specified by *Address*, *Width*, and *Count* is not valid for this PI System.

16 Legacy Region Protocol

This section describes the legacy region protocol that abstracts the platform capability for the BIOS memory region from 0xC0000 to 0xFFFFF. The Legacy Region Protocol is used to abstract the hardware control of the Option ROM and Compatibility 16-bit code region shadowing.

16.1 Legacy Region Protocol

The Legacy Region Protocol controls the read, write and boot-lock attributes for the region 0xC0000 to 0xFFFFF. The table below lists the functions that are included in the Legacy Region Protocol. See **EFI_LEGACY_REGION2_PROTOCOL** in Code Definitions for the definitions of these functions.

Table 24. Functions in Legacy Region Protocol

Function	Description
Decode()	Programs the chipset to decode or not decode regions in the 0xC0000 to 0xFFFFF range. Governs the read attribute.
Lock()	Programs the chipset to lock (write protect) regions in the 0xC0000 to 0xFFFFF range. Disables the write attribute.
BootLock()	Programs the chipset to boot-lock regions in the 0xC0000 to 0xFFFFF range. Enables the boot-lock attribute.
Unlock()	Programs the chipset to unlock regions in the 0xC0000 to 0xFFFFF range. Enables the write attribute.
GetInfo()	Get information about the granularity of the regions for each attribute.

16.2 Code Definitions

16.2.1 Legacy Region Protocol

EFI_LEGACY_REGION2_PROTOCOL

Summary

Abstracts the hardware control of the physical address region 0xC0000–0xFFFFF.

GUID

```
#define EFI_LEGACY_REGION2_PROTOCOL_GUID \
  { 0x70101eaf, 0x85, 0x440c, 0xb3, 0x56, 0x8e, 0xe3, 0x6f,\
  0xef, 0x24, 0xf0 }
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_REGION2_PROTOCOL {
  EFI_LEGACY_REGION2_DECODE                  Decode;
  EFI_LEGACY_REGION2_LOCK                    Lock;
  EFI_LEGACY_REGION2_BOOT_LOCK               BootLock;
  EFI_LEGACY_REGION2_UNLOCK                  UnLock;
  EFI_LEGACY_REGION_GET_INFO                 GetInfo;
} EFI_LEGACY_REGION2_PROTOCOL;
```

Parameters

Decode

Modify the read attribute of a memory region. See the **Decode()** function description.

Lock

Modify the write attribute of a memory region to prevent writes. See the **Lock()** function description.

BootLock

Modify the boot-lock attribute of a memory region to prevent future changes to the memory attributes for this region. See the **BootLock()** function description.

Unlock

Modify the write attribute of a memory region to allow writes. See the **Unlock()** function description.

GetInfo

Modify the write attribute of a memory region to allow writes. See the **GetInfo()** function description.

Description

The **EFI_LEGACY_REGION2_PROTOCOL** is used to abstract the hardware control of the memory attributes of the Option ROM shadowing region, 0xC0000 to 0xFFFFF.

There are three memory attributes that can be modified through this protocol: read, write and boot-lock. These protocols may be set in any combination.

EFI_LEGACY_REGION2_PROTOCOL.Decode()

Summary

Modify the hardware to allow (decode) or disallow (not decode) memory reads in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_DECODE) (
  IN   EFI_LEGACY_REGION2_PROTOCOL   *This,
  IN   UINT32                        Start,
  IN   UINT32                        Length,
  OUT  UINT32                        *Granularity ,
  IN   BOOLEAN                       *On
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

The beginning of the physical address of the region whose attributes should be modified.

Length

The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

On

Decode / Non-Decode flag.

Description

If the On parameter evaluates to **TRUE**, this function enables memory reads in the address range Start to (Start + Length - 1).

If the *On* parameter evaluates to **FALSE**, this function disables memory reads in the address range Start to (Start + Length - 1).

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.Lock()

Summary

Modify the hardware to disallow memory writes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_LOCK) (
  IN  EFI_LEGACY_REGION2_PROTOCOL *This,
  IN  UINT32                      Start,
  IN  UINT32                      Length,
  OUT UINT32                      *Granularity
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

The beginning of the physical address of the region whose attributes should be modified.

Length

The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function changes the attributes of a memory range to not allow writes.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.BootLock()

Summary

Modify the hardware to disallow memory attribute changes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_BOOT_LOCK) (
  IN   EFI_LEGACY_REGION2_PROTOCOL   *This,
  IN   UINT32                        Start,
  IN   UINT32                        Length,
  OUT  UINT32                        *Granularity
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

The beginning of the physical address of the region whose attributes should be modified.

Length

The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function makes the attributes of a region read only. Once a region is boot-locked with this function, the read and write attributes of that region cannot be changed until a power cycle has reset the boot-lock attribute. Calls to **Decode()**, **Lock()** and **Unlock()** will have no effect.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully locked.
EFI_INVALID_PARAMETER	If Start or Length describe an address not in the Legacy Region.
EFI_UNSUPPORTED	The chipset does not support locking the configuration registers in a way that will not affect memory regions outside the legacy memory region.

EFI_LEGACY_REGION2_PROTOCOL.UnLock()

Summary

Modify the hardware to allow memory writes in a region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION2_UNLOCK) (
  IN  EFI_LEGACY_REGION2_PROTOCOL  *This,
  IN  UINT32                       Start,
  IN  UINT32                       Length,
  OUT UINT32                       *Granularity
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

Start

The beginning of the physical address of the region whose attributes should be modified.

Length

The number of bytes of memory whose attributes should be modified. The actual number of bytes modified may be greater than the number specified.

Granularity

The number of bytes in the last region affected. This may be less than the total number of bytes affected if the starting address was not aligned to a region's starting address or if the length was greater than the number of bytes in the first region.

Description

This function changes the attributes of a memory range to allow writes.

Status Codes Returned

EFI_SUCCESS	The region's attributes were successfully modified.
EFI_INVALID_PARAMETER	If *Start* or *Length* describe an address not in the Legacy Region.

EFI_LEGACY_REGION2_PROTOCOL.GetInfo()

Summary

Get region information for the attributes of the Legacy Region.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_REGION_GET_INFO) (
  IN  EFI_LEGACY_REGION2_PROTOCOL    *This,
  OUT UINT32                         *DescriptorCount,
  OUT EFI_LEGACY_REGION_DESCRIPTOR   **Descriptor
  );
```

Parameters

This

Indicates the **EFI_LEGACY_REGION2_PROTOCOL** instance.

DescriptorCount

The number of region descriptor entries returned in the *Descriptor* buffer. Type **EFI_LEGACY_REGION_DESCRIPTOR** is defined in the "Related Definitions" section.

Descriptor

A pointer to a pointer used to return a buffer where the legacy region information is deposited. This buffer will contain a list of *DescriptorCount* number of region descriptors. This function will provide the memory for the buffer.

Description

This function is used to discover the granularity of the attributes for the memory in the legacy region. Each attribute may have a different granularity and the granularity may not be the same for all memory ranges in the legacy region.

Status Codes Returned

EFI_SUCCESS	The information structure was returned.
EFI_UNSUPPORTED	This function is not supported.

Related Definitions

```
typedef enum {
  LegacyRegionDecoded,
  LegacyRegionNotDecoded,
  LegacyRegionWriteEnabled,
  LegacyRegionWriteDisabled,
  LegacyRegionBootLocked,
  LegacyRegionNotLocked
} EFI_LEGACY_REGION_ATTRIBUTE;
```

LegacyRegionDecoded

This region is currently set to allow reads.

LegacyRegionNotDecoded

This region is currently set to not allow reads.

LegacyRegionWriteEnabled

This region is currently set to allow writes.

LegacyRegionWriteDisabled

This region is currently set to write protected.

LegacyRegionBootLocked

This region’s attributes are locked, cannot be modified until after a power cycle.

LegacyRegionNotLocked

This region’s attributes are not locked.

```
typedef struct {
  UINT32                          Start;
  UINT32                          Length;
  EFI_LEGACY_REGION_ATTRIBUTE     Attribute;
  UINT32                          Granularity;
} EFI_LEGACY_REGION_DESCRIPTOR;
```

Start

The beginning of the physical address of this region.

Length

The number of bytes in this region.

Attribute

Attribute of the Legacy Region Descriptor that describes the capabilities for that memory region.

Granularity

Describes the byte length programmability associated with the ***Start*** address and the specified ***Attribute*** setting.

17 I^{2}C Protocol Stack

17.1 Design Discussion

The Inter-Integrated Circuit (I^{2}C) protocol stack enables third party silicon vendors to write UEFI drivers for their products by decoupling the I^{2}C chip details from the I^{2}C controller and I^{2}C bus configuration details.

17.1.1 I^{2}C Bus Overview

The Inter-Integrated Circuit (I^{2}C) bus enables simple low speed communications between chips. The following sections describe the attributes of the I^{2}C bus configurations supported by the I^{2}C protocol stack and the **I2C-bus specification and user manual**.

17.1.1.1 Single Master

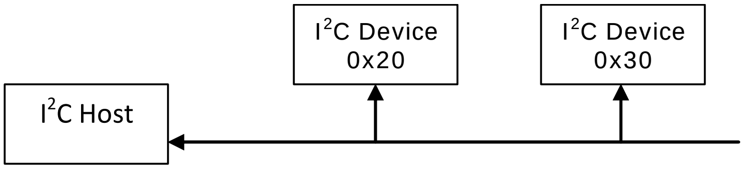


Figure 12. Simple 1^{2}C Bus

Figure 12 shows a simple I^{2}C bus configuration consisting of one host controller and two I^{2}C devices which use the same I^{2}C clock frequency. In this configuration the I^{2}C host controller gets initialized with a single clock frequency and performs transactions to the I^{2}C devices using their slave addresses.

17.1.1.2 Multiple I^{2}C Bus Frequencies

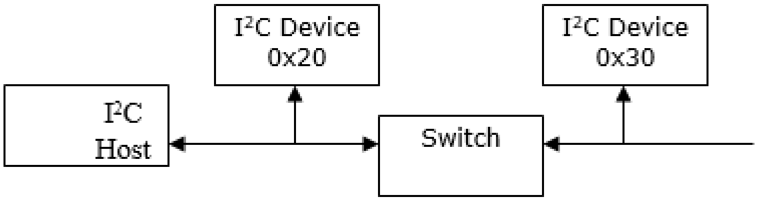


Figure 13. Multiple I^{2}C Bus Frequencies

Two I^{2}C bus configurations are shown in Figure 13, separated by a switch. This allows the I^{2}C bus to operate at two different frequencies depending on the state of the switch. Device requiring higher bus frequencies are placed closer to the I^{2}C host controller and are accessed when the switch is turned off. Devices using lower bus frequencies are placed after the switch and may only be accessed when the switch is on. Note that the I^{2}C bus frequency needs to be set to a frequency supported by all devices currently accessible by the I^{2}C host controller.

17.1.1.3 Limited Address Space

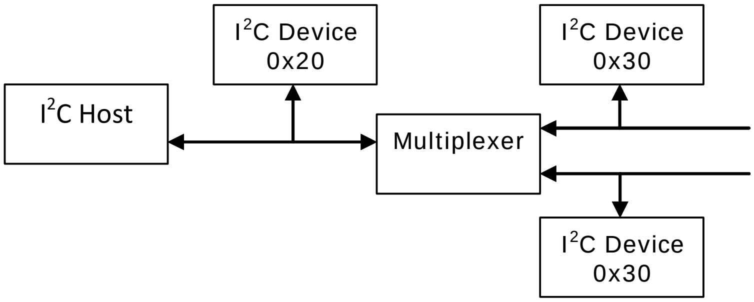


Figure 14. Limited address Space

I^{2}C devices have a limited number of address settings, sometimes only one. When the hardware design requires more I^{2}C devices than the address space supports a multiplexer may be introduced to create additional bus configurations (address spaces). Note that the host must first select the appropriate bus configuration before communicating with the I^{2}C device.

17.1.1.4 I^{2}C Bus Configurations

A bus configuration is a concept introduced by the I^{2}C protocol stack to configure the state of the switches and multiplexers in the I^{2}C bus. The I^{2}C protocol stack calls into the platform code with a value from zero (0) to N-1 to request the platform code enable a specific configuration of the switches and multiplexers. The platform code then sets the requested state for the switches and multiplexers and sets the I^{2}C clock frequency for this I^{2}C bus configuration. Upon return the I^{2}C protocol stack is able to access the I^{2}C devices in this configuration.

17.1.2 ^{2}C Protocol Stack Overview

The following is a representation of the I^{2}C protocol stack and an I^{2}C bus layout.

```
+-----------------+
|  Application  |
+-----------------+
        |
        | Third Party or UEFI
        |
        V
+-----------------+
|  Third Party  |
|  I2C Device   |
|    Driver     |
+-----------------+
        |
        |
        V
+-----------------+
| I2C IO Protocol |
+-----------------+
        |
        |
        V
+-----------------+   +------------------------+
| I2C Bus Driver  |-->| I2C Enumerate Protocol |
+-----------------+   +------------------------+
        |
        |
        V
+-------------------+
| I2C Host Protocol |---------.
+-------------------+         |
        |                 V
        |          +----------------------+
        |          | I2C Bus Configuration |
           |<-----------|  Management Protocol  |
           V           +-----------------------+
+---------------------+            |     |
| I2C Master Protocol |
+---------------------+            |     |
           |
Software   |                       |     |
-----------------------------------------------
Hardware   |                       |     |
           |
           V                       |     |
  +-----------------+
  | I2C Controller  |              |     |
  +-----------------+
           |                       |     |
----------------------
I2C Bus    |                       |     |
           |    +------------+
           +----| High speed |     |     |
           |    | I2C device |
           |    |    0x01    |     |     |
           |    +------------+
           |                       |     |
      +---------+  0
      | Switch  |<- - - - - - - - - `    |
      +---------+  1
           |                             |
           |    +------------+
           +----| Fast speed |           |
           |    | I2C device |
           |    |    0x02    |           |
           |    +------------+
           |                             |
    +-------------+
    | Multiplexer |< - - - - - - - - - - - `
    +-------------+
0 |     | 1
  |     |
  |     |
  |     |  +-------------+
  |     +----| Third Party |
  |     |  | I2C Device  |
  |     |  |  0x03, 0x04 |
  |     |  +-------------+
  |     |
  |
  |        +-------------+
  +------------| Third Party |
  |        | I2C Device  |
  |        |  0x03, 0x04 |
  |        +-------------+
  |
```

Figure 15. I^{2}C Protocol Stack

The platform hardware designer chooses the bus layout based upon the platform, I^{2}C chip and software requirements. The design uses switches to truncate the bus to enable higher bus frequencies for a subset of devices which are placed closer to the controller. When the switch is on, the extended bus must operate at a lower bus frequency. The design uses multiplexer to create separate address spaces enabling the use of multiple devices which would otherwise have conflicting addresses. See the **I2C-bus specification and user manual** for more details.

N.B. Some operating systems may prohibit the changing of switches and multiplexers in the I^{2}C bus. In this case the platform hardware and software designers must select a single I^{2}C bus configuration consisting of constant input values for the switches and multiplexers. The I^{2}C subsystem must be placed in the OS compatible I^{2}C bus configuration upon successful completion of **`ExitBootServices()`**.

The platform hardware designer needs to provide the platform software designer the following data for each I^{2}C bus:

1. Which controller controls this bus
2. A list of logic blocks contained in one or more I^{2}C devices:
    - I^{2}C device which contains this logic block
    - Logic block I^{2}C slave address
    - Logic block description
3. For each configuration of the switches and multiplexers in the I^{2}C bus

- What is the maximum frequency of operation for the I^{2}C bus
- What I^{2}C slave addresses are accessible

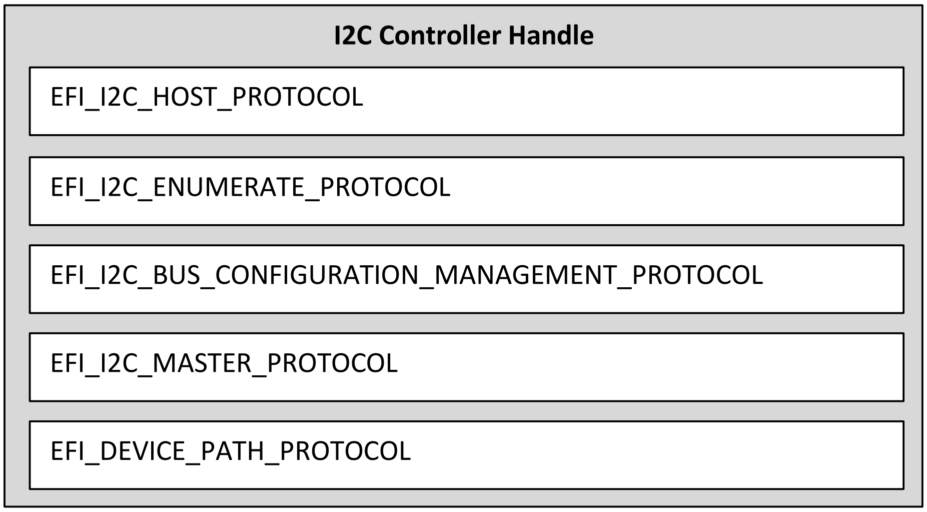
4. The settings for the switches and multiplexers when control is given to the operating system.

17.1.2.1 Handles

The I^{2}C protocol stack uses two groups of handles:

- I^{2}C controller handles
- I^{2}C device handles

Some bus driver (PCI, USB, etc.) or the platform specific code may expose a handle for each of the I^{2}C controllers. The platform specific code installs the I^{2}C bus configuration management and I^{2}C enumeration protocols on the controller handle. As the I^{2}C stack is initialized, additional protocols are placed on the I^{2}C controller handle. When the I^{2}C stack initialization is complete, the controller handle contains:


I2C Controller Handle

EFI_I2C_HOST_PROTOCOL

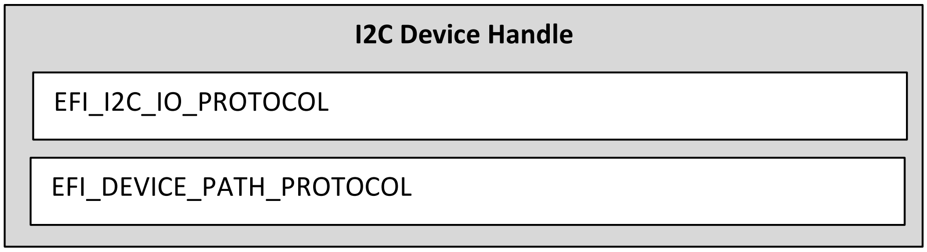
EFI_I2C_ENUMERATE_PROTOCOL

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL

EFI_I2C_MASTER_PROTOCOL

EFI_DEVICE_PATH_PROTOCOL

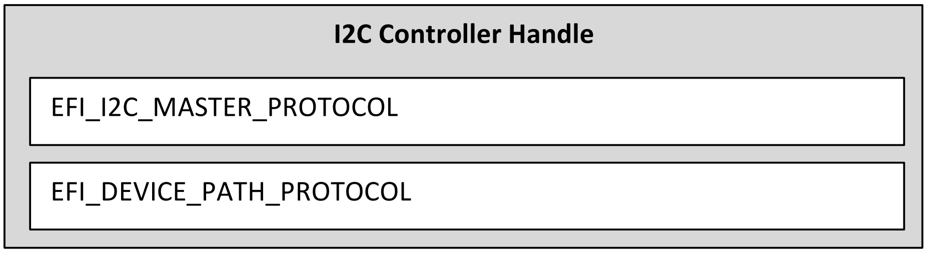
The I^{2}C Bus Driver uses the **EFI_I2C_ENUMERATE_PROTOCOL** to enumerate the set of I^{2}C devices connected to an I^{2}C controller, and creates an I^{2}C device handle for each I^{2}C device installing the following protocols on each:


I2C Device Handle

EFI_I2C_IO_PROTOCOL

EFI_DEVICE_PATH_PROTOCOL

It is possible for the SMBus Host Controller Protocol to be implemented using the services on an I^{2}C Controller Handle. The SMBus Host Controller Protocol does not support the concept of multiple bus configurations, so the state of the I^{2}C controller handle required for the SMBus Host Controller Protocol to be produced on an I^{2}C Controller Handle is as follows:



17.1.2.2 Driver Loading Order

A race condition potentially exists between the platform specific code and a layered SMBus driver when a driver for a PCI or USB I^{2}C controller installs the **EFI_I2C_MASTER_PROTOCOL** on its handle. The layered SMBus driver may start on this controller as soon as the **EFI_I2C_MASTER_PROTOCOL** is installed as long as the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** is not installed on the controller handle. However if the platform specific code wants to use this controller with the **EFI_I2C_HOST_PROTOCOL** then the platform specific code needs to prevent the SMBus driver from starting by installing the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL**. Note that the I^{2}C host protocol opens the **EFI_I2C_MASTER_PROTOCOL** only if the handle contains the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL**.

Chapter 10 of the *Universal Extensible Firmware Interface Specification* describes several ways for the platform specific code to adjust the driver load order. One possible way to eliminate this race condition is to use the version number for the driver binding protocol. The platform specific code implements the driver binding protocol's *Supported()* and *Start()* routines and sets the *version* field to a value in the range of **0xfffffff0 – 0xffffffff**. The SMBus driver should set the version field of the driver binding protocol to a value in the range of **0x00000010 – 0xffffffef**. This selection delays the SMBus driver to execute its *Supported()* and *Start()* routines after the platform specific code, enabling the platform specific code to install the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** and the **EFI_I2C_ENUMERATE_PROTOCOL** on the controller's handle.

17.1.2.3 Third Party I^{2}C Drivers

Third party I^{2}C drivers are I^{2}C chip specific but platform and host controller independent.

Third party I^{2}C driver writers, typically silicon vendors, need to provide:

- The vendor specific GUID that is used to select their driver.
- I^{2}C slave address array guidance (described below) when the I^{2}C device uses more than one I^{2}C slave address consisting of the order for the blocks of logic that get referenced by the entries in the slave address array.

The hardware version of the I^{2}C device, this value is passed to the third party I^{2}C driver to enable it to perform workarounds for the specific hardware version. It is recommended that value match the value in the ACPI _HRV tag. See the **Advanced Configuration and Power Interface Specification, Revision 5.0** for the field format and the **Plug and play support for I2C** web-page for restriction on values.

The third party I^{2}C driver uses relative addressing to abstract the platform specific details of the I^{2}C device. Using an example I^{2}C device containing an accelerometer and a magnetometer which consumes two I^{2}C slave addresses, one for each logic block. The third party I^{2}C driver writer may choose to write two drivers, one for each block of logic, in which case each driver refers to the single I^{2}C slave address using the relative value of zero (0). However if the third party I^{2}C driver writer chooses to write a single driver which consumes multiple I^{2}C slave addresses then the third party I^{2}C driver writer needs to convey the order of the I^{2}C slave address entries in the I^{2}C slave address array to the platform software designer. For the example:

0: Accelerometer

1: Magnetometer

The platform hardware designer picks the actual slave addresses from the I^{2}C device's data sheet and provides this information to the platform software designer. The platform software designer then places the I^{2}C slave addresses into the I^{2}C slave address array in the **EFI_I2C_ENUMERATE_PROTOCL** in the order specified by the third party I^{2}C driver writer. The third party I^{2}C driver writer uses the index into the I^{2}C slave address array as the relative I^{2}C slave address. The I^{2}C IO protocol uses the I^{2}C slave address array to translate the relative I^{2}C slave address into the platform specific I^{2}C slave address. The relative value always starts at zero (0) and its maximum value is the number of entries in I^{2}C slave address array minus one.

Each I^{2}C slave address entry is specified as a 32-bit integer to allow room for future I^{2}C slave address expansion. Only the I^{2}C master protocol knows the maximum I^{2}C slave address value. All other drivers and applications must look for the **EFI_NOT_FOUND** status for the indication that a reserve bit was set in the I^{2}C slave address.

31	30 8	7 0
0	Reserved (Must Be Zero)	7-bit Slave Address

31	30 10	9 0
1	Reserved (Must Be Zero)	10-bit Slave Address

17.1.2.3.1 Driver Binding Protocol Supported() API

The driver binding protocol's *Supported()* routine looks for controllers which declare the **EFI_I2C_IO_PROTOCOL** and match the device path supplied by the silicon vendor or third party I^2C driver writer to the platform integrator.

The third party I^2C device driver creates a GUID for a Vendor-Defined Hardware Device Path Node when describing the I^2C device. The third party I^2C device driver writer provides this GUID to the person writing the platform specific code to identify the type of I^2C device.

The third party I^2C driver which consumes the **EFI_I2C_IO_PROTOCOL** compares the known GUID with the GUID pointed to by the *DeviceGuid* field.

An example algorithm for the driver binding protocol *Supported()* routine:

1. Open the **EFI_I2C_IO_PROTOCOL** using **EFI_OPEN_PROTOCOL_BY_DRIVER**
2. If OpenProtocol() fails return the error status
3. Get the vendor GUID from the **EFI_I2C_IO_PROTOCOL**
4. Close the **EFI_I2C_IO_PROTOCOL**
5. Compare the expected vendor GUID to the GUID from the **EFI_I2C_IO_PROTOCOL** structure.
6. If the GUIDS don't match then return **EFI_NOT_SUPPORTED**
7. Return **EFI_SUCCESS**

17.1.2.3.2 Supporting Multiple Hardware Versions

Note that package markings are important to allow the platform integrator to verify the hardware revision after the part is integrated! The platform integrator includes the hardware revision information into the **EFI_I2C_ENUMERATE_PROTOCOL**. The I^2C bus driver gets this data during the I^2C device enumeration and makes it available to the third party I^2C device driver via the **EFI_I2C_IO_PROTOCOL**. There are a couple of ways in which the silicon vendor or third party I^2C driver writer may support multiple hardware versions of the I^2C device:

- Provide a different GUID value to the platform integrator for each hardware revision
- Provide a different hardware version value to the platform integrator with the devices

Each of the above methods describes an interface to the I^2C device. The interface specifies the number of slave addresses as well as the features and software workarounds for the I^2C device.

17.1.2.4 I^2C IO Protocol

The I2C IO protocol is platform, host controller, and I^2C chip independent.

The I^2C bus driver creates a handle for each of the I^2C devices returned by the I^2C enumerate protocol. The I^2C controller's device path is extended with the vendor GUID and unique ID value returned by the I^2C enumerate protocol and attached to the handle. The vendor GUID is used to extend the device path with a Vendor-define Hardware Device Path Node and the unique ID is used to further extend the device path with a Controller Device Path Node. If the unique ID is 0, then the

Controller Device Path Node is optional. The third party I²C device driver uses the device GUID to determine if it may connect.

When a third party I²C device driver or application calls `QueueRequest()`, the I²C IO protocol validates the `SlaveAddressIndex` (relative I²C address) for the I²C device and then converts the `SlaveAddressIndex` to a I²C slave address. The request is then passed to the I²C host protocol along with the tuple BusConfiguration:I²C slave address.

17.1.2.5 I²C Host Protocol

The I²C host protocol is platform, host controller, and I²C chip independent.

Note: *For proper operation of the I²C bus, only the I²C IO protocol and I²C test applications connect to the* **`EFI_I2C_HOST_PROTOCOL`**.

The I²C host protocol may access any device on the I²C bus. The I²C host protocol has the following responsibilities:

- Limits the number of requests to the I²C master protocol to one. The I²C host protocol holds on to additional requests until the I²C master protocol is available to process the request. The I²C requests are issued in FIFO order to the I²C master protocol.
- Enable the proper I²C bus configuration before starting the I²C request using the I²C master protocol

I²C devices are addressed as the tuple: BusConfiguration:SlaveAddress. I²C bus configuration zero (0) is the portion of the I²C bus that connects to the host controller. The bus configuration specifies the control values for the switches and multiplexers in the I²C bus. After the switches and multiplexers are properly configured, the I²C controller uses the slave address to access the requested I²C device.

Since the I²C protocol stack supports asynchronous transactions the I²C host protocol maintains a queue of I²C requests until the I²C controller is available them. When a request reaches the head of the queue the necessary bus configuration is enabled and then the request is sent to the I²C master protocol.

17.1.2.6 I²C Master Protocol

The I2C master protocol is I2C controller specific but platform independent.

This protocol is designed to allow the implementation to be built as a driver which may be delivered in binary form as an EFI image.

The master protocol manipulates the I2C controller to perform a transaction on the I2C bus. The I2C master protocol does not configure the I2C bus so it is up to the caller to ensure that the I2C bus is in the proper configuration before issuing the I²C request.

The I²C master protocol typically needs the following information:

- Host controller address

- Controller's input clock frequency

Depending upon the I2C controller, more data may be necessary. This protocol may use any method to get these values: hard coded values, PCD values, or may choose to communicate with the platform specific code using an undefined mechanism to get these values.

If the I2C master protocol requires data from the platform specific code then the I2C master protocol writer needs to provide the platform interface details to the platform software designer.

17.1.2.7 Platform Specific Code

The platform specific code installs the **EFI_I2C_ENUMERATE_PROTOCOL** to provide the I2C device descriptions to the I2C bus driver using the **EFI_I2C_DEVICE** structure. These descriptions include the bus configuration number required for the I2C device, the slave address array, the vendor GUID and a unique ID value.

The **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** enables the I2C host protocol to call into the platform specific code to enable a specific I2C bus configuration and set the I2C bus frequency. This protocol is required to get the I2C host protocol to start for the I2C controller's handle.

The platform software designer collects the data requirements from third party I2C driver writers, the vendor specific I2C master protocol writer, the **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** and **EFI_I2C_ENUMERATE_PROTOCOL**. The platform software designer gets the necessary data from the platform hardware designer. The platform software designer then builds the data structures and implements the necessary routines to construct the platform specific code for I^{2}C.

17.1.2.8 Switches and Multiplexers

There are some I2C switches and I2C multiplexers where the control is done via I2C commands. When the control inputs come via the same I^{2}C bus that is being configured then the platform specific code must use the **EFI_I2C_MASTER_PROTOCOL**. While the I^{2}C host protocol makes the call to *EnableI2cBusConfiguration* to configure the I^{2}C bus, the I^{2}C host protocol keeps the I2C master protocol idle, enabling the platform specific code to perform the necessary I2C configuration transactions.

If however the configuration control is done via an I2C device connected to a different I2C bus (host controller), then the platform software designer may choose between the following:

- Call into a third party I^{2}C driver to manipulate the I^{2}C bus control device.
- Call into the **EFI_I2C_IO_PROTOCOL** if no third party I^{2}C driver exists for the I^{2}C bus control device
- Call into the **EFI_I2C_HOST_PROTOCOL** if the platform does not expose the I^{2}C bus control device.

17.1.3 PCI Comparison

PCI provides several features to describe the device to the operating system as well decoupling the driver from the specific platform.

17.1.3.1 Device Description

PCI uses the *Vendor ID* and *Device ID* fields in configuration space to identify the piece of hardware. Where the Vendor ID is assigned by the PCI committee and the Device ID is assigned by the hardware manufacture.

PCI also uses the *Base Class*, *Sub Class* and *Programming Interface* fields to help identify the operating system driver.

The I^2C protocol stack uses the vendor GUID associated with the I^2C device to identify the UEFI driver. This GUID is supplied by the silicon vendor or third party I^2C driver writer to the platform integrator and gets included in the I^2C platform driver. The **EFI_I2C_ENUMERATE_PROTOCOL** provides this GUID to the I^2C bus driver during the I^2C bus enumeration.

The driver binding protocol's *Supported()* routine of the third party I^2C device driver looks for controllers which have the **EFI_I2C_IO_PROTOCOL** and have a match for the vendor GUID.

17.1.3.2 Hardware Features and Workarounds

PCI provides a *Revision ID* field to allow the driver to determine which version of hardware is present and which features and software workarounds are necessary to support this device.

The I^2C protocol stack uses the *HardwareRevision* field in the **EFI_I2C_IO_PROTOCOL** for this same purpose. It is recommended that this value match the _HRV value in the DSDT for this I^2C device. See the **Advanced Configuration and Power Interface Specification, Revision 5.0** for the field format and the **Plug and play support for I2C** web-page for restriction on values.

17.1.3.3 Device Relative Addressing

PCI provides **Base Address Registers** (BARs) to decouple the device driver software from the details of the platform's PCI bus configuration. Typically, all device register references are fixed offsets from one of the BAR addresses.

The I^2C protocol stack provides a similar mechanism using an index into an array of slave addresses. The silicon vendor or third party driver writer provides the structure of the array listing the major functions to the platform integrator. An example is:

0: Accelerometer

1: Compass

The platform integrator works with the platform's hardware designer to get the I^2C slave addresses of the I^2C device and builds the array which is included in the platform specific code. During I^2C device enumeration, this array is passed to the I^2C bus driver for use by the I^2C IO protocol.

The third party I^2C driver references the major components within the I^2C device using the index values, thus remaining platform independent. The I^2C IO protocol performs the array lookup, translating the index into an actual slave address on the I^2C bus.

Most I^2C devices only have a single I^2C slave address and thus the third party I^2C device driver will only use index zero (0). Also depending upon the I^2C device architecture, the silicon vendor or third party I^{2}C device writer may choose to write multiple drivers, each supporting a single I^{2}C slave address.

17.1.4 Hot Plug Support

I^{2}C protocol stack enables the platform specific code to support hot-plug with the following algorithm:

1. Describe all possible devices on all possible busses, including the hot-plug devices.
2. The platform specific code detects hot-plug events: Add and Remove
3. For a removal event:
   - The platform specific code opens the **EFI_I2C_IO_PROTOCOL** on the hot-plug device's handle exclusively. This operation tears down any upper layer protocols on this handle. Note that the open request may fail if I/O is pending in the lower protocols.
   - When the step above fails, delay below **TPL_NOTIFY** to allow the current I
   - ^{2}C transaction complete and then retry until the open is successful
   - After the open is successful, the platform specific code may use the I
   - ^{2}C IO protocol to perform I^{2}C transactions for device probing.
4. For an add event:
   - The platform specific code waits for completion any outstanding I/O that the platform specific code initiated on the hot-plug I^{2}C device.
   - The platform specific code closes the **EFI_I2C_IO_PROTOCOL**
   - The platform specific code issues a *ConnectController()* on the hot-plug device's handle. This causes the protocol stack which uses the hot-plug device to be reloaded.

17.2 DXE Code definitions

The I^{2}C protocol stack consists of the following protocols:

- **EFI_I2C_IO_PROTOCOL** – Third party silicon vendors use this protocol to access their I^{2}C device. This protocol enables a driver or application to perform I/O transactions to a single I^{2}C device independent of the I^{2}C bus configuration.
- **EFI_I2C_HOST_PROTOCOL** – The I^{2}C bus driver uses this protocol to produce the **EFI_I2C_IO_PROTOCOL** that provides access a device on the I^{2}C bus.
- **EFI_I2C_MASTER_PROTOCOL** – The I^{2}C host protocol uses this protocol to manipulate the I^{2}C host controller and perform transactions as a master on the I^{2}C bus.
- **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** – The I^{2}C host protocol uses this protocol to request the proper state for the switches and multiplexers in the I^{2}C bus and set the I^{2}C clock frequency.

- **EFI_I2C_ENUMERATE_PROTOCOL** – The I^{2}C bus driver uses this protocol to enumerate the devices on the I^{2}C bus, getting the bus configuration and an array of slave addresses for each of the I^{2}C devices.

The following sections describe these protocols in detail.

17.2.1 I^{2}C Master Protocol

EFI_I2C_MASTER_PROTOCOL

Summary

This protocol manipulates the I2C host controller to perform transactions as a master on the I2C bus using the current state of any switches or multiplexers in the I2C bus.

GUID

```
#define EFI_I2C_MASTER_PROTOCOL_GUID \
{ 0xcd72881f, 0x45b5, 0x4feb, { 0x98, 0xc8, 0x31, 0x3d, \
0xa8, 0x11, 0x74, 0x62 }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_MASTER_PROTOCOL {
  EFI_I2C_MASTER_PROTOCOL_SET_BUS_FREQUENCY  SetBusFrequency;
  EFI_I2C_MASTER_PROTOCOL_RESET              Reset;
  EFI_I2C_MASTER_PROTOCOL_START_REQUEST      StartRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES *I2cControllerCapabilities;
} EFI_I2C_MASTER_PROTOCOL;
```

Parameters

SetBusFrequency

Set the clock frequency for the I^{2}C bus.

Reset

Reset the I^{2}C host controller.

StartRequest

Start an I^{2}C transaction in master mode on the host controller.

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

The **EFI_I2C_MASTER_PROTOCOL** is typically used by the I^{2}C host protocol to perform transactions on the I^{2}C bus. This protocol may also be used to configure the I^{2}C clock frequency and use I^{2}C transactions to set the state of switches and multiplexers in the I^{2}C bus.

Related Definitions

31	30 8	7 0
0	Reserved (Must Be Zero)	7-bit Slave Address

31	30 10	9 0
1	Reserved (Must Be Zero)	10-bit Slave Address

A 10-bit slave address is or'ed with the following value enabling the I^{2}C protocol stack to address the duplicated address space between 0 and 127 in 10-bit mode.

```
#define I2C_ADDRESSING_10_BIT     0x80000000
```

The I^{2}C protocol stack uses the **EFI_I2C_REQUEST_PACKET** structure to describe I^{2}C transactions on the I^{2}C bus. The **EFI_I2C_OPERATION** describes a portion of the I^{2}C transaction. The transaction starts with a start bit followed by the first operation in the operation array. Subsequent operations are separated with repeated start bits and the last operation is followed by a stop bit which concludes the transaction.

```
typedef struct {
  UINTN              OperationCount;
  EFI_I2C_OPERATION  Operation[];
} EFI_I2C_REQUEST_PACKET;
```

Parameters

OperationCount

Number of elements in the operation array.

Operation

Description of the I^{2}C operation

Description

The **EFI_I2C_REQUEST_PACKET** describes a single I^{2}C transaction. The transaction starts with a start bit followed by the first operation in the operation array. Subsequent operations are separated with repeated start bits and the last operation is followed by a stop bit which concludes the transaction. Each operation is described by one of the elements in the *Operation* array.

```
typedef struct {
  UINT32  Flags;
  UINT32  LengthInBytes;
  UINT8   *Buffer;
} EFI_I2C_OPERATION;
```

Parameters

Flags

Flag bits qualify the I^2C operation.

Flag Bits:

```
///
/// Define the I2C flags
///
/// I2C read operation when set
#define I2C_FLAG_READ               0x00000001

///
/// Define the flags for SMBus operation
///
/// The following flags are also present in only the first I2C
operation
/// and are ignored when present in other operations.  These
flags
/// describe a particular SMB transaction as shown in the
following table.
///

/// SMBus operation
#define I2C_FLAG_SMBUS_OPERATION    0x00010000

/// SMBus block operation
///   The flag I2C_FLAG_SMBUS_BLOCK causes the I2C master
protocol to update
///   the LengthInBytes field of the operation in the request
packet with
///   the actual number of bytes read or written.  These values
are only
///   valid when the entire I2C transaction is successful.
///   This flag also changes the LengthInBytes meaning to be: A
maximum
///   of LengthInBytes is to be read from the device.  The first
byte
///   read contains the number of bytes remaining to be read,
plus an
///   optional PEC value.
#define I2C_FLAG_SMBUS_BLOCK        0x00020000

/// SMBus process call operation
#define I2C_FLAG_SMBUS_PROCESS_CALL 0x00040000

/// SMBus use packet error code (PEC)
///   Note that the I2C master protocol may clear the
I2C_FLAG_SMBUS_PEC bit
///   to indicate that the PEC value was checked by the hardware
and is
///   not appended to the returned read data.
///
```

```
#define I2C_FLAG_SMBUS_PEC          0x00080000

//--------------------------------------------------------------
--------
///
/// QuickRead:          OperationCount=1,
///                     LengthInBytes=0,   Flags=I2C_FLAG_READ
/// QuickWrite:         OperationCount=1,
///                     LengthInBytes=0,   Flags=0
///
///
/// ReceiveByte:        OperationCount=1,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            | I2C_FLAG_READ
/// ReceiveByte+PEC:    OperationCount=1,
///                     LengthInBytes=2,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            | I2C_FLAG_READ
///                                         | I2C_FLAG_SMBUS_PEC
///
///
/// SendByte:           OperationCount=1,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
/// SendByte+PEC:       OperationCount=1,
///                     LengthInBytes=2,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                         | I2C_FLAG_SMBUS_PEC
///
///
/// ReadDataByte:       OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                     LengthInBytes=1,   Flags=I2C_FLAG_READ
/// ReadDataByte+PEC:   OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                         | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=2,   Flags=I2C_FLAG_READ
///
///
/// WriteDataByte:      OperationCount=1,
///                     LengthInBytes=2,
Flags=I2C_FLAG_SMBUS_OPERATION
/// WriteDataByte+PEC:  OperationCount=1,
///                     LengthInBytes=3,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                         | I2C_FLAG_SMBUS_PEC
///
///
/// ReadDataWord:       OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                     LengthInBytes=2,   Flags=I2C_FLAG_READ
/// ReadDataWord+PEC:   OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                         | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=3,   Flags=I2C_FLAG_READ
///
///
/// WriteDataWord:      OperationCount=1,
///                     LengthInBytes=3,
Flags=I2C_FLAG_SMBUS_OPERATION
/// WriteDataWord+PEC:  OperationCount=1,
///                     LengthInBytes=4,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                         | I2C_FLAG_SMBUS_PEC
///
///
/// ReadBlock:          OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_BLOCK
///                     LengthInBytes=33,  Flags=I2C_FLAG_READ
/// ReadBlock+PEC:      OperationCount=2,
///                     LengthInBytes=1,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_BLOCK
///                                         | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=34,  Flags=I2C_FLAG_READ
///
///
/// WriteBlock:         OperationCount=1,
///                     LengthInBytes=N+2,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_BLOCK
/// WriteBlock+PEC:     OperationCount=1,
///                     LengthInBytes=N+3,
Flags=I2C_FLAG_SMBUS_OPERATION
```

```
///                                            |
I2C_FLAG_SMBUS_BLOCK
///                                         | I2C_FLAG_SMBUS_PEC
///
///
/// ProcessCall:        OperationCount=2,
///                     LengthInBytes=3,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_PROCESS_CALL
///                     LengthInBytes=2,   Flags=I2C_FLAG_READ
/// ProcessCall+PEC:    OperationCount=2,
///                     LengthInBytes=3,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_PROCESS_CALL
///                                         | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=3,   Flags=I2C_FLAG_READ
///
///
/// BlkProcessCall:     OperationCount=2,
///                     LengthInBytes=N+2,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_PROCESS_CALL
///                                            |
I2C_FLAG_SMBUS_BLOCK
///                     LengthInBytes=33,  Flags=I2C_FLAG_READ
/// BlkProcessCall+PEC: OperationCount=2,
///                     LengthInBytes=N+2,
Flags=I2C_FLAG_SMBUS_OPERATION
///                                            |
I2C_FLAG_SMBUS_PROCESS_CALL
///                                            |
I2C_FLAG_SMBUS_BLOCK
///                                         | I2C_FLAG_SMBUS_PEC
///                     LengthInBytes=34,  Flags=I2C_FLAG_READ
///
//--------------------------------------------------------------
--------
```

LengthInBytes

Number of bytes to send to or receive from the I^{2}C device. A ping (address only byte/ bytes) is indicated by setting the *LengthInBytes* to zero.

Buffer

Pointer to a buffer containing the data to send or to receive from the I^{2}C device. The *Buffer* must be at least *LengthInBytes* in size.

Description

The **EFI_I2C_OPERATION** describes a subset of an I^{2}C transaction in which the I^{2}C controller is either sending or receiving bytes from the bus. Some transactions will consist of a single operation while others will be two or more.

Note: *Some I^{2}C controllers do not support read or write ping (address only) operation and will return* **EFI_UNSUPPORTED** *status when these operations are requested.*

Note: *I^{2}C controllers which do not support complex transactions requiring multiple repeated start bits return* **EFI_UNSUPPORTED** *without processing any of the transaction.*

```
typedef struct {
  UINT32   StructureSizeInBytes;
  UINT32   MaximumReceiveBytes;
  UINT32   MaximumTransmitBytes;
  UINT32   MaximumTotalBytes;
} EFI_I2C_CONTROLLER_CAPABILITIES;
```

Parameters

StructureSizeInBytes

Length of this data structure in bytes

MaximumReceiveBytes;

The maximum number of bytes the I^{2}C host controller is able to receive from the I^{2}C bus.

MaximumTransmitBytes

The maximum number of bytes the I^{2}C host controller is able to send on the I^{2}C bus.

MaximumTotalBytes

The maximum number of bytes in the I^{2}C bus transaction.

Description

The **EFI_I2C_CONTROLLER_CAPABILITIES** specifies the capabilities of the I^{2}C host controller. The *StructureSizeInBytes* enables variations of this structure to be identified if there is need to extend this structure in the future.

EFI_I2C_MASTER_PROTOCOL.SetBusFrequency()

Summary

Set the frequency for the I^{2}C clock line.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_SET_BUS_FREQUENCY) (
  IN CONST EFI_I2C_MASTER_PROTOCOL    *This,
  IN OUT UINTN                        *BusClockHertz
  );
```

Parameters

This

Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

BusClockHertz

Pointer to the requested I^{2}C bus clock frequency in Hertz. Upon return this value contains the actual frequency in use by the I^{2}C controller.

Description

This routine must be called at or below **TPL_NOTIFY**.

The software and controller do a best case effort of using the specified frequency for the I^{2}C bus. If the frequency does not match exactly then the I^{2}C master protocol selects the next lower frequency to avoid exceeding the operating conditions for any of the I^{2}C devices on the bus. For example if 400 KHz was specified and the controller's divide network only supports 402 KHz or 398 KHz then the I^{2}C master protocol selects 398 KHz. If there are not lower frequencies available, then return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The bus frequency was set successfully.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_UNSUPPORTED	The controller does not support this frequency.

EFI_I2C_MASTER_PROTOCOL.Reset()

Summary

Reset the I^{2}C controller and configure it for use.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_RESET) (
  IN CONST EFI_I2C_MASTER_PROTOCOL   *This
  );
```

Parameters

This

Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

Description

This routine must be called at or below **TPL_NOTIFY**.

The I^{2}C controller is reset. The caller must call *SetBusFrequency()* after calling *Reset()*.

Status Codes Returned

EFI_SUCCESS	The reset completed successfully.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_DEVICE_ERROR	The reset operation failed.

EFI_I2C_MASTER_PROTOCOL.StartRequest()

Summary

Start an I^{2}C transaction on the host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_MASTER_PROTOCOL_START_REQUEST) (
  IN CONST EFI_I2C_MASTER_PROTOCOL   *This,
  IN UINTN                           SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET          *RequestPacket,
  IN EFI_EVENT                       Event           OPTIONAL,
  OUT EFI_STATUS                     *I2cStatus      OPTIONAL
  );
```

Parameters

This

Pointer to an **EFI_I2C_MASTER_PROTOCOL** structure.

SlaveAddress

Address of the device on the I^{2}C BUS. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^{2}C slave addresses and bits 0-9 are used for 10-bit I^{2}C slave addresses.

RequestPacket

Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction.

Event

Event to signal for asynchronous transactions, NULL for synchronous transactions

I2cStatus

Optional buffer to receive the I^{2}C transaction completion status

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

This function initiates an I^{2}C transaction on the controller. To enable proper error handling by the I^{2}C protocol stack, the I^{2}C master protocol does not support queuing but instead only manages one I^{2}C transaction at a time. This API requires that the I^{2}C bus is in the correct configuration for the I^{2}C transaction.

The transaction is performed by sending a start-bit and selecting the I^{2}C device with the specified I^{2}C slave address and then performing the specified I^{2}C operations. When multiple operations are requested they are separated with a repeated start bit and the slave address. The transaction is terminated with a stop bit.

When Event is NULL, *StartRequest* operates synchronously and returns the I^2C completion status as its return value.

When Event is not NULL, *StartRequest* synchronously returns **EFI_SUCCESS** indicating that the I^2C transaction was started asynchronously. The transaction status value is returned in the buffer pointed to by *I2cStatus* upon the completion of the I^2C transaction when I2cStatus is not NULL. After the transaction status is returned the *Event* is signaled.

Note: *The typical consumer of this API is the I^2C host protocol. Extreme care must be taken by other consumers of this API to prevent confusing the third party I^2C drivers due to a state change at the I^2C device which the third party I^2C drivers did not initiate. I^2C platform specific code may use this API within these guidelines.*

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully started when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_ALREADY_STARTED	The controller is busy with another transaction.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^2C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_NO_RESPONSE	The I^2C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^2C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.2 I^2C Host Protocol

EFI_I2C_HOST_PROTOCOL

Summary

This protocol provides callers with the ability to do I/O transactions to all of the devices on the I^2C bus.

GUID

```
#define EFI_I2C_HOST_PROTOCOL_GUID \
{ 0xa5aab9e3, 0xc727, 0x48cd, { 0x8b, 0xbf, 0x42, 0x72, \
0x33, 0x85, 0x49, 0x48 }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_HOST_PROTOCOL {
  EFI_I2C_HOST_PROTOCOL_QUEUE_REQUEST  QueueRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES  *I2cControllerCapabilities;
} EFI_I2C_HOST_PROTOCOL;
```

Parameters

QueueRequest

Queue an transaction for execution on the I^{2}C bus

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

The I^{2}C bus driver uses the services of the **EFI_I2C_HOST_PROTOCOL** to produce an instance of the **EFI_I2C_IO_PROTOCOL** for each I^{2}C device on an I^{2}C bus.

The **EFI_I2C_HOST_PROTOCOL** exposes an asynchronous interface to callers to perform transactions to any device on the I^{2}C bus. Internally, the I^{2}C host protocol manages the flow of the I^{2}C transactions to the host controller, keeping them in FIFO order. Prior to each transaction, the I^{2}C host protocol ensures that the switches and multiplexers are properly configured. The I^{2}C host protocol then starts the transaction on the host controller using the **EFI_I2C_MASTER_PROTOCOL**.

EFI_I2C_HOST_PROTOCOL.QueueRequest()

Summary

Queue an I^{2}C transaction for execution on the I^{2}C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_HOST_PROTOCOL_QUEUE_REQUEST) (
  IN CONST EFI_I2C_HOST_PROTOCOL  *This,
  IN UINTN                        I2cBusConfiguration,
  IN UINTN                        SlaveAddress,
  IN EFI_EVENT                    Event              OPTIONAL,
  IN EFI_I2C_REQUEST_PACKET       *RequestPacket,
  OUT EFI_STATUS                  *I2cStatus         OPTIONAL
  );
```

Parameters

This

Pointer to an **EFI_I2C_HOST_PROTOCOL** structure.

I2cBusConfiguration

I^{2}C bus configuration to access the I^{2}C device

SlaveAddress

Address of the device on the I^{2}C bus. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^{2}C slave addresses and bits 0-9 are used for 10-bit I^{2}C slave addresses.

Event

Event to signal for asynchronous transactions, NULL for synchronous transactions

RequestPacket

Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

I2cStatus

Optional buffer to receive the I^{2}C transaction completion status

Description

Queue an I^{2}C transaction for execution on the I^{2}C controller.

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

The I^{2}C host protocol uses the concept of I^{2}C bus configurations to describe the I^{2}C bus. An I^{2}C bus configuration is defined as a unique setting of the multiplexers and switches in the I^{2}C bus which enable access to one or more I^{2}C devices. When using a switch to divide a bus, due to bus frequency differences, the I^{2}C bus configuration management protocol defines an I^{2}C bus configuration for the I^{2}C devices on each side of the switch. When using a multiplexer, the I^{2}C bus configuration management defines an I^{2}C bus configuration for each of the selector values required to control the multiplexer. See Figure 1 in the **I2C -bus specification and user manual** for a complex I^{2}C bus configuration.

The I^{2}C host protocol processes all transactions in FIFO order. Prior to performing the transaction, the I^{2}C host protocol calls *EnableI2cBusConfiguration* to reconfigure the switches and multiplexers in the I^{2}C bus enabling access to the specified I^{2}C device. The EnableI2cBusConfiguration also selects the I^{2}C bus frequency for the I^{2}C device. After the I^{2}C bus is configured, the I^{2}C host protocol calls the I^{2}C master protocol to start the I^{2}C transaction.

When Event is NULL, *QueueRequest()* operates synchronously and returns the I^{2}C completion status as its return value.

When Event is not NULL, *QueueRequest()* synchronously returns **EFI_SUCCESS** indicating that the asynchronously I^{2}C transaction was queued. The values above are returned in the buffer pointed to by I2cStatus upon the completion of the I^{2}C transaction when I2cStatus is not NULL.

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully queued when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.3 I^{2}C I/O Protocol

EFI_I2C_IO_PROTOCOL

Summary

The EFI I^{2}C I/O protocol enables the user to manipulate a single I^{2}C device independent of the host controller and I^{2}C design.

GUID

```
#define EFI_I2C_IO_PROTOCOL_GUID \
{ 0xb60a3e6b, 0x18c4, 0x46e5, { 0xa2, 0x9a, 0xc9, 0xa1, \
0x06, 0x65, 0xa2, 0x8e }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_IO_PROTOCOL {
  EFI_I2C_IO_PROTOCOL_QUEUE_REQUEST QueueRequest;
  CONST EFI_GUID                    *DeviceGuid;
  UINT32                            DeviceIndex;
  UINT32                            HardwareRevision;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES  *I2cControllerCapabilities;
} EFI_I2C_IO_PROTOCOL;
```

Parameters

QueueRequest

Queue an I^{2}C transaction for execution on the I^{2}C device.

DeviceGuid

Unique value assigned by the silicon manufacture or the third party I^{2}C driver writer for the I^{2}C part. This value logically combines both the manufacture name and the I^{2}C part number into a single value specified as a GUID.

DeviceIndex

Unique ID of the I^{2}C part within the system

HardwareRevision

Hardware revision - ACPI _HRV value. See the **Advanced Configuration and Power Interface Specification, Revision 5.0** for the field format and the **Plug and play support for I2C** web-page for restriction on values.

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

- The I^{2}C IO protocol enables access to a specific device on the I^{2}C bus.
- Each I^{2}C device is identified uniquely in the system by the tuple *DeviceGuid*:*DeviceIndex*. The *DeviceGuid* represents the manufacture and part number and is provided by the silicon vendor or the third party I^{2}C device driver writer. The *DeviceIndex* identifies the part within the system by using a unique number and is created by the board designer or the writer of the **EFI_I2C_ENUMERATE_PROTOCOL**.

I^{2}C slave addressing is abstracted to validate addresses and limit operation to the specified I^{2}C device. The third party providing the I^{2}C device support provides an ordered list of slave addresses for the I^{2}C device required to implement the **EFI_I2C_ENUMERATE_PROTOCOL**. The order of the list must be preserved.

EFI_I2C_IO_PROTOCOL.QueueRequest()

Summary

Queue an I^{2}C transaction for execution on the I^{2}C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_PROTOCOL_QUEUE_REQUEST) (
  IN CONST EFI_I2C_IO_PROTOCOL  *This,
  IN UINTN                      SlaveAddressIndex,
  IN EFI_EVENT                  Event              OPTIONAL,
  IN EFI_I2C_REQUEST_PACKET     *RequestPacket,
  OUT EFI_STATUS                *I2cStatus        OPTIONAL
  );
```

Parameters

This

Pointer to an **EFI_I2C_IO_PROTOCOL** structure.

SlaveAddressIndex

Index value into an array of slave addresses for the I^{2}C device. The values in the array are specified by the board designer, with the third party I^{2}C device driver writer providing the slave address order.

For devices that have a single slave address, this value must be zero. If the I^{2}C device uses more than one slave address then the third party (upper level) I^{2}C driver writer needs to specify the order of entries in the slave address array.

Event

Event to signal for asynchronous transactions, NULL for synchronous transactions

RequestPacket

Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

I2cStatus

Optional buffer to receive the I^{2}C transaction completion status

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

This routine queues an I^{2}C transaction to the I^{2}C controller for execution on the I^{2}C bus.

When *Event* is NULL, *QueueRequest()* operates synchronously and returns the I^{2}C completion status as its return value.

When *Event* is not NULL, *QueueRequest()* synchronously returns **EFI_SUCCESS** indicating that the asynchronous I^{2}C transaction was queued. The values above are returned in the buffer pointed to by I2cStatus upon the completion of the I^{2}C transaction when I2cStatus is not NULL.

Status Codes Returned

EFI_SUCCESS	The asynchronous transaction was successfully queued when *Event* is not NULL.
EFI_SUCCESS	The transaction completed successfully when *Event* is NULL.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is NULL
EFI_NO_MAPPING	The **EFI_I2C_HOST_PROTOCOL** could not set the bus configuration required to access this I^{2}C device.
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address selected by *SlaveAddressIndex*. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.2.4 I^{2}C Bus Configuration Management Protocol

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL

Summary

The EFI I^{2}C bus configuration management protocol provides platform specific services that allow the I^{2}C host protocol to reconfigure the switches and multiplexers and set the clock frequency for the I^{2}C bus. This protocol also enables the I^{2}C host protocol to reset an I^{2}C device which may be locking up the I^{2}C bus by holding the clock or data line low.

GUID

```
#define EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_GUID \
{ 0x55b71fb5, 0x17c6, 0x410e, { 0xb5, 0xbd, 0x5f, 0xa2, \
0xe3, 0xd4, 0x46, 0x6b }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL {

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_ENABLE_I2C_BUS_CON
FIGURATION  EnableI2cBusConfiguration;
} EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL;
```

Parameters

EnableI2cBusConfiguration

Enable an I^{2}C bus configuration for use.

Description

The I^{2}C protocol stack uses the concept of an I^{2}C bus configuration as a way to describe a particular state of the switches and multiplexers in the I^{2}C bus.

A simple I^{2}C bus does not have any multiplexers or switches is described to the I^{2}C protocol stack with a single I^{2}C bus configuration which specifies the I^{2}C bus frequency.

An I^{2}C bus with switches and multiplexers use an I^{2}C bus configuration to describe each of the unique settings for the switches and multiplexers and the I^{2}C bus frequency. However the I^{2}C bus configuration management protocol only needs to define the I^{2}C bus configurations that the software uses, which may be a subset of the total.

The I^{2}C bus configuration description includes a list of I^{2}C devices which may be accessed when this I^{2}C bus configuration is enabled. I^{2}C devices before a switch or multiplexer must be included in one I^{2}C bus configuration while I^{2}C devices after a switch or multiplexer are on another I^{2}C bus configuration.

The I^{2}C bus configuration management protocol is an optional protocol. When the I^{2}C bus configuration protocol is not defined the I^{2}C host protocol does not start and the I^{2}C master protocol may be used for other purposes such as SMBus traffic. When the I^{2}C bus configuration protocol is available, the I^{2}C host protocol uses the I^{2}C bus configuration protocol to call into the platform specific code to set the switches and multiplexers and set the maximum I^{2}C bus frequency.

The platform designers determine the maximum I^{2}C bus frequency by selecting a frequency which supports all of the I^{2}C devices on the I^{2}C bus for the setting of switches and multiplexers. The platform designers must validate this against the I^{2}C device data sheets and any limits of the I^{2}C controller or bus length.

During I^{2}C device enumeration, the I^{2}C bus driver retrieves the I^{2}C bus configuration that must be used to perform I^{2}C transactions to each I^{2}C device. This I^{2}C bus configuration value is passed into the I^{2}C host protocol to identify the I^{2}C bus configuration required to access a specific I^{2}C device. The I^{2}C host protocol calls **EnableBusConfiguration()** to set the switches and multiplexers in the I^{2}C bus and the I^{2}C clock frequency. The I^{2}C host protocol may optimize calls to **EnableBusConfiguration()** by only making the call when the I^{2}C bus configuration value changes between I^{2}C requests.

When I^{2}C transactions are required on the same I^{2}C bus to change the state of multiplexers or switches, the I^{2}C master protocol must be used to perform the necessary I^{2}C transactions.

It is up to the platform specific code to choose the proper I^{2}C bus configuration when **ExitBootServices()** is called. Some operating systems are not able to manage the I^{2}C bus configurations and must use the I^{2}C bus configuration that is established by the platform firmware before **ExitBootServices()** returns.

EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL. EnableI2cBusConfiguration()

Summary

Enable access to an I^{2}C bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL_ENABLE_I2C_BUS_CO
NFIGURATION) (
  IN CONST EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL  *This,
  IN UINTN                                      I2cBusConfiguration,
  IN EFI_EVENT                                   Event        OPTIONAL,
  IN EFI_STATUS                                 *I2cStatus   OPTIONAL
  );
```

Parameters

This

Pointer to an **EFI_I2C_BUS_CONFIGURATION_MANAGEMENT_PROTOCOL** structure.

I2cBusConfiguration

Index of an I^{2}C bus configuration. All values in the range of zero to N-1 are valid where N is the total number of I^{2}C bus configurations for an I^{2}C bus.

Event

Event to signal when the transaction is complete

I2cStatus

Buffer to receive the transaction status.

Description

This routine must be called at or below **TPL_NOTIFY**. For synchronous requests this routine must be called at or below **TPL_CALLBACK**.

Reconfigure the switches and multiplexers in the I^{2}C bus to enable access to a specific I^{2}C bus configuration. Also select the maximum clock frequency for this I^{2}C bus configuration.

This routine uses the I^{2}C Master protocol to perform I^{2}C transactions on the local bus. This eliminates any recursion in the I^{2}C stack for configuration transactions on the same I^{2}C bus. This works because the local I^{2}C bus is idle while the I^{2}C bus configuration is being enabled.

If I^{2}C transactions must be performed on other I^{2}C busses, then the **EFI_I2C_HOST_PROTOCOL**, the **EFI_I2C_IO_PROTCOL**, or a third party I^{2}C driver interface for a specific device must be used. This requirement is because the I^{2}C host protocol controls the flow of requests to the I^{2}C controller. Use the **EFI_I2C_HOST_PROTOCOL** when the I^{2}C device is not enumerated by the **EFI_I2C_ENUMERATE_PROTOCOL**. Use a protocol produced by a third party driver when it is available or the **EFI_I2C_IO_PROTOCOL** when the third party driver is not available but the device is enumerated with the **EFI_I2C_ENUMERATE_PROTOCOL**.

When Event is NULL, *EnableI2cBusConfiguration* operates synchronously and returns the I^{2}C completion status as its return value. The values returned from *EnableI2cBusConfiguration* are:

Status Codes Returned

EFI_SUCCESS	The asynchronous bus configuration request was successfully started when *Event* is not NULL.
EFI_SUCCESS	The bus configuration request completed successfully when *Event* is NULL.
EFI_DEVICE_ERROR	The bus configuration failed.
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

17.2.5 I^{2}C Enumerate Protocol

EFI_I2C_ENUMERATE_PROTOCOL

Summary

Support the enumeration of the I^{2}C devices.

GUID

```
#define EFI_I2C_ENUMERATE_PROTOCOL_GUID \
{ 0xda8cd7c4, 0x1c00, 0x49e2, { 0x80, 0x3e, 0x52, 0x14, \
0xe7, 0x01, 0x89, 0x4c }}
```

Protocol Interface Structure

```
typedef struct _EFI_I2C_ENUMERATE_PROTOCOL {
  EFI_I2C_ENUMERATE_PROTOCOL_ENUMERATE          Enumerate;
  EFI_I2C_ENUMERATE_PROTOCOL_GET_BUS_FREQUENCY  GetBusFrequency;
} EFI_I2C_ENUMERATE_PROTOCOL;
```

Parameters

Enumerate

Traverse the set of I^{2}C devices on an I^{2}C bus. This routine returns the next I^{2}C device on an I^{2}C bus.

GetBusFrequency

Get the requested I^{2}C bus frequency for a specified bus configuration.

Description

The I^{2}C bus driver uses this protocol to enumerate the devices on the I^{2}C bus.

Related Definitions

```
typedef struct {
  CONST EFI_GUID   *DeviceGuid;
  UINT32           DeviceIndex;
  UINT32           HardwareRevision;
  UINT32           I2cBusConfiguration;
  UINT32           SlaveAddressCount;
  CONST UINT32     *SlaveAddressArray;
} EFI_I2C_DEVICE;
```

Parameters

DeviceGuid

Unique value assigned by the silicon manufacture or the third party I^{2}C driver writer for the I^{2}C part. This value logically combines both the manufacture name and the I^{2}C part number into a single value specified as a GUID.

DeviceIndex

Unique ID of the I^{2}C part within the system

HardwareRevision

Hardware revision - ACPI _HRV value. See the **Advanced Configuration and Power Interface Specification, Revision 5.0** for the field format and the **Plug and play support for I2C** web-page for restriction on values.

I2cBusConfiguration

I^{2}C bus configuration for the I^{2}C device

SlaveAddressCount

Number of slave addresses for the I^{2}C device.

SlaveAddressArray

Pointer to the array of slave addresses for the I^{2}C device.

Description

The **EFI_I2C_ENUMERATE_PROTOCOL** uses the **EFI_I2C_DEVICE** to describe the platform specific details associated with an I^{2}C device. This description is passed to the I^{2}C bus driver during enumeration where it is made available to the third party I^{2}C device driver via the **EFI_I2C_IO_PROTOCOL**.

EFI_I2C_ENUMERATE_PROTOCOL.Enumerate()

Summary

Enumerate the I^{2}C devices

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_ENUMERATE_PROTOCOL_ENUMERATE) (
  IN CONST EFI_I2C_ENUMERATE_PROTOCOL   *This,
  IN OUT CONST EFI_I2C_DEVICE           **Device
  );
```

Parameters

This

Pointer to an **EFI_I2C_ENUMERATE_PROTOCOL** structure.

Device

Pointer to a buffer containing an **EFI_I2C_DEVICE** structure. Enumeration is started by setting the initial **EFI_I2C_DEVICE** structure pointer to NULL. The buffer receives an **EFI_I2C_DEVICE** structure pointer to the next I^{2}C device.

Description

This function enables the caller to traverse the set of I^{2}C devices on an I^{2}C bus.

Status Codes Returned

EFI_SUCCESS	The platform data for the next device on the I^{2}C bus was returned successfully.
EFI_INVALID_PARAMETER	*Device* is NULL
EFI_NO_MAPPING	***Device*** does not point to a valid **EFI_I2C_DEVICE** structure returned in a previous call *Enumerate()*.

EFI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency()

Summary

Get the requested I^{2}C bus frequency for a specified bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_ENUMERATE_PROTOCOL_GET_BUS_FREQUENCY) (
  IN CONST EFI_I2C_ENUMERATE_PROTOCOL   *This,
  IN UINTN                              I2cBusConfiguration,
  OUT UINTN                             *BusClockHertz
  );
```

Parameters

This

Pointer to an **EFI_I2C_ENUMERATE_PROTOCOL** structure.

I2cBusConfiguration

I^{2}C bus configuration to access the I^{2}C device

BusClockHertz

Pointer to a buffer to receive the I^{2}C bus clock frequency in Hertz

Description

This function returns the requested I^{2}C bus clock frequency for the *I2cBusConfiguration*. This routine is provided for diagnostic purposes and is meant to be called after calling *Enumerate* to get the *I2cBusConfiguration* value.

Status Codes Returned

EFI_SUCCESS	The I^{2}C bus frequency was returned successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* was **NULL**
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

17.3 PEI Code definitions

For the Pre-EFI Initialization environment a subset of the I^{2}C stack is defined to support basic hardware initialization in the PEI phase. The **EFI_PEI_I2C_MASTER** PPI is defined to standardize access to the I^{2}C controller.

17.3.1 I^{2}C Master PPI

EFI_PEI_I2C_MASTER

Summary

This PPI manipulates the I2C host controller to perform transactions as a master on the I2C bus using the current state of any switches or multiplexers in the I2C bus.

GUID

```
#define EFI_PEI_I2C_MASTER_PPI_GUID \
{ 0xb3bfab9b, 0x9f9c, 0x4e8b, { 0xad, 0x37, 0x7f, 0x8c, \
0x51, 0xfc, 0x62, 0x80 }}
```

PEIM-to-PEIM Interface Structure

```
typedef struct _EFI_PEI_I2C_MASTER_PPI {
  EFI_PEI_I2C_MASTER_PPI_SET_BUS_FREQUENCY  SetBusFrequency;
  EFI_PEI_I2C_MASTER_PPI_RESET              Reset;
  EFI_PEI_I2C_MASTER_PPI_START_REQUEST      StartRequest;
  CONST EFI_PEI_I2C_CONTROLLER_CAPABILITIES *
I2cControllerCapabilities;
  EFI_GUID                                 Identifier;
} EFI_PEI_I2C_MASTER_PPI;
```

Parameters

SetBusFrequency

Set the clock frequency in Hertz for the I^{2}C bus.

Reset

Reset the I^{2}C host controller.

StartRequest

Start an I^{2}C transaction in master mode on the host controller.

I2cControllerCapabilities

Pointer to an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Identifier

Identifier which uniquely identifies thisI^{2}C controller in the system.

Description

The **EFI_PEI_I2C_MASTER** PPI enables the platform code to perform transactions on the I^{2}C bus.

EFI_PEI_I2C_MASTER_PPI.SetBusFrequency()

Summary

Set the frequency for the I^2C clock line.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_SET_BUS_FREQUENCY) (
  IN EFI_PEI_I2C_MASTER    *This,
  IN UINTN                 *BusClockHertz
  );
```

Parameters

This

Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

BusClockHertz

Pointer to the requested I^2C bus clock frequency in Hertz. Upon return this value contains the actual frequency in use by the I^2C controller.

Description

The software and controller do a best case effort of using the specified frequency for the I2C bus. If the frequency does not match exactly then the I2C master protocol selects the next lower frequency to avoid exceeding the operating conditions for any of the I2C devices on the bus. For example if 400 KHz was specified and the controller's divide network only supports 402 KHz or 398 KHz then the controller would be set to 398 KHz. If there are no lower frequencies available, then return **EFI_UNSUPPORTED**.

Status Codes Returned

EFI_SUCCESS	The bus frequency was set successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* is NULL
EFI_UNSUPPORTED	The controller does not support this frequency.

EFI_PEI_I2C_MASTER_PPI.Reset()

Summary

Reset the I^{2}C controller and configure it for use.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_RESET) (
  IN CONST EFI_PEI_I2C_MASTER   *This
  );
```

Parameters

This

Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

Description

The I^{2}C controller is reset. The caller must call **SetBusFrequency()** after calling **Reset()**.

Status Codes Returned

EFI_SUCCESS	The reset completed successfully.
EFI_DEVICE_ERROR	The reset operation failed.

EFI_PEI_I2C_MASTER_PPI.StartRequest()

Summary

Start an I^{2}C transaction on the host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_MASTER_PPI_START_REQUEST) (
  IN CONST EFI_PEI_I2C_MASTER   *This,
  IN UINTN                      SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET     *RequestPacket
  );
```

Parameters

This

Pointer to an **EFI_PEI_I2C_MASTER_PPI** structure.

SlaveAddress

Address of the device on the I^{2}C bus. Set the **I2C_ADDRESSING_10_BIT** when using 10-bit addresses, clear this bit for 7-bit addressing. Bits 0-6 are used for 7-bit I^{2}C slave addresses and bits 0-9 are used for 10-bit I^{2}C slave addresses.

RequestPacket

Pointer to an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction.

Description

This function initiates an I^{2}C transaction on the controller.

The transaction is performed by sending a start-bit and selecting the I^{2}C device with the specified I^{2}C slave address and then performing the specified I^{2}C operations. When multiple operations are requested they are separated with a repeated start bit and the slave address. The transaction is terminated with a stop bit. When the transaction completes, the status value is returned.

Status Codes Returned

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *RequestPacket->LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^2C error (NACK) during the transaction.
EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NO_RESPONSE	The I^2C device is not responding to the slave address. **EFI_DEVICE_ERROR** will be returned if the controller cannot distinguish when the NACK occurred.
EFI_NOT_FOUND	Reserved bit set in the *SlaveAddress* parameter
EFI_OUT_OF_RESOURCES	Insufficient memory for I^2C transaction
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.2 I^2C Host PPI

EFI_PEI_I2C_HOST

Summary

This PPI provides callers with the ability to do I/O transactions to all of the devices on the I^2C bus.

GUID

```
#define EFI_PEI_I2C_HOST_GUID \
{ 0x3a12e52d, 0x3bd2, 0x482c, 0xa6, 0x80, 0x0f, 0xeb, \
0x61, 0x9a, 0xeb, 0xef }
```

PEIM-to-PEIM Interface Structure

```
typedef struct _EFI_I2C_HOST_PPI {
  EFI_I2C_HOST_START_REQUEST            StartRequest;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES * I2cControllerCapabilities;
  UINTN                                 ControllerNumber;
};
```

Parameters

QueueRequest

Queue a transaction for execution on the I^2C bus

I2cControllerCapabilities

The address of an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^2C host controller.

ControllerNumber

Unique number identifying the I^2C controller in the system

Description

Please use **EFI_PEI_I2C_IO** as **EFI_PEI_I2C_HOST** is only intended to be used by the I^{2}C bus driver. The **EFI_PEI_I2C_HOST** requires the tuple **ControllerNumber:BusConfiguration:SlaveAddress** which is platform specific data to identify the I^{2}C device. **EFI_PEI_I2C_IO** eliminates the platform specific details.

The upper layer driver locates the correct **EFI_PEI_I2C_HOST** interface (I2cHost) by comparing the following field:

- I2cHost→ControllerNumber with the system unique value for the I^{2}C ControllerNumber

Prior to each transaction, the I^{2}C host driver ensures that the switches and multiplexers are properly configured. The I^{2}C host driver then starts the transaction on the I^{2}C controller calling the I^{2}C port driver interface (**EFI_PEI_I2C_MASTER**).

Related Definitions

The I^{2}C platform driver installs the following GUID after installing **EFI_PEI_I2C_ENUMERATE** and **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** PPIs for the necessary I^{2}C controllers. The following GUID resolves the dependency expressions for the I^{2}C port and host drivers enabling them to load and start their configuration.

Lack of **EFI_PEI_I2C_BUS_CONFIGURATION** prevents the I^{2}C host driver from loading, reserving the I^{2}C port driver for SMBus transactions.

GUID

```
#define EFI_PEI_I2C_BUS_CONFIGURED_GUID \
{ 0x9eade134, 0x6bb1, 0x421d, 0xac, 0xaf, 0x59, 0x0a, \
0x5d, 0x2e, 0xa6, 0x3a }
```

EFI_PEI_I2C_HOST.StartRequest()

Summary

Start a transaction on the I^{2}C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_HOST_START_REQUEST) (
  IN EFI_PEI_I2C_HOST        *This,
  IN UINTN                   I2cBusConfiguration,
  IN UINTN                   SlaveAddress,
  IN EFI_I2C_REQUEST_PACKET *RequestPacket
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_HOST** structure.

I2cBusConfiguration

I^{2}C bus configuration to access the I^{2}C device

SlaveAddress

Address of the device on the I^{2}C bus. Or in the value **I2C_ADDRESSING_10_BIT** when using 10-bit addresses.

RequestPacket

Address of an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

Description

Start an I2C transaction on the I2C controller.

N.B. The typical consumers of this API are the I^{2}C bus driver and on rare occasions I^{2}C test applications. Extreme care must be taken by other consumers of this API to prevent confusing the third party I^{2}C drivers due to a state change at the I^{2}C device which the third party I^{2}C drivers did not initiate. I^{2}C platform drivers may use this API within these guidelines.

This layer uses the concept of I^{2}C bus configurations to describe the I^{2}C bus. An I^{2}C bus configuration is defined as a unique setting of the multiplexers and switches in the I^{2}C bus which enable access to one or more I^{2}C devices. When using a switch to divide a bus, due to speed differences, the I^{2}C platform layer would define an I^{2}C bus configuration for the I^{2}C devices on each side of the switch. When using a multiplexer, the I^{2}C platform layer defines an I^{2}C bus configuration for each of the selector values required to control the multiplexer. See Figure 1 in the *I2C -bus specification and user manual* for a complex I^{2}C bus configuration.

The I^{2}C host driver calls the I^{2}C platform driver to reconfigure the switches and multiplexers in the I^{2}C bus enabling access to the specified I^{2}C device. The I^{2}C platform driver also selects the maximum bus speed for the device. After the I^{2}C bus is configured, the I^{2}C host driver calls the I^{2}C port driver to initialize the I^{2}C controller and start the I^{2}C transaction.

In event of timeout, the I2C host driver calls the I2C platform driver in an attempt to reset the host controller and the I2C device.

Status Codes Returned

The values returned from *StartRequest* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction. One possible cause is that the slave device is not present.
EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I^{2}C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.3 I^{2}C I/O PPI

EFI_PEI_I2C_IO

Summary

The EFI I^{2}C I/O PPI enables the user to manipulate a single I^{2}C device independent of the host controller and I^{2}C bus design.

GUID

```
#define EFI_PEI_I2C_IO_GUID \
{ 0x42179ed0, 0x2fa2, 0x47c0, 0x85, 0x7e, 0x8b, 0xc0, \
0x18, 0x81, 0xea, 0x97 }
```

PEIM-to-PEIM Interface Structure

```
typedef struct {
  EFI_PEI_I2C_IO_GET_DEVICE_INFO         GetDeviceInfo;
  EFI_PEI_I2C_IO_GET_DEVICE_INFO_ID_LIST GetDeviceInfoIdList;
  EFI_PEI_I2C_IO_START_REQUEST           StartRequest;
  EFI_I2C_DEVICE                         *I2cDevice;
  CONST EFI_I2C_CONTROLLER_CAPABILITIES  *
I2cControllerCapabilities;
} EFI_PEI_I2C_IO;
```

Parameters

GetDeviceInfo

Get a blob of data identified by a GUID.

GetDeviceInfoIdList

Get a list of the GUIDs associated with this I^{2}C device.

StartRequest

Start a transaction on the I^{2}C device.

I2cDevice

A pointer to the **EFI_I2C_DEVICE** structure contained within the I^{2}C platform driver.

I2cControllerCapabilities

The address of an **EFI_I2C_CONTROLLER_CAPABILITIES** data structure containing the capabilities of the I^{2}C host controller.

Description

The I^{2}C I/O PPI enables access to a specific device on the I^{2}C bus.

Each I^{2}C device is identified uniquely in the system by the tuple *DeviceGuid:DeviceIndex*. The *DeviceGuid* combines the manufacture and part number and is provided by the silicon vendor or the third party I^{2}C device driver writer. The *DeviceIndex* identifies the part within the system by using a unique number and is created by the board designer or the I^{2}C platform driver writer.

The upper layer I2C driver writer provides the following to the platform vendor:

- Vendor specific GUID for the I^{2}C part that is used to connect the upper layer driver to the device.
- Slave address array guidance when the I^{2}C device uses more than one slave address. This is used to access the blocks of hardware within the I^{2}C device.

The upper layer driver locates the correct **EFI_PEI_I2C_IO** interface (I2cIo) by comparing the following fields:

- I2cIo→Device.DeviceGuid with the vendor supplied GUID
- I2cIo→DeviceIndex with the system wide unique number assigned to the specific I^{2}C part.

I^{2}C slave addressing is abstracted to validate addresses and limit operation to the specified I^{2}C device. The third party providing the I^{2}C device support provides an ordered list of slave addresses for the I^{2}C device to the team building the platform layer. The platform team must preserve the order of the supplied list. *SlaveAddressCount* is the number of entries in this list or array within the platform layer. The third party device support references a slave address using an index into the list or array in the range of zero to *SlaveAddressCount* - 1.

EFI_I2C_IO_PROTOCOL.GetDeviceInfo()

Summary

Get a data blob associated with the I²C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_GET_DEVICE_INFO) (
  IN EFI_PEI_I2C_IO      *This,
  IN EFI_GUID            *DataGuid,
  IN OUT UINT32          *LengthInBytes,
  OUT VOID               *Buffer
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_IO** structure.

DataGuid

Address of the GUID associated with the data

LengthInBytes

Address of a value containing the length of the buffer in bytes on input and receiving the length of the data on output. If the input length was too small, the output length specifies the data length.

Buffer

Buffer address to receive the data

Description

This routine locates the specified data blob associated with the I^{2}C device.

Status Codes Returned

The values returned from *GetDeviceInfo* are:

EFI_SUCCESS	The data was returned successfully
EFI_BUFFER_TOO_SMALL	The specified buffer length is too small
EFI_INVALID_PARAMETER	*Buffer* is **NULL**
EFI_INVALID_PARAMETER	*DataGuid* is **NULL**
EFI_INVALID_PARAMETER	*LengthInBytes* is **NULL**
EFI_NOT_FOUND	Data blob was not found

EFI_I2C_IO_PROTOCOL.GetDeviceInfoIdList()

Summary

Get the list of data associated with the I^{2}C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_I2C_IO_GET_DEVICE_INFO_ID_LIST) (
  IN EFI_PEI_I2C_IO     *This,
  IN CONST EFI_GUID     ***GuidArray,
  IN UINTN              *GuidEntries
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_IO** structure.

GuidArray

Address to receive the list of GUIDs

GuidEntries

Address to receive the number of entries in the GUID array

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine returns an array of GUIDs identifying data associated with the I2C device. When the caller is done with the GUID array, the caller must call *FreePool* to return the GUID array to the heap.

Status Codes Returned

The values returned from *GetDeviceInfoIdList* are:

EFI_SUCCESS	The GUID array was returned successfully
EFI_INVALID_PARAMETER	*GuidArray* is **NULL**
EFI_INVALID_PARAMETER	*GuidEntries* is **NULL**
EFI_OUT_OF_RESOURCES	Memory allocation failure

EFI_PEI_I2C_IO.StartRequest()

Summary

Start an I^{2}C transaction on the I^{2}C device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_BUS_START_REQUEST) (
  IN EFI_PEI_I2C_IO          *This,
  IN UINTN                   SlaveAddressIndex,
  IN EFI_I2C_REQUEST_PACKET  *RequestPacket
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_IO** structure.

SlaveAddressIndex

Index value into an array of slave addresses for the I^{2}C device. The values in the array are specified by the board designer, with the I^{2}C device driver writer providing the slave address order.

For devices that have a single slave address, this value must be zero. If the I^{2}C device uses more than one slave address then the third party (upper level) I^{2}C driver writer needs to specify the order of entries in the slave address array.

RequestPacket

Address of an **EFI_I2C_REQUEST_PACKET** structure describing the I^{2}C transaction

Description

N.B. The typical consumers of this API are the third party I^{2}C drivers. Extreme care must be taken by other consumers of this API to prevent confusing the third party I^{2}C drivers due to a state change at the I^{2}C device which the third party I^{2}C drivers did not initiate. I^{2}C platform drivers may use this API within these guidelines.

This routine starts a transaction on the I^{2}C device.

Status Codes Returned

The values returned from *StartRequest* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction. One possible cause is that the slave device is not present.

EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I^2C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I^2C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^2C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

17.3.4 I^2C Bus Configuration Management PPI

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT

Summary

The EFI I^2C bus configuration management PPI enables the host driver to interact with the platform layer to reconfigure the switches and multiplexers and set the clock speed for the I^2C bus. This PPI also enables the I^2C host driver to reset an I^2C device which may be locking up the I^2C bus by holding the clock or data line low.

GUID

```
#define EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_GUID \
{ 0xe721de6f, 0x145b, 0x4532, 0xbd, 0x78, 0x9b, 0x40, \
0x95, 0xc7, 0x46, 0x97 }
```

PEIM-to-PEIM Interface Structure

```
typedef struct {

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_ENABLE_I2C_BUS_CONFIGUR
ATION EnableI2cBusConfiguration;
  EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_I2C_DEVICE_RESET
I2cDeviceReset;
} EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT;
```

Parameters

EnableI2cBusConfiguration

Enable an I^2C bus configuration for use.

I2cDeviceReset

Perform a platform specific reset for the specified I^2C part and the I^2C controller.

ControllerNumber

Unique number identifying the I^{2}C controller in the system.

Description

The I^{2}C driver stack uses the concept of an I^{2}C bus configuration as a way to describe a particular state of the switches and multiplexers in the I^{2}C bus.

A simple I^{2}C bus does not have any multiplexers or switches is described to the I^{2}C driver stack with a single I^{2}C bus configuration.

An I^{2}C bus with switches and multiplexers use an I^{2}C bus configuration to describe each of the unique settings for the switches and multiplexers. However the I^{2}C platform driver only needs to define the I^{2}C bus configurations that the software uses, which is a subset of the total.

The upper layer driver locates the correct **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** interface (I2cBusConfig) by comparing the following field:

- I2cBusConfigControllerNumber with the system unique value for the I^{2}C ControllerNumber

The I^{2}C bus configuration description includes a list of I^{2}C devices which may be accessed when this I^{2}C bus configuration is enabled. I^{2}C devices before a switch or multiplexer must be included in one I^{2}C bus configuration while I^{2}C devices after a switch or multiplexer are on another I^{2}C bus configuration.

The I^{2}C bus configuration management PPI is an optional PPI provided by the I^{2}C platform driver. The I^{2}C host driver only starts for this I^{2}C controller if the I^{2}C bus configuration management PPI is present. The I^{2}C host driver uses the I^{2}C bus configuration management PPI to call into the I^{2}C platform driver to set the switches and multiplexers and set the maximum I^{2}C bus frequency.

The platform designers determine the maximum I^{2}C bus frequency by selecting a frequency which supports all of the I^{2}C devices on the I^{2}C bus for the setting of switches and multiplexers. The platform designers must validate this against the I^{2}C device data sheets and any limits of the I^{2}C controller or bus length.

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT.EnableI2cBusConfiguration()

Summary

Enable access to an I^{2}C bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_ENABLE_I2C_BUS_CONFIGU
RATION) (
  IN EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT *This,
  IN UINTN                                     I2cBusConfiguration
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** structure.

I2cBusConfiguration

Index into a list or array of I^{2}C bus configurations

Description

Reconfigure the switches and multiplexers in the I^{2}C bus to enable access to a specific I^{2}C bus configuration. Also select the maximum clock frequency for this I^{2}C bus configuration.

This routine uses the I^{2}C Master PPI when the platform routine needs to perform I^{2}C transactions on the local bus. This eliminates any recursion in the I^{2}C stack for configuration transactions on the local bus.

The platform layer must perform I^{2}C transactions on other I^{2}C busses by using the **EFI_PEI_I2C_HOST** PPI or third party driver interface for the specific device. Use the **EFI_PEI_I2C_HOST** PPI when the device is not defined by the I^{2}C platform driver. Use the third party driver when it is available or **EFI_PEI_I2C_IO** when the third party driver is not available but the device is defined in the I^{2}C platform driver.

Status Codes Returned

The values returned from *EnableI2cBusConfiguration* are:

EFI_SUCCESS	The transaction completed successfully.
EFI_BAD_BUFFER_SIZE	The *LengthInBytes* value is too large.
EFI_DEVICE_ERROR	There was an I^{2}C error (NACK) during the transaction. One possible cause is that the slave device is not present.

EFI_INVALID_PARAMETER	*RequestPacket* is **NULL**
EFI_NOT_FOUND	I^{2}C slave address exceeds maximum address
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset!
EFI_NO_RESPONSE	The I^{2}C device is not responding to the slave address. **EFI_DEVICE_ERROR** may also be returned if the controller cannot distinguish when the NACK occurred.
EFI_OUT_OF_RESOURCES	Insufficient memory for I^{2}C transaction
EFI_TIMEOUT	The transaction did not complete within the specified timeout period.
EFI_UNSUPPORTED	The controller does not support the requested transaction.

EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT.I2cDeviceReset ()

Summary

Perform a platform specific reset for the specified I²C part and the I^{2}C controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI
*EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT_I2C_DEVICE_RESET) (
  IN CONST EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT *This,
  IN UINTN                                  I2cBusConfiguration,
  IN UINTN                                  SlaveAddress
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_BUS_CONFIGURATION_MANAGEMENT** structure.

I2cBusConfiguration

Index into a list or array of I^{2}C bus configurations

SlaveAddress

Address of the device on the I^{2}C bus.

Description

This error handling routine is called by the I^{2}C host driver when **EFI_TIMEOUT** status is returned by the I^{2}C port driver for an **EFI_I2C_REQUEST_PACKET**. This routine attempts to reset the failing I^{2}C device and the I^{2}C controller.

Status Codes Returned

The values returned from *I2cDeviceReset* are:

EFI_SUCCESS	The transaction completed successfully. Only the requested I^{2}C device and the I^{2}C controller were reset.
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value due to invalid platform data.
EFI_NO_MEDIA	State was lost because more than one device was reset! The host driver needs to return errors for the queue of pending **EFI_I2C_REQUEST_PACKETs**.

17.3.5 I^{2}C Enumerate PPI

EFI_PEI_I2C_ENUMERATE

Summary

Support the enumeration of the I^{2}C devices listed in the I^{2}C platform driver.

GUID

```
#define EFI_PEI_I2C_ENUMERATE_GUID \
{ 0xbe83f6f4, 0xe286, 0x4e70, 0xb4, 0x51, 0x1a, 0x2e, \
0x42, 0xdf, 0x31, 0x03 }
```

Protocol Interface Structure

```
typedef struct {
  EFI_PEI_I2C_ENUMERATE_ENUMERATE         Enumerate;
  EFI_PEI_I2C_ENUMERATE_GET_BUS_FREQUENCY GetBusFrequency;
} EFI_PEI_I2C_ENUMERATE;
```

Parameters

Enumerate

Walk the platform's list of I^{2}C devices on the bus. This routine returns the next I^{2}C device in the platform's list for this I^{2}C bus.

GetBusFrequency

Get the I^{2}C bus frequency for the bus configuration.

Description

The I^{2}C bus driver uses this PPI to enumerate the devices on the I^{2}C bus listed in the platform layer.

The upper layer driver locates the correct **EFI_PEI_I2C_ENUMERATE** interface (I2cEnumerate) by comparing the following field:

- *I2cEnumerateControllerNumber* with the system unique value for the I^{2}C *ControllerNumber*.

EFI_PEI_I2C_ENUMERATE_PROTOCOL.Enumerate()

Summary

Enumerate the I^{2}C devices

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_ENUMERATE_ENUMERATE) (
  IN EFI_PEI_I2C_ENUMERATE     *This,
  IN OUT CONST EFI_I2C_DEVICE  **Device
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_ENUMERATE** structure.

Device

Address of a buffer containing an **EFI_I2C_DEVICE** structure. Enumeration is started by setting the initial **EFI_I2C_DEVICE** structure address to NULL. The buffer receives an **EFI_I2C_DEVICE** structure address for the next I^{2}C device.

Description

This function walks the platform specific data to enumerate the I^{2}C devices on an I^{2}C bus.

Status Codes Returned

The values returned from *Enumerate* are:

EFI_SUCCESS	The platform data for the next device on the I^{2}C bus was returned successfully.
EFI_INVALID_PARAMETER	Device was **NULL**
EFI_NO_MAPPING	Device does not point to a valid **EFI_I2C_DEVICE** structure.

EFI_PEI_I2C_ENUMERATE_PROTOCOL.GetBusFrequency()

Summary

Get the I^{2}C bus frequency for the bus configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_I2C_ENUMERATE_GET_BUS_FREQUENCY) (
  IN EFI_PEI_I2C_ENUMERATE   *This,
  IN UNITN                   I2cBusConfiguration,
  IN OUT UINTN               *BusClockHertz
  );
```

Parameters

This

Address of an **EFI_PEI_I2C_ENUMERATE** structure.

I2cBusConfiguration

I^{2}C bus configuration to access the I^{2}C device

BusClockHertz

Address to receive the I^{2}C bus clock frequency in Hertz

Description

This function returns the I^{2}C bus clock frequency for the specified I^{2}C bus configuration.

Status Codes Returned

The values returned from *GetBusFrequency* are:

EFI_SUCCESS	The I^{2}C bus frequency was returned successfully.
EFI_INVALID_PARAMETER	*BusClockHertz* was **NULL**
EFI_NO_MAPPING	Invalid *I2cBusConfiguration* value

18 SPI Protocol Stack

18.1 Design Discussion

The SPI protocol stack enables third party silicon vendors to write UEFI drivers for their products by decoupling the SPI chip details from the SPI controller and SPI bus configuration details.

18.1.1 SPI Bus Overview

Each peripheral on the SPI bus share the clock, data out and data in lines. The peripheral is addressed by using a unique chip select line. Communications with the peripheral must be done at or below the maximum clock rate which the peripheral supports and must use the proper clock polarity and phase.

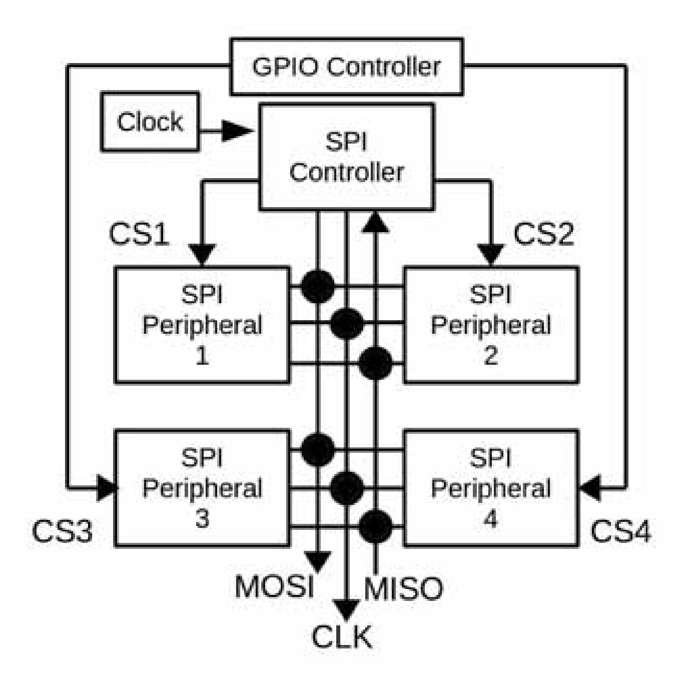


Figure 16. SPI Bus

The SPI controller must contain the data shift register and clock gating logic which honors clock phase, clock polarity and only presents clock pulses when valid data is on the SPI bus. The SPI controller must pause the clock while waiting for more data.

Independent logic blocks may provide the clock frequency used by the SPI controller as well as the GPIOs used for the SPI chip selects.

18.1.2 SPI Protocol Stack Overview

The SPI driver stack is being split on functional lines. Most of the complexity ends up in the SPI bus layer, simplifying the SPI peripheral, SPI controller and board layers.

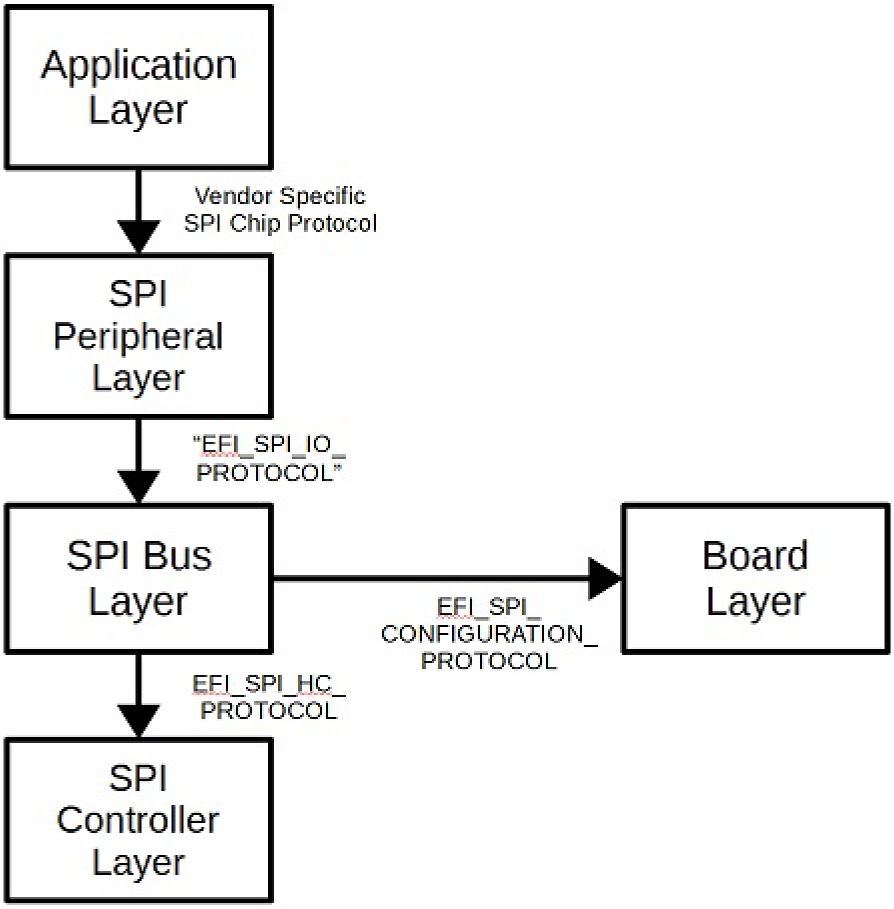


Figure 17. SPI Layers

The SPI protocol layers are:

- Application Layer - Applications using the SPI chips
- SPI Peripheral Layer - Converts an SPI chip request into one or more transactions on the SPI bus
- SPI Bus Layer - Handles:
  - SPI Peripheral Device Enumeration
  - SPI Transaction Management
  - SPI Controller Management
- SPI Host Controller Layer - Handles details of the SPI controller
- SPI Board Layer - Contains:
  - SPI bus descriptions
  - SPI part descriptions
  - Alternative SPI bus clock support
  - Alternative SPI chip select support

The SPI bus layer provides a data connection point with an **EFI_SPI_IO_PROTOCOL** data structure for each SPI peripheral. This data connection point exports the *SpiPeripheralDriverGuid* from the **EFI_SPI_PART** data structure. The SPI peripheral drivers connect to the connection points with the corresponding GUID.

An example:
A generic SPI flash driver is written and provides the GUID {5993c862-5c3f-4ae8-804d-8c89ad962c31} for use by SPI flash peripherals that meet the criteria specified by the developer of the SPI flash driver. The board developer chooses an SPI flash part, let's say a WinBond W25Q64FV. Ideally the SPI chip vendor would provide a header file containing the **EFI_SPI_PART** definition. When that is not available, the board developer could create an **EFI_SPI_PART** data structure and fill it with data from the datasheet as follows:

```
CONST EFI_SPI_PART Winbond_W25Q64FV = {
  L"Winbond",
  L"W25Q64FV",
  0,
  MHz(104), // Page 75, 3.0V - 3.6V
  FALSE // Page 6, Section 3.1
};
```

When the SPI bus layer creates the **EFI_SPI_IO_PROTOCOL** data structure for this device, the generic SPI flash driver is able to find and connect to it by calling **OpenProtocol** with the GUID specified above.

Some SPI chip examples:

- Maxim MAX3111E - UART and RS232 transceiver
- Maxim MAX6950 - Seven segment numeric LED controller
- Serial SPI NOR flash, one of:
  - Atmel AT25DF321 - 4 MiB SPI NOR flash
  - Winbond W2SQ80DV - 1 MiB SPI NOR flash
  - Winbond W25Q16DV - 2 MiB SPI NOR flash
  - Winbond W25Q32FV - 4 MiB SPI NOR flash
  - Winbond W25Q64FV - 8 MiB SPI NOR flash
  - Spansion S25FL164K - 16 MiB SPI NOR flash
  - Micron N2SQ128A - 32 MiB SPI NOR flash
  - Winbond W2SQ128FV - 32 MiB SPI NOR flash

The board vendor may provide example code which describes the SPI buses and SPI peripherals to simplify the configuration process. The example code may be modified by the board consumer to adjust for added SPI devices or SPI buses.

18.1.3 Application Layer

The application layer interacts with various chip specific drivers using vendor specific protocols.

Example applications are:

- A background application which reads gets the system time once per second and uses the MAX6950 driver to display the time on a four seven-segment displays.

- A background application which uses the Texas Instruments ADC108S102 driver to read a 10-bit voltage value from the analog-to-digital converter and displays the result on seven-segment displays driven by the MAX6950.

18.1.4 SPI Peripheral Layer

This layer provides vendor specific interfaces to the SPI chips. The upper interface to the SPI peripheral layer is chip specific and determined by the UEFI or Pl specifications where there is a standard or by the chip vendor when a standard does not exist.

The lower interface of the SPI peripheral layer connects to one or more of the SPI peripherals exposed by the SPI bus layer as an **EFI_SPI_IO_PROTOCOL** instance. The SPI peripheral driver interacts with the SPI chip by issuing data transactions to the SPI bus layer. These transactions make their way to the SPI controller layer where they are placed onto the SPI bus and data is exchanged with the SPI chip.

18.1.5 SPI I/O Interface

The SPI bus layer creates an **EFI_SPI_IO_PROTOCOL** instance for each SPI chip listed in the board layer. However unlike other protocols, the **EFI_SPI_IO_PROTOCOL** instance is identified by a GUID that is unique to the SPI chip driver to which it should connect. This differs from other bus protocols which produce a bus-specific GUID.

The rational behind this is decision is based upon:

1. No common device support

2. Performance and code size

3. SPI device enumeration performed infrequently

18.1.5.1 No Common Device Support

With other protocols such as PCI and USB, even though the devices differ, there is a common hardware support layer for peripheral identity, resource allocation and attaching the device to the bus. SPI chips differ in this respect because there are no hardware standards! Bus attachment is done by the board developer at a hardware level. From a software viewpoint as soon as chip select is asserted, the SPI chip is on the bus. Also with SPI there are no common commands that may be issued to identify or enable the chip. As such there is no advantage exposing the **EFI_SPI_IO_PROTOCOL** with a generic GUID.

18.1.5.2 Performance and Code Size

Times have changed since UEFI was originally architected and implemented. In todays world, the firmware engineers are being asked for sub-one-second boot times and a smaller firmware footprint with more functionality. Using a generic GUID for the **EFI_SPI_IO_PROTOCOL** requires that each SPI peripheral driver implement more code to verify some other identifier to determine if the driver should use this device. This additional check adds cost and complexity to the SPI peripheral driver. The costs are the development time to implement and debug the code as well as the CPU time to execute the code. This code also has a multiplicative effect on the firmware footprint.

18.1.5.3 SPI Device Enumeration Performed Infrequently

The claim is that SPI device enumeration will be performed infrequently.

- How often is SPI device enumeration a necessary operation?
- Is it a requirement that this operation be done using a generic GUID for **EFI_SPI_IO_PROTOCOL**?
- What are the use cases for this operation?

Eliminating the generic GUID for **EFI_SPI_IO_PROTOCOL** removes one way of doing SPI device enumeration. However SPI device enumeration is still possible.

SPI device enumeration is easiest done using the **EFI_SPI_CONFIGURATION_PROTOCOL**. From this protocol is possible to determine the SPI buses in the system and the devices which are attached to these buses. With a little extra work, calling *LocateHandleBuffer* it is possible to identify the handles which use the SPI driver GUID and match the **EFI_SPI_IO_PROTOCOL** interface.

Device paths may also be used to find handles which are attached to a specific SPI host controller. Each SPI peripheral attaches a **HW_CONTROLLER** node to the device path.

18.1.5.4 Synchronous Operation

All SPI I/O layer transactions are synchronous. No support is provided for asynchronous transactions.

18.1.5.5 SPI Transaction Management

The SPI I/O layer allocates a **EFI_SPI_BUS_TRANSACTION** data structure which contains the parameters that will be passed to the SPI host controller as part of this SPI transaction. The SPI bus layer uses this structure to control and complete the SPI transaction.

Synchronizing with the SPI bus layer schedules the SPI transaction on a free SPI host controller.

18.1.6 SPI Bus Layer

The SPI bus layer manages the SPI transactions for each of the host's SPI controllers. SPI peripheral drivers submit SPI transactions to the SPI bus layer which in turn submits them to the host's SPI controller.

The SPI transaction consists of:

1. Adjusting the clock speed, polarity and phase for an SPI peripheral
2. Use the chip select to enable the SPI peripheral, signaling the transaction start to the chip
3. Transfer the data in one or both directions simultaneously
4. Remove the chip select from the SPI peripheral signaling the transaction end to the chip
5. Optionally, shutdown the SPI controller's internal clock to reduce power

The SPI bus layer is responsible setting up the SPI clock and chip select. This ensures that the chip set up is done properly across all SPI controller drivers. The SPI bus layer uses the SPI chip data from the board layer to determine the clock phase and polarity. The clock frequency is the lowest frequency specified by:

- Maximum SPI controller clock frequency
- Supported SPI controller clock frequency < = SPI chip maximum clock frequency

- Supported SPI controller clock frequency < = non-zero *ClockHz*

After setting up the clock, the SPI bus layer asserts the appropriate chip select and then passes the SPI transaction to the SPI controller to start the data flow in both directions. Upon completion, the SPI bus layer deasserts the chip select and completes the SPI transaction to the SPI peripheral layer.

18.1.6.1 Half Duplex SPI controllers

Various SPI controllers support a half-duplex operation in addition to the full-duplex operation. The benefits of the half- duplex operation on the system are that less physical memory tied up during the operation and the memory bandwidth is cut in half for the operation.

It is beneficial to the system for the SPI architecture to support the half duplex operations of the SPI controller. Additionally it reduces code size and memory footprint by eliminating unnecessary buffers in the SPI peripheral drivers when half-duplex operations are performed on the SPI chip.

Write Then Read Operations

SPI flash chips and some SPI UARTs support write then read operations using SPI. The NXP SC16IS750/760 is an example of a UART designed for I2C and SPI which performs half-duplex operations which are a mix of transmit and receive.

SPI NOR flash chips such as the Winbond W25Q64FV also perform half-duplex operations which are a mix of transmit and receive. These operation consist of writing a command byte and possibly an address and then immediately reading data either from the status register or memory.

SPI Controller Support

Since SPI is inherently full-duplex, the SPI host controller should support full-duplex operation. Not all SPI controllers however are able to support half-duplex or write-then-read operation. In this case, the SPI bus layer converts the SPI transaction into a full-duplex transaction by allocating the necessary buffers and if necessary coping any write data into the new buffer. The SPI bus layer then hands the full-duplex transaction to the SPI host controller for processing. Upon completion, the SPI bus driver copies any necessary data from the full-duplex buffers into the SPI peripheral layer's receive buffer and then frees the allocated full-duplex buffers. This conversion allows the SPI peripheral drivers to choose transaction types which optimize system resources and performance in the general case.

SPI Controller To SPI Bus Connection

For DXE, the SPI host controller is identified by the device path. The SPI board layer includes copy of this device path in the *ControllerPath* field of the **EFI_SPI_BUS** structure. When the DXE version of the SPI bus layer locates an SPI host controller, the SPI bus layer matches the device path for the SPI host controller to the device field in the *ControllerPath* of the **EFI_SPI_BUS** structure in the board layer. Once this connection is made, the SPI bus layer can create the necessary SPI I/O interfaces.

18.1.7 SPI Host Controller Layer

The SPI host controller layer provides a simple interface to the SPI controller. This layer only handles SPI controller details for a single transaction.

The support at this layer is broken into three primary routines:

- Clock set up
- Chip selection
- Data movement

The SPI bus layer calls these routines to initiate and complete the SPI transaction.

18.1.7.1 Legacy SPI Flash Controller

The legacy SPI flash controller is designed to handle SPI NOR flash devices. This controller has several limitations and several security enhancements that impact the design of the SPI bus I/O layers and the SPI NOR flash peripheral driver. The security enhancements include:

- BIOS base address
- Protect range registers
- Controller configuration lock
  - Prefix type table
  - Opcode menu table
  - Opcode type table
  - BIOS base address
  - Protect range registers

The security enhancements are handled by the **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL**. This protocol provides the functions support the security features of the legacy SPI flash controller as well as functions to work around the flash targeted design of the controller.

The limitations of the legacy SPI flash controller include:

- 8-bit frames only
- Fixed clock rate
- No full-duplex transaction support
- No read-only transaction support
- Reads: 64-byte maximum transfer length
- Writes: 67-byte maximum transfer length
- Prefix opcode table
- Opcode menu table

These limitations are handled by:

- Setting frame size support to 8-bit only
- Using the legacy SPI flash controller's clock routine to validate the requested clock frequency
- Letting the legacy SPI flash controller's transaction routine fail the full-duplex and read-only transactions
- Setting the maximum transfer size to 64 bytes
- Adding a couple of flags to indicate that the opcode and 3 address bytes are included in the maximum transfer size
- Using the **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** to set the erase block opcode

- Using the **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** to set the write status prefix opcode

18.2 DXE Code Definitions

The SPI protocol stack consists of the following protocols:

- **EFI_LEGACY_SPI_FLASH_PROTOCOL** - The upper layers use this protocol to access the BIOS space address and protection registers of the legacy SPI flash controller.
- **EFI_SPI_NOR_FLASH_PROTOCOL** - The upper layers use this protocol to interact with SPI NOR flash devices.
- **EFI_SPI_IO_PROTOCOL** - The SPI peripheral drivers use this to interact with chips on the SPI bus.
- **EFI_SPI_HC_PROTOCOL** - The SPI bus layer uses this to interact with the host's SPI controller.
- **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** - The flash layer uses this protocol to invoke the additional functions provided by the legacy SPI controller.
- **EFI_SPI_CONFIGURATION_PROTOCOL** - The SPI bus layer uses this to interact with the board layer database and additional logic blocks for clock and GPIO controllers.

EFI_SPI_CONFIGURATION_PROTOCOL

Summary

Describe the details of the board's SPI buses to the SPI driver stack.

GUID

```
// {85a6d3e6-b65b-4afc-b38f-c6d54af6ddc8}
#define EFI_SPI_CONFIGURATION_GUID \
{ 0x85a6d3e6, 0xb65b, 0x4afc, { 0xb3, 0x8f, 0xc6, 0xd5, \
0x4a, 0xf6, 0xdd, 0xc8 }}
```

Protocol Interface Structure

```
typedef struct _EFI_SPI_CONFIGURATION_PROTOCOL {
  UINT32 BusCount;
  CONST EFI_SPI_BUS *CONST *CONST Buslist;
} EFI_SPI_CONFIGURATION_PROTOCOL;
```

Parameters

BusCount

The number of SPI buses on the board.

Buslist

The address of an array of **EFI_SPI_BUS** data structure addresses.

Description

The board layer uses the **EFI_SPI_CONFIGURATION_PROTOCOL** to expose the data tables which describe the board's SPI buses. The SPI bus layer uses these tables to configure the clock and chip select, and manage the SPI transactions on the SPI controllers.

The configuration tables describe:

- The number of SPI buses on the board
- Which SPI chips are connected to each SPI bus

For each SPI chip the configuration describes:

- The maximum clock frequency for the SPI part
- The clock polarity needed for the SPI part
- Whether the SPI controller uses a separate clock generator that needs to be set up
- The chip select polarity
- Whether the SPI controller or a GPIO pin is used for the chip select
- The data sampling edge for the SPI part.

Related Definitions

The **EFI_SPI_PERIPHERAL** and **EFI_SPI_BUS** data structures are defined later in this section.

EFI_SPI_CHIP_SELECT

Summary

Manipulate the chip select for an SPI device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_CHIP_SELECT) (
  IN CONST EFI_SPI_PERIPHERAL *SpiPeripheral,
  IN BOOLEAN PinValue
  );
```

Parameters

SpiPeripheral

The address of an **EFI_SPI_PERIPHERAL** data structure describing the SPI peripheral whose chip select pin is to be manipulated. The routine may access the *ChipSelectParameter* field to gain sufficient context to complete the operation.

PinValue

The value to be applied to the chip select line of the SPI peripheral.

Description

This routine must be called at or below **TPL_NOTIFY**.

Update the value of the chip select line for an SPI peripheral. The SPI bus layer calls this routine either in the board layer or in the SPI controller to manipulate the chip select pin at the start and end of an SPI transaction.

Status Codes Returned

EFI_SUCCESS	The chip select was set successfully
EFI_NOT_READY	Support for the chip select is not properly initialized
EFI_INVALID_PARAMETER	The *SpiPeripheral*->*ChipSelectParameter* value is invalid

EFI_SPI_PART

Summary

Describe the properties of an SPI chip.

Prototype

```
typedef struct _EFI_SPI_PART
{
  CONST CHAR16 *Vendor;
  CONST CHAR16 *PartNumber;
  UINT32 MinClockHz;
  UINT32 MaxClockHz;
  BOOLEAN ChipSelectPolarity;
} EFI_SPI_PART;
```

Parameters

Vendor

A Unicode string specifying the SPI chip vendor.

PartNumber

A Unicode string specifying the SPI chip part number.

MinClockHz

The minimum SPI bus clock frequency used to access this chip. This value may be specified in the chip's datasheet. If not, use the value of zero.

MaxClockHz

The maximum SPI bus clock frequency used to access this chip. This value is found in the chip's datasheet.

ChipSelectPolarity

Specify the polarity of the chip select pin. This value can be found in the SPI chip's datasheet. Specify *TRUE* when a one asserts the chip select and *FALSE* when a zero asserts the chip select.

Description

The **EFI_SPI_PART** data structure provides a description of an SPI part which is independent of the use on the board. This data is available directly from the part's datasheet and may be provided by the vendor.

EFI_SPI_PERIPHERAL

Summary

Describe the board specific properties associated with a specific SPI chip.

Prototype

```
typedef struct _EFI_SPI_PERIPHERAL
{
  CONST EFI_SPI_PERIPHERAL *NextSpiPeripheral;
  CONST CHAR16 *FriendlyName;
  CONST GUID *SpiPeripheralDriverGuid;
  CONST EFI_SPI_PART *SpiPart;
  UINT32 MaxClockHz;
  BOOLEAN ClockPolarity;
  BOOLEAN ClockPhase;
  UINT32 Attributes;
  CONST VOID *ConfigurationData;
  CONST EFI_SPI_BUS *SpiBus;
  EFI_SPI_CHIP_SELECT ChipSelect;
  VOID *ChipSelectParameter;
} EFI_SPI_PERIPHERAL;
```

Parameters

NextSpiPeripheral

Address of the next **EFI_SPI_PERIPHERAL** data structure. Specify NULL if the current data structure is the last one on the SPI bus.

FriendlyName

A unicode string describing the function of the SPI part.

SpiPeripheralDriverGuid

Address of a GUID provided by the vendor of the SPI peripheral driver. Instead of using a "**EFI_SPI_IO_PROTOCOL**" GUID, the SPI bus driver uses this GUID to identify an **EFI_SPI_IO_PROTOCOL** data structure and to provide the connection points for the SPI peripheral drivers. This reduces the comparison logic in the SPI peripheral **EFI_DRIVER_BINDING_PROTOCOL.Supported()** routine.

SpiPart

The address of an **EFI_SPI_PART** data structure which describes this chip.

MaxClockHz

The maximum clock frequency is specified in the **EFI_SPI_PART**. When this value is non-zero and less than the value in the **EFI_SPI_PART** then this value is used for the maximum clock frequency for the SPI part.

ClockPolarity

Specify the idle value of the clock as found in the datasheet. Use zero (0) if the clock's idle value is low or one (1) if the clock's idle value is high.

ClockPhase

Specify the clock delay after chip select. Specify zero (0) to delay an entire clock cycle or one (1) to delay only half a clock cycle.

Attributes

SPI peripheral attributes, select zero or more of:

- **SPI_PART_SUPPORTS_2_BIT_DATA_BUS_WIDTH** - The SPI peripheral is wired to support a two-bit data bus
- **SPI_PART_SUPPORTS_4_BIT_DATA_BUS_WIDTH** - The SPI peripheral is wired to support a four-bit data bus

ConfigurationData

Address of a vendor specific data structure containing additional board configuration details related to the SPI chip. The SPI peripheral layer uses this data structure when configuring the chip.

SpiBus

The address of an **EFI_SPI_BUS** data structure which describes the SPI bus to which this chip is connected.

ChipSelect

Address of the routine which controls the chip select pin for this SPI peripheral. Call the SPI host controller's chip select routine when this value is set to *NULL*.

ChipSelectParameter

Address of a data structure containing the additional values which describe the necessary control for the chip select. When *ChipSelect* is *NULL*, the declaration for this data structure is provided by the vendor of the host's SPI controller driver. The vendor's documentation specifies the necessary values to use for the chip select pin selection and control.

When *Chipselect* is not *NULL*, the declaration for this data structure is provided by the board layer.

Description

The **EFI_SPI_PERIPHERAL** data structure describes how a specific block of logic which is connected to the SPI bus. This data structure also selects which upper level driver is used to manipulate this SPI device. The *SpiPeripheraLDriverGuid* is available from the vendor of the SPI peripheral driver.

EFI_SPI_CLOCK

Summary

Set up the clock generator to produce the correct clock frequency, phase, and polarity for an SPI chip.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_CLOCK) (
  IN CONST EFI_SPI_PERIPHERAL *SpiPeripheral,
  IN UINT32 *ClockHz
  );
```

Parameters

SpiPeripheral

Pointer to a **EFI_SPI_PERIPHERAL** data structure from which the routine can access the *ClockParameter*, *ClockPhase*, and *ClockPolarity* fields. The routine also has access to the names for the SPI bus and chip which can be used during debugging.

ClockHz

Pointer to the requested clock frequency. The clock generator will choose a supported clock frequency which is less then or equal to this value. Specify zero to turn the clock generator off. The actual clock frequency supported by the clock generator will be returned.

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine updates the clock generator to generate the correct frequency and polarity for the SPI clock.

Status Codes Returned

EFI_SUCCESS	The clock was set up successfully
EFI_UNSUPPORTED	The SPI controller was not able to support the frequency requested by *ClockHz*

EFI_SPI_BUS

Summary

Describe the board specific details associated with an SPI bus.

Prototype

```
typedef struct _EFI_SPI_BUS {
  CONST CHAR16 *FriendlyName;
  CONST EFI_SPI_PERIPHERAL *Peripherallist;
  CONST EFI_DEVICE_PATH_PROTOCOL *ControllerPath;
  EFI_SPI_CLOCK Clock;
  VOID *ClockParameter;
} EFI_SPI_BUS;
```

Parameters

FriendlyName

A Unicode string describing the SPI bus

Peripheral List

Address of the first **EFI_SPI_PERIPHERAL** data structure connected to this bus. Specify *NULL* if there are no SPI peripherals connected to this bus.

ControllerPath

Address of an **EFI_DEVICE_PATH_PROTOCOL** data structure which uniquely describes the SPI controller.

Clock

Address of the routine which controls the clock used by the SPI bus for this SPI peripheral. The SPI host controller's clock routine is called when this value is set to NULL.

ClockParameter

Address of a data structure containing the additional values which describe the necessary control for the clock. When *Clock* is *NULL*, the declaration for this data structure is provided by the vendor of the host's SPI controller driver. When *Clock* is not *NULL*, the declaration for this data structure is provided by the board layer.

Description

The **EFI_SPI_BUS** data structure provides the connection details between the physical SPI bus and the **EFI_SPI_HC_PROTOCOL** instance which controls that SPI bus. This data structure also describes the details of how the clock is generated for that SPI bus. Finally this data structure provides the list of physical SPI devices which are attached to the SPI bus.

Summary

Macros to easily specify frequencies in hertz, kilohertz and megahertz.

Prototype

```
#define Hz(Frequency)     (Frequency)
#define KHz(Frequency)    (1000 * Hz(Frequency))
#define MHz(Frequency)    (1000 * KHz(Frequency))
```

EFI_SPI_NOR_FLASH_PROTOCOL

Summary

The **EFI_SPI_NOR_FLASH_PROTOCOL** exists in the SPI peripheral layer. This protocol manipulates the SPI NOR flash parts using a common set of commands. The board layer provides the interconnection and configuration details for the SPI NOR flash part. The SPI NOR flash driver uses this configuration data to expose a generic interface which provides the following APIs:

- Read manufacture and device ID
- Read data
- Read data using low frequency

- Read status
- Write data
- Erase 4 KiB blocks
- Erase 32 or 64 KiB blocks
- Write status

The **EFI_SPI_NOR_FLASH_PROTOCOL** also exposes some APIs to set the security features on the legacy SPI flash controller.

GUID

```
// {b57ec3fe-f833-4ba6-8578-2a7d6a87444b}
#define EFI_SPI_NOR_FLASH_PROTOCOL_GUID \
{ 0xb57ec3fe, 0xf833, 0x4ba6, { 0x85, 0x78, 0x2a, 0x7d, \
 0x6a, 0x87, 0x44, 0x4b }}
```

Protocol Interface Structure

```
struct _EFI_SPI_NOR_FLASH_PROTOCOL {
  CONST EFI_SPI_PERIPHERAL *SpiPeripheral;
  UINT32 FlashSize;
  UINT8 DeviceId [3];
  UINT32 EraseBlockBytes;
  EFI_SPI_NOR_FLASH_PROTOCOL_GET_FLASH_ID GetFlashId;
  EFI_SPI_NOR_FLASH_PROTOCOL_READ_DATA ReadData;
  EFI_SPI_NOR_FLASH_PROTOCOL_READ_DATA LfReadData;
  EFI_SPI_NOR_FLASH_PROTOCOL_READ_STATUS ReadStatus;
  EFI_SPI_NOR_FLASH_PROTOCOL_WRITE_STATUS WriteStatus;
  EFI_SPI_NOR_FLASH_PROTOCOL_WRITE_DATA WriteData;
  EFI_SPI_NOR_FLASH_PROTOCOL_ERASE Erase;
};
```

Parameters

SpiPeripheral

Pointer to an **EFI_SPI_PERIPHERAL** data structure

FlashSize

Flash size in bytes

DeviceId

Manufacturer and Device ID

EraseBlockBytes

Erase block size in bytes

18.2.0.1 SPI Flash Driver GUID

Use a pointer to *gEfiSpiNorFlashDriverGuid* in the **EFI_SPI_PERIPHERAL** structure to connect an SPI NOR flash part to the SPI flash driver.

EFI_SPI_NOR_FLASH_PROTOCOL.GetFlashId()

Summary

Read the 3 byte manufacture and device ID from the SPI flash.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_GET_FLASH_ID) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  OUT UINT8 *Buffer
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

Buffer

Pointer to a 3 byte buffer to receive the manufacturer and device ID.

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine reads the 3 byte manufacture and device ID from the flash part filling the buffer provided.

Status Codes Returned

EFI_SUCCESS	The manufacture and device ID was read successfully.
EFI_INVALID_PARAMETER	*Buffer* is *NULL*
EFI_DEVICE_ERROR	Invalid data received from SPI flash part.

EFI_SPI_NOR_FLASH_PROTOCOL.ReadData()

Summary

Read data from the SPI flash.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_READ_DATA) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 FlashAddress,
  IN UINT32 LengthInBytes, OUT UINT8 *Buffer
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

FlashAddress

Address in the flash to start reading

LengthInBytes

Read length in bytes

Buffer

Address of a buffer to receive the data

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine reads data from the SPI part in the buffer provided.

Status Codes Returned

EFI_SUCCESS	The data was read successfully.
EFI_INVALID_PARAMETER	*Buffer* is *NULL*
EFI_INVALID_PARAMETER	*FlashAddress >= This->FlashSize*
EFI_INVALID_PARAMETER	*LengthInBytes > This->FlashSize - FlashAddress*

EFI_SPI_NOR_FLASH_PROTOCOL.LfReadData()

Summary

Low frequency read data from the SPI flash.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_READ_DATA) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 FlashAddress,
  IN UINT32 LengthInBytes,
  OUT UINTS *Buffer
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

FlashAddress

Address in the flash to start reading

LengthInBytes

Read length in bytes

Buffer

Address of a buffer to receive the data

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine reads data from the SPI part in the buffer provided.

Status Codes Returned

EFI_SUCCESS	The data was read successfully.
EFI_INVALID_PARAMETER	*Buffer* is *NULL*
EFI_INVALID_PARAMETER	*FlashAddress >= This->FlashSize*
EFI_INVALID_PARAMETER	*LengthInBytes > This->FlashSize - FlashAddress*

EFIEFI_SPI_NOR_FLASH_PROTOCOL.ReadStatus()

Summary

Read the flash status register.

Prototype

```
typedef
EFI STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_READ_STATUS) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 LengthInBytes,
  OUT UINT8 *FlashStatus
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

LengthInBytes

Number of status bytes to read.

FlashStatus

Pointer to a buffer to receive the flash status.

Description

This routine must be called at or below **TPL_NOTIFY**. This routine reads the flash part status register.

Status Codes Returned

EFI_SUCCESS	The status register was read successfully.

EFI_SPI_NOR_FLASH_PROTOCOL.WriteStatus()

Summary

Write the flash status register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_WRITE_STATUS) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 LengthInBytes,
  IN UINT8 *FlashStatus
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

LengthInBytes

Number of status bytes to write.

FlashStatus

Pointer to a buffer containing the new status.

Description

This routine must be called at or below **TPL_NOTIFY**. This routine writes the flash part status register.

Status Codes Returned

EFI_SUCCESS	The status write was successful.
EFI_OUT_OF_RESOURCES	Failed to allocate the write buffer.

EFI_SPI_NOR_FLASH_PROTOCOL.WriteData()

Summary

Write data to the SPI flash.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_WRITE_DATA) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 FlashAddress,
  IN UINT32 LengthInBytes,
  IN UINT8 *Buffer
  );
```

Parameters

This

Pointer to an **EFI_SPI_NOR_FLASH_PROTOCOL** data structure.

FlashAddress

Address in the flash to start writing

LengthInBytes

Write length in bytes

Buffer

Address of a buffer containing the data

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine breaks up the write operation as necessary to write the data to the SPI part.

Status Codes Returned

EFI_SUCCESS	The data was written successfully.
EFI_INVALID_PARAMETER	*Buffer* is *NULL*
EFI_INVALID_PARAMETER	*FlashAddress >= This->FlashSize*
EFI_INVALID_PARAMETER	*LengthInBytes > This->FlashSize - FlashAddress*
EFI_OUT_OF_RESOURCES	Insufficient memory to copy buffer.

EFI_SPI_NOR_FLASH_PROTOCOL.Erase()

Summary

Efficiently erases one or more 4KiB regions in the SPI flash.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_NOR_FLASH_PROTOCOL_ERASE) (
  IN CONST EFI_SPI_NOR_FLASH_PROTOCOL *This,
  IN UINT32 FlashAddress,
  IN UINT32 BlockCount
  );
```

Parameters

This

Pointer to an EFI_SPI_NOR_FLASH_PROTOCOL data structure.

FlashAddress

Address within a 4 KiB block to start erasing

BlockCount

Number of 4 KiB blocks to erase

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine uses a combination of 4 KiB and larger blocks to erase the specified area.

Status Codes Returned

EFI_SUCCESS	The erase was completed successfully.
EFI_INVALID_PARAMETER	*FlashAddress >= This->FlashSize*
EFI_INVALID_PARAMETER	*BlockCount * 4 KiB > This->FlashSize - FlashAddress*

EFI_LEGACY_SPI_FLASH_PROTOCOL

Summary

The EFI_LEGACY _SPI_FLASH_PROTOCOL extends the EFI_SPI_NOR_FLASH_PROTOCOL with APIs to support the legacy SPI flash controller.

GUID

```
// {f01bed57-04bc-4f3f-9660-d6f2ea228259}
#define EFI_LEGACY_SPI_FLASH_PROTOCOL_GUID \
{ 0xf01bed57, 0x04bc, 0x4f3f, { 0x96, 0x60, 0xd6, 0xf2, \
  0xea, 0x22, 0x82, 0x59 }}
```

Protocol Interface Structure

```
struct _EFI_LEGACY_SPI_FLASH_PROTOCOL {
  EFI_SPI_NOR_FLASH_PROTOCOL FlashProtocol;
  ///
  /// Legacy flash (SP! host) controller support
  ///
  EFI_LEGACY_SPI_FLASH_PROTOCOL_BIOS_BASE_ADDRESS
BiosBaseAddress;
  EFI_LEGACY_SPI_FLASH_PROTOCOL_CLEAR_SPI_PROTECT
ClearSpiProtect;
  EFI_LEGACY_SPI_FLASH_PROTOCOL_IS_RANGE_PROTECTED
IsRangeProtected;
  EFI_LEGACY_SPI_FLASH_PROTOCOL_PROTECT_NEXT_RANGE
ProtectNextRange;
  EFI_LEGACY_SPI_FLASH_PROTOCOL_LOCK_CONTROLLER LockController;
};
```

EFI_LEGACY _SPI_FLASH_PROTOCOL.BiosBaseAddress()

Summary

Set the BIOS base address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_FLASH_PROTOCOL_BIOS_BASE_ADDRESS) (
  IN CONST EFI_LEGACY_SPI_FLASH_PROTOCOL *This,
  IN UINT32 BiosBaseAddress
  );
```

Parameters

This

Pointer to an EFI_LEGACY_SPI_FLASH_PROTOCOL data structure.

BiosBaseAddress

The BIOS base address.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS base address works with the protect range registers to protect portions of the SPI NOR flash from erase and write operations . The BIOS calls this API prior to passing control to the OS loader.

Status Codes Returned

EFI_SUCCESS	The BIOS base address was properly set
EFI_ACCESS_ERROR	The SPI controller is locked
EFI_INVALID_PARAMETER	*BiosBaseAddress > This->MaximumOffset*
EFI_UNSUPPORTED	The BIOS base address was already set
EFI_UNSUPPORTED	Not a legacy SPI host controller

EFI_LEGACY _SPI_FLASH_PROTOCOL.ClearSpiProtect()

Summary

Clear the SPI protect range registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_FLASH_PROTOCOL_CLEAR_SPI_PROTECT) (
  IN CONST EFI_LEGACY_SPI_FLASH_PROTOCOL *This
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_FLASH_PROTOCOL** data structure.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS uses this routine to set an initial condition on the SPI protect range registers.

Status Codes Returned

EFI_SUCCESS	The registers were successfully cleared
EFI_ACCESS_ERROR	The SPI controller is locked
EFI_UNSUPPORTED	Not a legacy SPI host controller

EFI_LEGACY _SPI_FLASH_PROTOCOL.IsRangeProtected()

Summary

Determine if the SPI range is protected.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_LEGACY_SPI_FLASH_PROTOCOL_IS_RANGE_PROTECTED) (
  IN CONST EFI_LEGACY_SPI_FLASH_PROTOCOL *This,
  IN UINT32 BiosAddress,
  IN UINT32 BlocksToProtect
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_FLASH_PROTOCOL** data structure.

BiosAddress

Address within a 4 KiB block to start protecting.

BlocksToProtect

The number of 4 KiB blocks to protect.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS uses this routine to verify a range in the SPI is protected.

Return Value

TRUE	The range is protected
FALSE	The range is not protected

EFI_LEGACY_SPI_FLASH_PROTOCOL.ProtectNextRange()

Summary

Set the next protect range register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_FLASH_PROTOCOL_PROTECT_NEXT_RANGE) (
  IN CONST EFI_LEGACY_SPI_FLASH_PROTOCOL *This,
  IN UINT32 BiosAddress,
  IN UINT32 BlocksToProtect
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_FLASH_PROTOCOL** data structure.

BiosAddress

Address within a 4 KiB block to start protecting.

BlocksToProtect

The number of 4 KiB blocks to protect.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS sets the protect range register to prevent write and erase operations to a portion of the SPI NOR flash device.

Status Codes Returned

EFI_SUCCESS	The register was successfully updated
EFI_ACCESS_ERROR	The SPI controller is locked
EFI_INVALID_PARAMETER	*BiosAddress < This->BiosBaseAddress*
EFI_INVALID_PARAMETER	*BlocksToProtect * 4 KiB > This->MaximumRangeBytes*
EFI_INVALID_PARAMETER	*BiosAddress - This->BiosBaseAddress + (BlocksToProtect * 4 KiB) > This->MaximumRangeBytes*
EFI_OUT_OF_RESOURCES	No protect range register available
EFI_UNSUPPORTED	Call *This*->*SetBaseAddress* because the BIOS base address is not set
EFI_UNSUPPORTED	Not a legacy SPI host controller

EFI_LEGACY _SPI_FLASH_PROTOCOL.LockController()

Summary

Lock the SPI controller configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_FLASH_PROTOCOL_LOCK_CONTROLLER) (
  IN CONST EFI_LEGACY_SPI_FLASH_PROTOCOL *This
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_FLASH_PROTOCOL** data structure.

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine locks the SPI controller's configuration so that the software is no longer able to update:

- Prefix table
- Opcode menu
- Opcode type table
- BIOS base address
- Protect range registers

Status Codes Returned

EFI_SUCCESS	The SPI controller was successfully locked
EFI_ALREADY_STARTED	The SPI controller was already locked
EFI_UNSUPPORTED	Not a legacy SPI host controller

EFI_SPI_IO_PROTOCOL

Summary

Support managed SPI data transactions between the SPI controller and an SPI chip.

GUID

The SPI peripheral layer provides the GUID for this interface!

Protocol Interface Structure

```
typedef struct _EFI_SPI_IO_PROTOCOL {
  CONST EFI_SPI_PERIPHERAL *SpiPeripheral;
  CONST EFI_SPI_PERIPHERAL *OriginalSpiPeripheral;
  UINT32 FrameSizeSupportMask;
  UINT32 MaximumTransferBytes;
  UINT32 Attributes;
  CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *LegacySpiProtocol;
  EFI_SPI_IO_PROTOCOL_TRANSACTION Transaction;
  EFI_SPI_IO_PROTOCOL_UPDATE_SPI_PERIPHERAL UpdateSpiPeripheral;
} EFI_SPI_IO_PROTOCOL;
```

Parameters

SpiPeripheral

Address of an **EFI_SPI_PERIPHERAL** data structure associated with this protocol instance.

OriginalSpiPeripheral

Address of the original **EFI_SPI_PERIPHERAL** data structure associated with this protocol instance.

FrameSizeSupportMask

Mask of frame sizes which the SPI I/O layer supports. Frame size of N-bits is supported when bit N-1 is set. The host controller must support a frame size of 8-bits. Frame sizes of 16, 24 and 32-bits are converted to 8-bit frame sizes by the SPI bus layer if the frame size is not supported by the SPI host controller.

MaximumTransferBytes

Maximum transfer size in bytes: 1 - 0xffffffff

Attributes

Transaction attributes: One or more from:

- **SPl_IO_SUPPORTS_2_B1T_DATA_BUS_W1DTH** - The SPI host and peripheral supports a two-bit data bus
- **SPI_IO_SUPPORTS_4_BIT_DATA_BUS_W1DTH** - The SPI host and peripheral supports a four-bit data bus
- **SPI_IO_TRANSFER_SIZE_INCLUDES_OPCODE** - Transfer size includes the opcode byte
- **SPI_IO_TRANSFER_SIZE_INCLUDES_ADDRESS** - Transfer size includes the 3 address bytes

LegacySpiProtocol;

Pointer to legacy SPI controller protocol

EFI_SPI_BUS_ TRANSACTION

The **EFI_SPI_BUS_TRANSACTION** data structure contains the description of the SPI transaction to perform on the host controller.

Prototype

```
typedef struct _EFI_SPI_BUS_TRANSACTION
{
  CONST EFI_SPI_PERIPHERAL *SpiPeripheral;
  EFI_SPI_TRANSACTION_TYPE TransactionType;
  BOOLEAN DebugTransaction;
  UINT32 BusWidth; UINT32 FrameSize;
  UINT32 WriteBytes; UINT8 *WriteBuffer;
  UINT32 ReadBytes; UINT8 *ReadBuffer;
} EFI_SPI_BUS_TRANSACTION;
```

Parameters

SpiPeripheral

Pointer to the SPI peripheral being manipulated.

TransactionType

Type of transaction specified by one of the **EFI_SPI_TRANSACTION_TYPE** values.

DebugTransaction

TRUE if the transaction is being debugged. Debugging may be turned on for a single SPI transaction. Only this transaction will display debugging messages. All other transactions with this value set to *FALSE* will not display any debugging messages.

BusWidth

SPI bus width in bits: 1, 2, 4

FrameSize

Frame size in bits, range: 1 - 32

WriteBytes

Length of the write buffer in bytes

WriteBuffer

Buffer containing data to send to the SPI peripheral

Frame sizes 1-8 bits: UINT8 (one byte) per frame

Frame sizes 7-16 bits: UINT16 (two bytes) per frame

Frame sizes

17-32 bits: UINT32 (four bytes) per frame

Read Bytes

Length of the read buffer in bytes

Read Buffer

Buffer to receive the data from the SPI peripheral

- Frame sizes 1-8 bits: UINT8 (one byte) per frame
- Frame sizes 7-16 bits: UINT16 (two bytes) per frame
- Frame sizes 17-32 bits: UINT32 (four bytes) per frame

EFI_SPI_IO_PROTOCOL.Transaction()

Summary

Initiate an SPI transaction between the host and an SPI peripheral.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_IO_PROTOCOL_TRANSACTION) (
  IN CONST EFI_SPI_IO_PROTOCOL *This,
  IN EFI_SPI_TRANSACTION_TYPE TransactionType,
  IN BOOLEAN DebugTransaction,
  IN UINT32 ClockHz OPTIONAL,
  IN UINT32 BusWidth,
  IN UINT32 FrameSize,
  IN UINT32 WriteBytes,
  IN UINT8 *WriteBuffer,
  IN UINT32 ReadBytes,
  OUT UINT8 *ReadBuffer
  );
```

Parameters

This

Pointer to an **EFI_SPI_IO_PROTOCOL** structure.

TransactionType

```
typedef enum _EFI_SPI_TRANSACTION_TYPE {
  SPI_TRANSACTION_FULL_DUPLEX = 0,
  SPI_TRANSACTION_WRITE_ONLY,
  SPI_TRANSACTION_READ_ONLY,
  SPI_TRANSACTION_WRITE_THEN_READ
} EFI_SPI_TRANSACTION_TYPE;
```

Type of SPI transaction specified by one of the **EFI_SPI_TRANSACTION_ TYPE** values:

- **SPI_TRANSACTION_FULL_DUPLEX** - Data flowing in both direction between the host and SPI peripheral. *ReadBytes* must equal *WriteBytes* and both *ReadBuffer* and *WriteBuffer* must be provided.
- **SPI_TRANSACTION_WRITE_ONLY** - Data flowing from the host to the SPI peripheral. *ReadBytes* must be zero. *WriteBytes* must be non-zero and *WriteBuffer* must be provided.
- **SPI_TRANSACTION_READ_ONLY** - Data flowing from the SPI peripheral to the host. *WriteBytes* must be zero. *ReadBytes* must be non-zero and *ReadBuffer* must be provided.
- **SPI_TRANSACTION_WRITE_THEN_READ** - Data first flowing from the host to the SPI peripheral and then data flows from the SPI peripheral to the host. These types of operations get used for SPI flash devices when control data (opcode, address) must be passed to the SPI peripheral to specify the data to be read.

DebugTransaction

Set *TRUE* only when debugging is desired. Debugging may be turned on for a single SPI transaction. Only this transaction will display debugging messages. All other transactions with this value set to *FALSE* will not display any debugging messages.

ClockHz

Specify the *ClockHz* value as zero (0) to use the maximum clock frequency supported by the SPI controller and part. Specify a non-zero value only when a specific SPI transaction requires a reduced clock rate.

BusWidth

Width of the SPI bus in bits: 1, 2, 4

FrameSize

Frame size in bits, range: 1 - 32

WriteBytes

The length of the *WriteBuffer* in bytes. Specify zero for read-only operations.

WriteBuffer

The buffer containing data to be sent from the host to the SPI chip. Specify *NULL* for read only operations.

- Frame sizes 1-8 bits: UINT8 (one byte) per frame
- Frame sizes 7-16 bits: UINT16 (two bytes) per frame
- Frame sizes 17-32 bits: UINT32 (four bytes) per frame The transmit frame is in the least significant N bits.

ReadBytes

The length of the *ReadBuffer* in bytes. Specify zero for write-only operations.

ReadBuffer

The buffer to receive data from the SPI chip during the transaction. Specify *NULL* for write only operations.

- Frame sizes 1-8 bits: UINT8 (one byte) per frame
- Frame sizes 7-16 bits: UINT16 (two bytes) per frame
- Frame sizes 17-32 bits: UINT32 (four bytes) per frame The received frame is in the least significant N bits.

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine works with the SPI bus layer to pass the SPI transaction to the SPI controller for execution on the SPI bus. There are four types of supported transactions supported by this routine:

- Full Duplex: *WriteBuffer* and *ReadBuffer* are the same size.
- Write Only: *WriteBuffer* contains data for SPI peripheral, *ReadBytes* = 0
- Read Only: *ReadBuffer* to receive data from SPI peripheral, *WriteBytes* = 0
- Write Then Read: *WriteBuffer* contains control data to write to SPI peripheral before data is placed into the *ReadBuffer*. Both *WriteBytes* and *ReadBytes* must be non-zero.

Status Codes Returned

EFI_SUCCESS	The SPI transaction completed successfully

EFI_BAD_BUFFER_SIZE	The *WriteBytes* value was invalid
EFI_BAD_BUFFER_SIZE	The *ReadBytes* value was invalid
EFI_INVALID_PARAMETER	*TransactionType* is not valid
EFI_INVALID_PARAMETER	*BusWidth* not supported by SPI peripheral or SPI host controller
EFI_INVALID_PARAMETER	*WriteBytes* is non-zero and *WriteBuffer* is *NULL*
EFI_INVALID_PARAMETER	*ReadBytes* non-zero and *ReadBuffer* is *NULL*
EFI_INVALID_PARAMETER	*ReadBuffer != WriteBuffer* for full-duplex type
EFI_INVALID_PARAMETER	*WriteBuffer* was *NULL*
EFI_INVALID_PARAMETER	*TPL* is too high
EFI_OUT_OF_RESOURCES	Insufficient memory for SPI transaction
EFI_UNSUPPORTED	The *FrameSize* is not supported by the SPI bus layer or the SPI host controller
EFI_UNSUPPORTED	The SPI controller was not able to support the frequency requested by *ClockHz*

EFI_SPI_IO_PROTOCOL.UpdateSpiPeripheral()

Summary

Update the SPI peripheral associated with this SPI I/O instance.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_IO_PROTOCOL_UPDATE_SPI_PERIPHERAL) (
  IN CONST EFI_SPI_IO_PROTOCOL *This,
  IN CONST EFI_SPI_PERIPHERAL *SpiPeripheral
  );
```

Parameters

This

Pointer to an **EFI_SPI_IO_PROTOCOL** structure.

SpiPeripheral

Pointer to an **EFI_SPI_PERIPHERAL** structure.

Description

Support socketed SPI parts by allowing the SPI peripheral driver to replace the SPI peripheral after the connection is made. An example use is socketed SPI NOR flash parts, where the size and parameters change depending upon device is in the socket.

Status Codes Returned

EFI_SUCCESS	The SPI peripheral was updated successfully
EFI_INVALID_PARAMETER	The SpiPeripheral value is *NULL*

EFI_INVALID_PARAMETER	The `SpiPeripheral->SpiBus` is *NULL*
EFI_INVALID_PARAMETER	The `SpiPeripheral->SpiBus` pointing at wrong bus
EFI_INVALID_PARAMETER	The `SpiPeripheral->SpiPart` is *NULL*

EFI_SPI_HC_PROTOCOL

Summary

Support an SPI data transaction between the SPI controller and an SPI chip.

GUID

```
// {c74e5db2-fa96-4ae2-b399-15977fe3002d}
#define EFI_SPI_HOST_GUID \
{ 0xc74e5db2, 0xfa96, 0x4ae2, { 0xb3, 0x99, 0x15, 0x97, \
  0x7f, 0xe3, 0x0, 0x2d }}
```

Protocol Interface Structure

```
typedef struct _EFI_SPI_HC_PROTOCOL {
  UINT32 Attributes;
  UINT32 FrameSizeSupportMask;
  UINT32 MaximumTransferBytes;
  EFI_SPI_HC_PROTOCOL_CHIP_SELECT ChipSelect;
  EFI_SPI_HC_PROTOCOL_CLOCK Clock;
  EFI_SPI_HC_PROTOCOL_TRANSACTION Transaction;
} EFI_SPI_HC_PROTOCOL;
```

Parameters

Attributes

Host control attributes, may have zero or more of the following set:

- **HC_SUPPORTS_WRITE_ONLY_OPERATIONS**
- **HC_SUPPORTS_READ_ONLY_OPERATIONS**
- **HC_SUPPORTS_WRITE_THEN_READ_OPERATIONS**
- **HC_TX_FRAME_IN_MOST_SIGNIFICANT_BITS** - The SPI host controller requires the transmit frame to be in most significant bits instead of least significant bits. The host driver will adjust the frames if necessary.
- **HC_RX_FRAME_IN_MOST_SIGNIFICANT_BITS** - The SPI host controller places the receive frame to be in most significant bits instead of least significant bits. The host driver will adjust the frames to be in the least significant bits if necessary.
- **HC_SUPPORTS_2_BIT_DATA_BUS_W1DTH** - The SPI controller supports a two-bit data bus
- **HC_SUPPORTS_4_B1T_DATA_BUS_WIDTH** - The SPI controller supports a four-bit data bus
- **HC_TRANSFER_SIZE_INCLUDES_OPCODE** - Transfer size includes the opcode byte
- **HC_TRANSFER_SIZE_INCLUDES_ADDRESS** - Transfer size includes the 3 address bytes

The SPI host controller must support full-duplex (receive while sending) operation. The SPI host controller must support a one-bit bus width.

FrameSizeSupportMask

Mask of frame sizes which the SPI host controller supports. Frame size of N-bits is supported when bit N-1 is set. The host controller must support a frame size of 8-bits.

MaximumTransferBytes

Maximum transfer size in bytes: 1 - 0xffffffff

EFI_SPI_HC_PROTOCOL.ChipSelect()

Summary

Assert or deassert the SPI chip select.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_HC_PROTOCOL_CHIP_SELECT) (
  IN CONST EFI_SPI_HC_PROTOCOL *This,
  IN CONST EFI_SPI_PERIPHERAL *SpiPeripheral,
  IN BOOLEAN PinValue
  );
```

Parameters

This

Pointer to an **EFI_SPI_HC_PROTOCOL** structure.

SpiPeripheral

The address of an **EFI_SPI_PERIPHERAL** data structure describing the SPI peripheral whose chip select pin is to be manipulated. The routine may access the *ChipSelectParameter* field to gain sufficient context to complete the operation.

PinValue

The value to be applied to the chip select line of the SPI peripheral.

Description

This routine is called at **TPL_NOTIFY**.

Update the value of the chip select line for an SPI peripheral. The SPI bus layer calls this routine either in the board layer or in the SPI controller to manipulate the chip select pin at the start and end of an SPI transaction.

Status Codes Returned

EFI_SUCCESS	The chip select was set as requested
EFI_NOT_READY	Support for the chip select is not properly initialized
EFI_INVALID_PARAMETER	The *ChipSelect* value or its contents are invalid

EFI_SPI_HC_PROTOCOL.Clock()

Summary

Set up the clock generator to produce the correct clock frequency, phase and polarity for an SPI chip.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_HC_PROTOCOL_CLOCK) (
  IN CONST EFI_SPI_HC_PROTOCOL *This,
  IN CONST EFI_SPI_PERIPHERAL *SpiPeripheral,
  IN UINT32 *ClockHz
  );
```

Parameters

This

Pointer to an **EFI_SPI_HC_PROTOCOL** structure.

SpiPeripheral

Pointer to a **EFI_SPI_PERIPHERAL** data structure from which the routine can access the *CLockParameter*, *CLockPhase* and *CLockPolarity* fields. The routine also has access to the names for the SPI bus and chip which can be used during debugging.

ClockHz

Pointer to the requested clock frequency. The SPI host controller will choose a supported clock frequency which is less then or equal to this value. Specify zero to turn the clock generator off. The actual clock frequency supported by the SPI host controller will be returned.

Description

This routine is called at **TPL_NOTIFY**.

This routine updates the clock generator to generate the correct frequency and polarity for the SPI clock.

Status Codes Returned

EFI_SUCCESS	The clock was set up successfully
EFI_UNSUPPORTED	The SPI controller was not able to support the frequency requested by *ClockHz*

EFI_SPI_HC_PROTOCOL.Transaction()

Summary

Perform the SPI transaction on the SPI peripheral using the SPI host controller.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SPI_HC_PROTOCOL_TRANSACTION) (
  IN CONST EFI_SPI_HC_PROTOCOL *This,
  IN EFI_SPI_BUS_TRANSACTION *BusTransaction
  );
```

Parameters

This

Pointer to an **EFI_SPI_HC_PROTOCOL** structure.

BusTransaction

Pointer to a **EFI_SPI_BUS_TRANSACTION** containing the description of the SPI transaction to perform.

Description

This routine is called at **TPL_NOTIFY**.

This routine synchronously returns **EFI_SUCCESS** indicating that the asynchronous SPI transaction was started. The routine then waits for completion of the SPI transaction prior to returning the final transaction status.

Status Codes Returned

EFI_SUCCESS	The transaction completed successfully
EFI_BAD_BUFFER_SIZE	The *BusTransaction->WriteBytes* value is invalid
EFI_BAD_BUFFER_SIZE	The *BusTransaction->ReadinBytes* value is invalid
EFI_UNSUPPORTED	The *BusTransaction->TransactionType* is unsupported

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL

Summary

Support the extra features of the legacy SPI flash controller.

GUID

```
// {39136fc7-lall-49de-bf35-0e78ddb524fc}
#define EFI_LEGACY_SPI_CONTROLLER_GUID \
{ 0x39136fc7, 0xlall, 0x49de, { 0xbf, 0x35, 0x0e, 0x78, \
0xdd, 0xb5, 0x24, 0xfc }}
```

Protocol Interface Structure

```
typedef struct _EFI_LEGACY_SPI_CONTROLLER_PROTOCOL {
  UINT32 MaximumOffset; UINT32 MaximumRangeBytes;
  UINT32 RangeRegisterCount;
  EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_ERASE_BLOCK_OPCODE  EraseBlockOpcode;
  EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_WRITE_STATUS_PREFIX WriteStatusPrefix;
```

```
    EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_BIOS_BASE_ADDRESSBiosBaseAddress;
    EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_CLEAR_SPI_PROTECTClearSpiProtect;
    EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_IS_RANGE_PROTECTEDIsRangeProtected;
    EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_PROTECT_NEXT_RANGEProtectNextRange;
    EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_LOCK_CONTROLLERLockController;
  } EFI_LEGACY_SPI_CONTROLLER_PROTOCOL;
```

Parameters

MaximumOffset

Maximum offset from the BIOS base address that is able to be protected.

MaximumRangeBytes

Maximum number of bytes that can be protected by one range register.

RangeRegisterCount

The number of registers available for protecting the BIOS.

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.EraseBlockOpcode()

Summary

Set the erase block opcode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_ERASE_BLOCK_OPCODE) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This,
  IN UINT8 EraseBlockOpcode
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

EraseBlockOpcode

Erase block opcode to be placed into the opcode menu table.

Description

This routine must be called at or below **TPL_NOTIFY**.

The menu table contains SPI transaction opcodes which are accessible after the legacy SPI flash controller's configuration is locked. The board layer specifies the erase block size for the SPI NOR flash part. The SPI NOR flash peripheral driver selects the erase block opcode which matches the erase block size and uses this API to load the opcode into the opcode menu table.

Status Codes Returned

EFI_SUCCESS	The opcode menu table was updated

EFI_ACCESS_ERROR	The SPI controller is locked

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.WriteStatusPrefix()

Summary

Set the write status prefix opcode.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_WRITE_STATUS_PREFIX) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This,
  IN UINT8 WriteStatusPrefix
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

WriteStatusPrefix

Prefix opcode for the write status command.

Description

This routine must be called at or below **TPL_NOTIFY**.

The prefix table contains SPI transaction write prefix opcodes which are accessible after the legacy SPI flash controller's configuration is locked. The board layer specifies the write status prefix opcode for the SPI NOR flash part. The SPI NOR flash peripheral driver uses this API to load the opcode into the prefix table.

Status Codes Returned

EFI_SUCCESS	The prefix table was updated
EFI_ACCESS_ERROR	The SPI controller is locked

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.BiosBaseAddress()

Summary

Set the BIOS base address.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_BIOS_BASE_ADDRESS) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This,
  IN UINT32 BiosBaseAddress
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

BiosBaseAddress

The BIOS base address.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS base address works with the protect range registers to protect portions of the SPI NOR flash from erase and write operations. The BIOS calls this API prior to passing control to the OS loader.

Status Codes Returned

EFI_SUCCESS	The BIOS base address was properly set
EFI_ACCESS_ERROR	The SPI controller is locked
EFI_INVALID_PARAMETER	The BIOS base address is greater than *This->MaximumOffset*
EFI_UNSUPPORTED	The BIOS base address was already set

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.ClearSpiProtect()

Summary

Clear the SPI protect range registers.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_CLEAR_SPI_PROTECT) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

Description

This routine must be called at or below TPL_NOTIFY.

The BIOS uses this routine to set an initial condition on the SPI protect range registers.

Status Codes Returned

EFI_SUCCESS	The registers were successfully cleared
EFI_ACCESS_ERROR	The SPI controller is locked

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.IsRangeProtected()

Summary

Determine if the SPI range is protected.

Prototype

```
typedef
BOOLEAN
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_IS_RANGE_PROTECTED) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This,
  IN UINT32 BiosAddress,
  IN UINT32 BlocksToProtect
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

BiosAddress

Address within a 4 KiB block to start protecting.

BytesToProtect

The number of 4 KiB blocks to protect.

Description

This routine must be called at or below TPL_NOTIFY.

The BIOS uses this routine to verify a range in the SPI is protected.

Return Value

TRUE	The range is protected
FALSE	The range is not protected

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.ProtectNextRange()

Summary

Set the next protect range register.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_PROTECT_NEXT_RANGE) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This,
  IN UINT32 BiosAddress,
  IN UINT32 BlocksToProtect
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

BiosAddress

Address within a 4 KiB block to start protecting.

BlocksToProtect

The number of 4 KiB blocks to protect.

Description

This routine must be called at or below **TPL_NOTIFY**.

The BIOS sets the protect range register to prevent write and erase operations to a portion of the SPI NOR flash device.

Status Codes Returned

EFI_SUCCESS	The register was successfully updated
EFI_ACCESS_ERROR	The SPI controller is locked
EFI_INVALID_PARAMETER	*BiosAddress < This->BiosBaseAddress*
EFI_INVALID_PARAMETER	*BlocksToProtect * 4 KiB > This->MaximumRangeBytes*
EFI_INVALID_PARAMETER	*BiosAddress - This->BiosBaseAddress + (BlocksToProtect * 4 KiB) > This->MaximumRangeBytes*
EFI_OUT_OF_RESOURCES	No protect range register available
EFI_UNSUPPORTED	Call *This->SetBaseAddress* because the BIOS base address is not set

EFI_LEGACY_SPI_CONTROLLER_PROTOCOL.LockController()

Summary

Lock the SPI controller configuration.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_LEGACY_SPI_CONTROLLER_PROTOCOL_LOCK_CONTROLLER) (
  IN CONST EFI_LEGACY_SPI_CONTROLLER_PROTOCOL *This
  );
```

Parameters

This

Pointer to an **EFI_LEGACY_SPI_CONTROLLER_PROTOCOL** structure.

Description

This routine must be called at or below **TPL_NOTIFY**.

This routine locks the SPI controller's configuration so that the software is no longer able to update:

- Prefix table
- Opcode menu
- Opcode type table
- BIOS base address
- Protect range registers

Status Codes Returned

EFI_SUCCESS	The SPI controller was successfully locked
EFI_ALREADY_STARTED	The SPI controller was already locked

Appendix A Error Codes

A.1 Error Code Definitions

For 32-bit architecture:

```
#define EFI_INTERRUPT_PENDING                 0xa0000000
#define EFI_WARN_INTERRUPT_SOURCE_PENDING     0x20000000
#define EFI_WARN_INTERRUPT_SOURCE_QUIESCED    0x20000001
```

For 64-bit architecture:

```
#define EFI_INTERRUPT_PENDING                 0xa000000000000000
#define EFI_WARN_INTERRUPT_SOURCE_PENDING     0x2000000000000000
#define EFI_WARN_INTERRUPT_SOURCE_QUIESCED    0x2000000000000001
```