VOLUME 3: Platform Initialization

Shared Architectural Elelments

Version 1.2 Errata B

July1, 2010

Revision History

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

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

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

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

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

Specification Volumes

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

VOLUME 1: Pre-EFI Initialization Core Interface

VOLUME 2: Driver Execution Environment Core Interface

VOLUME 3: Shared Architectural Elements

VOLUME 4: System Management Mode

VOLUME 5: Standards

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

Contents

Figures

Tables

1
Platform Intialization Shared Architectural Elements

1.1 Overview

This volume describes the basic concepts behind Platform Initialization (PI) firmware storage and Hand-Off Blocks implementation.

The basic Platform Initialization (PI) firmware storage concepts include:

- Firmware Volumes
- Firmware File Systems
- Firmware Files
- Standard Binary Layout
- Pre-EFI Initialization (PEI) PEIM-to-PEIM Interfaces (PPIs)
- Driver Execution Environment (DXE) Protocols

The core code that is required for an implementation of Hand-Off Blocks (HOBs) in the Platform Initialization (PI) Architecture specifications are also shown. A HOB is a binary data structure that passes system state information from the HOB producer phase to the HOB consumer phase in the PI Architecture. This HOB specification does the following:

- Describes the basic components of HOBs and the rules for constructing them
- Provides code definitions for the HOB data types and structures that are architecturally required by the PI Architecture specifications

1.2 Target Audience

This document is intended for the following readers:

- Independent hardware vendors (IHVs) and original equipment manufacturers (OEMs) who will be implementing firmware components that are stored in firmware volumes
- Firmware developers who create firmware products or those who modify these products for use in platforms

1.3 Conventions Used in this Document

This document uses the typographic and illustrative conventions described below.

1.3.1 Data Structure Descriptions

Supported processors are “little endian” machines. This distinction means that the low-order byte of a multibyte data item in memory is at the lowest address, while the high-order byte is at the highest

address. Some supported processors may be configured for both “little endian” and “big endian” operation. All implementations designed to conform to this specification will use “little endian” operation.

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

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

STRUCTURE NAME: The formal name of the data structure.

Summary: A brief description of the data structure.

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

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

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

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

1.3.2 Pseudo-Code Conventions

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

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

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

1.3.3 Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text	The normal text typeface is used for the vast majority of the descriptive text in a specification.
Plain text (blue)	In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.

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

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

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

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

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

In general, for an implementation to be conformant with this specification, the implementation must include functional elements that match in all respects the complete description of the required

facility descriptions presented as part of the specification. Any part of the specification that is not explicitly marked as “optional” is considered a required facility.

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

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

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

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

2
Firmware Storage Design Discussion

2.1 Firmware Storage Introduction

This specification describes how files should be stored and accessed within non-volatile storage. Firmware implementations must support the standard PI Firmware Volume and Firmware File System format (described below), but may support additional storage formats.

2.1.1 Firmware Devices

A *firmware device* is a persistent physical repository that contains firmware code and/or data. It is typically a flash component but may be some other type of persistent storage. A single physical firmware device may be divided into smaller pieces to form multiple logical firmware devices. Similarly, multiple physical firmware devices may be aggregated into one larger logical firmware device.

This section describes the characteristics of typical physical firmware devices.

2.1.1.1 Flash

Flash devices are the most common non-volatile repository for firmware volumes. Flash devices are often divided into sectors (or blocks) of possibly differing sizes, each with different run-time characteristics. Flash devices have several unique qualities that are reflected in the design of the firmware file system:

- Flash devices can be erased on a sector-by-sector basis. After an erasure, all bits within a sector return to their *erase value*, either all 0 or all 1.
- Flash devices can be written on a bit-by-bit basis if the change is from its erase value to the non-erase value. For example, if the erase value is 1, then a bit with the value 1 can be changed to 0.
- Flash devices can only change from a non-erase value to an erase value by performing an erase operation on an entire flash sector.
- Some flash devices can enable or disable reads and writes to the entire flash device or to individual flash sectors.
- Some flash devices can lock the current enable or disable state of reads and writes until the next reset.
- Flash writes and erases are often longer operations than reads.
- Flash devices often place restrictions on the operations that can be performed while a write or erase is occurring.

2.1.2 Firmware Volumes

A Firmware Volume (FV) is a logical firmware device. In this specification, the basic storage repository for data and/or code is the firmware volume. Each firmware volume is organized into a file system. As such, the file is the base unit of storage for firmware.

Each firmware volume has the following attributes:

- **Name.** Each volume has a name consisting of an UEFI Globally Unique Identifier (GUID).
- **Size.** Each volume has a size, which describes the total size of all volume data, including any header, files, and free space.
- **Format.** Each volume has a format, which describes the Firmware File System used in the body of the volume.
- **Memory Mapped?** Some volumes may be memory-mapped, which indicates that the entire contents of the volume appear at once in the memory address space of the processor.
- **Sticky Write?** Some volumes may require special erase cycles in order to change bits from a non-erase value to an erase value.
- **Erase Polarity.** If a volume supports "Sticky Write," then all bits within the volume will return to this value (0 or 1) after an erase cycle.
- **Alignment.** The first byte of a volume is required to be aligned on some power-of-two boundary. At a minimum, this must be greater than or equal to the highest file alignment value.
- **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 15.

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 7.
- **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.

- **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 12.

Non-standard file types are supported through the use of the OEM file types. See "Firmware File Types" on page 7 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 1. Defined File Types

Name	Value	Description
EFI_FV_FILETYPE_RAW	0x01	Binary data
EFI_FV_FILETYPE_FREEFORM	0x02	Sectioned data
EFI_FV_FILETYPE_SECURITY_CORE	0x03	Platform core code used during the SEC phase
EFI_FV_FILETYPE_PEI_CORE	0x04	PEI Foundation
EFI_FV_FILETYPE_DXE_CORE	0x05	DXE Foundation
EFI_FV_FILETYPE_PEIM	0x06	PEI module (PEIM)
EFI_FV_FILETYPE_DRIVER	0x07	DXE driver
EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER	0x08	Combined PEIM/DXE driver
EFI_FV_FILETYPE_APPLICATION	0x09	Application
EFI_FV_FILETYPE_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.

Name	Value	Description
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.

A single **EFI_SECTION_FREEFORM_SUBTYPE_GUID** section may be included in a file of type **EFI_FV_FILETYPE_FREEFORM** to provide additional file type differentiation. While it is permissible to omit the **EFI_SECTION_FREEFORM_SUBTYPE_GUID** section entirely, there must never be more than one instance of it.

2.1.4.1.8 EFI_FV_FILETYPE_FFS_PAD

A pad file is an FFS-defined file type that is used to pad the location of the file that follows it in the storage file. The normal state of any valid (not deleted or invalidated) file is that both its header and data are valid. This status is indicated using the *State* bits with *State* **= 00000111b**. Pad files differ from all other types of files in that any pad file in this state must *not* have any data written into the data space. It is essentially a file filled with free space.

Standard firmware file system services will not return the handle of any pad files, nor will they permit explicit creation of such files. The *Name* field of the **EFI_FFS_FILE_HEADER** structure is considered invalid for pad files and will not be used in any operation that requires name comparisons.

2.1.4.1.9 EFI_FV_FILETYPE_PEIM

The file type **EFI_FV_FILETYPE_PEIM** denotes a file that is a PEI module (PEIM). A PEI module is dispatched by the PEI Foundation based on its dependencies during execution of the PEI phase. See the *Platform Initialization Pre-EFI Initialization Core Interface Specification* for details on PEI operation.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain one and only one executable section. This section must have one of the following types:
  —**EFI_SECTION_PE32**
  —**EFI_SECTION_PIC**
  —**EFI_SECTION_TE**
- The file must contain no more than one **EFI_SECTION_VERSION** section.
- The file must contain no more than one **EFI_SECTION_PEI_DEPEX** section.

As long as the above rules are followed, the file may contain other leaf and encapsulation sections as required or enabled by the platform design. Care must be taken to ensure that additional encapsulations do not render the file inaccessible due to execute-in-place requirements.

2.1.4.1.10 EFI_FV_FILETYPE_PEI_CORE

The file type **EFI_FV_FILETYPE_PEI_CORE** denotes the PEI Foundation file. This image is entered upon completion of the SEC phase of a PI Architecture-compliant boot cycle.

This file type is a sectioned file that must be constructed in accordance with the following rules:

- The file must contain one and only one executable section. This section must have one of the following types:
  —**EFI_SECTION_PE32**
  —**EFI_SECTION_PIC**
  —**EFI_SECTION_TE**
- The file must contain no more than one **EFI_SECTION_VERSION** section.

As long as the above rules are followed, the file may contain other leaf and encapsulations as required/enabled by the platform design.

2.1.4.1.11 EFI_FV_FILETYPE_RAW

The file type **EFI_FV_FILETYPE_RAW** denotes a file that does not contain sections and is treated as a raw data file. The consumer of this type of file must have *a priori* knowledge of its format and content. Because there are no sections, there are no construction rules.

2.1.4.1.12 EFI_FV_FILETYPE_SECURITY_CODE

The file type **EFI_FV_FILETYPE_SECURITY_CORE** denotes code and data that comprise the first part of PI Architecture firmware to execute. Its format is undefined with respect to the PI Architecture, as differing platform architectures may have varied requirements.

2.1.4.1.13 EFI_FV_FILETYPE_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 13.
- **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.

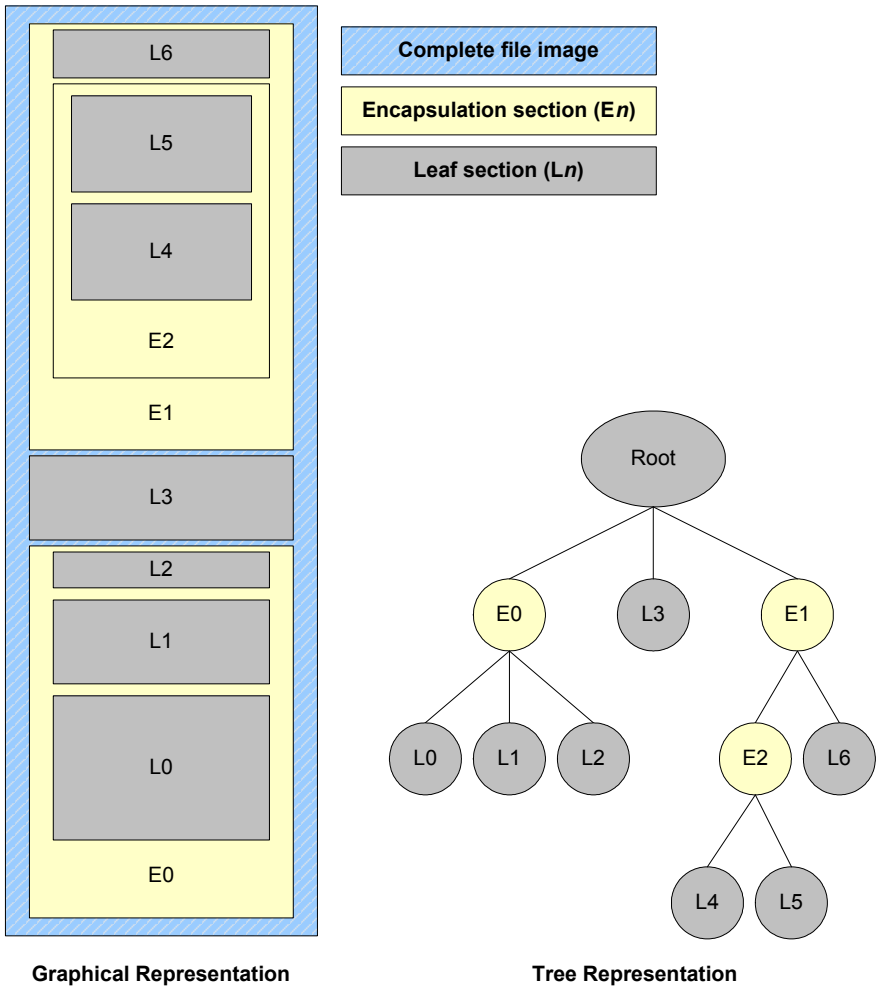

Figure 1. Example File Image (Graphical and Tree Representations)

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 2 lists the defined architectural section types.

Table 2. 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 5. 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.

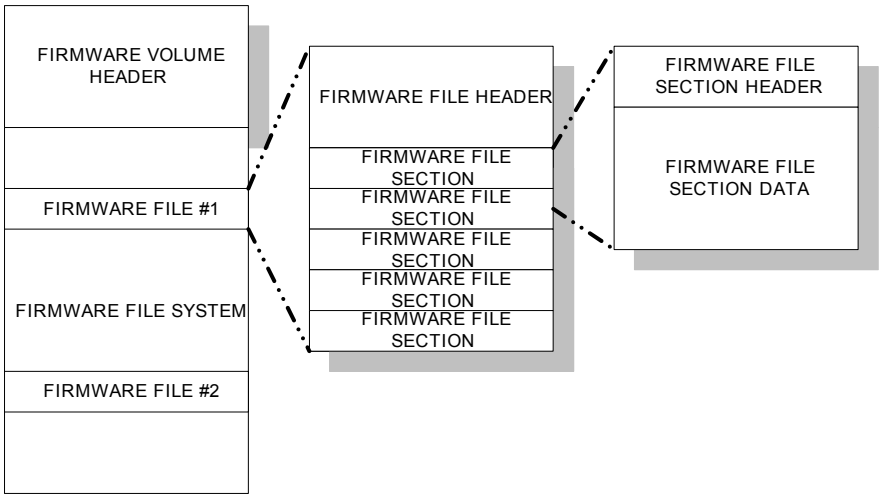

Figure 2. The Firmware Volume Format

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. Typically, this contains the **`EFI_FIRMWARE_FILE_SYSTEM2_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 14. 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
- 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 29 for more information on the firmware volume header. For the FFS file system, the GUID is defined as **EFI_FIRMWARE_FILE_SYSTEM2_GUID** on page 35.

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 36.

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 8-byte, aligned 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 means it 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 **EFI_FFS_FILE_HEADER** on page 37.

Figure 3 illustrates the layout of a typical PI Architecture Firmware File.

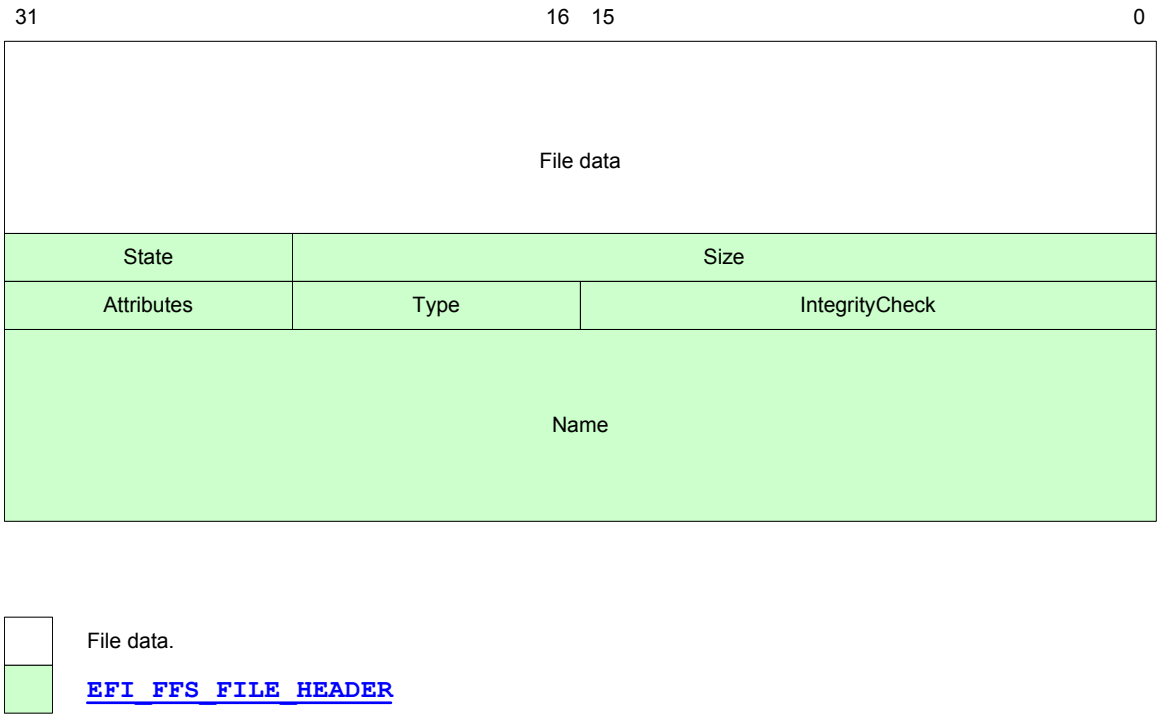

Figure 3. Typical FFS File Layout

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 are 4-byte aligned within the parent 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. Figure 4 shows the general format of a section.

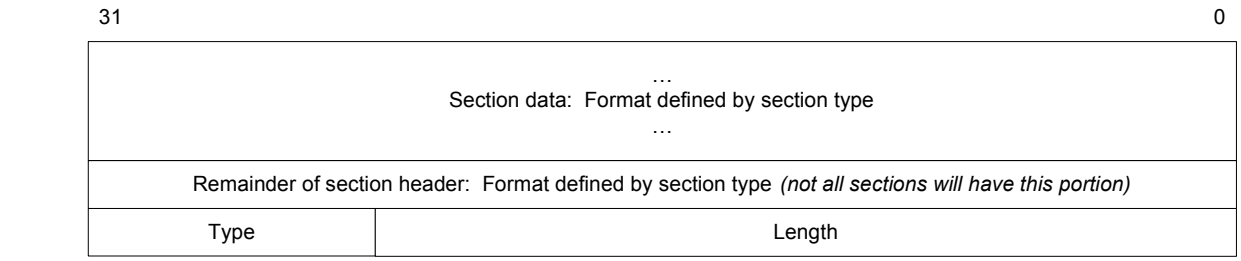

Figure 4. General Section Format

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) {
    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 18.

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 29.

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 29

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 5 illustrates the steps to create a new file, which are detailed below the figure.

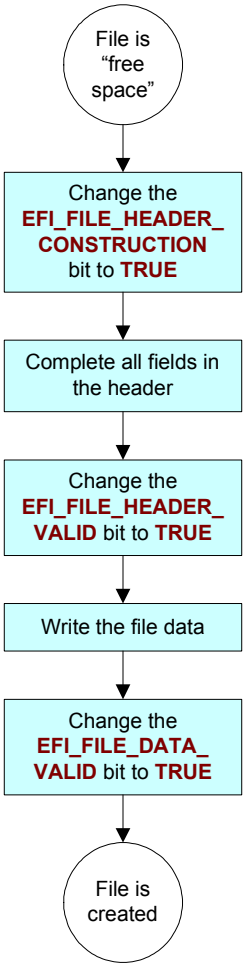

Figure 5. Creating a File

As shown in Figure 5, the following steps are required to create a new file:

1. Allocate space in the firmware volume for a new **EFI_FFS_FILE_HEADER** and complete all fields of the header (except for the *State* field, which is updated independently from the rest of the header). This allocation is done by interpreting the free space as a file header and changing the **EFI_FILE_HEADER_CONSTRUCTION** bit to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00000001b,** which indicates the header construction

has begun but has not yet been completed. This value has the effect of "claiming" the FFS header space from the firmware volume free space.

While in this state, the following fields of the FFS header are initialized and written to the firmware volume:

- *Name*
- *IntegrityCheck.Header*
- *Type*
- *Attributes*
- *Size*

The value of *IntegrityCheck.Header* is calculated as described in **EFI_FFS_FILE_HEADER** on page 37.

2. Mark the new header as complete and write the file data. To mark the header as complete, the **EFI_FILE_HEADER_VALID** bit is changed to **TRUE**. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00000011b,** which indicates the header construction is complete, but the file data has not yet been written. This value has the effect of "claiming" the full length of the file from the firmware volume free space. Once the **EFI_FILE_HEADER_VALID** bit is set, no further changes to the following fields may be made:

   - *Name*
   - *IntegrityCheck.Header*
   - *Type*
   - *Attributes*
   - *Size*

   While in this state, the file data and *IntegrityCheck.File* are written to the firmware volume. The order in which these are written does not matter. The calculation of the value for *IntegrityCheck.File* is described in **EFI_FFS_FILE_HEADER** on page 37.

3. Mark the data as valid. To mark the data as valid, the **EFI_FILE_DATA_VALID** bit is changed to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00000111b,** which indicates the file data is fully written and is valid.

2.2.8.4 Deleting a File

Any file with **EFI_FILE_HEADER_VALID** set to **TRUE** and **EFI_FILE_HEADER_INVALID** and **EFI_FILE_DELETED** set to **FALSE** is a candidate for deletion.

To delete a file, the **EFI_FILE_DELETED** bit is set to the **TRUE** state. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 0001xx11b**, which indicates the file is marked deleted. Its header is still valid, however, in as much as its length field is used in locating the next file in the firmware volume.

Note: *The* **EFI_FILE_HEADER_INVALID** *bit must be left in the* **FALSE** *state.*

2.2.8.5 Updating a File

A file update is a special case of file creation where the file being added already exists in the firmware volume. At all times during a file update, only one of the files, either the new one or the old one, is valid at any given time. This validation is possible by using the **EFI_FILE_MARKED_FOR_UPDATE** bit in the old file.

Figure 6 illustrates the steps to update a file, which are detailed below the figure.

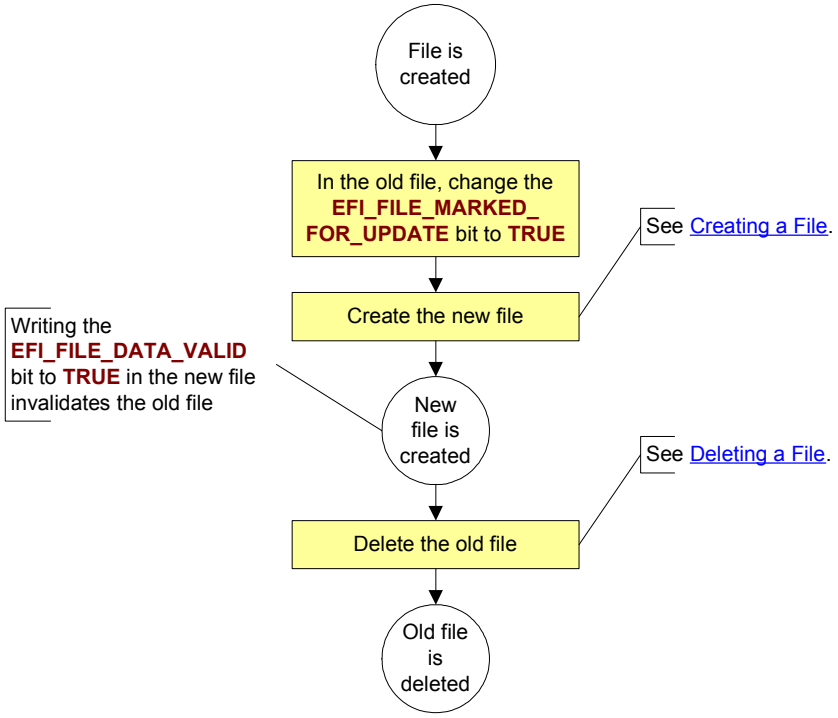

Figure 6. Updating a File

As shown in Figure 6, 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 24. 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 25.

3
Firmware Storage Code Definitions

3.1 Firmware Storage Code Definitions Introduction

This section provides the code definitions for:

- The PI Architecture Firmware Storage binary formats for volumes, file system, files, and file sections.
- The PEI interfaces that support firmware volumes, firmware file systems, firmware files, and firmware file sections.
- The DXE protocols that support firmware volumes, firmware file systems, firmware files, and firmware file sections.

3.2 Firmware Storage Formats

3.2.1 Firmware Volume

EFI_FIRMWARE_VOLUME_HEADER

Summary

Describes the features and layout of the firmware volume.

Prototype

```
typedef struct {
  UINT8                 ZeroVector[16];
  EFI_GUID              FileSystemGuid;
  UINT64                FvLength;
  UINT32                Signature;
  EFI_FVB_ATTRIBUTES_2  Attributes;
  UINT16                HeaderLength;
  UINT16                Checksum;
  UINT16                ExtHeaderOffset;
  UINT8                 Reserved[1];
  UINT8                 Revision;
  EFI_FV_BLOCK_MAP      BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

Parameters

ZeroVector

The first 16 bytes are reserved to allow for the reset vector of processors whose reset vector is at address 0.

FileSystemGuid

Declares the file system with which the firmware volume is formatted. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *Unified Extensible Firmware Interface Specification*, version 2.0 (UEFI 2.0 specification).

FvLength

Length in bytes of the complete firmware volume, including the header.

Signature

Set to **{'_','F','V','H'}**.

Attributes

Declares capabilities and power-on defaults for the firmware volume. Current state is determined using the **GetAttributes()** function and is not maintained in the *Attributes* field of the firmware volume header. Type **EFI_FVB_ATTRIBUTES_2** is defined in "Related Definitions" below.

HeaderLength

Length in bytes of the complete firmware volume header.

Checksum

A 16-bit checksum of the firmware volume header. A valid header sums to zero.

ExtHeaderOffset

Offset, relative to the start of the header, of the extended header (**EFI_FIRMWARE_VOLUME_EXT_HEADER**) or zero if there is no extended header. The extended header is followed by zero or more variable length extension entries. Each extension entry is prefixed with the **EFI_FIRMWARE_VOLUME_EXT_ENTRY** structure (see "Related Definitions" below), which defines the type and size of the extension entry. The extended header is always 32-bit aligned relative to the start of the file header

.If there is an instance of the **EFI_FIRMWARE_VOLUME_EXT_ENTRY**, 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_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_ALIGNMNET_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 3 describes the fields in the above definition:

Table 3. Descriptions of `EFI_FVB_ATTRIBUTES_2`

Attribute	Description
EFI_FVB2_READ_DISABLED_CAP	**TRUE** if reads from the firmware volume may be disabled.
EFI_FVB2_READ_ENABLED_CAP	**TRUE** if reads from the firmware volume may be enabled.
EFI_FVB2_READ_STATUS	**TRUE** if reads from the firmware volume are currently enabled.
EFI_FVB2_WRITE_DISABLED_CAP	**TRUE** if writes to the firmware volume may be disabled.
EFI_FVB2_WRITE_ENABLED_CAP	**TRUE** if writes to the firmware volume may be enabled.
EFI_FVB2_WRITE_STATUS	**TRUE** if writes to the firmware volume are currently enabled.
EFI_FVB2_LOCK_CAP	**TRUE** if firmware volume attributes may be locked down.

Attribute	Description
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 98.
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 98.
EFI_FVB2_READ_LOCK_CAP	**TRUE** if the firmware volume's read-status can be locked.
EFI_FVB2_READ_LOCK_STATUS	**TRUE** If the firmware volume's read-status is locked.
EFI_FVB2_WRITE_LOCK_CAP	**TRUE** if the firmware volume's write status can be locked.
EFI_FVB2_WRITE_LOCK_STATUS	**TRUE** if the firmware volume's write-status is locked.
EFI_FVB2_ALIGNMENT	The first byte of the firmware volume must be placed at an address which is an even multiple of 2^(*this field*). For example, a value of 5 in this field would mean a required alignment of 32 bytes.

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_TYPE0x01
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.

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 29 for more information on the firmware volume header. For the FFS file system, the GUID is defined below.

GUID

```
// {8C8CE578-8A3D-4f1c-9935-896185C32DD3}
#define EFI_FIRMWARE_FILE_SYSTEM2_GUID \
    { 0x8c8ce578, 0x8a3d, 0x4f1c, \
    0x99, 0x35, 0x89, 0x61, 0x85, 0xc3, 0x2d, 0xd3 }
```

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;
```

Parameters

Name

This GUID is the file name. It is used to uniquely identify the file. There may be only one instance of a file with the file name GUID of *Name* in any given firmware volume, except if the file type is **EFI_FV_FILETYPE_FFS_PAD**.

IntegrityCheck

Used to verify the integrity of the file. Type **EFI_FFS_INTEGRITY_CHECK** is defined in "Related Definitions" below.

Type

Identifies the type of file. Type **EFI_FV_FILETYPE** is defined in "Related Definitions," below. FFS-specific file types are defined in **EFI_FV_FILETYPE_FFS_PAD**.

Attributes

Declares various file attribute bits. Type **EFI_FFS_FILE_ATTRIBUTES** is defined in "Related Definitions" below.

Size

The length of the file in bytes, including the FFS header. The length of the file data is either **(***Size* **– sizeof(EFI_FFS_FILE_HEADER))**. This calculation means a zero-length file has a *Size* of 24 bytes, which is **sizeof(EFI_FFS_FILE_HEADER)**.

Size is *not* required to be a multiple of 8 bytes. Given a file *F*, the next file header is located at the next 8-byte aligned firmware volume offset following the last byte of the file *F*.

State

Used to track the state of the file throughout the life of the file from creation to deletion. Type **EFI_FFS_FILE_STATE** is defined in "Related Definitions" below. See "File Integrity and State " on page 22 for an explanation of how these bits are used.

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 22 for more details.

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 22 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_FIXED                0x04
#define FFS_ATTRIB_DATA_ALIGNMENT       0x38
#define FFS_ATTRIB_CHECKSUM             0x40
```

Figure 7 depicts the bit allocation of the *Attributes* field in an FFS file's header.

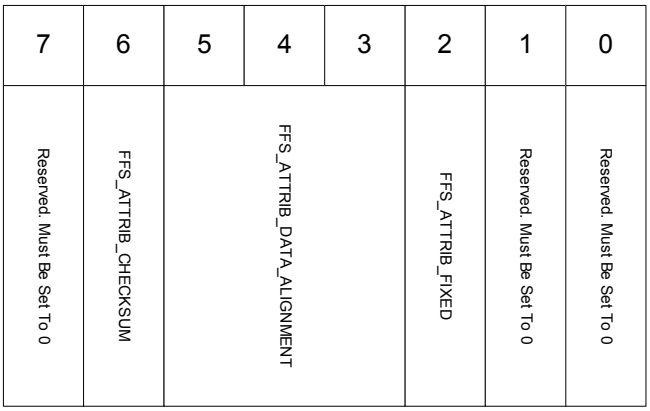

Figure 7. Bit Allocation of FFS *Attributes*

Table 4 provides descriptions of the fields in the above definition.

Table 4. Bit Allocation Definitions

Value	Definition
FFS_ATTRIB_FIXED	Indicates that the file may not be moved from its present location.
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 10 for more information regarding pad files.
FFS_ATTRIB_CHECKSUM	Determines the interpretation of *IntegrityCheck.Checksum.File*. See the *IntegrityCheck* definition above for specific usage.

Table 5 maps all FFS-supported alignments to **FFS_ATTRIB_DATA_ALIGNMENT** values and firmware volume interface alignment values.

Table 5. Supported FFS Alignments

Required Alignment (bytes)	Alignment Value in FFS *Attributes* Field	Alignment Value in Firmware Volume Interfaces
1	0	0
2	0	1
4	0	2
8	0	3
16	1	4
128	2	7
512	3	9
1KB	4	10
4KB	5	12
32KB	6	15
64KB	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 22 for an explanation of how these bits are used. Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** on page 29.

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;
```

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.

Description

The type **EFI_COMMON_SECTION_HEADER** defines the common header for all the section types.

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;
```

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_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;
```

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.

Related Definitions

```
//*************************************************************
// CompressionType values
//*************************************************************
#define EFI_NOT_COMPRESSED             0x00
#define EFI_STANDARD_COMPRESSION       0x01
```

Table 6 describes the fields in the above definition.

Table 6. 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;
```

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_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;
```

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_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;
```

Description

A *firmware volume image section* is a leaf section that contains a PI Firmware Volume Image.

EFI_SECTION_FREEFORM_SUBTYPE_GUID

Summary

A leaf section type that contains a single **EFI_GUID**.

Prototype

```
typedef struct {
  EFI_COMMON_SECTION_HEADER CommonHeader;
  EFI_GUID                  SubTypeGuid;
} EFI_FREEFORM_SUBTYPE_GUID_SECTION;
```

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**. 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 10 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;
```

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.

Related Definitions

```
//*******************************************************
// Bit values for GuidedSectionHeader.Attributes
//*******************************************************
#define EFI_GUIDED_SECTION_PROCESSING_REQUIRED     0x01
#define EFI_GUIDED_SECTION_AUTH_STATUS_VALID       0x02
```

Table 7 describes the fields in the above definition.

Table 7. Descriptions of Fields for *GuidedSectionHeader.Attributes*

Field	Description
EFI_GUIDED_SECTION_ PROCESSING_REQUIRED	Set to 1 if the section requires processing to obtain meaningful data from the section contents. Processing would be required, for example, if the section contents were encrypted or compressed. If the **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** bit is cleared to zero, it is possible to retrieve the section's contents without processing in the absence of an associated instance of the **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** (DXE) or **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI** (PEI).. In this case, the beginning of the encapsulated section stream is indicated by the value of *DataOffset*.
EFI_GUIDED_SECTION_AUTH_ STATUS_VALID	Set to 1 if the section contains authentication data that is reported through the *AuthenticationStatus* parameter returned from the **GUIDed Section Extraction Protocol**. If the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit is clear, the *AuthenticationStatus* parameter is not used.

All other bits are reserved and must be set to zero. Together, the **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** and **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bits provide the necessary data to set the proper bits of the *AuthenticationStatus* output parameter in the event that no **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** is available and the data is still returned.

EFI_SECTION_PE32

Summary

A leaf section type that contains a complete PE32+ image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_PE32_SECTION;
```

Description

The *PE32+ image section* is a leaf section that contains a complete PE32+ image. Normal UEFI executables are stored within PE32+ images.

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;
```

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_SECTION_PIC

Summary

A leaf section type that contains a position-independent-code (PIC) image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_PIC_SECTION;
```

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_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;
```

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_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;
```

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_SECTION_TE

Summary

A leaf section that contains a Terse Executable (TE) image.

Prototype

```
typedef EFI_COMMON_SECTION_HEADER EFI_TE_SECTION;
```

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_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;
```

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_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;
```

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.

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_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_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 all sections of a specific type.

EFI_PEI_FIRMWARE_VOLUME_PPI.ProcessVolume()

Summary

Process a firmware volume and create a volume handle.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_PROCESS_FV) (
IN  CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
IN  VOID                              *Buffer,
IN  UINTN                             BufferSize,
OUT EFI_PEI_FV_HANDLE                 *FvHandle
);
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

Buffer

Points to the start of the buffer.

BufferSize

Size of the buffer.

FvHandle

Points to the returned firmware volume handle. The firmware volume handle must be unique within the system. The type **EFI_PEI_FV_HANDLE** is defined in the PEI Services **FfsFindNextVolume()**.

Description

Create a volume handle from the information in the buffer. For memory-mapped firmware volumes, *Buffer* and *BufferSize* refer to the start of the firmware volume and the firmware volume size. For non memory-mapped firmware volumes, this points to a buffer which contains the necessary information for creating the firmware volume handle. Normally, these values are derived from the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI**.

Status Codes Returned

EFI_SUCCESS	Firmware volume handle created.
EFI_VOLUME_CORRUPTED	Volume was corrupt.

EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByType()

Summary

Finds the next file of the specified type.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_FILE_TYPE) (
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN EFI_FV_FILETYPE                   SearchType,
  IN EFI_PEI_FV_HANDLE                 FvHandle,
  IN OUT EFI_PEI_FILE_HANDLE           *FileHandle
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

SearchType

A filter to find only files of this type. Type **EFI_FV_FILETYPE_ALL** causes no filtering to be done.

FvHandle

Handle of firmware volume in which to search.

FileHandle

Points to the current handle from which to begin searching or NULL to start at the beginning of the firmware volume. Updated upon return to reflect the file found.

Description

This service enables PEI modules to discover additional firmware files. The *FileHandle* must be unique within the system.

Status Codes Returned

EFI_SUCCESS	The file was found.
EFI_NOT_FOUND	The file was not found. *FileHandle* contains NULL.

EFI_PEI_FIRMWARE_VOLUME_PPI.FindFileByName()

Summary

Find a file within a volume by its name.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_FILE_NAME) (
  IN  CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN  CONST EFI_GUID                    *FileName,
  IN  EFI_PEI_FV_HANDLE                 *FvHandle,
  OUT EFI_PEI_FILE_HANDLE               *FileHandle
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

FileName

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

FvHandle

Upon entry, the pointer to the firmware volume to search or **NULL** if all firmware volumes should be searched. Upon exit, the actual firmware volume in which the file was found.

FileHandle

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

Description

This service searches for files with a specific name, within either the specified firmware volume or all firmware volumes. The behavior of files with file types **EFI_FV_FILETYPE_FFS_MIN** and **EFI_FV_FILETYPE_FFS_MAX** depends on the firmware file system. For more information on the specific behavior for the standard PI firmware file system, see section 1.1.4.1.6 of the PI Specification, Volume 3.

Status Codes Returned

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

EFI_PEI_FIRMWARE_VOLUME_PPI.GetFileInfo()

Summary

Returns information about a specific file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_GET_FILE_INFO) (
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN EFI_PEI_FILE_HANDLE               FileHandle,
  OUT EFI_FV_FILE_INFO                 *FileInfo
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

FileHandle

Handle of the file.

FileInfo

Upon exit, points to the file's information.

Description

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

Status Codes Returned

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

EFI_PEI_FIRMWARE_VOLUME_PPI.GetVolumeInfo()

Summary

Return information about the firmware volume.

Prototypes

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_GET_INFO)(
  IN CONST EFI_PEI_FIRMWARE_VOLUME_PPI   *This,
  IN EFI_PEI_FV_HANDLE                   FvHandle,
  OUT EFI_FV_INFO                        *VolumeInfo
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

FvHandle

Handle to the firmware handle.

VolumeInfo

Points to the returned firmware volume information.

Description

This function returns information about the firmware volume.

Status Codes Returned

EFI_SUCCESS	Information returned successfully.
EFI_INVALID_PARAMETER	*FvHandle* does not indicate a valid firmware volume or *VolumeInfo* is **NULL**

EFI_PEI_FIRMWARE_VOLUME_PPI.FindSectionByType()

Summary

Find the next matching section in the firmware file.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_FV_FIND_SECTION) (
  IN  CONST EFI_PEI_FIRMWARE_VOLUME_PPI *This,
  IN  EFI_SECTION_TYPE                  SearchType,
  IN  EFI_PEI_FILE_HANDLE               FileHandle,
  OUT VOID                              **SectionData
  );
```

Parameters

This

Points to this instance of the **EFI_PEI_FIRMWARE_VOLUME_PPI**.

SearchType

A filter to find only sections of this type.

FileHandle

Handle of firmware file in which to search.

SectionData

Updated upon return to point to the section found.

Description

This service enables PEI modules to discover sections of a given type within a valid file.

Status Codes Returns

EFI_SUCCESS	Section was found.
EFI_NOT_FOUND	Section of the specified type was not found. *SectionData* contains **NULL**.

EFI_PEI_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 72

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. 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), 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.

Any behavior PEIM which requires to be executed from code permanent memory should include wait for **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI** and **EFI_PEI_LOAD_FILE_PPI** to be installed.

Status Codes Returned

EFI_SUCCESS	The image was loaded successfully.
EFI_OUT_OF_RESOURCES	There was not enough memory.
EFI_LOAD_ERROR	There was no supported image in the file
EFI_INVALID_PARAMETER	*FileHandle* was not a valid firmware file handle.
EFI_INVALID_PARAMETER	*EntryPoint* was NULL.
EFI_UNSUPPORTED	An image requires relocations or is not memory mapped.

EFI_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.

Protocol Interface Structure

```
typedef struct {
  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

```
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 8.

Table 8. *AuthenticationStatus* **Bit Definitions**

Bit	Definition
xx00	Image was not signed.
xxx1	Platform security policy override. Assumes same meaning as 0010 (the image was signed, the signature was tested, and the signature passed authentication test).
0010	Image was signed, the signature was tested, and the signature passed authentication test.
0110	Image was signed and the signature was not tested. This can occur if there is no GUIDed Section Extraction Protocol available to process a GUID-defined section, but it was still possible to retrieve the data from the GUID-defined section directly.
1010	Image was signed, the signature was tested, and the signature failed the authentication test.
1110	To generate this code, there must be at least two layers of GUIDed encapsulations. In one layer, the *AuthenticationStatus* was returned as 0110; in another layer, it was returned as 1010. When these two results are OR-ed together, the aggregate result is 1110.

Status Codes Returned

EFI_SUCCESS	The *InputSection* was successfully processed and the section contents were returned.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The GUID in *InputSection* does not match this instance of the GUIDed Section Extraction PPI.

EFI_PEI_DECOMPRESS_PPI

Summary

Provides decompression services to the PEI Foundatoin.

GUID

```
#define EFI_PEI_DECOMPRESS_PPI_GUID \
{ 0x1a36e4e7, 0xfab6, 0x476a, \
{ 0x8e, 0x75, 0x69, 0x5a, 0x5, 0x76, 0xfd, 0xd7 } }
```

Prototype

```
struct _EFI_PEI_DECOMPRESS_PPI {
  EFI_PEI_DECOMPRESS_DECOMPRESS   Decompress;
} EFI_PEI_DECOMPRESS_PPI;
```

Members

Decompress

Decompress a single compression section in a firmware file. See **Decompress()** for more information.

Description

This PPI's single member function decompresses a compression encapsulated section. It is used by the PEI Foundation to process sectioned files. Prior to the installation of this PPI, compression sections will be ignored.

EFI_PEI_DECOMPRESS_PPI.Decompress()

Summary

Decompress a single section.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DECOMPRESS_DECOMPRESS)(
  IN  CONST EFI_PEI_DECOMPRESS_PPI       *This,
  IN  CONST EFI_COMPRESSION_SECTION      *InputSection,
  OUT VOID                               **OutputBuffer,
  OUT UINTN                              *OutputSize
);
```

Parameters

This

Points to this instance of the **EFI_PEI_DECOMPRESS_PEI PPI**.

InputSection

Points to the compressed section.

OutputBuffer

Holds the returned pointer to the decompressed sections.

OutputSize

Holds the returned size of the decompress section streams.

Description

Decompresses the data in a compressed section and returns it as a series of standard PI Firmware File Sections. The required memory is allocated from permanent memory.

Status Codes Returned

EFI_SUCCESS	The section was decompressed successfully. *OutputBuffer* contains the resulting data and *OutputSize* contains the resulting size.
EFI_OUT_OF_RESOURCES	Unable to allocate sufficient memory to hold the decompressed data.
EFI_UNSUPPORTED	The compression type specified in the compression header is unsupported.

3.4 DXE

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_FV_GET_ATTRIBUTES         GetVolumeAttributes;
  EFI_FV_SET_ATTRIBUTES         SetVolumeAttributes;
  EFI_FV_READ_FILE              ReadFile;
  EFI_FV_READ_SECTION           ReadSection;
  EFI_FV_WRITE_FILE             WriteFile;
  EFI_FV_GET_NEXT_FILE          GetNextFile;
  UINT32                        KeySize;
  EFI_HANDLE                    ParentHandle;
  EFI_FV_GET_INFO               GetInfo;
  EFI_FV_SET_INFO               SetInfo;
} EFI_FIRMWARE_VOLUME2_PROTOCOL;
```

Parameters

GetVolumeAttributes

Retrieves volume capabilities and current settings. See the **GetVolumeAttributes()** function description.

SetVolumeAttributes

Modifies the current settings of the firmware volume. See the **SetVolumeAttributes()** function description.

ReadFile

Reads an entire file from the firmware volume. See the **ReadFile()** function description.

ReadSection

Reads a single section from a file into a buffer. See the **ReadSection()** function description.

WriteFile

Writes an entire file into the firmware volume. See the **WriteFile()** function description.

GetNextFile

Provides service to allow searching the firmware volume. See the **GetNextFile()** function description.

KeySize

Data field that indicates the size in bytes of the *Key* input buffer for the **GetNextFile()** API.

ParentHandle

Handle of the parent firmware volume. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

GetInfo

Gets the requested file or volume information. See the **GetInfo()** function description.

SetInfo

Sets the requested file information. See the **SetInfo()** function description.

Description

The Firmware Volume Protocol contains the file-level abstraction to the firmware volume as well as some firmware volume attribute reporting and configuration services. The Firmware Volume Protocol is the interface used by all parts of DXE that are not directly involved with managing the firmware volume itself. This abstraction allows many varied types of firmware volume implementations. A firmware volume may be a flash device or it may be a file in the UEFI system partition, for example. This level of firmware volume implementation detail is not visible to the consumers of the Firmware Volume Protocol.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_GET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  OUT EFI_FV_ATTRIBUTES                   *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

FvAttributes

Pointer to an **EFI_FV_ATTRIBUTES** in which the attributes and current settings are returned. Type **EFI_FV_ATTRIBUTES** is defined in "Related Definitions" below.

Description

Because of constraints imposed by the underlying firmware storage, an instance of the Firmware Volume Protocol may not be to able to support all possible variations of this architecture. These constraints and the current state of the firmware volume are exposed to the caller using the **GetVolumeAttributes()** function.

GetVolumeAttributes() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **GetVolumeAttributes()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Related Definitions

```
//*******************************************************
// EFI_FV_ATTRIBUTES
//*******************************************************
typedef UINT64 EFI_FV_ATTRIBUTES;

//*******************************************************
// EFI_FV_ATTRIBUTES bit definitions
//*******************************************************

// EFI_FV_ATTRIBUTES bit semantics
#define EFI_FV2_READ_DISABLE_CAP      0x0000000000000001
#define EFI_FV2_READ_ENABLE_CAP       0x0000000000000002
#define EFI_FV2_READ_STATUS           0x0000000000000004
```

```
#define EFI_FV2_WRITE_DISABLE_CAP      0x0000000000000008
#define EFI_FV2_WRITE_ENABLE_CAP       0x0000000000000010
#define EFI_FV2_WRITE_STATUS           0x0000000000000020

#define EFI_FV2_LOCK_CAP               0x0000000000000040
#define EFI_FV2_LOCK_STATUS            0x0000000000000080
#define EFI_FV2_WRITE_POLICY_RELIABLE  0x0000000000000100

#define EFI_FV2_READ_LOCK_CAP          0x0000000000001000
#define EFI_FV2_READ_LOCK_STATUS       0x0000000000002000
#define EFI_FV2_WRITE_LOCK_CAP         0x0000000000004000
#define EFI_FV2_WRITE_LOCK_STATUS      0x0000000000008000
#define EFI_FV2_ALIGNMENT              0x00000000001F0000

#define EFI_FV2_ALIGNMENT_1            0x0000000000000000
#define EFI_FV2_ALIGNMENT_2            0x0000000000010000
#define EFI_FV2_ALIGNMENT_4            0x0000000000020000
#define EFI_FV2_ALIGNMENT_8            0x0000000000030000
#define EFI_FV2_ALIGNMENT_16           0x0000000000040000
#define EFI_FV2_ALIGNMENT_32           0x0000000000050000
#define EFI_FV2_ALIGNMENT_64           0x0000000000060000
#define EFI_FV2_ALIGNMENT_128          0x0000000000070000
#define EFI_FV2_ALIGNMENT_256          0x0000000000080000
#define EFI_FV2_ALIGNMENT_512          0x0000000000090000
#define EFI_FV2_ALIGNMENT_1K           0x00000000000A0000
#define EFI_FV2_ALIGNMENT_2K           0x00000000000B0000
#define EFI_FV2_ALIGNMENT_4K           0x00000000000C0000
#define EFI_FV2_ALIGNMENT_8K           0x00000000000D0000
#define EFI_FV2_ALIGNMENT_16K          0x00000000000E0000
#define EFI_FV2_ALIGNMENT_32K          0x00000000000F0000
#define EFI_FV2_ALIGNMENT_64K          0x0000000000100000
#define EFI_FV2_ALIGNMENT_128K         0x0000000000110000
#define EFI_FV2_ALIGNMENT_256K         0x0000000000120000
#define EFI_FV2_ALIGNMENT_512K         0x0000000000130000
#define EFI_FV2_ALIGNMENT_1M           0x0000000000140000
#define EFI_FV2_ALIGNMENT_2M           0x0000000000150000
#define EFI_FV2_ALIGNMENT_4M           0x0000000000160000
#define EFI_FV2_ALIGNMENT_8M           0x0000000000170000
#define EFI_FV2_ALIGNMENT_16M          0x0000000000180000
#define EFI_FV2_ALIGNMENT_32M          0x0000000000190000
#define EFI_FV2_ALIGNMENT_64M          0x00000000001A0000
#define EFI_FV2_ALIGNMENT_128M         0x00000000001B0000
#define EFI_FV2_ALIGNMENT_256M         0x00000000001C0000
#define EFI_FV2_ALIGNMENT_512M         0x00000000001D0000
#define EFI_FV2_ALIGNMENT_1G           0x00000000001E0000
```

```
#define EFI_FV2_ALIGNMENT_2G          0x00000000001F0000
```

Table 9 describes the fields in the above definition.

Table 9. Descriptions of Fields for `EFI_FV_ATTRIBUTES`

Field	Description
EFI_FV_READ_DISABLED_CAP	Set to 1 if it is possible to disable reads from the firmware volume.
EFI_FV_READ_ENABLED_CAP	Set to 1 if it is possible to enable reads from the firmware volume.
EFI_FV_READ_STATUS	Indicates the current read state of the firmware volume. Set to 1 if reads from the firmware volume are enabled.
EFI_FV_WRITE_DISABLED_CAP	Set to 1 if it is possible to disable writes to the firmware volume.
EFI_FV_WRITE_ENABLED_CAP	Set to 1 if it is possible to enable writes to the firmware volume.
EFI_FV_WRITE_STATUS	Indicates the current state of the firmware volume. Set to 1 if writes to the firmware volume are enabled.
EFI_FV_LOCK_CAP	Set to 1 if it is possible to lock firmware volume read/write attributes.
EFI_FV_LOCK_STATUS	Set to 1 if firmware volume attributes are locked down.
EFI_FV_WRITE_POLICY_RELIABLE	Set to 1 if the firmware volume supports “reliable” writes..
EFI_FV_READ_LOCK_CAP	Set to 1 if it is possible to lock the read status for the firmware volume.
EFI_FV_READ_LOCK_STATUS	Indicates the current read lock state of the firmware volume. Set to 1 if the read lock is currently enabled.
EFI_FV_WRITE_LOCK_CAP	Set to 1 if it is possible to lock the write status for the firmware volume.
EFI_FV_WRITE_LOCK_STATUS	Indicates the current write lock state of the firmware volume. Set to 1 if the write lock is currently enabled.
EFI_FV_ALIGNMENT	The first byte of the firmware volume must be at an address which is an even multiple of the alignment specified.

All other bits are reserved and are cleared to zero.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME2_PROTOCOL.SetVolumeAttributes()

Summary

Modifies the current settings of the firmware volume according to the input parameter.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_SET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  IN OUT EFI_FV_ATTRIBUTES          *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

FvAttributes

On input, *FvAttributes* is a pointer to an **EFI_FV_ATTRIBUTES** containing the desired firmware volume settings. On successful return, it contains the new settings of the firmware volume. On unsuccessful return, *FvAttributes* is not modified and the firmware volume settings are not changed. Type **EFI_FV_ATTRIBUTES** is defined in **GetVolumeAttributes()**.

Description

The **SetVolumeAttributes()** function is used to set configurable firmware volume attributes. Only **EFI_FV_READ_STATUS**, **EFI_FV_WRITE_STATUS**, and **EFI_FV_LOCK_STATUS** may be modified, and then only in accordance with the declared capabilities. All other bits of **FvAttributes* are ignored on input. On successful return, all bits of **FvAttributes* are valid and it contains the completed **EFI_FV_ATTRIBUTES** for the volume.

To modify an attribute, the corresponding status bit in the **EFI_FV_ATTRIBUTES** is set to the desired value on input. The **EFI_FV_LOCK_STATUS** bit does not affect the ability to read or write the firmware volume. Rather, once the **EFI_FV_LOCK_STATUS** bit is set, it prevents further modification to all the attribute bits.

SetVolumeAttributes() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **SetVolumeAttributes()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The requested firmware volume attributes were set and the resulting **EFI_FV_ATTRIBUTES** is returned in *FvAttributes*.

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

EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadFile()

Summary

Retrieves a file and/or file information from the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_READ_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL   *This,
  IN CONST EFI_GUID                        *NameGuid,
  IN OUT VOID                              **Buffer,
  IN OUT UINTN                             *BufferSize,
  OUT EFI_FV_FILETYPE                      *FoundType,
  OUT EFI_FV_FILE_ATTRIBUTES               *FileAttributes,
  OUT UINT32                               *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NameGuid

Pointer to an **EFI_GUID**, which is the file name. All firmware file names are **EFI_GUID**s. A single firmware volume must not have two valid files with the same file name **EFI_GUID**. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Buffer

Pointer to a pointer to a buffer in which the file contents are returned, not including the file header. See "Description" below for more details on the use of the *Buffer* parameter.

BufferSize

Pointer to a caller-allocated **UINTN.** It indicates the size of the memory represented by **Buffer*. See "Description" below for more details on the use of the *BufferSize* parameter.

FoundType

Pointer to a caller-allocated **EFI_FV_FILETYPE**. See "Firmware File Types" on page 7 for **EFI_FV_FILETYPE** related definitions.

FileAttributes

Pointer to a caller-allocated **EFI_FV_FILE_ATTRIBUTES**. Type **EFI_FV_FILE_ATTRIBUTES** is defined in "Related Definitions" below.

AuthenticationStatus

Pointer to a caller-allocated **UINT32** in which the authentication status is returned. See "Related Definitions" in **EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()** 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
```

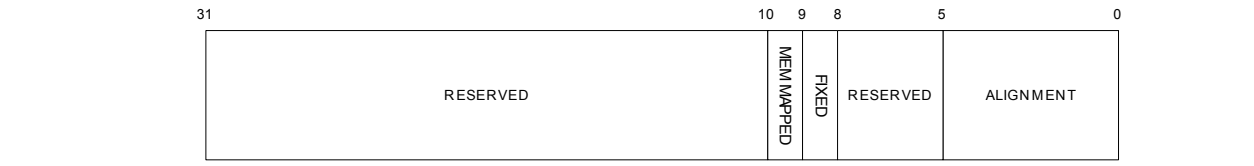

Figure 8. EFI_FV_FILE_ATTRIBUTES fields

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 10. All other values are reserved.

Table 10. Supported Alignments for EFI_FV_FILE_ATTRIB_ALIGNMENT

Required Alignment (bytes)	Alignment Value in *Attributes* Field
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1KB	10
2KB	11
4KB	12
8KB	13
16KB	14

Required Alignment (bytes)	Alignment Value in *Attributes* Field
32KB	15
64KB	16
128 KB	17
256 KB	18
512 KB	19
1 MB	20
2 MB	21
4 MB	22
8 MB	23
16 MB	24
32 MB	25
64 MB	26
128 MB	27
256 MB	28
512 MB	29
1 GB	30
2 GB	31

The **EFI_FV_FILE_ATTRIB_FIXED** attribute indicates that the file has a fixed location and should not be moved (1) or may be moved to any address consistent with the alignment specified in **EFI_FV_FILE_ATTRIB_ALIGNMENT**.

The **EFI_FV_FILE_ATTRIB_MEMORY_MAPPED** attribute indicates that the file is memory mapped in the firmware volume and thus its contents may be accessed directly. If this is clear, then *Buffer* is invalid. This value can be derived from the **EFI_FV_ATTRIBUTES** value returned by **EFI_FIRMWARE_VOLUME2_PROTOCOL.GetVolumeAttributes()** or the PEI Service **FfsGetVolumeInfo()**.

Status Codes Returned

EFI_SUCCESS	The call completed successfully.
EFI_WARN_BUFFER_TOO_SMALL	The buffer is too small to contain the requested output. The buffer is filled and the output is truncated.
EFI_OUT_OF_RESOURCES	An allocation failure occurred.
EFI_NOT_FOUND	*Name* was not found in the firmware volume.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.

EFI_FIRMWARE_VOLUME2_PROTOCOL.ReadSection()

Summary

Locates the requested section within a file and returns it in a buffer.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_READ_SECTION) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL   *This,
  IN CONST EFI_GUID                        *NameGuid,
  IN EFI_SECTION_TYPE                      SectionType,
  IN UINTN                                 SectionInstance,
  IN OUT VOID                              **Buffer,
  IN OUT UINTN                             *BufferSize,
  OUT UINT32                               *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NameGuid

Pointer to an **EFI_GUID**, which indicates the file name from which the requested section will be read. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

SectionType

Indicates the section type to return. *SectionType* in conjunction with *SectionInstance* indicates which section to return. Type **EFI_SECTION_TYPE** is defined in **EFI_COMMON_SECTION_HEADER**.

SectionInstance

Indicates which instance of sections with a type of *SectionType* to return. *SectionType* in conjunction with *SectionInstance* indicates which section to return. *SectionInstance* is zero based.

Buffer

Pointer to a pointer to a buffer in which the section contents are returned, not including the section header. See “Description” below for more details on the usage of the *Buffer* parameter.

BufferSize

Pointer to a caller-allocated **UINTN.** It indicates the size of the memory represented by **Buffer*. See “Description” below for more details on the usage of the *BufferSize* parameter.

AuthenticationStatus

Pointer to a caller-allocated **UINT32** in which the authentication status is returned. See **EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()** for more information.

Description

ReadSection() is used to retrieve a specific section from a file within a firmware volume. The section returned is determined using a depth-first, left-to-right search algorithm through all sections found in the specified file. See "Firmware File Sections" on page 12 for more details about sections.

The output buffer is specified by a double indirection of the *Buffer* parameter. The input value of **Buffer* is used to determine if the output buffer is caller allocated or is dynamically allocated by **ReadSection()**.

If the input value of **Buffer!=NULL*, it indicates that the output buffer is caller allocated. In this case, the input value of **BufferSize* indicates the size of the caller-allocated output buffer. If the output buffer is not large enough to contain the entire requested output, it is filled up to the point that the output buffer is exhausted and **EFI_WARN_BUFFER_TOO_SMALL** is returned, and then **BufferSize* is returned with the size that is required to successfully complete the read. All other output parameters are returned with valid values.

If the input value of **Buffer==NULL*, it indicates the output buffer is to be allocated by **ReadSection()**. In this case, **ReadSection()** will allocate an appropriately sized buffer from boot services pool memory, which will be returned in **Buffer*. The size of the new buffer is returned in **BufferSize* and all other output parameters are returned with valid values.

ReadSection() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **ReadSection()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The call completed successfully.
EFI_WARN_BUFFER_ TOO_SMALL	The caller-allocated buffer is too small to contain the requested output. The buffer is filled and the output is truncated.
EFI_OUT_OF_RESOURCES	An allocation failure occurred.
EFI_NOT_FOUND	The requested file was not found in the firmware volume.
EFI_NOT_FOUND	The requested section was not found in the specified file.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.
EFI_PROTOCOL_ERROR	The requested section was not found, but the file could not be fully parsed because a required **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** was not found. It is possible the requested section exists within the file and could be successfully extracted once the required **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** is published.

EFI_FIRMWARE_VOLUME2_PROTOCOL.WriteFile()

Summary

Writes one or more files to the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_WRITE_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN UINT32                              NumberOfFiles,
  IN EFI_FV_WRITE_POLICY                 WritePolicy,
  IN EFI_FV_WRITE_FILE_DATA              *FileData
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

NumberOfFiles

Indicates the number of elements in the array pointed to by *FileData*.

WritePolicy

Indicates the level of reliability for the write in the event of a power failure or other system failure during the write operation. Type **EFI_FV_WRITE_POLICY** is defined in "Related Definitions" below.

FileData

Pointer to an array of **EFI_FV_WRITE_FILE_DATA.** Each element of *FileData[]* represents a file to be written. Type **EFI_FV_WRITE_FILE_DATA** is defined in "Related Definitions" below.

Description

WriteFile() is used to write one or more files to a firmware volume. Each file to be written is described by an **EFI_FV_WRITE_FILE_DATA** structure.

The caller must ensure that any required alignment for all files listed in the *FileData* array is compatible with the firmware volume. Firmware volume capabilities can be determined using the **GetVolumeAttributes()** call.

Similarly, if the *WritePolicy* is set to **EFI_FV_RELIABLE_WRITE**, the caller must check the firmware volume capabilities to ensure **EFI_FV_RELIABLE_WRITE** is supported by the firmware volume. **EFI_FV_UNRELIABLE_WRITE** must always be supported.

Writing a file with a size of zero (*FileData[n].BufferSize == 0*) deletes the file from the firmware volume if it exists. Deleting a file must be done one at a time. Deleting a file as part of a multiple file write is not allowed.

WriteFile() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **WriteFile()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Related Definitions

```
//*******************************************************
// EFI_FV_WRITE_POLICY
//*******************************************************
typedef UINT32 EFI_FV_WRITE_POLICY

#define EFI_FV_UNRELIABLE_WRITE     0x00000000
#define EFI_FV_RELIABLE_WRITE       0x00000001
```

All other values of **EFI_FV_WRITE_POLICY** are reserved. Table 11 describes the fields in the above definition.

Table 11. Description of fields for EFI_FV_WRITE_POLICY

Field	Description
EFI_FV_UNRELIABLE_WRITE	This value in the *WritePolicy* parameter indicates that there is no required reliability if a power failure or other system failure occurs during a write operation. Updates may leave a combination of old and new files. Data loss, including complete loss of all files involved, is also permissible. In essence, no guarantees are made regarding what files will be present following a system failure during a write with a *WritePolicy* of **EFI_FV_UNRELIABLE_WRITE**. The advantage of this mode is that it can be implemented to use much less space in the storage media. Space-constrained firmware volumes may be able to support writes where it would be otherwise impossible.
EFI_FV_RELIABLE_WRITE	This value in the *WritePolicy* parameter indicates that, on the next initialization of the firmware volume following a power failure or other system failure during a write, all files listed in the *FileData* array are completely written and are valid, or none is written and the state of the firmware volume is the same as it was before the write operation was attempted.

```
//*******************************************************
// EFI_FV_WRITE_FILE_DATA
//*******************************************************

typedef struct {
  EFI_GUID                  *NameGuid,
  EFI_FV_FILETYPE           Type,
  EFI_FV_FILE_ATTRIBUTES    FileAttributes
  VOID                      *Buffer,
  UINT32                    BufferSize
} EFI_FV_WRITE_FILE_DATA;
```

NameGuid

Pointer to a GUID, which is the file name to be written. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Type

Indicates the type of file to be written. Type **EFI_FV_FILETYPE** is defined in "Related Definitions" of **EFI_FFS_FILE_HEADER** on page 37.

FileAttributes

Indicates the attributes for the file to be written. Type **EFI_FV_FILE_ATTRIBUTES** is defined in **ReadFile()**.

Buffer

Pointer to a buffer containing the file to be written.

BufferSize

Indicates the size of the file image contained in *Buffer*.

Status Codes Returned

EFI_SUCCESS	The write completed successfully.
EFI_OUT_OF_RESOURCES	The firmware volume does not have enough free space to storefile(s).
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_WRITE_PROTECTED	The firmware volume is configured to disallow writes.
EFI_NOT_FOUND	A delete was requested, but the requested file was not found in the firmware volume.
EFI_INVALID_PARAMETER	A delete was requested with a multiple file write.
EFI_INVALID_PARAMETER	An unsupported *WritePolicy* was requested.
EFI_INVALID_PARAMETER	An unknown file type was specifiedspecified or the specified file type is not supported by the firmware file system.
EFI_INVALID_PARAMETER	A file system specific error has occurred.

Other than **EFI_DEVICE_ERROR**, all error codes imply the firmware volume has not been modified. In the case of **EFI_DEVICE_ERROR**, the firmware volume may have been corrupted and appropriate repair steps must be taken.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetNextFile()

Summary

Retrieves information about the next file in the firmware volume store that matches the search criteria.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FV_GET_NEXT_FILE) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL  *This,
  IN OUT VOID                             *Key,
  IN OUT EFI_FV_FILETYPE                  *FileType,
  OUT EFI_GUID                            *NameGuid,
  OUT EFI_FV_FILE_ATTRIBUTES              *Attributes,
  OUT UINTN                               *Size
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance.

Key

Pointer to a caller-allocated buffer that contains implementation-specific data that is used to track where to begin the search for the next file. The size of the buffer must be at least *This->KeySize* bytes long. To re-initialize the search and begin from the beginning of the firmware volume, the entire buffer must be cleared to zero. Other than clearing the buffer to initiate a new search, the caller must not modify the data in the buffer between calls to **GetNextFile()**.

FileType

Pointer to a caller-allocated **EFI_FV_FILETYPE**. The **GetNextFile()** API can filter its search for files based on the value of the **FileType* input. A **FileType* input of **EFI_FV_FILETYPE_ALL** causes **GetNextFile()** to search for files of all types. If a file is found, the file's type is returned in **FileType*. **FileType* is not modified if no file is found. See"Related Definitions" of **EFI_FFS_FILE_HEADER** on page 37.

NameGuid

Pointer to a caller-allocated **EFI_GUID**. If a matching file is found, the file's name is returned in **NameGuid*. If no matching file is found, **NameGuid* is not modified. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Attributes

Pointer to a caller-allocated **EFI_FV_FILE_ATTRIBUTES**. If a matching file is found, the file's attributes are returned in **Attributes*. If no matching file is

found, **Attributes* is not modified. Type **EFI_FV_FILE_ATTRIBUTES** is defined in **ReadFile()**.

Size

Pointer to a caller-allocated **UINTN**. If a matching file is found, the file's size is returned in **Size*. If no matching file is found, **Size* is not modified.

Description

GetNextFile() is the interface that is used to search a firmware volume for a particular file. It is called successively until the desired file is located or the function returns **EFI_NOT_FOUND**.

To filter uninteresting files from the output, the type of file to search for may be specified in **FileType*. For example, if **FileType* is **EFI_FV_FILETYPE_DRIVER**, only files of this type will be returned in the output. If **FileType* is **EFI_FV_FILETYPE_ALL**, no filtering of file types is done.The behavior of files with file types **EFI_FV_FILETYPE_FFS_MIN** and **EFI_FV_FILETYPE_FFS_MAX** depends on the firmware file system. For more information on the specific behavior for the standard PI firmware file system, see section 1.1.4.1.6 of the PI Specification, Volume 3.

The *Key* parameter is used to indicate a starting point of the search. If the buffer **Key* is completely initialized to zero, the search re-initialized and starts at the beginning. Subsequent calls to **GetNextFile()** must maintain the value of **Key* returned by the immediately previous call. The actual contents of **Key* are implementation specific and no semantic content is implied.

GetNextFile() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **GetNextFile()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	**The output parameters are filled with data obtained from the first matching file that was found.**
EFI_NOT_FOUND	No files of type *FileType* were found.
EFI_DEVICE_ERROR	A hardware error occurred when attempting to access the firmware volume.
EFI_ACCESS_DENIED	The firmware volume is configured to disallow reads.

EFI_FIRMWARE_VOLUME2_PROTOCOL.GetInfo()

Summary

Return information about a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FV_GET_INFO) (
  IN     CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN     CONST EFI_GUID                      *InformationType,
  IN OUT UINTN                               *BufferSize,
  OUT    VOID                                *Buffer
  );
```

Parameters

This

A pointer to the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance that is the file handle the requested information is for.

InformationType

The type identifier for the information being requested. Type **EFI_GUID** is defined in the UEFI 2.0 specification.

BufferSize

On input, the size of Buffer. On output, the amount of data returned in *Buffer*. In both cases, the size is measured in bytes.

Buffer

A pointer to the data buffer to return. The buffer's type is indicated by InformationType.

Description

The **GetInfo()** function returns information of type *InformationType* for the requested firmware volume. If the volume does not support the requested information type, then **EFI_UNSUPPORTED** is returned. If the buffer is not large enough to hold the requested structure, **EFI_BUFFER_TOO_SMALL** is returned and the *BufferSize* is set to the size of buffer that is required to make the request. The information types defined by this specification are required information types that all file systems must support.

Status Codes Returned

EFI_SUCCESS	The information was retrieved.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.

EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_BUFFER_TOO_SMALL	The *BufferSize* is too small to read the current directory entry. *BufferSize* has been updated with the size needed to complete the request.

EFI_FIRMWARE_VOLUME2_PROTOCOL.SetInfo()

Summary

Sets information about a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FV_SET_INFO) (
  IN CONST EFI_FIRMWARE_VOLUME2_PROTOCOL *This,
  IN CONST EFI_GUID                      *InformationType,
  IN UINTN                               BufferSize,
  IN CONST VOID                          *Buffer
);
```

Parameters

This

A pointer to the **EFI_FIRMWARE_VOLUME2_PROTOCOL** instance that is the file handle the information is for.

InformationType

The type identifier for the information being set. Type **EFI_GUID** is defined in the UEFI 2.0 specification.

BufferSize

The size, in bytes, of *Buffer*.

Buffer

A pointer to the data buffer to write. The buffer's type is indicated by *InformationType*.

Description

The **SetInfo()** function sets information of type *InformationType* on the requested firmware volume.

Status Codes Returned

EFI_SUCCESS	The information was set.
EFI_UNSUPPORTED	The *InformationType* is not known.
EFI_NO_MEDIA	The device has no medium.
EFI_DEVICE_ERROR	The device reported an error.
EFI_VOLUME_CORRUPTED	The file system structures are corrupted.
EFI_WRITE_PROTECTED	The media is read only.
EFI_VOLUME_FULL	The volume is full.

EFI_BAD_BUFFER_SIZE	*BufferSize* is smaller than the size of the type indicated by *InformationType*.

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 {
  EFI_FVB_GET_ATTRIBUTES              GetAttributes;
  EFI_FVB_SET_ATTRIBUTES              SetAttributes;
  EFI_FVB_GET_PHYSICAL_ADDRESS        GetPhysicalAddress;
  EFI_FVB_GET_BLOCK_SIZE              GetBlockSize;
  EFI_FVB_READ                        Read;
  EFI_FVB_WRITE                       Write;
  EFI_FVB_ERASE_BLOCKS                EraseBlocks;
  EFI_HANDLE                          ParentHandle;
} EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL;
```

Parameters

GetAttributes

Retrieves the current volume attributes. See the **GetAttributes()** function description.

SetAttributes

Sets the current volume attributes. See the **SetAttributes()** function description.

GetPhysicalAddress

Retrieves the memory-mapped address of the firmware volume. See the **GetPhysicalAddress()** function description.

GetBlockSize

Retrieves the size for a specific block. Also returns the number of consecutive similarly sized blocks. See the **GetBlockSize()** function description.

Read

Reads *n* bytes into a buffer from the firmware volume hardware. See the **Read()** function description.

Write

Writes *n* bytes from a buffer into the firmware volume hardware. See the **Write()** function description.

EraseBlocks

Erases specified block(s) and sets all values as indicated by the **EFI_FVB_ERASE_POLARITY** bit. See the **EraseBlocks()** function description. Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

ParentHandle

Handle of the parent firmware volume. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the UEFI 2.0 specification.

Description

The Firmware Volume Block Protocol is the low-level interface to a firmware volume. File-level access to a firmware volume should not be done using the Firmware Volume Block Protocol. Normal access to a firmware volume must use the Firmware Volume Protocol. Typically, only the file system driver that produces the Firmware Volume Protocol will bind to the Firmware Volume Block Protocol.

The Firmware Volume Block Protocol provides the following:

- Byte-level read/write functionality.
- Block-level erase functionality.
- It further exposes device-hardening features, such as may be required to protect the firmware from unwanted overwriting and/or erasure.
- It is useful to layer a file system driver on top of the Firmware Volume Block Protocol. This file system driver produces the Firmware Volume Protocol, which provides file-level access to a firmware volume. The Firmware Volume Protocol abstracts the file system that is used to format the firmware volume and the hardware device-hardening features that may be present.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_ATTRIBUTES) (
  IN  CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL    *This,
  OUT EFI_FVB_ATTRIBUTES_2                         *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Attributes

Pointer to **EFI_FVB_ATTRIBUTES_2** in which the attributes and current settings are returned. Type **EFI_FVB_ATTRIBUTES_2** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

Description

The **GetAttributes()** function retrieves the attributes and current settings of the block.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.SetAttributes()

Summary

Modifies the current settings of the firmware volume according to the input parameter.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_SET_ATTRIBUTES) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL    *This,
  IN OUT EFI_FVB_ATTRIBUTES_2                     *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Attributes

On input, *Attributes* is a pointer to **EFI_FVB_ATTRIBUTES_2** that contains the desired firmware volume settings. On successful return, it contains the new settings of the firmware volume. Type **EFI_FVB_ATTRIBUTES_2** is defined in **EFI_FIRMWARE_VOLUME_HEADER**.

Description

The **SetAttributes()** function sets configurable firmware volume attributes and returns the new settings of the firmware volume.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.
EFI_INVALID_PARAMETER	The attributes requested are in conflict with the capabilities as declared in the firmware volume header.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetPhysicalAddress ()

Summary

Retrieves the physical address of a memory-mapped firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_PHYSICAL_ADDRESS) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL   *This,
  OUT EFI_PHYSICAL_ADDRESS                       *Address
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Address

Pointer to a caller-allocated **EFI_PHYSICAL_ADDRESS** that, on successful return from **GetPhysicalAddress()**, contains the base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

Description

The **GetPhysicalAddress()** function retrieves the base address of a memory-mapped firmware volume. This function should be called only for memory-mapped firmware volumes.

Status Codes Returned

EFI_SUCCESS	The firmware volume base address is returned.
EFI_UNSUPPORTED	The firmware volume is not memory mapped.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.GetBlockSize()

Summary

Retrieves the size in bytes of a specific block within a firmware volume.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_GET_BLOCK_SIZE) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL *This,
  IN EFI_LBA                                   Lba,
  OUT UINTN                                    *BlockSize,
  OUT UINTN                                    *NumberOfBlocks
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

Indicates the block for which to return the size. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

BlockSize

Pointer to a caller-allocated **UINTN** in which the size of the block is returned.

NumberOfBlocks

Pointer to a caller-allocated **UINTN** in which the number of consecutive blocks, starting with *Lba*, is returned. All blocks in this range have a size of *BlockSize*.

Description

The **GetBlockSize()** function retrieves the size of the requested block. It also returns the number of additional blocks with the identical size. The **GetBlockSize()** function is used to retrieve the block map (see **EFI_FIRMWARE_VOLUME_HEADER**).

Status Codes Returned

EFI_SUCCESS	The firmware volume base address is returned.
EFI_INVALID_PARAMETER	The requested LBA is out of range.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Read()

Summary

Reads the specified number of bytes into a buffer from the specified block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_FVB_READ) (
  IN  CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL *This,
  IN  EFI_LBA                                   Lba,
  IN  UINTN                                     Offset,
  IN  OUT UINTN                                 *NumBytes,
  OUT UINT8                                     *Buffer,
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

The starting logical block index from which to read. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

Offset

Offset into the block at which to begin reading.

NumBytes

Pointer to a **UINTN**. At entry, **NumBytes* contains the total size of the buffer. At exit, **NumBytes* contains the total number of bytes read.

Buffer

Pointer to a caller-allocated buffer that will be used to hold the data that is read.

Description

The **Read()** function reads the requested number of bytes from the requested block and stores them in the provided buffer.

Implementations should be mindful that the firmware volume might be in the *ReadDisabled* state. If it is in this state, the **Read()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the buffer.

The **Read()** function must also prevent spanning block boundaries. If a read is requested that would span a block boundary, the read must read up to the boundary but not beyond. The output parameter *NumBytes* must be set to correctly indicate the number of bytes actually read. The caller must be aware that a read may be partially completed.

Status Codes Returned

EFI_SUCCESS	The firmware volume was read successfully and contents are in *Buffer*.
EFI_BAD_BUFFER_SIZE	Read attempted across an LBA boundary. On output, *NumBytes* contains the total number of bytes returned in *Buffer*.
EFI_ACCESS_DENIED	The firmware volume is in the *ReadDisabled* state.
EFI_DEVICE_ERROR	The block device is not functioning correctly and could not be read.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.Write()

Summary

Writes the specified number of bytes from the input buffer to the block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_WRITE) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL *This,
  IN EFI_LBA                                   Lba,
  IN UINTN                                     Offset,
  IN OUT UINTN                                 *NumBytes,
  IN UINT8                                     *Buffer
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

Lba

The starting logical block index to write to. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the UEFI 2.0 specification.

Offset

Offset into the block at which to begin writing.

NumBytes

Pointer to a **UINTN**. At entry, **NumBytes* contains the total size of the buffer. At exit, **NumBytes* contains the total number of bytes actually written.

Buffer

Pointer to a caller-allocated buffer that contains the source for the write.

Description

The **Write()** function writes the specified number of bytes from the provided buffer to the specified block and offset.

If the firmware volume is sticky write, the caller must ensure that all the bits of the specified range to write are in the **EFI_FVB_ERASE_POLARITY** state before calling the **Write()** function, or else the result will be unpredictable. This unpredictability arises because, for a sticky-write firmware volume, a write may negate a bit in the **EFI_FVB_ERASE_POLARITY** state but it cannot flip it back again. In general, before calling the **Write()** function, the caller should call the **EraseBlocks()** function first to erase the specified block to write. A block erase cycle will transition bits from the **(NOT)EFI_FVB_ERASE_POLARITY** state back to the **EFI_FVB_ERASE_POLARITY** state.

Implementations should be mindful that the firmware volume might be in the *WriteDisabled* state. If it is in this state, the **Write()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the firmware volume.

The **Write()** function must also prevent spanning block boundaries. If a write is requested that spans a block boundary, the write must store up to the boundary but not beyond. The output parameter *NumBytes* must be set to correctly indicate the number of bytes actually written. The caller must be aware that a write may be partially completed.

All writes, partial or otherwise, must be fully flushed to the hardware before the **Write()** service returns.

Status Codes Returned

EFI_SUCCESS	The firmware volume was written successfully.
EFI_BAD_BUFFER_SIZE	The write was attempted across an LBA boundary. On output, *NumBytes* contains the total number of bytes actually written.
EFI_ACCESS_DENIED	The firmware volume is in the *WriteDisabled* state.
EFI_DEVICE_ERROR	The block device is malfunctioning and could not be written.

EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL.EraseBlocks()

Summary

Erases and initializes a firmware volume block.

Prototype

```
typedef
EFI_STATUS
(EFIAPI * EFI_FVB_ERASE_BLOCKS) (
  IN CONST EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL  *This,
  …
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK2_PROTOCOL** instance.

…

The variable argument list is a list of tuples. Each tuple describes a range of LBAs to erase and consists of the following:

- An **EFI_LBA** that indicates the starting LBA
- A **UINTN** that indicates the number of blocks to erase

The list is terminated with an **EFI_LBA_LIST_TERMINATOR**. Type **EFI_LBA_LIST_TERMINATOR** is defined in "Related Definitions" below.

For example, the following indicates that two ranges of blocks (5–7 and 10–11) are to be erased:

```
EraseBlocks (This, 5, 3, 10, 2, EFI_LBA_LIST_TERMINATOR);
```

Description

The **EraseBlocks()** function erases one or more blocks as denoted by the variable argument list. The entire parameter list of blocks must be verified before erasing any blocks. If a block is requested that does not exist within the associated firmware volume (it has a larger index than the last block of the firmware volume), the **EraseBlocks()** function must return the status code **EFI_INVALID_PARAMETER** without modifying the contents of the firmware volume.

Implementations should be mindful that the firmware volume might be in the *WriteDisabled* state. If it is in this state, the **EraseBlocks()** function must return the status code **EFI_ACCESS_DENIED** without modifying the contents of the firmware volume.

All calls to **EraseBlocks()** must be fully flushed to the hardware before the **EraseBlocks()** service returns.

Related Definitions

```
//*******************************************
// EFI_LBA_LIST_TERMINATOR
//*******************************************
#define EFI_LBA_LIST_TERMINATOR 0xFFFFFFFFFFFFFFFF
```

Status Codes Returned

EFI_SUCCESS	The erase request was successfully completed.
EFI_ACCESS_DENIED	The firmware volume is in the *WriteDisabled* state.
EFI_DEVICE_ERROR	The block device is not functioning correctly and could not be written. The firmware device may have been partially erased.
EFI_INVALID_PARAMETER	One or more of the LBAs listed in the variable argument list do not exist in the firmware volume.

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_EXTRACT_GUIDED_SECTION     ExtractSection;
} EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL;
```

Parameters

ExtractSection

Takes the GUIDed section as input and produces the section stream data. See the **ExtractSection()** function description.

EFI_GUIDED_SECTION_EXTRACTION_PROCOCOL.ExtractSection()

Summary

Processes the input section and returns the data contained therein along with the authentication status.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_EXTRACT_GUIDED_SECTION)(
  IN CONST EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL *This,
  IN CONST VOID                 *InputSection,
  OUT VOID                      **OutputBuffer,
  OUT UINTN                     *OutputSize,
  OUT UINT32                    *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** instance.

InputSection

Buffer containing the input GUIDed section to be processed.

OutputBuffer

**OutputBuffer* is allocated from boot services pool memory and contains the new section stream. The caller is responsible for freeing this buffer.

OutputSize

A pointer to a caller-allocated **UINTN** in which the size of **OutputBuffer* allocation is stored. If the function returns anything other than **EFI_SUCCESS**, the value of **OutputSize* is undefined.

AuthenticationStatus

A pointer to a caller-allocated **UINT32** that indicates the authentication status of the output buffer. If the input section's *GuidedSectionHeader.Attributes* field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit as clear, **AuthenticationStatus* must return zero. Both local bits (19:16) and aggregate bits (3:0) in *AuthenticationStatus* are returned by **ExtractSection()**. These bits reflect the status of the extraction operation. The bit pattern in both regions must be the same, as the local and aggregate authentication statuses have equivalent meaning at this level. If the function returns anything other than **EFI_SUCCESS**, the value of **AuthenticationStatus* is undefined.

Description

The **ExtractSection()** function processes the input section and allocates a buffer from the pool in which it returns the section contents.

If the section being extracted contains authentication information (the section's **GuidedSectionHeader.Attributes** field has the **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit set), the values returned in *AuthenticationStatus* must reflect the results of the authentication operation.

Depending on the algorithm and size of the encapsulated data, the time that is required to do a full authentication may be prohibitively long for some classes of systems. To indicate this, use **EFI_SECURITY_POLICY_PROTOCOL_GUID**, which may be published by the security policy driver (see the *Platform Initialization Driver Execution Environment Core Interface Specification* for more details and the GUID definition). If the **EFI_SECURITY_POLICY_PROTOCOL_GUID** exists in the handle database, then, if possible, full authentication should be skipped and the section contents simply returned in the *OutputBuffer*. In this case, the **EFI_AUTH_STATUS_PLATFORM_OVERRIDE** bit *AuthenticationStatus* must be set on return. See "Related Definitions" in **EFI_PEI_GUIDED_SECTION_EXTRACTION_PPI.ExtractSection()** on page 72 for the definition of type **EFI_AUTH_STATUS_PLATFORM_OVERRIDE**.

ExtractSection() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **ExtractSection()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the UEFI 2.0 specification.

Status Codes Returned

EFI_SUCCESS	The *InputSection* was successfully processed and the section contents were returned.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The GUID in *InputSection* does not match this instance of the GUIDed Section Extraction Protocol.

4
HOB Design Discussion

4.1 Explanation of HOB Terms

Because HOBs are the key architectural mechanism that is used to hand off system information in the early preboot stages and because not all implementations of the PI Architecture will use the Pre-EFI Initialization (PEI) and Driver Execution Environment (DXE) phases, this specification refrains from using the PEI and DXE nomenclature used in other PI specifications.

Instead, this specification uses the following terms to refer to the phases that deal with HOBs:

- HOB producer phase
- HOB consumer phase

The *HOB producer phase* is the preboot phase in which HOBs and the HOB list are created. The *HOB consumer phase* is the preboot phase to which the HOB list is passed and then consumed.

If the PI Architecture implementation incorporates the PEI and DXE, the HOB producer phase is the PEI phase and the HOB consumer phase is the DXE phase. The producer and consumer can change, however, depending on the implementation.

The following table translates the terminology used in this specification with that used in other PI specifications.

Table 12. 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 and the semantics for creating them, see "HOB Code Definitions" on page 117.

4.3 Example HOB Producer Phase Memory Map and Usage

Figure 9 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.

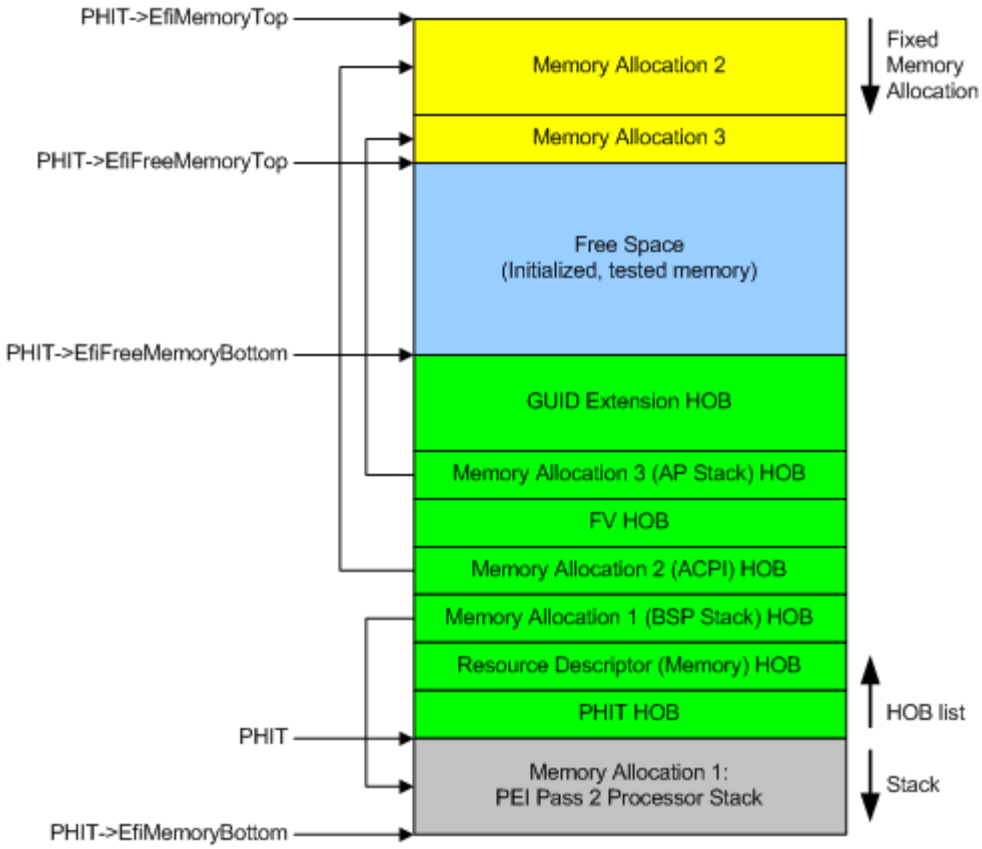

Figure 9. Example HOB Producer Phase Memory Map and Usage

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 *NewHobSize* <= PHIT->*EfiFreeMemoryTop* - PHIT->*EfiFreeMemoryBottom*).
3. Construct the HOB at PHIT->*EfiFreeMemoryBottom*.
4. Set PHIT->*EfiFreeMemoryBottom* = PHIT->*EfiFreeMemoryBottom* + *NewHobSize*.

5
HOB Code Definitions

5.1 HOB Introduction

This section contains the basic definitions of various HOBs. All HOBs consist of a generic header, **EFI_HOB_GENERIC_HEADER**, that specifies the type and length of the HOB. Each HOB has additional data beyond the generic header, according to the HOB type. The following data types and structures are defined in this section:

- **EFI_HOB_GENERIC_HEADER**
- **EFI_HOB_HANDOFF_INFO_TABLE**
- **EFI_HOB_MEMORY_ALLOCATION**
- **EFI_HOB_MEMORY_ALLOCATION_STACK**
- **EFI_HOB_MEMORY_ALLOCATION_BSP_STORE**
- **EFI_HOB_MEMORY_ALLOCATION_MODULE**
- **EFI_HOB_RESOURCE_DESCRIPTOR**
- **EFI_HOB_GUID_TYPE**
- **EFI_HOB_FIRMWARE_VOLUME**
- **EFI_HOB_FIRMWARE_VOLUME2**
- **EFI_HOB_CPU**
- **EFI_HOB_MEMORY_POOL**
- **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_UNUSED                 0xFFFE
#define EFI_HOB_TYPE_END_OF_HOB_LIST        0xffff
```

Other values for *HobType* are reserved for future use by this specification.

5.3 PHIT HOB

EFI_HOB_HANDOFF_INFO_TABLE (PHIT HOB)

Summary

Contains general state information used by the HOB producer phase. This HOB must be the first one in the HOB list.

Prototype

```
typedef struct _EFI_HOB_HANDOFF_INFO_TABLE {
  EFI_HOB_GENERIC_HEADER  Header;
  UINT32                  Version;
  EFI_BOOT_MODE           BootMode;
  EFI_PHYSICAL_ADDRESS    EfiMemoryTop;
  EFI_PHYSICAL_ADDRESS    EfiMemoryBottom;
  EFI_PHYSICAL_ADDRESS    EfiFreeMemoryTop;
  EFI_PHYSICAL_ADDRESS    EfiFreeMemoryBottom;
  EFI_PHYSICAL_ADDRESS    EfiEndOfHobList;
} EFI_HOB_HANDOFF_INFO_TABLE;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_HANDOFF**.

Version

The version number pertaining to the PHIT HOB definition. See "Related Definitions" below for the version numbers defined by this specification. This value is 4 bytes in length to provide an 8-byte aligned entry when it is combined with the 4-byte *BootMode*.

BootMode

The system boot mode as determined during the HOB producer phase. Type **EFI_BOOT_MODE** is a **UINT32**; if the PI Architecture-compliant implementation incorporates the PEI phase, the possible bit values are defined in the *Platform Initialization Pre-EFI Initialization Core Interface Specification* (PEI CIS).

EfiMemoryTop

The highest address location of memory that is allocated for use by the HOB producer phase. This address must be 4-KB 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-KB 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-KB 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.

EFI_HOB_LOAD_PEIM

Summary

Describes request of the module to be loaded to the permanent memory once it is available. Unlike most of the other HOBs, this HOB is produced and consumed during the HOB producer phase.

Prototype

```
typedef struct _EFI_HOB_LOAD_PEIM {
  EFI_HOB_GENERIC_HEADER  Header;
  EFI_PEI_FILE_HANDLE     FileHandle;
  EFI_PEIM_ENTRY_POINT2   EntryPoint;
  EFI_PEIM_ENTRY_POINT2   InMemEntryPoint;
} EFI_HOB_LOAD_PEIM;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_LOAD_PEIM**.

FileHandle

File handle of the module to be loaded.

EntryPoint

Normal module entry point

InMemEntryPoint

Function to be called once PEIM is loaded to the permanent memory.

Description

The Load PEIM HOB is created by the PEIM to request loading to the permanent memory once it is available.

The Load PEIM HOB may only be created during the execution of the PEIM entry point.

Any Load PEIM HOBs created before permanent memory and LoadFile PPI are available will be processed when permanent memory and LoadFile PPI become available.

Any Load PEIM HOBs created after permanent memory and LoadFile PPI are available will be processed when control returns from the entry point of the PEIM, which created the HOB back to the PEI Dispatcher.

The PEI Foundation processes Load PEIM HOB as follows:

- Loads FFS file corresponding to **FileHandle** to the permanent memory using LoadFile PPI, unless file is already in memory.
- Calculates address of the function **InMemoryEntryPoint** in permanent memory and makes a call to that address.

5.5 Resource Descriptor HOB

EFI_HOB_RESOURCE_DESCRIPTOR

Summary

Describes the resource properties of all fixed, nonrelocatable resource ranges found on the processor host bus during the HOB producer phase.

Prototype

```
typedef struct _EFI_HOB_RESOURCE_DESCRIPTOR {
  EFI_HOB_GENERIC_HEADER        Header;
  EFI_GUID                      Owner;
  EFI_RESOURCE_TYPE             ResourceType;
  EFI_RESOURCE_ATTRIBUTE_TYPE   ResourceAttribute;
  EFI_PHYSICAL_ADDRESS          PhysicalStart;
  UINT64                        ResourceLength;
} EFI_HOB_RESOURCE_DESCRIPTOR;
```

Parameters

Header

The HOB generic header. *Header.HobType* = **EFI_HOB_TYPE_RESOURCE_DESCRIPTOR**.

Owner

A GUID representing the owner of the resource. This GUID is used by HOB consumer phase components to correlate device ownership of a resource.

ResourceType

Resource type enumeration as defined by **EFI_RESOURCE_TYPE**. Type **EFI_RESOURCE_TYPE** is defined in "Related Definitions" below.

ResourceAttribute

Resource attributes as defined by **EFI_RESOURCE_ATTRIBUTE_TYPE**. Type **EFI_RESOURCE_ATTRIBUTE_TYPE** is defined in "Related Definitions" below.

PhysicalStart

Physical start address of the resource region. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the UEFI 2.0 specification.

ResourceLength

Number of bytes of the resource region.

Description

The resource descriptor HOB describes the resource properties of all fixed, nonrelocatable resource ranges found on the processor host bus during the HOB producer phase. This HOB type does not describe how memory is used but instead describes the attributes of the physical memory present.

The HOB consumer phase reads all resource descriptor HOBs when it established the initial Global Coherency Domain (GCD) map. The minimum requirement for the HOB producer phase is that executable content in the HOB producer phase report one of the following:

- The resources that are necessary to start the HOB consumer phase
- The fixed resources that are not captured by HOB consumer phase driver components that were started prior to the dynamic system configuration performed by the platform boot-policy phase

For example, executable content in the HOB producer phase should report any physical memory found during the HOB producer phase. Another example is reporting the Boot Firmware Volume (BFV) that contains firmware volume(s). Executable content in the HOB producer phase does not need to report fixed system resources such as I/O port 70h/71h (real-time clock) because these fixed resources can be allocated from the GCD by a platform-specific chipset driver loading in the HOB consumer phase prior to the platform boot-policy phase, for example.

Current thinking is that the GCD does not track the HOB's *Owner* GUID, so a HOB consumer phase component that assumes ownership of a device's resource must deallocate the resource initialized by the HOB producer phase from the GCD before attempting to assign the devices resource to itself in the HOB consumer phase.

Related Definitions

There can only be a single *ResourceType* field, characterized as follows.

```
//*******************************************************
// EFI_RESOURCE_TYPE
//*******************************************************

typedef UINT32 EFI_RESOURCE_TYPE;

#define EFI_RESOURCE_SYSTEM_MEMORY          0x00000000
#define EFI_RESOURCE_MEMORY_MAPPED_IO       0x00000001
#define EFI_RESOURCE_IO                     0x00000002
#define EFI_RESOURCE_FIRMWARE_DEVICE        0x00000003
#define EFI_RESOURCE_MEMORY_MAPPED_IO_PORT  0x00000004
#define EFI_RESOURCE_MEMORY_RESERVED        0x00000005
#define EFI_RESOURCE_IO_RESERVED            0x00000006
#define EFI_RESOURCE_MAX_MEMORY_TYPE        0x00000007
```

The following table describes the fields listed in the above definition.

EFI_RESOURCE_SYSTEM_MEMORY	Memory that persists out of the HOB producer phase.
EFI_RESOURCE_MEMORY_MAPPED_IO	Memory-mapped I/O that is programmed in the HOB producer phase.
EFI_RESOURCE_IO	Processor I/O space.
EFI_RESOURCE_FIRMWARE_DEVICE	Memory-mapped firmware devices.
EFI_RESOURCE_MEMORY_MAPPED_IO_PORT	Memory that is decoded to produce I/O cycles.
EFI_RESOURCE_MEMORY_RESERVED	Reserved memory address space.
EFI_RESOURCE_IO_RESERVED	Reserved I/O address space.

EFI_RESOURCE_MAX_MEMORY_TYPE	Any reported HOB value of this type or greater should be deemed illegal. This value could increase with successive revisions of this specification, so the “illegality” will also be based upon the revision field of the PHIT HOB.

The *ResourceAttribute* field is characterized as follows:

```
//*******************************************************
// EFI_RESOURCE_ATTRIBUTE_TYPE
//*******************************************************

typedef UINT32 EFI_RESOURCE_ATTRIBUTE_TYPE;


// These types can be ORed together as needed.
//
// The first three enumerations describe settings
//
#define EFI_RESOURCE_ATTRIBUTE_PRESENT      0x00000001
#define EFI_RESOURCE_ATTRIBUTE_INITIALIZED  0x00000002
#define EFI_RESOURCE_ATTRIBUTE_TESTED       0x00000004

#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_READ_PROTECTED    0x00000080
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTED   0x00000100
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTED
                                                 0x00000200
#define EFI_RESOURCE_ATTRIBUTE_READ_PROTECTABLE
                                                 0x00100000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE
                                                 0x00200000
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE
                                                 0x00400000

// The rest of the settings describe capabilities
//

#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
```

The following table describes the fields listed in the above definition.

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.
EFI_RESOURCE_ATTRIBUTE_ EXECUTION_PROTECTED	Physical memory protection attribute: The memory region is execution protected.
EFI_RESOURCE_ATTRIBUTE_ UNCACHEABLE	Memory cacheability attribute: The memory does not support caching.
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 13 specifies the resource attributes applicable to each resource type.

Table 13. 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		
Uncacheable	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 Unused HOB

EFI_HOB_TYPE_UNUSED

Summary

Indicates that the contents of the HOB can be ignored.

Prototype

```
#define EFI_HOB_TYPE_UNUSED             0xFFFE
```

Description

This HOB type means that the contents of the HOB can be ignored. This type is necessary to support the simple, allocate-only architecture of HOBs that have no delete service. The consumer of the HOB list should ignore HOB entries with this type field.

An agent that wishes to make a HOB entry ignorable should set its type to the prototype defined above.

5.11 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**.