Intel® Platform Innovation Framework for EFI Firmware Volume Block 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

1 Introduction

Overview

This specification defines a format for storing firmware volumes in block access type devices for an implementation of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the "Framework"). It is designed to scale to many types of block devices. This specification does the following:

- Defines a firmware volume
- Describes how to implement the Firmware Volume Block Protocol and firmware volume Hand-Off Blocks (HOBs)
- Defines a firmware volume header structure and a block-oriented protocol interface that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*

Target Audience

This document is intended for the following readers:

- Independent hardware vendors (IHVs) and original equipment manufacturers (OEMs) who will be implementing firmware components that are stored in firmware volumes
- BIOS developers, either those who create general-purpose BIOS and other firmware products or those who modify these products for use in Intel® architecture–based products

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 Volume Block Protocol

The Firmware Volume Block Protocol provides block-level access to a firmware volume. Although the Firmware Volume Block Protocol represents an abstraction of the firmware device, it is not intended to be a generic and primitive abstraction to the firmware device. An implementation may choose to implement an arbitrary number of abstractions beneath the Firmware Volume Block driver as required to satisfy platform requirements.

The Firmware Volume Block Protocol provides the following:

- Byte-level read/write functionality
- Block-level erase functionality

It further exposes device-hardening features, such as may be required to protect the firmware from unwanted overwriting and/or erasure.

It is useful to layer a file system driver on top of the Firmware Volume Block Protocol. This file system driver produces the Firmware Volume Protocol, which provides file-level access to a firmware volume. The Firmware Volume Protocol abstracts the file system that is used to format the firmware volume and the hardware device-hardening features that may be present.

For more information, including information on the Firmware Volume Protocol, see the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification.*

Firmware Volume HOB

The Pre-EEFI Initialization (PEI) phase must produce a firmware volume Hand-Off Block (HOB) for each firmware volume. The firmware volume HOB details the location of firmware volumes that contain firmware files. It includes a base address and length. In particular, the DXE Foundation will use these HOBs to discover drivers to execute and the DXE Initial Program Load (IPL) PEIM will use this HOB to discover the location of the DXE Foundation firmware file.

See the *Intel® Platform Innovation Framework for EFI Hand-Off Block (HOB) Specification* for the definition and explanation of the firmware volume HOB.

3
Code Definitions

Introduction

This section contains the basic definitions for storing firmware volumes in block access type devices. The following protocols and data types are defined in this section:

- **EFI_FIRMWARE_VOLUME_HEADER**
- **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL**

This section also contains the definitions for additional data types and structures that are subordinate to the structures in which they are called. The following types or structures can be found in "Related Definitions" of the parent data structure or protocol definition:

- **EFI_FVB_ATTRIBUTES**

Firmware Volume Header

EFI_FIRMWARE_VOLUME_HEADER

Summary

Describes the features and layout of the firmware volume.

Prototype

NOTE

The following prototype uses Backus-Naur Form (BNF) instead of a C data structure because the firmware volume header has a variable length, which is not possible to describe using a C data structure. Memory-mapped firmware volumes must be aligned on an 8-byte boundary.

```
FvHeader:
     < UINT8                 ZeroVector[16] >
     < EFI_GUID              FileSystemGuid >
     < UINT64                FvLength >
     < UINT32                Signature >
     < EFI_FVB_ATTRIBUTES    Attributes >
     < UINT16                HeaderLength >
     < UINT16                Checksum >
     < UINT8                 Reserved[3] >
     < UINT8                 Revision >
     FvBlockMap

FvBlockMap:
     FvBlockMapEntry
     { FvBlockMapEntry }
     FvBlockMapTerminator

FvBlockMapEntry:
     < UINT32 NumBlocks >
     < UINT32 BlockLength >

FvBlockMapTerminator:
     < (UINT32) 0 >     // FvBlockMapEntry.NumBlocks = 0
     < (UINT32) 0 >     // FvBlockMapEntry.BlockLength = 0
```

Parameters

ZeroVector

The first 16 bytes are reserved to allow for the reset vector of processors whose reset vector is at address 0 (Intel® processors based on Intel® XScale™ technology).

FileSystemGuid

Declares the file system with which the firmware volume is formatted. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

FvLength

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

Signature

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

Attributes

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

HeaderLength

Length in bytes of the complete firmware volume header.

Checksum

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

Reserved

In this version of the specification, this field must always be set to zero.

Revision

Set to 1. Future versions of this specification may define new header fields and will increment the *Revision* field accordingly.

FvBlockMap[]

An array of run-length encoded *FvBlockMapEntry* structures. The array is terminated with an entry of **{0,0}**.

FvBlockMapEntry.NumBlocks

The number of blocks in the run.

FvBlockMapEntry.BlockLength

The length of each block in the run.

Description

A firmware volume based on a block device begins with a header that describes the features and layout of the firmware volume. This header includes a description of the capabilities, state, and block map of the device.

The block map is a run-length-encoded array of logical block definitions. This design allows a reasonable mechanism of describing the block layout of typical firmware devices. Each block can be referenced by its logical block address (LBA). The LBA is a zero-based enumeration of all of the blocks—i.e., LBA 0 is the first block, LBA 1 is the second block, and LBA *n* is the (*n*-1) device.

The header is always located at the beginning of LBA 0.

Related Definitions

```
//*******************************************
// EFI_FVB_ATTRIBUTES
//*******************************************

typedef UINT32 EFI_FVB_ATTRIBUTES


// Attributes bit definitions

#define EFI_FVB_READ_DISABLED_CAP     0x00000001
#define EFI_FVB_READ_ENABLED_CAP      0x00000002
#define EFI_FVB_READ_STATUS           0x00000004

#define EFI_FVB_WRITE_DISABLED_CAP    0x00000008
#define EFI_FVB_WRITE_ENABLED_CAP     0x00000010
#define EFI_FVB_WRITE_STATUS          0x00000020

#define EFI_FVB_LOCK_CAP              0x00000040
#define EFI_FVB_LOCK_STATUS           0x00000080

#define EFI_FVB_STICKY_WRITE          0x00000200
#define EFI_FVB_MEMORY_MAPPED         0x00000400
#define EFI_FVB_ERASE_POLARITY        0x00000800

#define EFI_FVB_ALIGNMENT_CAP         0x00008000
#define EFI_FVB_ALIGNMENT_2           0x00010000
#define EFI_FVB_ALIGNMENT_4           0x00020000
#define EFI_FVB_ALIGNMENT_8           0x00040000
#define EFI_FVB_ALIGNMENT_16          0x00080000
#define EFI_FVB_ALIGNMENT_32          0x00100000
#define EFI_FVB_ALIGNMENT_64          0x00200000
#define EFI_FVB_ALIGNMENT_128         0x00400000
#define EFI_FVB_ALIGNMENT_256         0x00800000
#define EFI_FVB_ALIGNMENT_512         0x01000000
#define EFI_FVB_ALIGNMENT_1K          0x02000000
```

```
#define EFI_FVB_ALIGNMENT_2K          0x04000000
#define EFI_FVB_ALIGNMENT_4K          0x08000000
#define EFI_FVB_ALIGNMENT_8K          0x10000000
#define EFI_FVB_ALIGNMENT_16K         0x20000000
#define EFI_FVB_ALIGNMENT_32K         0x40000000
#define EFI_FVB_ALIGNMENT_64K         0x80000000
```

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

EFI_FVB_READ_DISABLED_CAP	**TRUE** if reads from the firmware volume may be disabled.
EFI_FVB_READ_ENABLED_CAP	**TRUE** if reads from the firmware volume may be enabled.
EFI_FVB_READ_STATUS	**TRUE** if reads from the firmware volume are currently enabled.
EFI_FVB_WRITE_DISABLED_CAP	**TRUE** if writes to the firmware volume may be disabled.
EFI_FVB_WRITE_ENABLED_CAP	**TRUE** if writes to the firmware volume may be enabled.
EFI_FVB_WRITE_STATUS	**TRUE** if writes to the firmware volume are currently enabled.
EFI_FVB_LOCK_CAP	**TRUE** if firmware volume attributes may be locked down.
EFI_FVB_LOCK_STATUS	**TRUE** if firmware volume attributes are currently locked down.
EFI_FVB_STICKY_WRITE	**TRUE** if a block erase is required to transition bits from **(NOT)EFI_FVB_ERASE_POLARITY** to **EFI_FVB_ERASE_POLARITY**. That is to say, after erasure a write may negate a bit in the **EFI_FVB_ERASE_POLARITY** state, but a write cannot flip it back again. A block erase cycle is required to transition bits from the **(NOT)EFI_FVB_ERASE_POLARITY** state back to the **EFI_FVB_ERASE_POLARITY** state. See the Firmware Volume Block Protocol.
EFI_FVB_MEMORY_MAPPED	**TRUE** if firmware volume is memory mapped.
EFI_FVB_ERASE_POLARITY	Value of all bits after erasure. See the Firmware Volume Block Protocol.
EFI_FVB_ALIGNMENT_CAP	**TRUE** if firmware volume supports alignment attributes for files. If **EFI_FVB_ALIGNMENT_CAP** is **FALSE**, then all **EFI_FVB_ALIGNMENT_{alignment value}** bits must be zero.
EFI_FVB_ALIGNMENT_{alignment_value}	Each if these bits indicates whether or not the firmware volume supports the *alignment_value.* **TRUE** indicates the *alignment_value* is supported.

All other **EFI_FVB_ATTRIBUTES** bits are reserved and must be zero.

Firmware Volume Block Protocol

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL

Summary

This protocol provides control over block-oriented firmware devices. Typically, the FFS (or an alternate file system) driver consumes the Firmware Volume Block Protocol and produces the Firmware Volume Protocol.

GUID

```
// 0xDE28BC59-6228-41BD-BDF6-A3B9ADB58DA1

#define FW_VOLUME_BLOCK_PROTOCOL_GUID \
{ 0xDE28BC59, 0x6228, 0x41BD, 0xBD, 0xF6, 0xA3, 0xB9, \
  0xAD,0xB5, 0x8D, 0xA1 }
```

Protocol Interface Structure

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

Parameters

GetAttributes

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

SetAttributes

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

GetPhysicalAddress

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

GetBlockSize

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

Read

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

Write

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

EraseBlocks

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

ParentHandle

Handle of the parent firmware volume. Type **EFI_HANDLE** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

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

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. GetAttributes()

Summary

Returns the attributes and current settings of the firmware volume.

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FVB_GET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL   *This,
  OUT EFI_FVB_ATTRIBUTES                  *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Attributes

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

Description

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

Status Codes Returned

EFI_SUCCESS	The firmware volume attributes were returned.

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. SetAttributes()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FVB_SET_ATTRIBUTES) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL    *This,
  IN OUT EFI_FVB_ATTRIBUTES                *Attributes
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Attributes

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

Description

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

Status Codes Returned

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. GetPhysicalAddress()

Summary

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

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FVB_GET_PHYSICAL_ADDRESS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  OUT EFI_PHYSICAL_ADDRESS               *Address
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Address

Pointer to a caller-allocated **EFI_PHYSICAL_ADDRESS** that, on successful return from **GetPhysicalAddress()**, contains the base address of the firmware volume. Type **EFI_PHYSICAL_ADDRESS** is defined in **AllocatePages()** in the *EFI 1.10 Specification.*

Description

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

Status Codes Returned

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. GetBlockSize()

Summary

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

Prototype

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

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Lba

Indicates the block for which to return the size. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the *EFI 1.10 Specification.*

BlockSize

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

NumberOfBlocks

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

Description

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

Status Codes Returned

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. Read()

Summary

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

Prototype

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

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Lba

The starting logical block index from which to read. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the *EFI 1.10 Specification.*

Offset

Offset into the block at which to begin reading.

NumBytes

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

Buffer

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

Description

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

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

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

Status Codes Returned

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. Write()

Summary

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

Prototype

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

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

Lba

The starting logical block index to write to. Type **EFI_LBA** is defined in the **BLOCK_IO** Protocol (section 11.6) in the *EFI 1.10 Specification.*

Offset

Offset into the block at which to begin writing.

NumBytes

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

Buffer

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

Description

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

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

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

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

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

Status Codes Returned

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

EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL. EraseBlocks()

Summary

Erases and initializes a firmware volume block.

Prototype

```
EFI_STATUS
(EFIAPI * EFI_FVB_ERASE_BLOCKS) (
  IN EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL  *This,
  …
  );
```

Parameters

This

Indicates the **EFI_FIRMWARE_VOLUME_BLOCK_PROTOCOL** instance.

…

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

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

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

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

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

Description

The **EraseBlocks()** function erases one or more blocks as denoted by the variable argument list.

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

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

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

Related Definitions

```
//*******************************************
// EFI_LBA_LIST_TERMINATOR
//*******************************************

#define EFI_LBA_LIST_TERMINATOR 0xFFFFFFFFFFFFFFFF
```

Status Codes Returned

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