Intel® Platform Innovation Framework for EFI Firmware Volume Specification

Version 0.9

September 16, 2003

Intel order number xxxxxx-001

Revision History

Revision	Revision History	Date
0.9	First public release.	9/16/03

intel

Contents

Figures

Tables

1 Introduction

Overview

This specification defines the Framework image format and its associated file access protocols that are required for an implementation the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). This specification does the following:

- Describes the Firmware Volume Protocol, the Framework firmware image format, and Framework file types
- Provides code definitions for services, functions, and data types that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Scope

This specification defines the following:

- Firmware storage interfaces
- The associated binary format that may be accessed using these interfaces

It does not, however, define the binary format of the data as it actually exists in the storage media.

Rationale

Unlike a traditional legacy BIOS, which generally is monolithic and contains few independent components, Framework-based firmware is highly modular, consisting of many small, independently linked components. A new approach to firmware storage is needed to ensure the following:

- Efficient usage of the firmware devices
- A flexible storage strategy that allow various components to be found and retrieved without *a priori* knowledge of exactly where to find them or the methods that are required to retrieve them

Conventions Used in This Document

This document uses the typographic and illustrative conventions described below.

Data Structure Descriptions

Intel® processors based on 32-bit Intel® architecture (IA-32) 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. Processors of the Intel® Itanium® processor family 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.

Protocol Descriptions

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

Protocol Name: The formal name of the protocol interface.

Summary: A brief description of the protocol interface.

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

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

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

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

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

Procedure Descriptions

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

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

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

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

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

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

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

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 *Extensible Firmware Interface Specification*.

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

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

See the master Framework glossary in the Framework Interoperability and Component Specifications help system for definitions of terms and abbreviations that are used in this document or that might be useful in understanding the descriptions presented in this document.

See the master Framework references in the Interoperability and Component Specifications help system for a complete list of the additional documents and specifications that are required or suggested for interpreting the information presented in this document.

The Framework Interoperability and Component Specifications help system is available at the following URL:

http://www.intel.com/technology/framework/spec.htm

2
Design Discussion

Firmware Volumes

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. A logical firmware device is called a *firmware volume*. In the Framework, 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 Framework firmware.

If the files contained in a firmware volume must be accessed from either the Security (SEC) or Pre-EFI Initialization (PEI) phases or early in the Driver Execution Environment (DXE) phase, the firmware volume must be memory mapped and follow the Framework Firmware File System (FFS) format, which is defined in the *Intel® Platform Innovation Framework for EFI Firmware File System Specification.* The SEC, PEI, and DXE phases must be able to parse the FFS and Framework firmware image format as necessary. As such, the FFS is architectural for these types of firmware volumes.

Firmware Volume Protocol

Firmware Volume Protocol Overview

The DXE phase accesses firmware volumes using the file abstraction contained in the Firmware Volume Protocol. The Firmware Volume Protocol allows DXE to access all types of firmware volumes, including the following:

- Firmware volumes that are not memory mapped
- Firmware volumes that do not implement the FFS

Firmware Volume Protocol Stacks

Firmware Volume Protocol Stack: Typical

Typically, the Firmware Volume Protocol will be produced by a file system driver and will layer on top of the Firmware Volume Block Protocol to access the firmware volume hardware. This implementation yields the protocol stack shown in the figure below.

See the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification* for more information on the Firmware Volume Block Protocol.

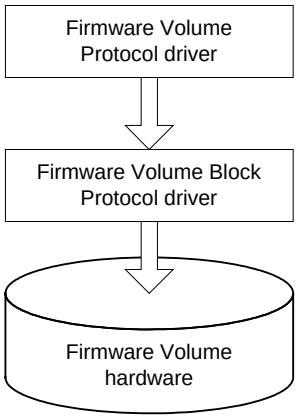

Figure 2-1. Firmware Volume Protocol Stack (Typical)

Firmware Volume Protocol Stack: Memory-Mapped Firmware Volume Hardware

However, there is an exception to this typical stack. If the firmware volume hardware is memory mapped, the Firmware Volume Protocol accesses the firmware volume at its memory address for reads. All other operations still go through the Firmware Volume Block Protocol. This scenario yields the protocol stack shown in the figure below.

See the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification* for more information on the Firmware Volume Block Protocol.

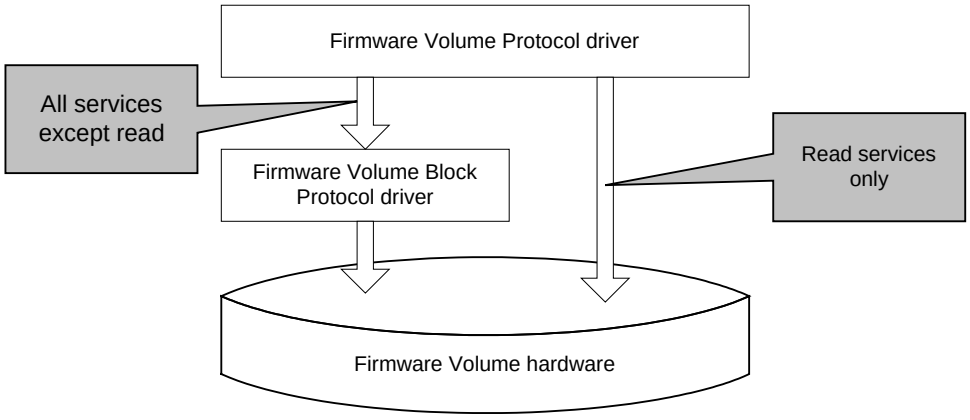

Figure 2-2. Firmware Volume Protocol Stack (Memory-Mapped Firmware Volume Hardware)

Firmware Volume Protocol Stack: Direct Interface with Hardware

The only other case is the degenerate case where the Firmware Volume Protocol subsumes all functionality and interfaces with the hardware directly, as shown in the figure below.

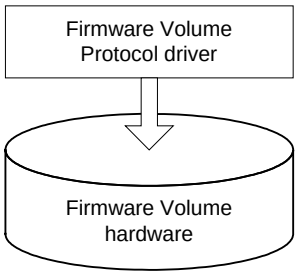

Figure 2-3. Firmware Volume Protocol Stack (Direct Interface with Hardware)

Framework Firmware Image Format

Framework Firmware Image Format Introduction

Regardless of the underlying file system implementation, consumers of the Firmware Volume Protocol must know what the binary format of the file data is. The underlying storage is likely to be FFS, but it may be any of the following:

- FAT32
- NTFS
- NFS
- FTP
- Any one of many other ways files are represented

In an operating system context, the contents of a file do not change depending on the type of file system in which they are stored. Assume an executable program named “HelloWorld.” The program image “HelloWorld” is exactly the same whether it is loaded from a FAT12 floppy or an NFS drive.

File Sections

File Sections

File Sections

Many file formats have separate discrete “parts” within them. These “parts” are called *file sections*, or just *sections* for short.

All sections begin with a header that declares the type and length of the section. The section headers must be 4 bytes aligned within the parent file’s image.

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. In the reciprocal relationship, 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.

Example File Image

The figure below is an example file image comprised of sections. It shows the same file in two ways:

- Graphically
- As a tree

The portion labeled “Graphical Representation” graphically shows the encapsulation of sections within the file, while the “Tree Representation” portion shows a tree representation of the same file.

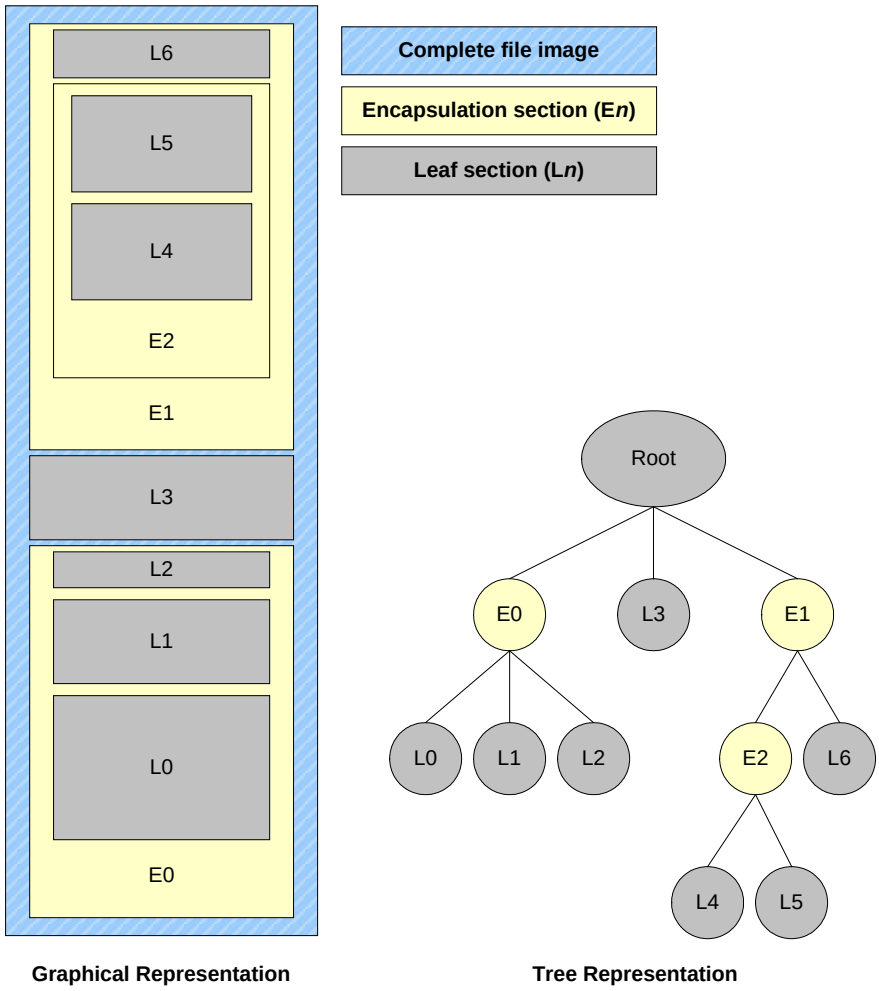

Figure 2-4. Example File Image (Graphical and Tree Representations)

In the example shown in the figure above, 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), in turn, has two children that are both leaves (L4 and L5).

Section Layout

Each section begins with a section header, followed by data defined by the section type.

The section headers are 4 bytes 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. Since it is not possible to describe variable-sized structures in the C

programming language, Backus-Naur Form (BNF) is used to describe section types that have variable lengths. C data structures are considered terminals with respect to the BNF description.

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. The figure below shows the general format of a section.

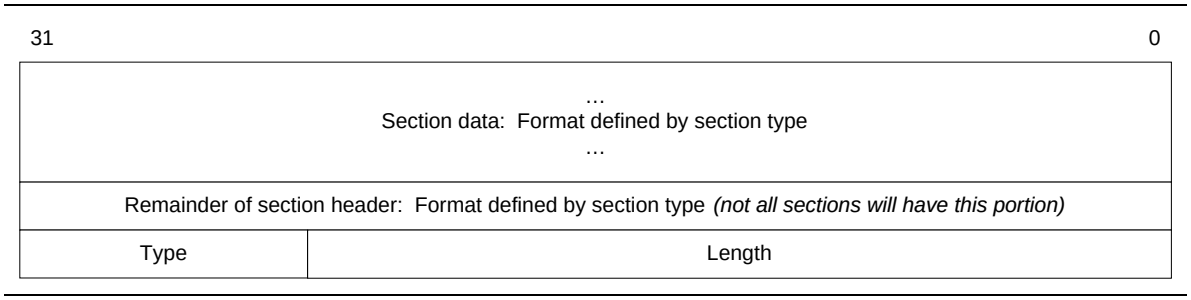

Figure 2-5. General Section Format

Architectural Section Types

This specification defines the following architectural types of sections.

Encapsulation Section Types

- Compression
- GUID-defined

Leaf Section Types

- PE32+ image
- Position-independent code (PIC) image
- Terse Executable (TE)
- DXE dependency expression
- Version
- User interface file name
- Compatibility16 image
- Firmware volume image
- Free-form subtype GUID
- Raw
- PEI dependency expression

See Code Definitions: Framework Firmware Image Format for the definitions of the section types listed above.

Section Extraction Protocols

Section Extraction Protocol Overview

Because some types of files may be arbitrarily complex with respect to encapsulation sections, a code-friendly way of retrieving sections is necessary to facilitate a reasonable implementation of the Firmware Volume Protocol. The Section Extraction Protocol is the API that abstracts the complexities of file construction and provides a straightforward mechanism to extract sections from files.

It is expected that drivers producing the Firmware Volume Protocol will be the only consumers of the Section Extraction Protocol. All other consumers of file sections must use the Firmware Volume Protocol's **`ReadFile()`** API. Furthermore, it is expected that all caching of firmware files and sections thereof will be done within the implementation of the Section Extraction Protocol. These two guidelines enable both performance and code size optimization, as well as preventing cache coherency problems with respect to firmware files.

GUIDed Section Extraction Protocol Overview

The GUIDed Section Extraction Protocol is used by the section extraction driver to enable extraction of GUIDed sections. It is essentially a "plug-in" to enable extensibility to section extraction.

File Types

File Types Overview

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 Framework image format defines the contents of a file that is returned by the firmware volume interface.

The Framework image format 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. In an FFS firmware volume, the file type is captured in the *`Type`* field of the FFS file header, **`EFI_FFS_FILE_HEADER`**; see the *Intel® Platform Innovation Framework for EFI Firmware File System Specification* for the definition of the FFS file header.

This specification defines the following ten architectural file types:

- **`EFI_FV_FILETYPE_RAW`**
- **`EFI_FV_FILETYPE_FREEFORM`**
- **`EFI_FV_FILETYPE_SECURITY_CORE`**
- **`EFI_FV_FILETYPE_PEI_CORE`**
- **`EFI_FV_FILETYPE_DXE_CORE`**
- **`EFI_FV_FILETYPE_PEIM`**

- **EFI_FV_FILETYPE_DRIVER**
- **EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER**
- **EFI_FV_FILETYPE_APPLICATION**
- **EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE**

An additional file type, **EFI_FV_FILETYPE_ALL**, is defined in “Code Definitions,” but it is only a pseudo type; see its code definition for details.

3
Code Definitions

Introduction

This section contains the basic definitions of the Framework image format and its associated file access protocols. The following protocols and data types are defined in this section:

- **EFI_FIRMWARE_VOLUME_PROTOCOL**
- **EFI_COMMON_SECTION_HEADER** and the defined section types
- **EFI_SECTION_EXTRACTION_PROTOCOL**
- **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL**
- **EFI_FV_FILETYPE** and the defined Framework file types

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 or function definition:

- **EFI_FV_ATTRIBUTES**
- **EFI_FV_FILE_ATTRIBUTES**
- **EFI_FV_WRITE_POLICY**
- **EFI_SECTION_TYPE**
- **EFI_COMPRESSION_SECTION_HEADER**

Firmware Volume Protocol

EFI_FIRMWARE_VOLUME_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. The Firmware Volume Protocol also provides mechanisms for determining and modifying some attributes of the firmware volume.

GUID

```
// 389F751F-1838-4388-8390-CD8154BD27F8

#define EFI_FIRMWARE_VOLUME_PROTOCOL_GUID    \
  { 0x389F751F, 0x1838, 0x4388, 0x83, 0x90, 0xCD, 0x81, \
    0x54, 0xBD, 0x27, 0xF8 }
```

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_FIRMWARE_VOLUME_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 *EFI 1.10 Specification.*

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 EFI system partition, for example. This level of firmware volume implementation detail is not visible to the consumers of the Firmware Volume Protocol.

EFI_FIRMWARE_VOLUME_PROTOCOL. GetVolumeAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FV_GET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_PROTOCOL  *This,
  OUT EFI_FV_ATTRIBUTES            *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_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 *EFI 1.10 Specification.*

Related Definitions

```
//*******************************************************
// EFI_FV_ATTRIBUTES
//*******************************************************

typedef UINT64 EFI_FV_ATTRIBUTES;



//*******************************************************
// EFI_FV_ATTRIBUTES bit definitions
//*******************************************************
#define EFI_FV_READ_DISABLE_CAP      0x0000000000000001
#define EFI_FV_READ_ENABLE_CAP       0x0000000000000002
#define EFI_FV_READ_STATUS           0x0000000000000004
```

```
#define EFI_FV_WRITE_DISABLE_CAP     0x0000000000000008
#define EFI_FV_WRITE_ENABLE_CAP      0x0000000000000010
#define EFI_FV_WRITE_STATUS          0x0000000000000020

#define EFI_FV_LOCK_CAP              0x0000000000000040
#define EFI_FV_LOCK_STATUS           0x0000000000000080
#define EFI_FV_WRITE_POLICY_RELIABLE 0x0000000000000100

#define EFI_FV_ALIGNMENT_CAP         0x0000000000008000
#define EFI_FV_ALIGNMENT_2           0x0000000000010000
#define EFI_FV_ALIGNMENT_4           0x0000000000020000
#define EFI_FV_ALIGNMENT_8           0x0000000000040000
#define EFI_FV_ALIGNMENT_16          0x0000000000080000
#define EFI_FV_ALIGNMENT_32          0x0000000000100000
#define EFI_FV_ALIGNMENT_64          0x0000000000200000
#define EFI_FV_ALIGNMENT_128         0x0000000000400000
#define EFI_FV_ALIGNMENT_256         0x0000000000800000
#define EFI_FV_ALIGNMENT_512         0x0000000001000000
#define EFI_FV_ALIGNMENT_1K          0x0000000002000000
#define EFI_FV_ALIGNMENT_2K          0x0000000004000000
#define EFI_FV_ALIGNMENT_4K          0x0000000008000000
#define EFI_FV_ALIGNMENT_8K          0x0000000010000000
#define EFI_FV_ALIGNMENT_16K         0x0000000020000000
#define EFI_FV_ALIGNMENT_32K         0x0000000040000000
#define EFI_FV_ALIGNMENT_64K         0x0000000080000000

// EFI_FV_ATTRIBUTES bit semantics
```

Following is a description of the fields in the above definition.

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. See **EFI_FIRMWARE_VOLUME_PROTOCOL.WriteFile()**.
EFI_FV_ALIGNMENT_CAP	Set to 1 if the firmware volume supports alignment attributes for files. If **EFI_FV_ALIGNMENT_CAP==0**, then all **EFI_FV_ALIGNMENT_{alignment value}** bits are cleared to zero.
EFI_FV_ALIGNMENT_{alignment_value}	Each if these bits indicates whether or not the firmware volume supports the *alignment_value.* A value of 1 indicates the *alignment_value* is supported.

All other bits are reserved and are cleared to zero.

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME_PROTOCOL. SetVolumeAttributes()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FV_SET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_PROTOCOL  *This,
  IN OUT EFI_FV_ATTRIBUTES         *FvAttributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_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 *EFI 1.10 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_VOLUME_PROTOCOL.ReadFile()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FV_READ_FILE) (
  IN EFI_FIRMWARE_VOLUME_PROTOCOL    *This,
  IN 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_VOLUME_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 *EFI 1.10 Specification.*

Buffer

Pointer to a pointer to a buffer in which the file or section contents are returned. 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 Code Definitions: File Types 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: Framework Firmware Image 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 *EFI 1.10 Specification.*

Related Definitions

```
//*******************************************************
// EFI_FV_FILE_ATTRIBUTES
//*******************************************************

typedef UINT32 EFI_FV_FILE_ATTRIBUTES;


#define EFI_FV_FILE_ATTRIB_ALIGNMENT        0x0000001F
```

31 5	4 0
Reserved – must be 0	**EFI_FV_FILE_ATTRIB_ALIGNMENT**

The *Reserved* field must be set to zero.

The **EFI_FV_FILE_ATTRIB_ALIGNMENT** field indicates that the beginning of the data 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 64 KB alignment. The supported alignments are described in the table below. All other values are reserved.

Table 3-1. 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
1 KB	10
2 KB	11
4 KB	12
8 KB	13
16 KB	14
32 KB	15
64 KB	16

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_VOLUME_PROTOCOL. ReadSection()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FV_READ_SECTION) (
  IN EFI_FIRMWARE_VOLUME_PROTOCOL    *This,
  IN 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_VOLUME_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 *EFI 1.10 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 file or 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 Code Definitions: Framework Firmware Image Format 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 *EFI 1.10 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 **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 **GUIDED_SECTION_EXTRACTION_PROTOCOL** is published.

EFI_FIRMWARE_VOLUME_PROTOCOL.WriteFile()

Summary

Writes one or more files to the firmware volume.

Prototype

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

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_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 *EFI 1.10 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. Following is a description of the fields in the above definition.

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 *EFI 1.10 Specification.*

Type

Indicates the type of file to be written. Type **EFI_FV_FILETYPE** is defined in Code Definitions: File Types.

FileAttributes

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

Buffer

Pointer to a buffer containing the file to be written.

BufferSize

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

Status Codes Returned

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

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FV_GET_NEXT_FILE) (
  IN EFI_FIRMWARE_VOLUME_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_VOLUME_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 reinitialize 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 Code Definitions: File Types for **EFI_FV_FILETYPE** related definitions.

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 *EFI 1.10 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 *Key* parameter is used to indicate a starting point of the search. If the buffer **Key* is completely initialized to zero, the search reinitialized 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 *EFI 1.10 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.

Framework Firmware Image Format

File Sections

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

//*************************************************************
// 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
```

All other values are reserved for future use.

Encapsulation Sections

EFI_SECTION_COMPRESSION

Summary

An encapsulation section type in which the section data is compressed.

Prototype

```
CompressionSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        < EFI_COMPRESSION_SECTION_HEADER CompressionHeader >
        { CompressedData }

CompressedData :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Usual common section header. *CommonHeader.Type* = **EFI_SECTION_COMPRESSION**.

CompressionHeader

Compression-section-specific header. Type **EFI_COMPRESSION_SECTION_HEADER** is defined in "Related Definitions" below.

CompressedData

An array of zero or more bytes. Data is compressed using the compression algorithm indicated by *CompressionHeader.CompressionType*. Once decompressed, it can be interpreted as a section stream.

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 *CompressionHeader.CompressionType* parameter. The decompressed image is then interpreted as a section stream.

Related Definitions

```
//*************************************************************
// EFI_COMPRESSION_SECTION_HEADER
//*************************************************************

typedef struct {
  UINT32                    UncompressedLength;
  UINT8                     CompressionType;
} EFI_COMPRESSION_SECTION_HEADER;
```

UncompressedLength

UINT32 that indicates the size of the section data after decompression.

CompressionType

Indicates what compression algorithm is used.

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

Following is a description of the fields in the above definition.

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 *EFI Specification.*

EFI_SECTION_GUID_DEFINED

Summary

An encapsulation section type in which the method of encapsulation is defined by an identifying GUID.

Prototype

```
GuidDefinedSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        GuidedSectionHeader
        { Data }

GuidedSectionHeader :
< EFI_GUID SectionDefinitionGuid >
        < UINT16 DataOffset >
        < UINT16 Attributes >
        { GuidSpecificHeaderFields }

GuidSpecificHeaderFields :
        < UINT8 > { < UINT8 > }

Data :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_GUID_DEFINED**.

GuidedSectionHeader.SectionDefinitionGuid

GUID that defines the format of the data that follows. It is in essence a vendor-defined section type. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

GuidedSectionHeader.DataOffset

Contains the offset in bytes from the beginning of the common header to the first byte of the data.

GuidedSectionHeader.Attributes

Bit field that declares some specific characteristics of the section contents. The bits are defined as shown in "Related Definitions" below.

GuidSpecificHeaderFields

Zero or more bytes of data that is 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 the GUIDed Section Extraction Protocol is required to extract the contents of this encapsulation section. (See **EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL** for details of the GUIDed Section Extraction Protocol)

The GUID-defined section enables original equipment manufacturers (OEMs) to define 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
```

Following is a description of the fields in the above definition:

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 GUIDed Section Extraction Protocol. In this case, the beginning of the encapsulated section stream is indicated by the value of *GuidedSectionHeader.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. See **EFI_GUIDED_SECTION_EXTRACTION_PROCOCOL. ExtractSection()** for details.

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 for **EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()** 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.

Leaf Sections

EFI_SECTION_PE32

Summary

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

Prototype

```
Pe32Section :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        { Pe32Image }

Pe32Image :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_PE32**.

Pe32Image

PE32+ image.

Description

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

EFI_SECTION_PIC

Summary

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

Prototype

```
PicSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        { PicImage }

PicImage :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_PIC**.

PicImage

Position-independent PE32+ image with relocation information stripped from the image.

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, Pre-EFI Initialization Module (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_TE

Summary

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

Prototype

```
TESection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        TEImage
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_TE**.

TEImage

TE image as defined in the *Intel® Platform Innovation Framework for EFI Architecture Specification.*

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 Framework 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_DXE_DEPEX

Summary

A leaf section type that is used to determine the dispatch order for a DXE driver.

Prototype

```
DxeDepexSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        DepexImage
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_DXE_DEPEX**.

DepexImage

DXE driver dependency expression as defined in the *Intel® Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification* (DXE CIS).

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 DXE CIS for details regarding the format of the dependency expression.

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

```
VersionSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        < UINT16 BuildNumber >
        VersionString

VersionString :
        { INT16 } UnicodeNull

UnicodeNull : < 0x0000 >   // Constant INT16 with value of zero
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_VERSION**.

BuildNumber

A **UINT16** that represents a particular build. Subsequent builds have monotonically increasing build numbers relative to earlier builds.

VersionString

A null-terminated Unicode string that contains a text representation of the version. If there is no text representation of the version, then an empty string must be provided.

Description

A *version section* is a leaf section that contains a numeric build number and an optional Unicode string that represents the file revision.

To facilitate versioning of PEIMs, DXE drivers, and other files, a version section may be included in a file. There must never be more than one version section contained within a file.

EFI_SECTION_USER_INTERFACE

Summary

A leaf section type that contains a Unicode string that contains a human-readable file name.

Prototype

```
UserInterfaceFileNameSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        FileNameString

FileNameString :
        { INT16 } UnicodeNull

UnicodeNull : < 0x0000 >   // Constant INT16 with value of zero
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_USER_INTERFACE**.

FileNameString

A null-terminated Unicode string that contains the human readable file name.

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_COMPATIBILITY16

Summary

A leaf section type that contains an IA-32 16-bit executable image.

Prototype

```
Compatibility16Section :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        { Compatibility16Image }

Compatibility16Image :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_COMPATIBILITY16**.

Compatibility16Image

Compatibility16 image.

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 Framework firmware is stored in a 16-bit executable image.

EFI_SECTION_FIRMWARE_VOLUME_IMAGE

Summary

A leaf section type that contains a firmware volume image.

Prototype

```
FirmwareVolumeImageSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        { FvImage }

FvImage :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_FIRMWARE_VOLUME_IMAGE**.

FvImage

Complete firmware volume image that is suitable for copying into memory and producing a Firmware Volume Protocol to retrieve the files that are contained within. The image must contain a valid firmware volume header as defined by the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification.*

Description

A *firmware volume image section* is a leaf section that contains a firmware volume image. It is used for capsule updates and crisis recovery. See the following specifications for details:

- Intel® Platform Innovation Framework for EFI Capsule Specification
- Intel® Platform Innovation Framework for EFI Recovery Specification

EFI_SECTION_FREEFORM_SUBTYPE_GUID

Summary

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

Prototype

```
FreeformSubtypeGuidSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        < EFI_GUID SubtypeGuid >
```

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 in essence a vendor-defined file type. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 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** in “Code Definitions” for more details about **EFI_FV_FILETYPE_FREEFORM** files.

EFI_SECTION_RAW

Summary

A leaf section type that contains an array of zero or more bytes.

Prototype

```
RawSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        { Data }

Data :
        < UINT8 > { < UINT8 > }
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_RAW**.

Data

Zero or more bytes of arbitrary data.

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_PEI_DEPEX

Summary

A leaf section type that is used to determine dispatch order for a PEIM.

Prototype

```
PeiDepexSection :
        < EFI_COMMON_SECTION_HEADER CommonHeader >
        DepexImage
```

Parameters

CommonHeader

Common section header. *CommonHeader.Type* = **EFI_SECTION_PEI_DEPEX**.

DepexImage

PEIM dependency expression as defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

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 PEI CIS for details regarding the format of the dependency expression.

Section Extraction Protocol

EFI_SECTION_EXTRACTION_PROTOCOL

Summary

The Section Extraction Protocol provides a simple method of extracting sections from arbitrarily complex files. The caller is responsible for removing any file-system-specific information from the contents of the file to produce a clean "section stream." More generally, a section stream is simply a buffer containing one or more adjacent sections. Typically, a section stream will simply be the contents of a sectioned file.

GUID

```
// 448F5DA4-6DD7-4FE1-9307-69224192215D

#define EFI_SECTION_EXTRACTION_PROTOCOL_GUID              \
  { 0x448F5DA4, 0x6DD7, 0x4FE1, 0x93, 0x07, 0x69, 0x22, \
    0x41, 0x92, 0x21, 0x5D }
```

Protocol Interface Structure

```
typedef struct _EFI_SECTION_EXTRACTION_PROTOCOL {
  EFI_OPEN_SECTION_STREAM        OpenSectionStream;
  EFI_GET_SECTION                GetSection;
  EFI_CLOSE_SECTION_STREAM       CloseSectionStream;
} EFI_SECTION_EXTRACTION_PROTOCOL;
```

Parameters

OpenSectionStream

Takes a bounded stream of sections and returns a section stream handle. See the **OpenSectionStream()** function description.

GetSection

Given a section stream handle, retrieves the requested section and meta-data from the section stream. See the **GetSection()** function description.

CloseSectionStream

Given a section stream handle, closes the section stream. See the **CloseSectionStream()** function description.

EFI_SECTION_EXTRACTION_PROCOCOL.OpenSectionStream()

Summary

Creates and returns a new section stream handle to represent the new section stream.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_OPEN_SECTION_STREAM)(
  IN EFI_SECTION_EXTRACTION_PROTOCOL  *This,
  IN UINTN                            SectionStreamLength,
  IN VOID                             *SectionStream,
  OUT UINTN                           *SectionStreamHandle
  );
```

Parameters

This

Indicates the **EFI_SECTION_EXTRACTION_PROTOCOL** instance.

SectionStreamLength

Size in bytes of the section stream.

SectionStream

Buffer containing the new section stream.

SectionStreamHandle

A pointer to a caller-allocated **UINTN** that, on output, contains the new section stream handle.

Description

The **OpenSectionStream()** function creates and returns a new section stream handle to represent the new section stream. This handle is used to access the section stream in the **GetSection()** and **CloseSectionStream()** APIs.

OpenSectionStream() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **OpenSectionStream()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the *EFI 1.10 Specification.*

Status Codes Returned

EFI_SUCCESS	The *SectionStream* was successfully processed and the section stream handle was returned.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The *SectionStream* does not end coincident with the last section in the stream. The section stream may be corrupt or the value of *SectionStreamLength* may be incorrect.

EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()

Summary

Reads and returns a single section from a section stream.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_GET_SECTION)(
  IN EFI_SECTION_EXTRACTION_PROTOCOL  *This,
  IN UINTN                            SectionStreamHandle,
  IN EFI_SECTION_TYPE                 *SectionType,
  IN EFI_GUID                         *SectionDefinitionGuid,  OPTIONAL
  IN UINTN                            SectionInstance,
  IN OUT VOID                         **Buffer,
  IN OUT UINTN                        *BufferSize,
  OUT UINT32                          *AuthenticationStatus
  );
```

Parameters

This

Indicates the **EFI_SECTION_EXTRACTION_PROTOCOL** instance.

SectionStreamHandle

Indicates from which section stream to read.

SectionType

Pointer to an **EFI_SECTION_TYPE**. If *SectionType* == **NULL**, the contents of the entire section stream are returned in *Buffer*. If *SectionType* is not **NULL**, only the requested section is returned. **EFI_SECTION_ALL** matches all section types and can be used as a wild card to extract all sections in order.

SectionDefinitionGuid

Pointer to an **EFI_GUID**. If *SectionType* == **EFI_SECTION_GUID_DEFINED**, *SectionDefinitionGuid* indicates what section GUID to search for. If *SectionType* *!=* **EFI_SECTION_GUID_DEFINED**, then *SectionDefinitionGuid* is unused and is ignored.

See **EFI_GUID_DEFINED_SECTION** for details about GUID-defined sections. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification*.

SectionInstance

Indicates which instance of the requested section type to return when *SectionType* is not **NULL**. The file's section layout can be thought of as a tree that is built recursively left to right. *SectionInstance* is zero based and is calculated using a left-to-right, depth-first search algorithm of the file's section layout. See Code Definitions: Framework Firmware Image Format for more information. If *SectionType* is **NULL**, *SectionInstance* is ignored.

SectionStreamHandle

A pointer to a caller-allocated **UINTN** that, on output, contains the new section stream handle.

Buffer

Pointer to a pointer to a buffer in which the section contents are returned. See "Description" below for more details on the use of the *Buffer* parameter.

BufferSize

Pointer to a caller-allocated **UINTN**. On input, **BufferSize* indicates the size in bytes of the memory region pointed to by *Buffer*. On output, **BufferSize* contains the number of bytes that are required to read the section.

AuthenticationStatus

Pointer to a caller-allocated **UINT32** in which any meta-data from encapsulation GUID-defined sections is returned. See **EFI_GUID_DEFINED_SECTION** for more information on GUID-defined sections.

Description

The **GetSection()** function is used to retrieve a section from within a section stream. The stream can be thought of as a tree with encapsulation sections as interior nodes and terminal leaf sections. This tree is built and searched left to right, depth first.

GetSection() will retrieve both encapsulation sections and leaf sections in their entirety, exclusive of the section header.

Since the requested section may be contained within compression and/or GUIDed encapsulations, the implementation must be capable of processing these encapsulations to produce the requested section. While decompression of an encapsulation compression section is completely transparent, the results of all encapsulation GUIDed sections used for authentication must be exposed to the caller so the caller can make appropriate policy decisions. The authentication results are passed back using the *AuthenticationStatus* output variable. See the parameter description above for a full description of this output.

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

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 **GetSection()**. In this case, **GetSection()** 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.

GetSection() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **GetSection()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the *EFI 1.10 Specification.*

Related Definitions

```
//*******************************************************
// Bit values for AuthenticationStatus
//*******************************************************
#define EFI_AGGREGATE_AUTH_STATUS_PLATFORM_OVERRIDE 0x000001
#define EFI_AGGREGATE_AUTH_STATUS_IMAGE_SIGNED      0x000002
#define EFI_AGGREGATE_AUTH_STATUS_NOT_TESTED        0x000004
#define EFI_AGGREGATE_AUTH_STATUS_TEST_FAILED       0x000008

#define EFI_LOCAL_AUTH_STATUS_PLATFORM_OVERRIDE     0x010000
#define EFI_LOCAL_AUTH_STATUS_IMAGE_SIGNED          0x020000
#define EFI_LOCAL_AUTH_STATUS_NOT_TESTED            0x040000
#define EFI_LOCAL_AUTH_STATUS_TEST_FAILED           0x080000

// All other bits are reserved and must be 0.
```

The bit definitions above are in two groups:

- **Bits 3:0:** Indicate the aggregate *AuthenticationStatus* for the data retrieved. This aggregate is the bit-wise **OR** of all *AuthenticationStatus* values from all layers of GUID-defined sections that were encountered when retrieving the data and any default *AuthenticationStatus* the implementation has associated with the firmware volume.
- **Bits 19:16:** Indicate the *AuthenticationStatus* of the data's immediate parent.

Any section that does not generate its own *AuthenticationStatus* data (i.e., any leaf section, or compression section, or any GUID-defined encapsulation section that has the *Attributes:* **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit clear) inherits the full *AuthenticationStatus* from its immediate parent.

GUID-defined encapsulation sections that have the *Attributes:* **EFI_GUIDED_SECTION_AUTH_STATUS_VALID** bit set generate *AuthenticationStatus* data. As such, this local status data is used to refresh bits 19:16. This local status must also be bit-wise **OR**ed with the aggregate status.

The table below describes the possible values for each grouping of four *AuthenticationStatus* bits.

Table 3-2. Possible *AuthenticationStatus* Bit Values

	Bits 3:0	Bits 19:16
xx00	Image was not signed at any layer.	Image was not locally signed.
xxx1	Platform security policy override… Assumes same meaning as 0010 (image was signed, signature was tested, and signature passed authentication test). The Section Extraction Protocol cannot produce this result, but it can be generated from the Firmware Volume interface if a default Firmware Volume *AuthenticationStatus* is implemented.	Same meaning as for bits 3:0.
0010	One or more section encapsulations of the image were signed, the signatures were tested, and all signatures passed their respective authentication tests. This value is the cumulative or aggregate result of all authentication encapsulations.	Same meaning as bits 3:0, except it represents the local authentication status of the encapsulation from which the data was retrieved.
0110	Image was signed, and the signature was not tested. This case 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. This value is the cumulative or aggregate result of all authentication encapsulations.	Same meaning as bits 3:0, except it represents the local authentication status of the encapsulation from which the data was retrieved.
1010	Image was signed, signature was tested, and signature failed the authentication test. This value is the cumulative or aggregate result of all authentication encapsulations.	Same meaning as bits 3:0, except it represents the local authentication status of the encapsulation from which the data was retrieved.
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.	This value is invalid for bits 19:16 because this value requires at least two layers of authentication and bits 19:16 represent only the most local result.

Status Codes Returned

EFI_SUCCESS	The *SectionStream* was successfully processed and the section contents were returned in *Buffer*.
EFI_PROTOCOL_ERROR	A GUID-defined section was encountered in the section stream with its **EFI_GUIDED_SECTION_PROCESSING_REQUIRED** bit set, but there was no corresponding GUIDed Section Extraction Protocol in the handle database. **Buffer* is unmodified.
EFI_NOT_FOUND	An error was encountered when parsing the *SectionStream*, which indicates that the *SectionStream* is not correctly formatted.
EFI_NOT_FOUND	The requested section does not exist. **Buffer* is unmodified.
EFI_OUT_OF_RESOURCES	The system has insufficient resources to process the request.
EFI_INVALID_PARAMETER	The *SectionStreamHandle* does not exist.
EFI_WARN_BUFFER_ TOO_SMALL	The size of the input buffer is insufficient to contain the requested section. The input buffer is filled and section contents are truncated.

EFI_SECTION_EXTRACTION_PROCOCOL.CloseSectionStream()

Summary

Deletes a section stream handle and returns all associated resources to the system.

Prototype

```
EFI_STATUS
(EFIAPI *EFI_CLOSE_SECTION_STREAM)(
  IN EFI_SECTION_EXTRACTION_PROTOCOL  *This,
  IN UINTN                            SectionStreamHandle
  );
```

Parameters

This

Indicates the **EFI_SECTION_EXTRACTION_PROTOCOL** instance.

SectionStreamHandle

Indicates the section stream to close.

Description

The **CloseSectionStream()** function deletes a section stream handle and frees all associated system resources.

CloseSectionStream() is callable only from **EFI_TPL_NOTIFY** and below. Behavior of **CloseSectionStream()** at any **EFI_TPL** above **EFI_TPL_NOTIFY** is undefined. Type **EFI_TPL** is defined in **RaiseTPL()** in the *EFI 1.10 Specification.*

Status Codes Returned

EFI_SUCCESS	The *SectionStream* was successfully processed and the section stream handle was returned.
EFI_INVALID_PARAMETER	The *SectionStreamHandle* does not exist.

GUIDed Section Extraction Protocol

EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL

Summary

If a GUID-defined section is encountered when doing section extraction, the section extraction driver calls the appropriate instance of the GUIDed Section Extraction Protocol to extract the section stream contained therein.

GUID

Typically, protocol interface structures are identified by associating them with a GUID. Each instance of a protocol with a given GUID must have the same interface structure. While all instances of the GUIDed Section Extraction Protocol must have the same interface structure, they do not all have the same GUID. The GUID that is associated with an instance of the GUIDed Section Extraction Protocol is used to correlate it with the GUIDed section type that it is intended to process.

Protocol Interface Structure

```
typedef struct {
  EFI_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

```
EFI_STATUS
(EFIAPI *EFI_EXTRACT_GUIDED_SECTION)(
  IN EFI_GUIDED_SECTION_EXTRACTION_PROTOCOL *This,
  IN 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. See **EFI_SECTION_EXTRACTION_PROCOCOL.GetSection()** for more details. 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 enable a platform policy of *Authenticate on capsule update but not during normal boot*, the platform must be able to indicate this policy to the GUIDed Section Extraction Protocol (see the *Intel® Platform Innovation Framework for EFI Capsule Specification* for more details). This indication is done using **EFI_SECURITY_POLICY_PROTOCOL_GUID**, which may be published by the security policy driver (see the DXE CIS 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 the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS) 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 *EFI 1.10 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.

File Types

EFI_FV_FILETYPE

Summary

Given the various uses for firmware files, different file types are defined. Each file type is encoded as an 8-bit unsigned integer and carries with it a set of construction rules regarding file section organization.

Prototype

```
typedef UINT8 EFI_FV_FILETYPE;
```

Related Definitions

Following is the enumeration of file types:

```
#define EFI_FV_FILETYPE_ALL                    0x00
#define EFI_FV_FILETYPE_RAW                    0x01
#define EFI_FV_FILETYPE_FREEFORM               0x02
#define EFI_FV_FILETYPE_SECURITY_CORE          0x03
#define EFI_FV_FILETYPE_PEI_CORE               0x04
#define EFI_FV_FILETYPE_DXE_CORE               0x05
#define EFI_FV_FILETYPE_PEIM                   0x06
#define EFI_FV_FILETYPE_DRIVER                 0x07
#define EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER   0x08
#define EFI_FV_FILETYPE_APPLICATION            0x09
// The value 0x0A is reserved and should not be used
#define EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE  0x0B
```

File type values between 0xE0 and 0xEF are reserved for use by the test architecture. File types between 0xF0 and 0xFF are reserved for use by the file system. See the *Intel® Platform Innovation Framework for EFI Firmware File System Specification* and applicable test architecture specifications for details of these file types. All other values are reserved for future use.

EFI_FV_FILETYPE_ALL

Summary

The value **EFI_FV_FILETYPE_ALL** is a pseudo type and reserved for use with the **EFI_FIRMWARE_VOLUME_PROTOCOL.GetNextFile()** API. **EFI_FV_FILETYPE_ALL** is not a legal file type for an actual file. When specified to **GetNextFile()**, it indicates that no filtering on file type is to be done.

Prototype

```
#define EFI_FV_FILETYPE_ALL          0x00
```

EFI_FV_FILETYPE_RAW

Summary

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.

Prototype

```
#define EFI_FV_FILETYPE_RAW            0x01
```

EFI_FV_FILETYPE_FREEFORM

Summary

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.

Prototype

```
#define EFI_FV_FILETYPE_FREEFORM           0x02
```

Description

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.

There are no other construction rules.

EFI_FV_FILETYPE_SECURITY_CORE

Summary

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

Prototype

```
#define EFI_FV_FILETYPE_SECURITY_CORE          0x03
```

EFI_FV_FILETYPE_PEI_CORE

Summary

The file type **EFI_FV_FILETYPE_PEI_CORE** denotes a file that is the PEI Foundation. This image is entered upon completion of the SEC phase of a Framework boot cycle.

Prototype

```
#define EFI_FV_FILETYPE_PEI_CORE          0x04
```

Description

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

1. The file must contain one and only one executable section. This section must have one of the following types:
2. **EFI_SECTION_PE32**
3. **EFI_SECTION_PIC**
4. **EFI_SECTION_TE**

2. 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.

EFI_FV_FILETYPE_DXE_CORE

Summary

The file type **EFI_FV_FILETYPE_DXE_CORE** denotes the file that is the DXE Foundation. This image is the one entered upon completion of the PEI phase of an EFI boot cycle.

Prototype

```
#define EFI_FV_FILETYPE_DXE_CORE          0x05
```

Description

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

1. The file must contain at one and only one executable section, which must have a type of **EFI_SECTION_PE32**.
2. 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.

EFI_FV_FILETYPE_PEIM

Summary

The file type **EFI_FV_FILETYPE_PEIM** denotes a file that is a PEIM. A PEIM is dispatched by the PEI Foundation based on its imports and exports during execution of the PEI phase. See the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS) for details on PEI operation.

Prototype

```
#define EFI_FV_FILETYPE_PEIM          0x06
```

Description

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

1. The file must contain one and only one executable section. This section must have one of the following types:
2. **EFI_SECTION_PE32**
3. **EFI_SECTION_PIC**
4. **EFI_SECTION_TE**
2. The file must contain no more than one **EFI_SECTION_VERSION** section.
3. 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.

EFI_FV_FILETYPE_DRIVER

Summary

The file type **EFI_FV_FILETYPE_DRIVER** denotes a file that contains a PE32 image that can be dispatched by the DXE Dispatcher.

Prototype

```
#define EFI_FV_FILETYPE_DRIVER        0x07
```

Description

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

1. The file must contain at least one **EFI_SECTION_PE32** section. There are no restrictions on encapsulation of this section.
2. The file must contain no more than one **EFI_SECTION_VERSION** section.
3. 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 DXE CIS for details.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER

Summary

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.

Prototype

```
#define EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER      0x08
```

Description

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:

1. 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.
2. The file must not contain more than one **EFI_SECTION_DXE_DEPEX** section.
3. The file must not contain more than one **EFI_SECTION_PEI_DEPEX** section.
4. 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.

EFI_FV_FILETYPE_APPLICATION

Summary

The file type **EFI_FV_FILETYPE_APPLICATION** denotes a file that contains a PE32 image that can be loaded using the EFI Boot Service **LoadImage()**. Files of type **EFI_FV_FILETYPE_APPLICATION** are not dispatched by the DXE Dispatcher.

Prototype

```
#define EFI_FV_FILETYPE_APPLICATION                0x09
```

Description

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

1. 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 DXE CIS for details.

The file may contain other leaf and encapsulation sections as required or enabled by the platform design.

EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE

Prototype

The file type **EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE** denotes a file that contains one or more firmware volume images. This special file type is used to produce firmware volumes in conjunction with a firmware update or crisis recovery. See the following specifications for details:

- *Intel® Platform Innovation Framework for EFI Capsule Specification*
- *Intel® Platform Innovation Framework for EFI Recovery Specification*

Summary

```
#define EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE      0x0B
```

Description

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

1. 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.