



VOLUME 3: Platform Initialization

Shared Architectural Elements

Version 1.4 Errata A

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.

Copyright 2006 - 2016 Unified EFI, Inc. All Rights Reserved.

Revision History

Revision	Revision History	Date
1.0	Initial public release.	8/21/06
1.0 errata	Mantis tickets: <ul style="list-style-type: none"> • M47 dxe_dispatcher_load_image_behavior • M48 Make spec more consistent GUID & filename. • M155 FV_FILE and FV_ONLY: Change subtype number back to the original one. • M171 Remove 10 us lower bound restriction for the TickPeriod in the Metronome • M178 Remove references to tail in file header and made file checksum for the data • M183 Vol 1-Vol 5: Make spec more consistent. • M192 Change PAD files to have an undefined GUID file name and update all FV 	10/29/07
1.1	Mantis tickets: <ul style="list-style-type: none"> • M39 (Updates PCI Hostbridge & PCI Platform) • M41 (Duplicate 167) • M42 Add the definition of the DXE CIS Capsule AP & Variable AP • M43 (SMBios) • M46 (SMM error codes) • M163 (Add Volume 4--SMM) • M167 (Vol2: adds the DXE Boot Services Protocols--new Chapter 12) • M179 (S3 boot script) • M180 (PMI ECR) • M195 (Remove PMI references from SMM CIS) • M196 (disposable-section type to the FFS) 	11/05/07
1.1 correction	Restore (missing) MP protocol	03/12/08
1.1 Errata	Revises typographical errors and minor omissions--see Errata for details	04/25/08

1.1 Errata	<p>Mantis tickets</p> <ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 247 Clarification regarding use of dependency expression section types with firmware volume image files • 399 SMBIOS Protocol Errata • 405 PIWG Volume 5 incorrectly refers to EFI_PCI_OVERRIDE_PROTOCOL • 422 TEMPORARY_RAM_SUPPORT_PPI is misnamed • 428 Volume 5 PCI issue • 430 Clarify behavior w/ the FV extended header 	02/23/09
1.2	<ul style="list-style-type: none"> • 271 Support For Large Firmware Files And Firmware File Sections • 284 CPU I/O protocol update • 286 Legacy Region protocol • 289 Recovery API • 292 PCD Specification Update • 354 ACPI Manipulation Protocol • 355 EFI_SIO_PROTOCOL Errata • 365 UEFI Capsule HOB • 382 IDE Controller Specification • 385 Report Status Code Router Specification • 386 Status Code Specification 	01/19/09

1.2	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 407 Comment: additional change to LMA Pseudo-Register • 441 Comment: PI Volume 3, Incorrect Struct Declaration (esp PCD_PPI) • 455 Comment: Errata - Clarification of InstallPeiMemory() • 465 Comment: Errata on PMI interface • 466 Comment: Vol 4 EXTENDED_SAL_PROC definition • 467 Comments: PI1.1 errata • 480 Comment: FIX to PCD_PROTOCOL and PCD_PPI 	05/13/09

1.2 errata	<ul style="list-style-type: none">• 345 PI1.0 errata• 468 Issues on proposed PI1.2 ACPI System Description Table Protocol• 492 Add Resource HOB Protectability Attributes• 494 Vol. 2 Appendix A Clean up• 495 Vol 1: update HOB reference• 380 PI1.1 errata from SMM development• 501 Clean Up SetMemoryAttributes() language Per Mantis 489 (from USWG)• 502 Disk info• 503 typo• 504 remove support for fixed address resources• 509 PCI errata – execution phase• 510 PCI errata - platform policy• 511 PIC TE Image clarification/errata• 520 PI Errata• 521Add help text for EFI_PCD_PROTOCOL for GetNextTokenSpace• 525 Itanium ESAL, MCA/INIT/PMI errata• 526 PI SMM errata• 529 PCD issues in Volume 3 of the PI1.2 Specification• 541 Volume 5 Typo• 543 Clarification around usage of FV Extended header• 550 Naming conflicts w/ PI SMM	12/16/09
------------	---	----------

1.2 errata A	<ul style="list-style-type: none"> • 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 & error on stage 3 recovery description • 571 duplicate definition of EFI_AP_PROCEDURE in DXE MP (volume2) and SMM (volume 4) • 581 EFI_HOB_TYPE_LOAD_PEIM ambiguity • 591ACPI Protocol Name collision • 592 More SMM name conflicts • 593 A couple of ISA I/O clarifications • 594 ATA/ATAPI clarification • 595 SMM driver entry point clarification • 596 Clarify ESAL return codes • 602 SEC->PEI hand-off update • 604 EFI_NOT_SUPPORTED versus EFI_UNSUPPORTED 	2/24/10
1.2 errata B	<ul style="list-style-type: none"> • 628 ACPI SDT protocol errata • 629 Typos in PCD GetSize() • 630EFI_SMM_PCI_ROOT_BRIDGE_IO_PROTOCOL service clarification • 631 System Management System Table (SMST) MP-related field clarification 	5/27/10

1.2 Errata C	<ul style="list-style-type: none"> • 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 • 821 Location conflict for EFI_RESOURCE_ATTRIBUTE_XXX_PROTECTABLE #defines • 823 Clarify max length of SMBIOS Strings in SMBIOS Protocol • 824 EFI_SMM_SW_DISPATCH2_PROTOCOL.Register() Errata • 837 ARM Vector table can not support arbitrary 32-bit address • 838 Vol 3 EFI_FVB2_ALIGNMNET_512K should be EFI_FVB2_ALIGNMENT_512K • 840 Vol 3 Table 5 Supported FFS Alignments contains values not supported by FFS • 844 correct references to Platform Initialization Hand-Off Block Specification 	10/27/11
--------------	---	----------

1.2.1	<ul style="list-style-type: none"> • 527 PI Volume 2 DXE Security Architecture Protocol (SAP) clarification • 562 Add SetMemoryCapabilities to GCD interface • 719 End of DXE event • 731 Volume 4 SMM - clarify the meaning of NumberOfCpus • 737 Remove SMM Communication ACPI Table definition . • 753 SIO PEI and UEFI-Driver Model Architecture • 769 Signed PI sections • 813 Add a new EFI_GET_PCD_INFO_PROTOCOL and EFI_GET_PCD_INFO_PPI instance. • 818 New SAP2 return code • 822 Method to disable Temporary RAM when Temp RAM Migration is not required • 833 Method to Reserve Interrupt and Exception Vectors • 839 Add support for weakly aligned FVs • 892 EFI_PCI_ENUMERATION_COMPLETE_GUID Protocol • 894 SAP2 Update • 895 Status Code Data Structures Errata • 902 Errata on signed firmware volume/file • 903 SMIManage Update • 906 Volume 3 errata - Freeform type • 916 Service table revisions 	05/02/12
1.2.1 Errata A	<ul style="list-style-type: none"> • 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 • 969 Vol 1 errata: TE Header parameters 	10/26/12
1.2.1 Errata A	<ul style="list-style-type: none"> • 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 • 969 Vol 1 errata: TE Header parameters 	10/26/12
1.3	<ul style="list-style-type: none"> • 945 Integrated Circuit (I2C) Bus Protocol • 998 PI Status Code additions • 999 PCI enumeration complete GUID • 1005 NVMe Disk Info guid • 1006 Security Ppi Fixes • 1025 PI table revisions 	3/29/13

1.3 Errata	<ul style="list-style-type: none"> • 1041 typo in HOB Overview • 1067 PI1.3 Errata for SetBootMode • 1068 Updates to PEI Service table/M1006 • 1069 SIO Errata - pnp end node definition • 1070 Typo in SIO chapter • 1072 Errata – SMM register protocol notify clarification/errata • 1093 Extended File Size Errata • 1095 typos/errata • 1097 PI SMM GPI Errata • 1098 Errata on I2C IO status code • 1099 I2C Protocol stop behavior errata • 1104 ACPI System Description Table Protocol Errata • 1105 ACPI errata - supported table revision • 1177 PI errata - make CPU IO optional • 1178 errata - allow PEI to report an additional memory type • 1283 Errata - clarify sequencing of events 	2/19/15
1.4	<ul style="list-style-type: none"> • 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

1.4 Errata A	<ul style="list-style-type: none"> • 1596 Mantis1489 GCD issue • 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 dxs 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
--------------	--	---------

Specification Volumes

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

VOLUME 1: Pre-EFI Initialization Core Interface

VOLUME 2: Driver Execution Environment Core Interface

VOLUME 3: Shared Architectural Elements

VOLUME 4: System Management Mode

VOLUME 5: Standards

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

Contents

1	Platform Intialization	
	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	7
2.1	Firmware Storage Introduction.....	7
2.1.1	Firmware Devices	7
2.1.2	Firmware Volumes	7
2.1.3	Firmware File System	8
2.1.4	Firmware Files.....	8
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	18
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	24
2.2.7	File Integrity and State	25
2.2.8	File State Transitions	26
3	Firmware Storage Code Definitions.....	31
3.1	Firmware Storage Code Definitions Introduction	31
3.2	Firmware Storage Formats	31
3.2.1	Firmware Volume	31
	EFI_FIRMWARE_VOLUME_HEADER	31
3.2.2	Firmware File System	37
	EFI_FIRMWARE_FILE_SYSTEM2_GUID	37
	EFI_FIRMWARE_FILE_SYSTEM3_GUID	38
	EFI_FFS_VOLUME_TOP_FILE_GUID	39
3.2.3	Firmware File	40
	EFI_FFS_FILE_HEADER	40

3.2.4 Firmware File Section	46
EFI_COMMON_SECTION_HEADER	46
3.2.5 Firmware File Section Types.....	48
EFI_SECTION_COMPATIBILITY16	48
EFI_SECTION_COMPRESSION.....	49
EFI_SECTION_DISPOSABLE	51
EFI_SECTION_DXE_DEPEX	52
EFI_SECTION_FIRMWARE_VOLUME_IMAGE	53
EFI_SECTION_FREEFORM_SUBTYPE_GUID.....	54
EFI_SECTION_GUID_DEFINED	55
EFI Signed Sections.....	57
EFI_SECTION_PE32.....	58
EFI_SECTION_PEI_DEPEX.....	59
EFI_SECTION_PIC.....	60
EFI_SECTION_RAW	61
EFI_SECTION_SMM_DEPEX	62
EFI_SECTION_TE	63
EFI_SECTION_USER_INTERFACE	64
EFI_SECTION_VERSION.....	65
3.3 PEI	66
EFI_PEI_FIRMWARE_VOLUME_INFO_PPI.....	66
EFI_PEI_FIRMWARE_VOLUME_INFO2_PPI	67
3.3.1 PEI Firmware Volume PPI	68
EFI_PEI_FIRMWARE_VOLUME_PPI	68
EFI_PEI_FIRMWARE_VOLUME_PPI.ProcessVolume()	70
EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByType()	71
EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByName()	72
EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo()	73
EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo2()	74
EFI_PEI_FIRMWARE_VOLUME_PPI.GetVolumeInfo()	75
EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType()	76
EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType2()	77
3.3.2 PEI Load File PPI.....	78
EFI_PEI_LOAD_FILE_PPI.....	78
EFI_PEI_LOAD_FILE_PPI.LoadFile().....	79
3.3.3 PEI Guided Section Extraction PPI	80
EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI	80
EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()	82
3.3.4 PEI Decompress PPI	84
EFI_PEI_DECOMPRESS_PPI.....	84
EFI_PEI_DECOMPRESS_PPI.Decompress()	85
3.4 DXE.....	86
3.4.1 Firmware Volume2 Protocol.....	86
EFI_FIRMWARE_VOLUME2_PROTOCOL.....	86
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes()	88
EFI_FIRMWARE_VOLUME2_PROTOCOL.SetVolumeAttributes().....	91
EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadFile()	93

EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadSection()	97
EFI_FIRMWARE_VOLUME2_PROTOCOL.WriteFile()	99
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetNextFile()	102
EFI_FIRMWARE_VOLUME2_PROTOCOL.GetInfo()	104
EFI_FIRMWARE_VOLUME2_PROTOCOL.SetInfo()	106
3.4.2 Firmware Volume Block2 Protocol	107
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL	107
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetAttributes()	109
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.SetAttributes()	110
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetPhysicalAddress()	111
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetBlockSize()	112
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Read()	113
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Write()	115
EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.EraseBlocks()	117
3.4.3 Guided Section Extraction Protocol	118
EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL	118
EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL.ExtractSection()	119
4	
HOB Design Discussion	121
4.1 Explanation of HOB Terms	121
4.2 HOB Overview	121
4.3 Example HOB Producer Phase Memory Map and Usage	122
4.4 HOB List	122
4.5 Constructing the HOB List	123
4.5.1 Constructing the Initial HOB List	123
4.5.2 HOB Construction Rules	123
4.5.3 Adding to the HOB List	124
5	
HOB Code Definitions	125
5.1 HOB Introduction	125
5.2 HOB Generic Header	126
EFI_HOB_GENERIC_HEADER	126
5.3 PHIT HOB	128
EFI_HOB_HANDOFF_INFO_TABLE (PHIT HOB)	128
5.4 Memory Allocation HOB	130
5.4.1 Memory Allocation HOB	130
EFI_HOB_MEMORY_ALLOCATION	130
5.4.2 Boot-Strap Processor (BSP) Stack Memory Allocation HOB	133
EFI_HOB_MEMORY_ALLOCATION_STACK	133
5.4.3 Boot-Strap Processor (BSP) BSPSTORE Memory Allocation HOB	135
EFI_HOB_MEMORY_ALLOCATION_BSP_STORE	135
5.4.4 Memory Allocation Module HOB	136
EFI_HOB_MEMORY_ALLOCATION_MODULE	136
5.5 Resource Descriptor HOB	137
EFI_HOB_RESOURCE_DESCRIPTOR	137
5.6 GUID Extension HOB	144

EFI_HOB_GUID_TYPE	144
5.7 Firmware Volume HOB	145
EFI_HOB_FIRMWARE_VOLUME	145
EFI_HOB_FIRMWARE_VOLUME2	146
5.8 CPU HOB	147
EFI_HOB_CPU	147
5.9 Memory Pool HOB	148
EFI_HOB_MEMORY_POOL	148
5.10 UEFI Capsule HOB	148
EFI_HOB_UEFI_CAPSULE	148
5.11 Unused HOB	150
EFI_HOB_TYPE_UNUSED	150
5.12 End of HOB List HOB	151
EFI_HOB_TYPE_END_OF_HOB_LIST	151

6

Platform Initialization

Status Codes	147
6.1 Status Codes Overview	147
6.1.1 Organization of the Status Codes Specification	147
6.2 Terms	147
6.3 Types of Status Codes	148
6.3.1 Status Code Classes	150
6.3.2 Instance Number	150
6.4 Hardware Classes	151
6.4.1 Computing Unit Class	151
6.4.2 User-Accessible Peripheral Class	160
6.4.3 Subclasses	161
6.5 Software Classes	179
6.5.1 Host Software Class	179
6.5.2 Instance Number	179
6.5.3 Progress Code Operations	179
6.5.4 Error Code Operations	180
6.5.5 Subclasses	181
6.5.6 Runtime (RT) Subclass	190
6.6 Code Definitions	198
6.6.1 Data Structures	198
6.6.2 Extended Data Header	198
EFI_STATUS_CODE_DATA	198
EFI_STATUS_CODE_DATA_TYPE_STRING_GUID	200
EFI_STATUS_CODE_SPECIFIC_DATA_GUID	203
6.6.3 Enumeration Schemes	203
6.6.4 Common Extended Data Formats	204
EFI_DEVICE_PATH_EXTENDED_DATA	205
EFI_DEVICE_HANDLE_EXTENDED_DATA	206
EFI_RESOURCE_ALLOC_FAILURE_ERROR_DATA	207
6.7 Class Definitions	208

6.7.1 Computing Unit Class	209
EFI_COMPUTING_UNIT_VOLTAGE_ERROR_DATA.....	215
EFI_COMPUTING_UNIT_MICROCODE_UPDATE_ERROR_DATA.....	217
EFI_COMPUTING_UNIT_TIMER_EXPIRED_ERROR_DATA.....	218
EFI_HOST_PROCESSOR_MISMATCH_ERROR_DATA.....	219
EFI_COMPUTING_UNIT_THERMAL_ERROR_DATA.....	221
EFI_CACHE_INIT_DATA.....	222
EFI_COMPUTING_UNIT_CPU_DISABLED_ERROR_DATA	223
EFI_MEMORY_EXTENDED_ERROR_DATA	224
EFI_STATUS_CODE_DIMM_NUMBER.....	227
EFI_MEMORY_MODULE_MISMATCH_ERROR_DATA	229
EFI_MEMORY_RANGE_EXTENDED_DATA.....	230
6.7.2 User-Accessible Peripherals Class	230
6.7.3 I/O Bus Class	236
6.7.4 Software Classes	242
EFI_DEBUG_ASSERT_DATA.....	261
EFI_STATUS_CODE_EXCEP_EXTENDED_DATA.....	262
EFI_STATUS_CODE_START_EXTENDED_DATA	264
EFI_LEGACY_OPROM_EXTENDED_DATA	265

7

Report Status Code Routers 267

7.1 Overview	267
7.2 Code Definitions.....	267
7.2.1 Report Status Code Handler Protocol.....	267
EFI_RSC_HANDLER_PROTOCOL.....	267
EFI_RSC_HANDLER_PROTOCOL.Register().....	269
EFI_RSC_HANDLER_PROTOCOL.Unregister().....	270
7.2.2 Report Status Code Handler PPI	270
EFI_PEI_RSC_HANDLER_PPI	270
EFI_PEI_RSC_HANDLER_PPI.Register().....	272
EFI_PEI_RSC_HANDLER_PPI.Unregister()	273
7.2.3 SMM Report Status Code Handler Protocol	273
EFI_SMM_RSC_HANDLER_PROTOCOL	273
EFI_SMM_RSC_HANDLER_PROTOCOL.Register().....	275
EFI_SMM_RSC_HANDLER_PROTOCOL.Unregister()	276

8

PCD 277

8.1 PCD Protocol Definitions	277
8.1.1 PCD Protocol	277
EFI_PCD_PROTOCOL	277
EFI_PCD_PROTOCOL.SetSku ().....	280
EFI_PCD_PROTOCOL.Get8 ().....	281
EFI_PCD_PROTOCOL.Get16 ().....	282
EFI_PCD_PROTOCOL.Get32 ().....	283
EFI_PCD_PROTOCOL.Get64 ().....	284
EFI_PCD_PROTOCOL.GetPtr ()	285

EFI_PCD_PROTOCOL.GetBool ()	286
EFI_PCD_PROTOCOL.GetSize ()	287
EFI_PCD_PROTOCOL.Set8 ()	288
EFI_PCD_PROTOCOL.Set16 ()	289
EFI_PCD_PROTOCOL.Set32 ()	290
EFI_PCD_PROTOCOL.Set64 ()	291
EFI_PCD_PROTOCOL.SetPtr ()	292
EFI_PCD_PROTOCOL.SetBool ()	293
EFI_PCD_PROTOCOL.CallbackOnSet ()	294
EFI_PCD_PROTOCOL.CancelCallback ()	295
EFI_PCD_PROTOCOL.GetNextToken ()	296
EFI_PCD_PROTOCOL.GetNextTokenSpace ()	297
8.1.2 Get PCD Information Protocol	297
EFI_GET_PCD_INFO_PROTOCOL	297
EFI_GET_PCD_INFO_PROTOCOL.GetInfo ()	299
EFI_GET_PCD_INFO_PROTOCOL.GetSku ()	301
8.2 PCD PPI Definitions	301
8.2.1 PCD PPI	301
EFI_PEI_PCD_PPI	301
EFI_PEI_PCD_PPI.SetSku ()	304
EFI_PEI_PCD_PPI.Get8 ()	305
EFI_PEI_PCD_PPI.Get16 ()	306
EFI_PEI_PCD_PPI.Get32 ()	307
EFI_PEI_PCD_PPI.Get64 ()	308
EFI_PEI_PCD_PPI.GetPtr ()	309
EFI_PEI_PCD_PPI.GetBool ()	310
EFI_PEI_PCD_PPI.GetSize ()	311
EFI_PEI_PCD_PPI.Set8 ()	312
EFI_PEI_PCD_PPI.Set16 ()	313
EFI_PEI_PCD_PPI.Set32 ()	314
EFI_PEI_PCD_PPI.Set64 ()	315
EFI_PEI_PCD_PPI.SetPtr ()	316
EFI_PEI_PCD_PPI.SetBool()	318
EFI_PEI_PCD_PPI.CallbackOnSet ()	319
EFI_PEI_PCD_PPI.CancelCallback ()	320
EFI_PEI_PCD_PPI.GetNextToken ()	321
EFI_PEI_PCD_PPI.GetNextTokenSpace ()	322
8.2.2 Get PCD Information PPI	322
EFI_GET_PCD_INFO_PPI	322
EFI_GET_PCD_INFO_PPI.GetInfo ()	324
EFI_GET_PCD_INFO_PPI.GetSku ()	325

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	27
Figure 8. Updating a File.....	29
Figure 9. Bit Allocation of FFS <i>Attributes</i>	43
Figure 10. EFI_FV_FILE_ATTRIBUTES fields	95
Figure 11. Example HOB Producer Phase Memory Map and Usage	122
Figure 12. Hierarchy of Status Code Operations	149
Figure 13. Status Code Services	267

Tables

Table 1. SI prefixes	5
Table 2. Binary prefixes	5
Table 3. Defined File Types	9
Table 4. Architectural Section Types	16
Table 5. Descriptions of EFI_FVB_ATTRIBUTES_2	34
Table 6. Bit Allocation Definitions	44
Table 7. Supported FFS Alignments	44
Table 8. Description of Fields for <i>CompressionType</i>	50
Table 9. Descriptions of Fields for <i>GuidedSectionHeader.Attributes</i>	56
Table 10. <i>AuthenticationStatus</i> Bit Definitions	83
Table 11. Descriptions of Fields for EFI_FV_ATTRIBUTES	90
Table 12. Supported Alignments for EFI_FV_FILE_ATTRIB_ALIGNMENT	95
Table 13. Description of fields for EFI_FV_WRITE_POLICY	100
Table 14. Translation of HOB Specification Terminology	121
Table 15. EFI_RESOURCE_ATTRIBUTE_TYPE fields	141
Table 16. HOB Producer Phase Resource Types	143
Table 17. Organization of This Specification	147
Table 18. Class Definitions	150
Table 19. Progress Code Operations: Computing Unit Class	151
Table 20. Error Code Operations: Computing Unit Class	151
Table 21. Computing Unit Class: Subclasses	152
Table 22. Progress and Error Code Operations: Computing Unit Unspecified Subclass ...	153
Table 23. Progress and Error Code Operations: Host Processor Subclass	153
Table 24. Progress and Error Code Operations: Service Processor Subclass	155
Table 25. Progress and Error Code Operations: Cache Subclass	156
Table 26. Progress and Error Code Operations: Memory Subclass	157
Table 27. Progress and Error Code Operations: Chipset Subclass	159
Table 28. Progress Code Operations: User-Accessible Peripheral Class	160
Table 29. Error Code Operations: User-Accessible Peripheral Class	160
Table 30. Defined Subclasses: User-Accessible Peripheral Class	161
Table 31. Progress and Error Code Operations: Peripheral Unspecified Subclass	162
Table 32. Progress and Error Code Operations: Keyboard Subclass	163
Table 33. Progress and Error Code Operations: Mouse Subclass	164
Table 34. Progress and Error Code Operations: Local Console Subclass	165
Table 35. Progress and Error Code Operations: Remote Console Subclass	165
Table 36. Progress and Error Code Operations: Serial Port Subclass	166
Table 37. Progress and Error Code Operations: Parallel Port Subclass	166
Table 38. Progress and Error Code Operations: Fixed Media Subclass	167
Table 39. Progress and Error Code Operations: Removable Media Subclass	167
Table 40. Progress and Error Code Operations: Audio Input Subclass	168
Table 41. Progress and Error Code Operations: Audio Output Subclass	168
Table 42. Progress and Error Code Operations: LCD Device Subclass	168
Table 43. Progress and Error Code Operations: Network Device Subclass	169

Table 44. Progress Code Operations: I/O Bus Class	169
Table 45. Error Code Operations: I/O Bus Class	170
Table 46. Defined Subclasses: I/O Bus Class	172
Table 47. Progress and Error Code Operations: I/O Bus Unspecified Subclass	172
Table 48. Progress and Error Code Operations: PCI Subclass	173
Table 49. Progress and Error Code Operations: USB Subclass	174
Table 50. Progress and Error Code Operations: IBA Subclass	174
Table 51. Progress and Error Code Operations: AGP Subclass	175
Table 52. Progress and Error Code Operations: PC Card Subclass	175
Table 53. Progress and Error Code Operations: LPC Subclass	176
Table 54. Progress and Error Code Operations: SCSI Subclass	176
Table 55. Progress and Error Code Operations: ATA/ATAPI/SATA Subclass	176
Table 56. Progress and Error Code Operations: FC Subclass	177
Table 57. Progress and Error Code Operations: IP Network Subclass	178
Table 58. Progress and Error Code Operations: SMBus Subclass	178
Table 59. Progress and Error Code Operations: I2C Subclass	178
Table 60. Progress Code Operations: Host Software Class	179
Table 61. Error Code Operations: Host Software Class	180
Table 62. Defined Subclasses: Host Software Class	182
Table 63. Progress and Error Code Operations: Host Software Unspecified Subclass	183
Table 64. Progress and Error Code Operations: SEC Subclass	183
Table 65. Progress and Error Code Operations: PEI Foundation Subclass	184
Table 66. Progress and Error Code Operations: PEI Module Subclass	185
Table 67. Progress and Error Code Operations: DXE Foundation Subclass	186
Table 68. Progress and Error Code Operations: DXE Boot Service Driver Subclass	187
Table 69. Progress and Error Code Operations: DXE Runtime Service Driver Subclass ..	189
Table 70. Progress and Error Code Operations: SMM Driver Subclass	189
Table 71. Progress and Error Code Operations: UEFI Application Subclass	189
Table 72. Progress and Error Code Operations: OS Loader Subclass	190
Table 73. Progress and Error Code Operations: Runtime Subclass	190
Table 74. Progress and Error Code Operations: PEI Subclass	191
Table 75. Progress and Error Code Operations: Boot Services Subclass	193
Table 76. Progress and Error Code Operations: Runtime Services Subclass	195
Table 77. Progress and Error Code Operations: DXE Services Subclass	197
Table 78. Progress Code Enumeration Scheme	203
Table 79. Debug Code Enumeration Scheme	204
Table 80. Class Definitions	208
Table 81. Defined Subclasses: Computing Unit Class	209
Table 82. Description of EFI_CPU_STATE_CHANGE_CAUSE fields	224
Table 83. Definitions to describe Group Operations	227
Table 84. Defined Subclasses: User-Accessible Peripheral Class	230
Table 85. Defined Subclasses: I/O Bus Class	236
Table 86. Defined Subclasses: Host Software Class	

Platform Initialization

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.
<i>Italic</i>	In text, an <i>Italic</i> 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.
<u>Bold Monospace</u>	In the online help version of this specification, words in a <u>Bold Monospace</u> 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 <i>not</i> 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.
<i>Italic Monospace</i>	In code or in text, words in <i>Italic Monospace</i> 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.

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 18](#).

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 9](#).
- **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 9](#) 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

Name	Value	Description
EFI_FV_FILETYPE_APPLICATION	0x09	Application
EFI_FV_FILETYPE_SMM	0x0A	Contains a PE32+ image that will be loaded into SMRAM.
EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE	0x0B	Firmware volume image
EFI_FV_FILETYPE_COMBINED_SMM_DXE	0x0C	Contains PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into SMRAM.
EFI_FV_FILETYPE_SMM_CORE	0x0D	SMM Foundation
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_SMM_DXE** denotes a file that contains a PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into SMRAM.

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_SMM_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_PFI_CORE

The file type **EFI_FV_FILETYPE_PFI_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_CODE** 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_SMM

The file type **EFI_FV_FILETYPE_SMM** denotes a file that contains a PE32+ image that will be loaded into SMRAM.

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_SMM_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_SMM_CORE

The file type **EFI_FV_FILETYPE_DXE_CORE** denotes the SMM Foundation file. This image will be loaded by SMM IPL into SMRAM.

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.

2.1.5 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.

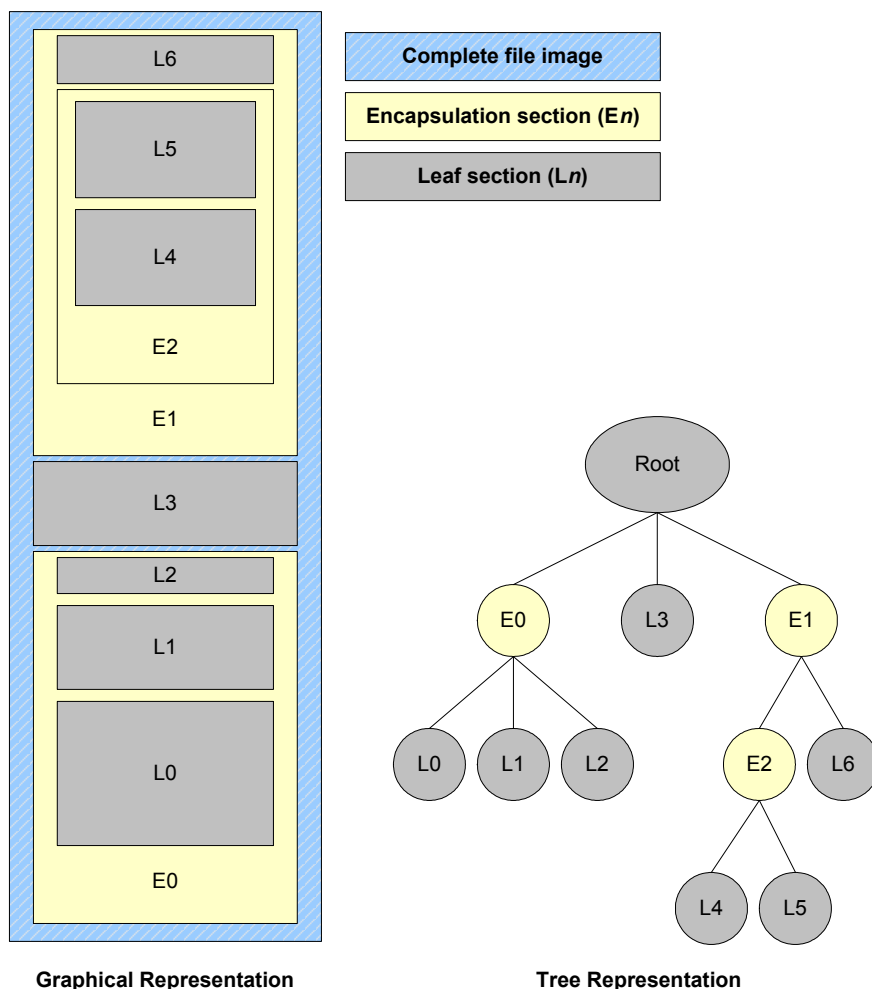


Figure 1. Example File Image (Graphical and Tree Representations)

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_SMM_DEPEX	0x1c	Leaf section type for determining the dispatch order for an SMM driver

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 7](#). 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.

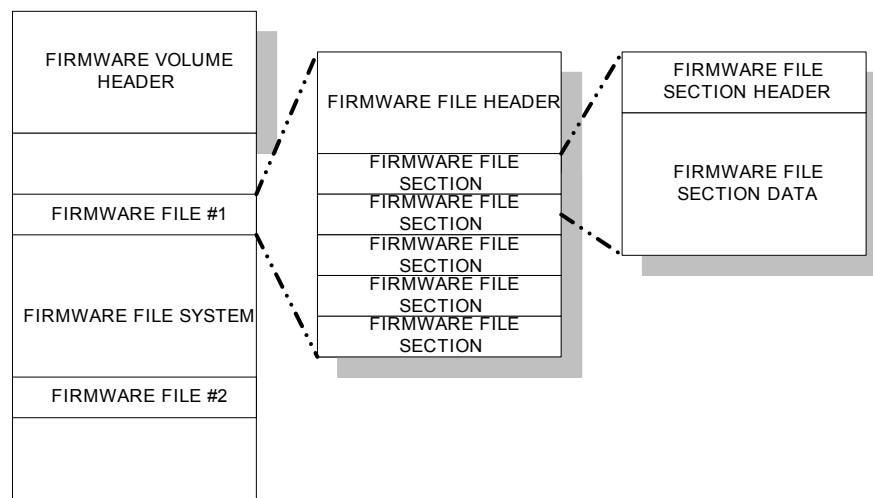


Figure 2. The Firmware Volume Format

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 31](#) 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 37](#) 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 39](#).

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-byte boundary 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:

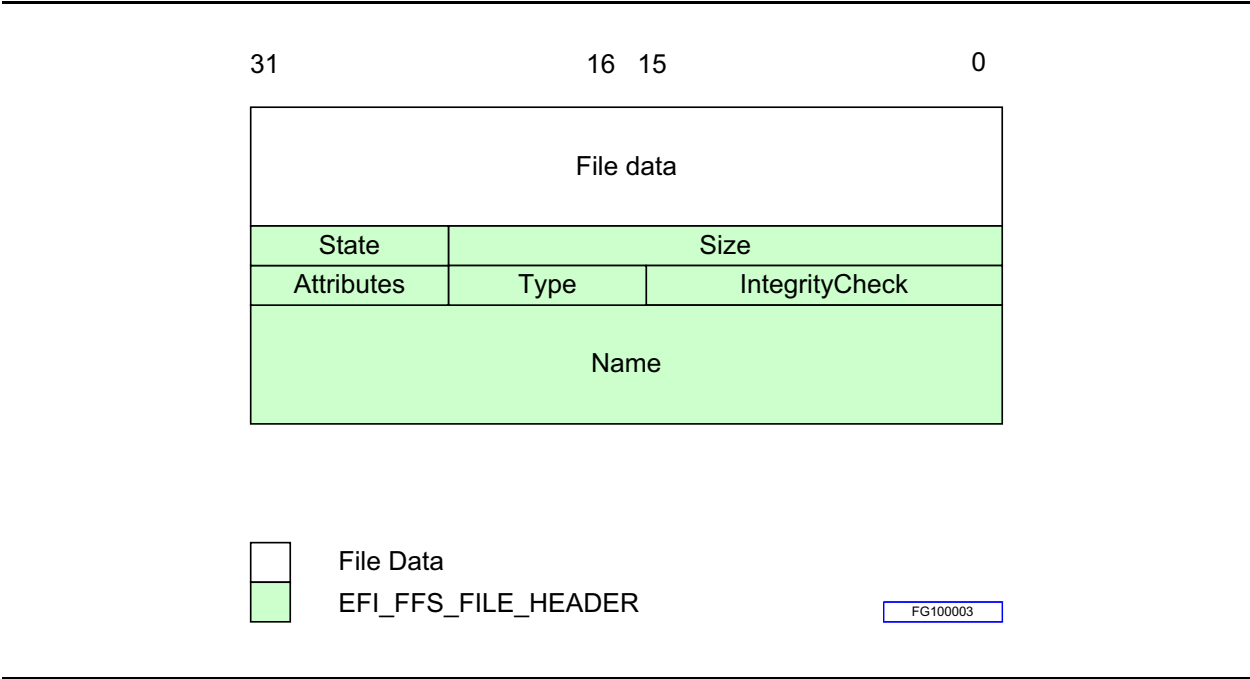


Figure 3. Typical FFS File Layout

Figure 4 illustrates the layout of a PI Architecture Firmware File **larger** than 16 Mb:

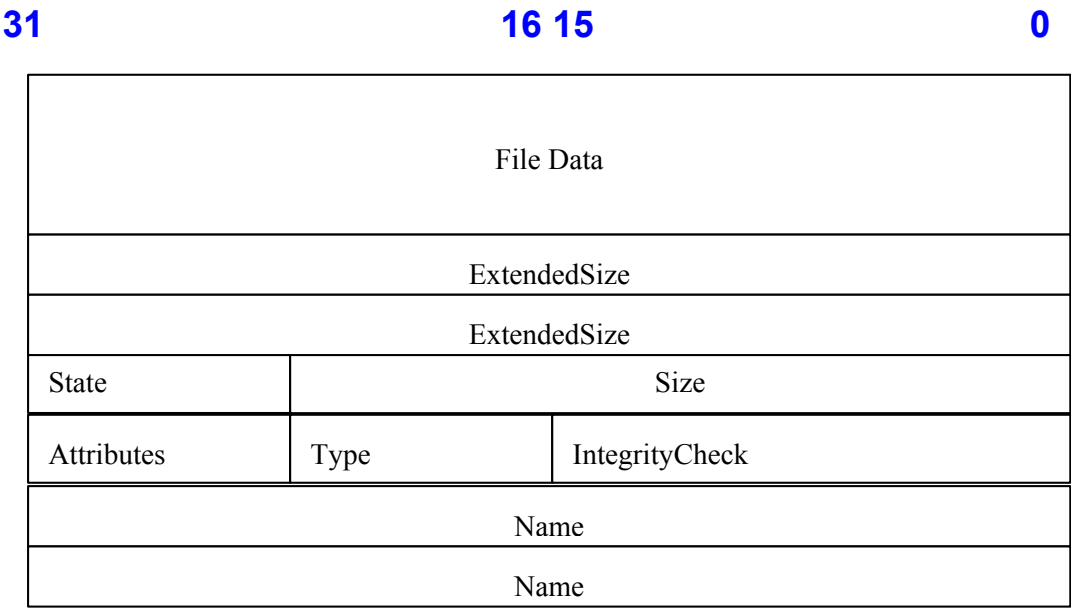


Figure 4. File Header 2 layout for files larger than 16Mb

2.2.4 Firmware File Section Format

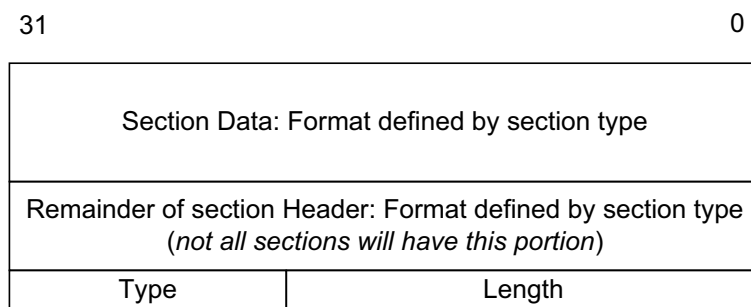
This section describes the standard firmware file section layout.

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.

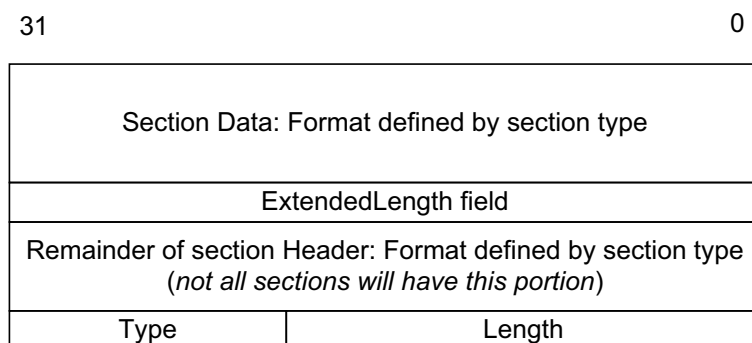
Many section types are variable in length and are more accurately described as data streams rather than data structures.

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 0xFFFFFFFF 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.



FG100005

Figure 5. Format of a section (below 16Mb)



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 31](#).

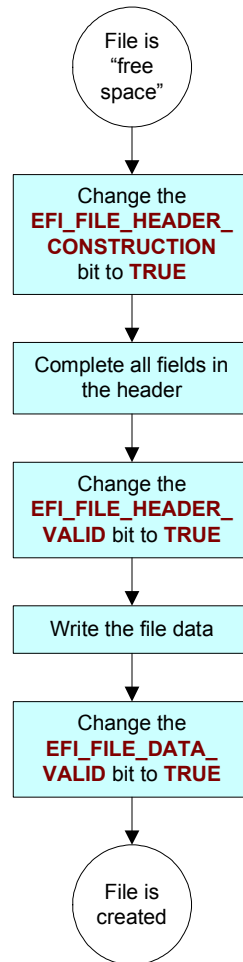
2.2.8.2 Initial State

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 31](#).

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.

**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* = **0000001b**, 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 contain 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* = **0000011b**, 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 40](#).

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* = **0000111b**, 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.

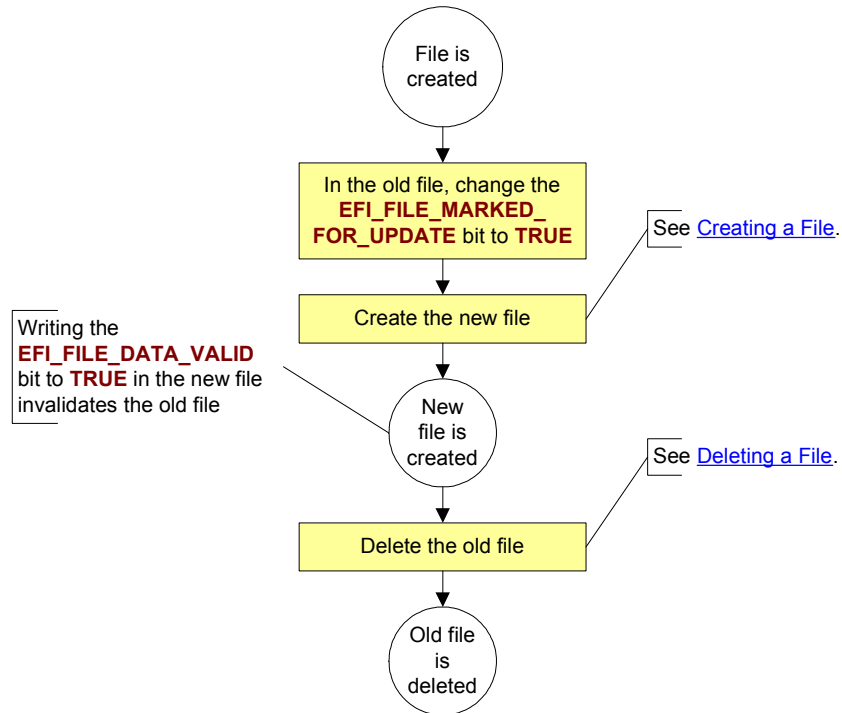


Figure 8. Updating a File

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 26](#). 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 28](#).

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.

Attribute	Description
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 107.
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 107.
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 $2^{\text{(this field)}}$. 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 array 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.

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 31](#) 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 31](#) 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 $(Size - \text{sizeof}(\text{EFI_FFS_FILE_HEADER}))$. This calculation means a zero-length file has a *Size* of 24 bytes, which is

$\text{sizeof}(\text{EFI_FFS_FILE_HEADER})$.

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 25](#) 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 0xFFFFF 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 25](#) 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 25](#) 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	Reserved. Must be set to 0	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	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 64KB alignment. FFS does not support this full range. The table below maps all FFS supported alignments to FFS_ATTRIB_DATA_ALIGNMENT 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 <i>IntegrityCheck.Checksum.File</i> . See the <i>IntegrityCheck</i> definition above for specific usage.

Table 7 maps all FFS-supported alignments to **FFS_ATTRIB_DATA_ALIGNMENT** values and firmware volume interface alignment values.

Table 7. Supported FFS Alignments

Required Alignment (bytes)	Alignment Value in FFS <i>Attributes</i> Field	Alignment Value in Firmware Volume Interfaces
1	0	0
16	1	4
128	2	7
512	3	9
1KiB	4	10
4KiB	5	12
32KiB	6	15
64KiB	7	16

```

//*****
// 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 25](#) for an explanation of how these bits are used. Type **EFI_FVB_ERASE_POLARITY** is defined in [EFI FIRMWARE VOLUME HEADER](#) on [page 31](#).

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 0xFFFFFFFF then *ExtendedSize* contains the size of the section. If *Size* is not equal to 0xFFFFFFFF 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 0xFFFFFFFF the size of the section header is sizeof (**EFI_COMMON_SECTION_HEADER2**). If *Size* is not equal to 0xFFFFFFFF 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_SMM_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_PPI_GUIDED_SECTION_EXTRACTION_PPI (PEI).. In this case, the beginning of the encapsulated section stream is indicated by the value of <i>DataOffset</i> .
EFI_GUIDED_SECTION_AUTH_STATUS_VALID	Set to 1 if the section contains authentication data that is reported through the <i>AuthenticationStatus</i> parameter returned from the GUIDED Section Extraction Protocol . If the EFI_GUIDED_SECTION_AUTH_STATUS_VALID bit is clear, the <i>AuthenticationStatus</i> 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_SMM_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for an SMM driver.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_SMM_DEPEX_SECTION;  
typedef EFI_COMMON_SECTION_HEADER2 EFI_SMM_DEPEX_SECTION2;
```

Description

The *SMM dependency expression section* is a leaf section that contains a dependency expression that is used to determine the dispatch order for SMM drivers. Before the SMRAM invocation of the SMM 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 SMM protocol database. **EFI_SMM_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. <i>FileHandle</i> 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	<i>FileHandle</i> or <i>FileName</i> 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 <i>FileHandle</i> does not represent a valid file.
EFI_INVALID_PARAMETER	If <i>FileInfo</i> 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 <i>FileHandle</i> does not represent a valid file.
EFI_INVALID_PARAMETER	If <i>FileInfo</i> 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	<i>FvHandle</i> does not indicate a valid firmware volume or <i>VolumeInfo</i> 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. <i>SectionData</i> 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. <i>SectionData</i> 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 82](#)

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	<i>FileHandle</i> was not a valid firmware file handle.
EFI_INVALID_PARAMETER	<i>EntryPoint</i> 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 <i>AuthenticationStatus</i> 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 <i>InputSection</i> 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 <i>InputSection</i> 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 Foundation.

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_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. <i>OutputBuffer</i> contains the resulting data and <i>OutputSize</i> 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 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               0x000000000001F000

#define EFI_FV2_ALIGNMENT_1             0x0000000000000000
#define EFI_FV2_ALIGNMENT_2             0x0000000000001000
#define EFI_FV2_ALIGNMENT_4             0x0000000000002000
#define EFI_FV2_ALIGNMENT_8             0x0000000000003000
#define EFI_FV2_ALIGNMENT_16            0x0000000000004000
#define EFI_FV2_ALIGNMENT_32            0x0000000000005000
#define EFI_FV2_ALIGNMENT_64            0x0000000000006000
#define EFI_FV2_ALIGNMENT_128           0x0000000000007000
#define EFI_FV2_ALIGNMENT_256           0x0000000000008000
#define EFI_FV2_ALIGNMENT_512           0x0000000000009000
#define EFI_FV2_ALIGNMENT_1K            0x000000000000A000
#define EFI_FV2_ALIGNMENT_2K            0x000000000000B000
#define EFI_FV2_ALIGNMENT_4K            0x000000000000C000
#define EFI_FV2_ALIGNMENT_8K            0x000000000000D000
#define EFI_FV2_ALIGNMENT_16K           0x000000000000E000
#define EFI_FV2_ALIGNMENT_32K           0x000000000000F000
#define EFI_FV2_ALIGNMENT_64K           0x0000000000010000
#define EFI_FV2_ALIGNMENT_128K          0x0000000000011000
#define EFI_FV2_ALIGNMENT_256K          0x0000000000012000
#define EFI_FV2_ALIGNMENT_512K          0x0000000000013000
#define EFI_FV2_ALIGNMENT_1M            0x0000000000014000
#define EFI_FV2_ALIGNMENT_2M            0x0000000000015000
#define EFI_FV2_ALIGNMENT_4M            0x0000000000016000
#define EFI_FV2_ALIGNMENT_8M            0x0000000000017000
#define EFI_FV2_ALIGNMENT_16M           0x0000000000018000
#define EFI_FV2_ALIGNMENT_32M           0x0000000000019000
#define EFI_FV2_ALIGNMENT_64M           0x000000000001A000
#define EFI_FV2_ALIGNMENT_128M          0x000000000001B000
#define EFI_FV2_ALIGNMENT_256M          0x000000000001C000
#define EFI_FV2_ALIGNMENT_512M          0x000000000001D000
#define EFI_FV2_ALIGNMENT_1G            0x000000000001E000

```

```
#define EFI_FV2_ALIGNMENT_2G 0x000000000001F0000
```

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 <i>FvAttributes</i> .
-------------	---

EFI_INVALID_PARAMETER	<i>FvAttributes:EFI_FV_READ_STATUS</i> is set to 1 on input, but the device does not support enabling reads (<i>FvAttributes:EFI_FV_READ_ENABLE_CAP</i> is clear on return from GetVolumeAttributes()). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	<i>FvAttributes:EFI_FV_READ_STATUS</i> is cleared to 0 on input, but the device does not support disabling reads (<i>FvAttributes:EFI_FV_READ_DISABLE_CAP</i> is clear on return from GetVolumeAttributes()). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	<i>FvAttributes:EFI_FV_WRITE_STATUS</i> is set to 1 on input, but the device does not support enabling writes (<i>FvAttributes:EFI_FV_WRITE_ENABLE_CAP</i> is clear on return from GetVolumeAttributes()). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	<i>FvAttributes:EFI_FV_WRITE_STATUS</i> is cleared to 0 on input, but the device does not support disabling writes (<i>FvAttributes:EFI_FV_WRITE_DISABLE_CAP</i> is clear on return from GetVolumeAttributes()). Actual volume attributes are unchanged.
EFI_INVALID_PARAMETER	<i>FvAttributes:EFI_FV_LOCK_STATUS</i> is set on input, but the device does not support locking (<i>FvAttributes:EFI_FV_LOCK_CAP</i> is clear on return from GetVolumeAttributes()). Actual volume attributes are unchanged.
EFI_ACCESS_DENIED	Device is locked and does not allow attribute modification (<i>FvAttributes:EFI_FV_LOCK_STATUS</i> 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_GUIDS**. 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 9](#) 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_PROTOCOL.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 <i>Attributes</i> 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

Required Alignment (bytes)	Alignment Value in <i>Attributes</i> Field
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	<i>Name</i> 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_PROTOCOL.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[n]* 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 <i>WritePolicy</i> 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 <i>WritePolicy</i> 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 <i>WritePolicy</i> 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 <i>FileData</i> 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 40](#).

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 store file(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 <i>WritePolicy</i> was requested.
EFI_INVALID_PARAMETER	An unknown file type was specified 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 40](#).

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 <i>FileType</i> 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 <i>InformationType</i> 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 <i>BufferSize</i> is too small to read the current directory entry. <i>BufferSize</i> 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 <i>InformationType</i> 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	<i>BufferSize</i> is smaller than the size of the type indicated by <i>InformationType</i> .

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 <i>Buffer</i> .
EFI_BAD_BUFFER_SIZE	Read attempted across an LBA boundary. On output, <i>NumBytes</i> contains the total number of bytes returned in <i>Buffer</i> .
EFI_ACCESS_DENIED	The firmware volume is in the <i>ReadDisabled</i> 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, <i>NumBytes</i> contains the total number of bytes actually written.
EFI_ACCESS_DENIED	The firmware volume is in the <i>WriteDisabled</i> 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 <i>WriteDisabled</i> 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_PROTOCOL.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 82](#) 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 <i>InputSection</i> 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 <i>InputSection</i> does not match this instance of the GUIDed Section Extraction Protocol.

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.

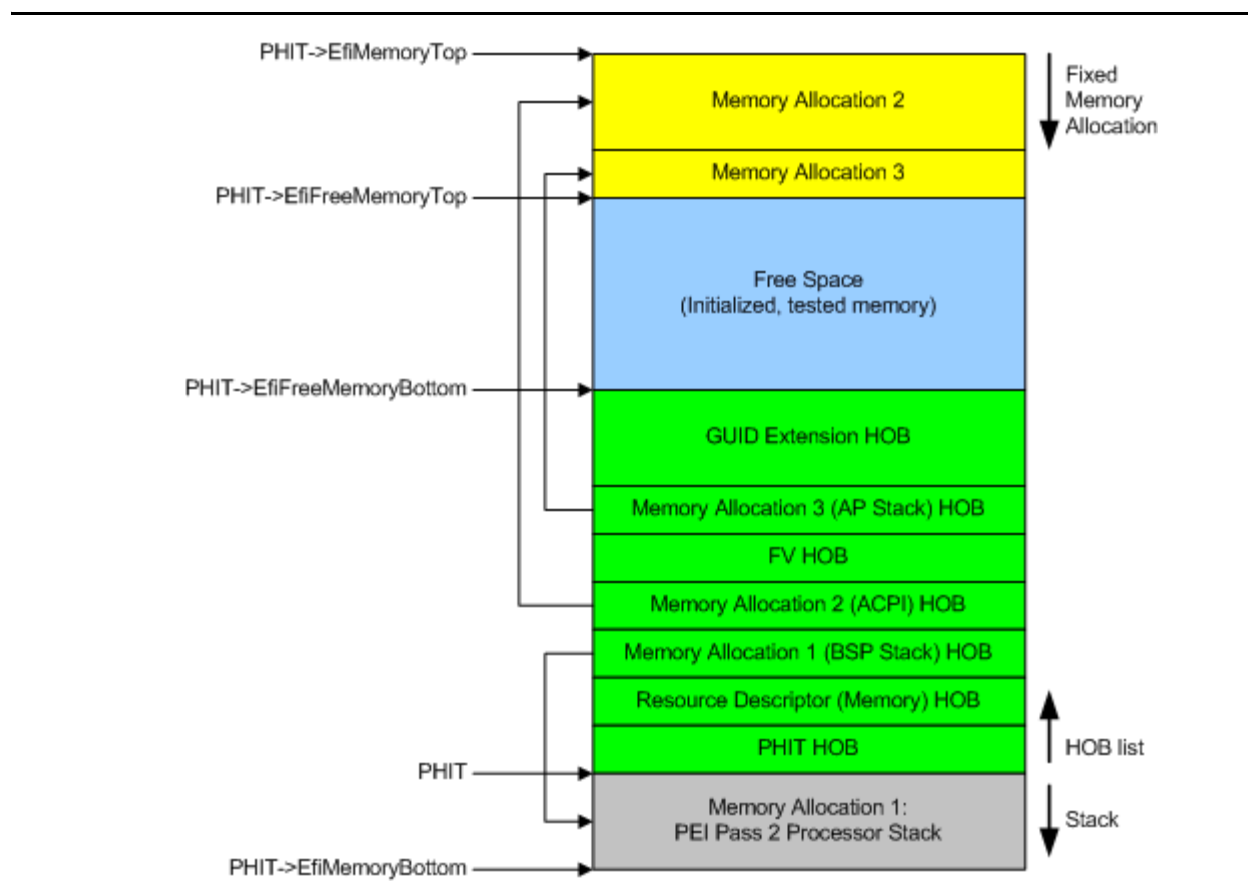


Figure 11. Example HOB Producer Phase Memory Map and Usage

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 consumer-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 $\text{NewHobSize} \leq \text{PHIT} \rightarrow \text{EfiFreeMemoryTop} - \text{PHIT} \rightarrow \text{EfiFreeMemoryBottom}$.
3. Construct the HOB at $\text{PHIT} \rightarrow \text{EfiFreeMemoryBottom}$.
4. Set $\text{PHIT} \rightarrow \text{EfiFreeMemoryBottom} = \text{PHIT} \rightarrow \text{EfiFreeMemoryBottom} + \text{NewHobSize}$.

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_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_UNUSED                 0xFFFF
#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 device's 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.

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 a 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 0xFFFF
```

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**.

6

Platform Initialization 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.

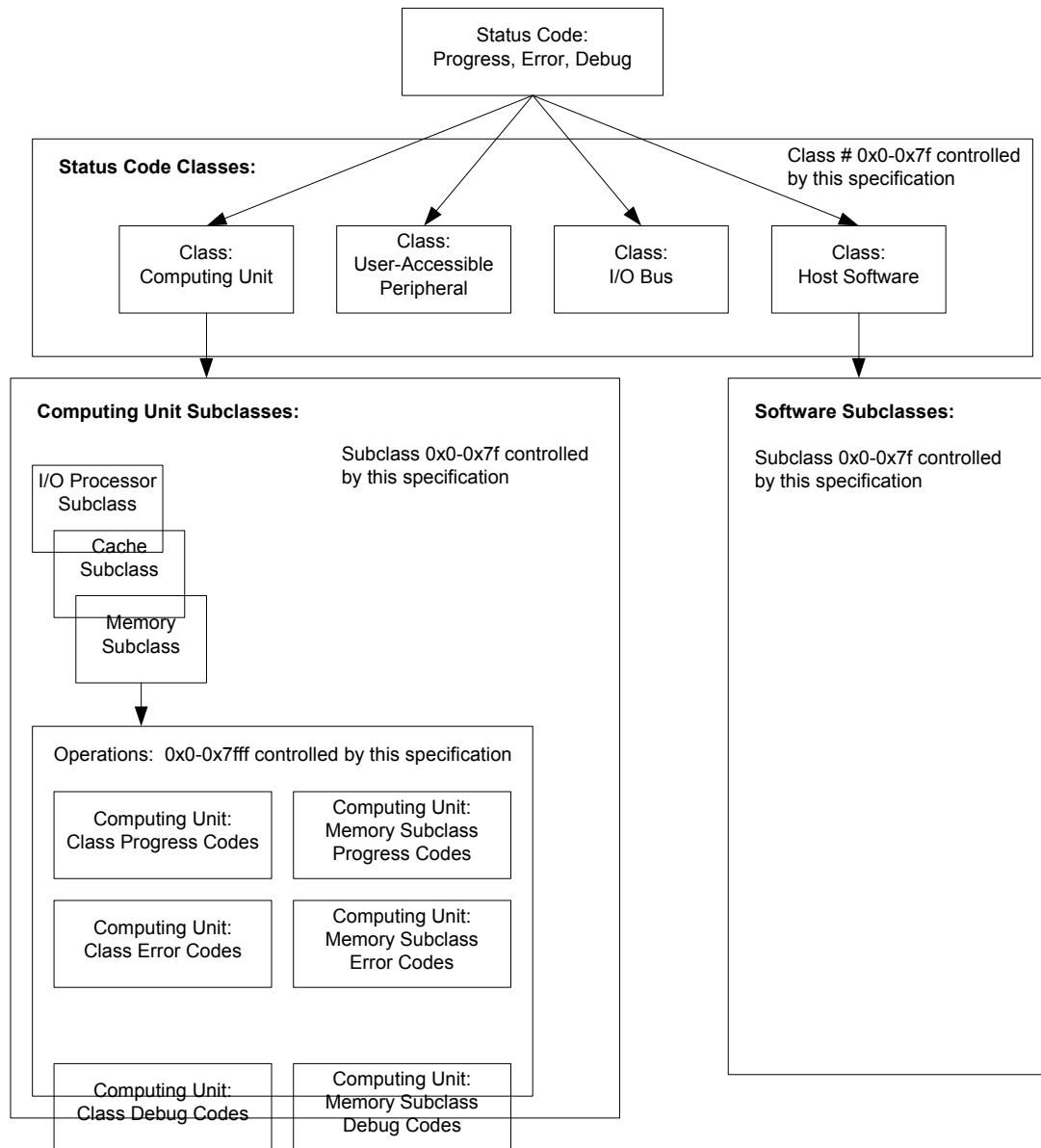


Figure 12. Hierarchy of Status Code Operations

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_PROCESSOR	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 6.7.1.1 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 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 6.7.1.1 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 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_UPDATE	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_DETECTION	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 miscomparisons 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_SB_ERROR	South Bridge initialization error in DXE	None
Error	0x1003–0x7FFF	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.
0x0006–0x0FFF	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

6.4.3 Subclasses

6.4.3.1 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.
0x0D–0x7F	Reserved for future use by this specification.	
0x80–0xFF	Reserved for OEM use.	

6.4.3.1.1 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.

6.4.3.1.2 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
	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 6.7.2 for definitions of the extended error data listed above.

6.4.3.1.3 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.

6.4.3.1.4 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.

6.4.3.1.5 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.

6.4.3.1.6 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.

6.4.3.1.7 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.

6.4.3.1.8 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.

6.4.3.1.9 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.

6.4.3.1.10 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.

6.4.3.1.11 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.

6.4.3.1.12 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.

6.4.3.1.13 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.

6.4.3.1.14 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

6.4.3.1.15 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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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 <code>EFI_DEVICE_PATH_EXTENDED_DATA</code> .
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.

Platform Initialization Status Codes

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_EXTENDED_DATA.
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_ALLOC	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

Type of code	Operation	Description	Extended data
	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 6.7.3 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 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 <code>EFI_LOADED_IMAGE_PROTOCOL</code> on this handle. See <code>EFI_DEVICE_HANDLE_EXTENDED_DATA</code> .
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 <code>EFI_DEBUG_ASSERT_DATA</code>
EFI_SW_EC_ILLEGAL_HARDWARE_STATE	The instance detected an illegal hardware state.	None.

Operation	Description	Extended Data
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.
EFI_SW_EC_EVENT_LOG_FULL	System event log is full.	None.
0x0012–0x00FF	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 an x64 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.
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
	0x1007–0x7FFF	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
	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

Type of Code	Operation	Description	Extended Data
	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
	0x1008–0x7FFF	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_BINDING 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_OPMOM_INIT	Initializing a legacy Option ROM (OpROM).	See EFI_LEGACY_OPMOM_EXTENDED_DATA.
	EFI_SW_DXE_BS_PC_READY_TO_BOOT_EVENT	The EFI_EVENT_GROUP_READY_TO_BOOT event was signaled. See the <i>UEFI Specification</i> .	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 <i>UEFI Specification</i> .	None

Type of Code	Operation	Description	Extended Data
	EFI_SW_DXE_BS_PC_VIRTUAL_ADDRESS_CHANGE_EVENT	The EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE event was signaled. See the <i>UEFI Specification</i> .	None
	0x1000–0x7FFF	Reserved for future use by this specification.	NA
Error	EFI_SW_DXE_BS_EC_LEGACY_OPRM_NO_SPACE	Not enough memory available to shadow a legacy option ROM.	See EFI_LEGACY_OPRM_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_SW_DXE_BS_EC_BOOT_OPTION_FAILED	Error during boot option launch (StartImage returned error)	
	EFI_SW_DXE_BS_EC_INVALID_IDE_PASSWORD	Invalid hard driver password has been provided	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.

Type of Code	Operation	Description	Extended Data
	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
	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 capsule 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	<p>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.</p> <p>These codes are reserved by this specification.</p>

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 6.3 for 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)

```

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. Description 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
0x0D–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)
```

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

//
// 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)  
  
//  
// 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)

```



```

#define EFI_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

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)

//
// 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)

//
// 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

//
// 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)

//
// 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_ERROR EXCEPT_AARCH64_ERROR
```

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_DATA**

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_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*.

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.

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.

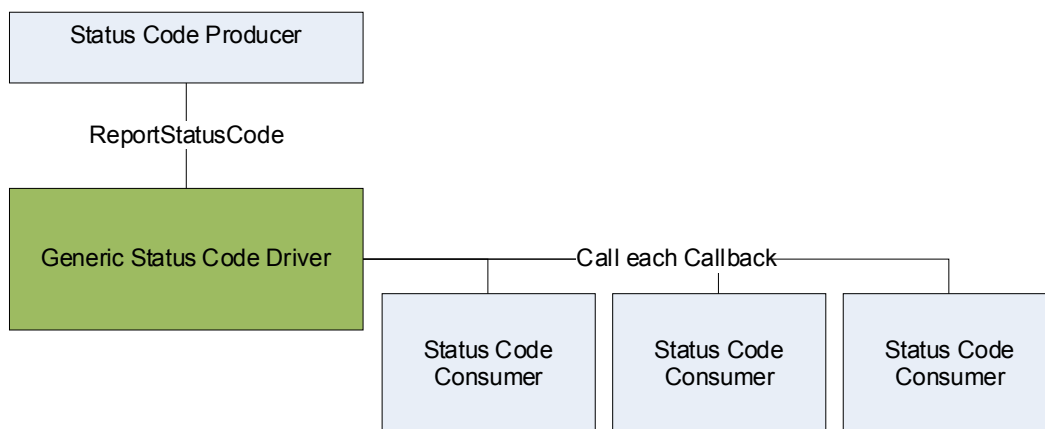


Figure 13. Status Code Services

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.

8.1 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 is 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 <i>GetBool ()</i> 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 <i>GetBool()</i> 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 <i>GetBool()</i> 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 <i>GetBool()</i> 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 <i>SizeOfValue</i> 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 <i>CallbackToken</i> 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 <i>CallbackToken</i> .
EFI_INVALID_PARAMETER	The PCD service did not match the <i>CallbackFunction</i> 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
(EFI_API *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 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 <i>SizeofValue</i> 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 <i>CallbackToken</i> 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 <i>CallbackToken</i> .
EFI_INVALID_PARAMETER	The PCD service did not match the <i>CallbackFunction</i> 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.

