Intel® Platform Innovation Framework for EFI Firmware File System 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

Contents

Figures

Tables

1 Introduction

Overview

This specification defines the core code that is required for an implementation of the Firmware File System (FFS) of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the “Framework”). This FFS specification does the following:

- Describes the basic components of the FFS
- Defines basic operations that may be performed with the FFS
- Provides code definitions for FFS-related data types and structures that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification*
- Provides pseudo code that describes methods for initializing the FFS and accessing file prior to the FFS being initialized

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.

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

Introduction

The Framework Firmware File System (FFS) is a binary layout of file storage for firmware volumes. It is a flat file system in that there is no provision for any directory hierarchy; rather, files all exist in the root directly. Files are stored, in essence, end to end without any directory entry to describe which files are present. Parsing the contents of a firmware volume to obtain a listing of files present requires walking the firmware volume from beginning to end. This process is abstracted from consumers by the Firmware Volume Protocol, which is expected to be produced by the FFS driver.

All files stored with the FFS must follow the Framework image format described in the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification.*

The file header provides for several levels of integrity checking to help detect file corruption, should it occur for some reason. Authentication (verifying the origin) of the files is not supported directly by the FFS, but it is supported by the Framework image format.

This section explains the following:

- FFS file format
- FFS file integrity and state
- FFS-defined file types
- Volume Top File (VTF)

See Code Definitions for the type definitions of any code that is referenced in this section. See the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification* for the definition of the Firmware Volume Protocol and the Framework image format.

File Format

Overview

This section describes the binary format of the FFS, including the following:

- FFS GUID
- FFS file image

See Code Definitions: File Format for the corresponding code definitions that are described in this section.

FFS GUID

The firmware volume header contains a data field for the file system Globally Unique Identifier (GUID). See the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification* for more information on the firmware volume header. For the FFS file system, the GUID is defined as **EFI_FIRMWARE_FILE_SYSTEM_GUID**; see Code Definitions for the GUID definition.

FFS File Image

All FFS files begin with a header that is 8 bytes aligned with respect to the beginning of the firmware volume. FFS files can contain the following parts:

- Header
- Data
- Tail

It is possible to create a file that has only a header and no data, which means it consumes 24 bytes of space. This type of file is known as a *zero-length file*.

If the file contains data, the data immediately follows the header. The format of the data within a file is defined by the `Type` field in **`EFI_FFS_FILE_HEADER`**.

If indicated in the `Attributes` field of **`EFI_FFS_FILE_HEADER`**, the last two bytes of the file are defined to be the *tail*. The tail is used for file integrity checking and is optional. Zero-length files (files with only a header but no data area) and pad files do not have a tail.

See the **`EFI_FFS_FILE_HEADER`** and **`EFI_FFS_FILE_TAIL`** definitions in Code Definitions: File Format for more information.

The figure below illustrates the layout of a typical FFS file.

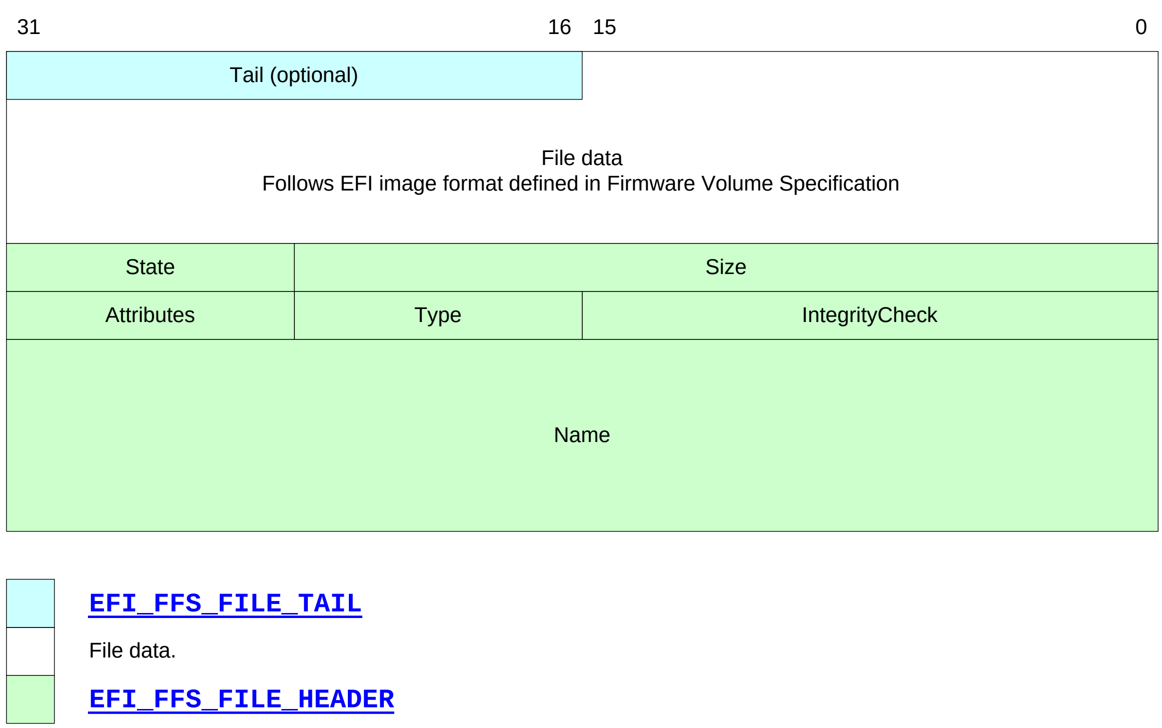

Figure 2-1. Typical FFS File Layout

FFS File Integrity and State

Detecting FFS File Corruption

File corruption, regardless of the cause, must be detectable so that appropriate file system repair steps may be taken. File corruption can come from several sources but generally falls into three categories:

- General failure
- Erase failure
- Write failure

A *general failure* is defined to be apparently random corruption of the storage media. This corruption can be caused by storage media design problems or storage media degradation, for example. This type of failure can be as subtle as changing a single bit within the contents of a file. With good system design and reliable storage media, general failures should not happen. Even so, the FFS enables detection of this type of failure.

An *erase failure* occurs when a block erase of firmware volume media is not completed due to a power failure or other system failure. While the erase operation is not defined, it is expected that most implementations of FFS that allow file write and delete operations will also implement a mechanism to reclaim deleted files and coalesce free space. If this operation is not completed correctly, the file system can be left in an inconsistent state.

Similarly, a *write failure* occurs when a file system write is in progress and is not completed due to a power failure or other system failure. This type of failure can leave the file system in an inconsistent state.

All of these failures are detectable during FFS initialization, and, depending on the nature of the failure, many recovery strategies are possible. Careful sequencing of the `State` bits during normal file transitions is sufficient to enable subsequent detection of write failures. However, the `State` bits alone are not sufficient to detect all occurrences of general and/or erase failures. These types of failures require additional support, which is enabled with the file header `IntegrityCheck` field.

See Pseudo Code: FFS Initialization for sample code that provides a method of FFS initialization that can detect FFS file corruption, regardless of the cause.

File State Transitions

Overview

There are three basic operations that may be done with the FFS:

- Creating a file
- Deleting a file
- Updating a file

All state transitions must be done carefully at all times to ensure that a power failure never results in a corrupted firmware volume. This transition is managed using the *State* field in the file header.

For the purposes of the examples below, positive decode logic is assumed (**EFI_FVB_ERASE_POLARITY = 0**). In actual use, the **EFI_FVB_ERASE_POLARITY** in the firmware volume header is referenced to determine the truth value of all FFS *State* bits. Note that Intel® flash memory technologies erase to one. All *State* bit transitions must be atomic operations. Further, except when specifically noted, only the most significant *State* bit that is **TRUE** has meaning. Lower-order *State* bits are superceded by higher-order *State* bits.

Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** in the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification.*

Initial State

The initial condition is that of "free space." All free space in a firmware volume must be initialized such that all bits in the free space contain the value of **EFI_FVB_ERASE_POLARITY**. As such, if the free space is interpreted as an FFS file header, all *State* bits are **FALSE**.

Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** in the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification.*

Creating a File

A new file is created by allocating space from the firmware volume immediately beyond the end of the preceding file (or the firmware volume header if the file is the first one in the firmware volume). The figure below illustrates the steps to create a new file, which are detailed below the figure.

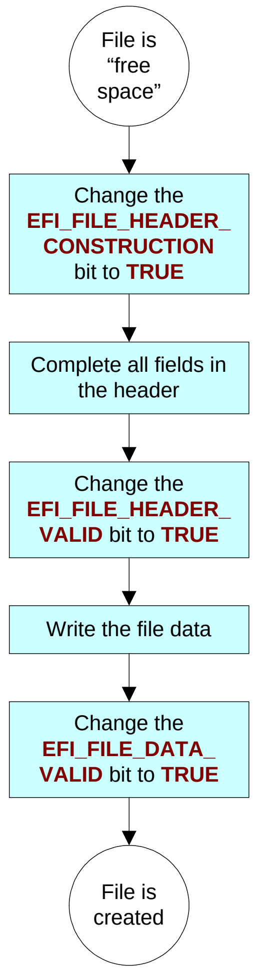

Figure 2-2. Creating a File

As shown in the figure above, the following steps are required to create a new file:

1. Allocate space in the firmware volume for a new **EFI_FFS_FILE_HEADER** and complete all fields of the header (except for the *State* field, which is updated independently from the rest of the header). This allocation is done by interpreting the free space as a file header and changing the **EFI_FILE_HEADER_CONSTRUCTION** bit to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00000001b,** which indicates the header construction has begun but has not yet been completed. This value has the effect of "claiming" the FFS header space from the firmware volume free space.

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

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

The value of *IntegrityCheck.Header* is calculated as described in **EFI_FFS_FILE_HEADER** in "Code Definitions."

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

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

   While in this state, the file data, *IntegrityCheck.File*, and the file tail are written to the firmware volume. The order in which these are written does not matter. The calculation of the values for *IntegrityCheck.File* and the file tail are described in **EFI_FFS_FILE_HEADER** and **EFI_FFS_FILE_TAIL** in "Code Definitions." If the **FFS_ATTRIB_TAIL_PRESENT** bit of the *Attributes* field is clear, the file tail does not exist. If the **FFS_ATTRIB_TAIL_PRESENT** bit of the *Attributes* field is set, the value of *IntegrityCheck.File* must be included in the calculation of the tail value.
3. Mark the data as valid. To mark the data as valid, the **EFI_FILE_DATA_VALID** bit is changed to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00000111b,** which indicates the file data is fully written and is valid.

See Updating Multiple Files in Lockstep for details on creating and updating multiple files.

Deleting a File

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

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

NOTE

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

Updating a File

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

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

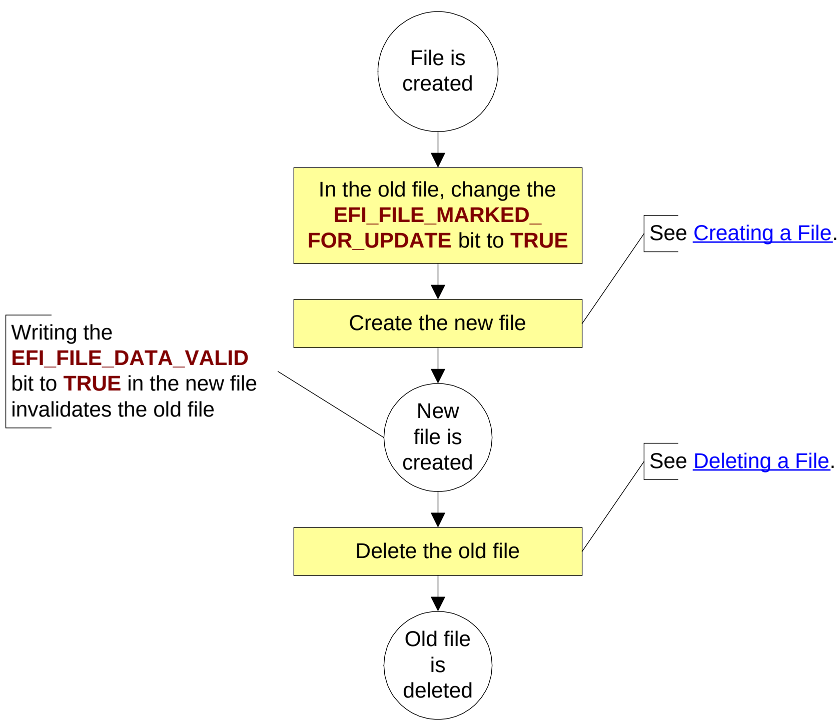

Figure 2-3. Updating a File

As shown in the figure above, the following steps are required to update a file:

1. Set the **EFI_FILE_MARKED_FOR_UPDATE** bit to **TRUE** in the old file. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00001111b,** which indicates the file is marked for update. A file in this state remains valid as long as no other file in the firmware volume has the same name and a *State* of **000001xxb.**
2. Create the new file following the steps described in Creating a File. When the new file becomes valid, the old file that was marked for update becomes invalid. That is to say, a file marked for update is valid only as long as there is no file with the same name in the firmware volume that has a *State* of **000001xxb.** In this way, only one of the files, either the new or the old, is valid at any given time. The act of writing the **EFI_FILE_DATA_VALID** bit in the new file's *State* field has the additional effect of invalidating the old file.
3. Delete the old file following the steps described in Deleting a File.

See Updating Multiple Files in Lockstep for details on creating and updating multiple files.

FFS-Defined File Types

Overview

The *Intel® Platform Innovation Framework for EFI Firmware Volume Specification* defines a number of file types and associated image formats. It also reserves file types 0xF0 to 0xFF for definition by the file system. The table below lists the FFS definitions for these file types. The rest of this section describes pad files.

Table 2-1. FFS-Defined File Types

Type	Name
0xF0	Pad file. See the Pad Files section.
0xF1–0xFF	Reserved for future use.

Pad Files (File Type 0xF0)

Pad File Overview

A *pad file* gets its name from one of its common uses. It can be used to pad the location of the file that follows it in the storage media. This padding may be done for a variety of reasons, including the following:

- Fixing the location of a file in a firmware volume
- Consuming space before a Volume Top File
- Guaranteeing data alignment for a file with the alignments bits set in the *`Attributes`* field
- Performing file update operations where multiple files within a firmware volume must be updated in lockstep

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

The **`FFS_ATTRIB_TAIL_PRESENT`** bit in the *`Attributes`* field must be clear for pad files. This restriction is because if the **`FFS_ATTRIB_TAIL_PRESENT`** bit were set, it would not be possible to reclaim the free space from the pad file (see Reclaiming Pad Free Space). Because the file is free space, an extended check of the file is simply a check for any nonfree data.

Reclaiming a Pad File's Free Space

Because a pad file's data space is not used, it is desirable to reclaim this free space for use if possible. The free space is reclaimed by using two of the pad file's *`State`* bits.

Because the data area of a pad file with *`State`* **`= 00000111b`** is guaranteed to be unperturbed free space, the conventional use of the **`EFI_FILE_MARKED_FOR_UPDATE`** bit makes no sense. In pad files, the meaning of this bit is overloaded to indicate that the data area is *not* unperturbed free space and that it may have had some data written to it. This overloading is the key to reclaiming the free space contained in a pad file. The figure below illustrates the steps to reclaim a pad file's free space, which are detailed below the figure.

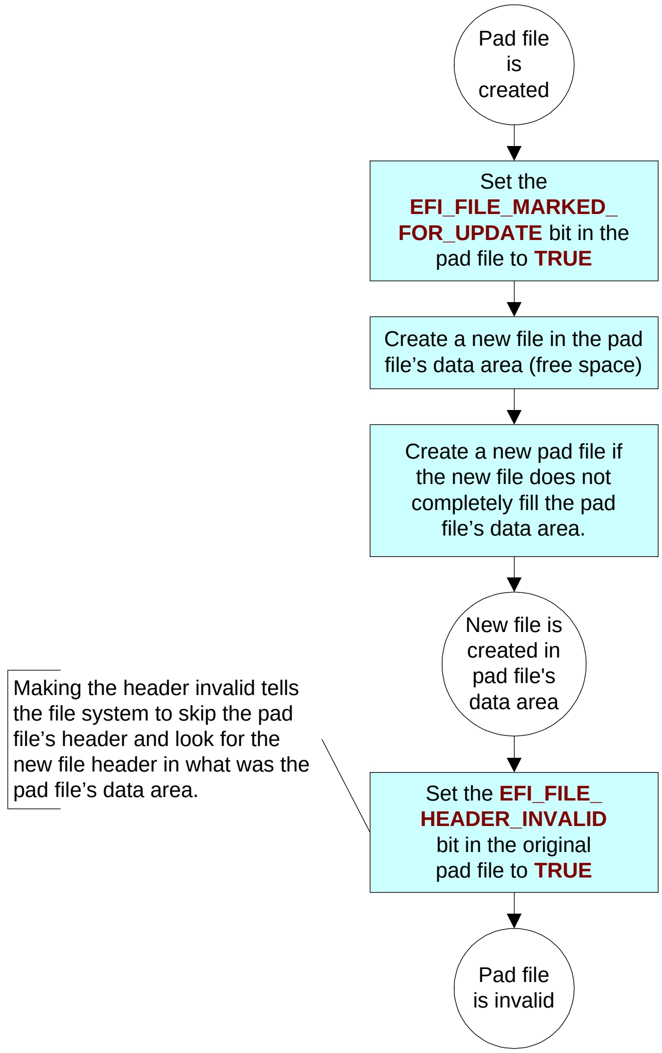

Figure 2-4. Reclaiming a Pad File's Free Space

As shown in the figure above, the following steps are required to reclaim the free space contained within a pad file:

1. Set the **EFI_FILE_MARKED_FOR_UPDATE** in the pad file to **TRUE**. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00001111b**, which indicates the pad file's data area is *not* guaranteed to be unperturbed free space.
2. Create a completely new file in the pad file's data area (free space). If the new file does not have any special alignment requirement, it is created at the lowest address within the pad file. If there *is* an alignment requirement, it may be necessary to precede the desired file with another pad file, all written to the original pad file's data area. Regardless, the new file(s) must be written completely, including the file header and data. The *State* of this file is written such that *State* **= 00000111b**. Because it is really part of the pad file's data area, it is not yet visible as part of the FFS.
3. If the new file created in step 2 does not completely fill the pad file's data area, another pad file must be created to fill this space. This file is created in the same manner as in step 2, except the beginning of the new pad file's header follows the data for the file created in step 2.
4. Set the **EFI_FILE_HEADER_INVALID** bit in the original pad file to **TRUE**. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* **= 00101111b**, which indicates the pad file's header is invalid. Because the pad file's header is now invalid, the *Length* field in the pad file's header is also no longer valid. The effect of making the header invalid is to skip only the pad file's header and look for another file header in what was the pad file's data area. Because the new file's header exists at this location, it is correctly interpreted as a valid file.

Updating a File Using a Pad File's Free Space

Updating a file using a pad file's free space is very similar to a normal file update, which is described in Updating a File in File State Transitions. The figure below illustrates the steps to update a file using a pad file's free space, which are detailed below the figure.

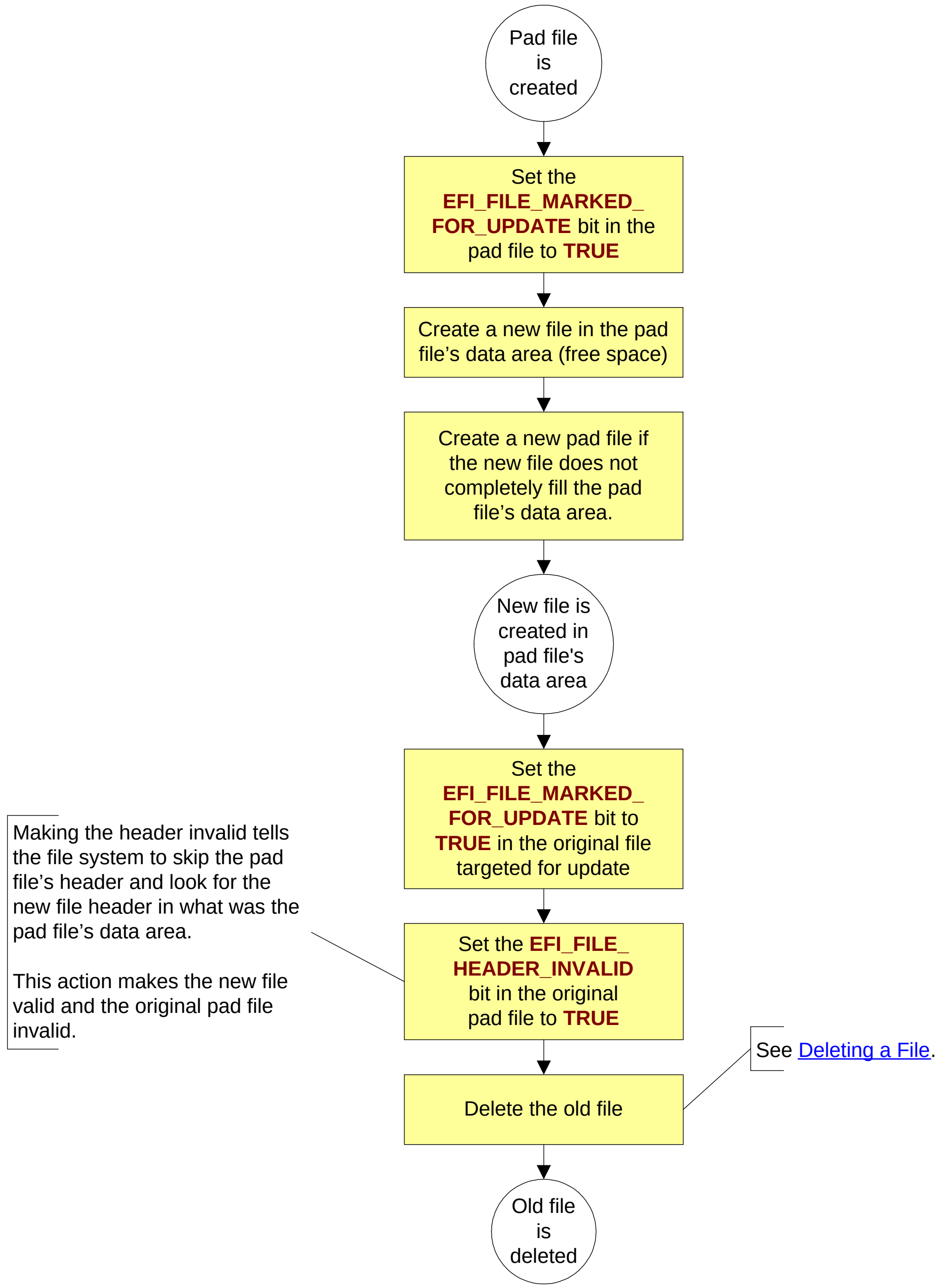

Figure 2-5. Updating a File Using a Pad File's Free Space

As shown in the figure above, the following steps are required to update a file using a pad file's free space:

1. Set the **EFI_FILE_MARKED_FOR_UPDATE** in the pad file to **TRUE.** The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* = **00001111b**, which indicates the pad file's data area is *not* guaranteed to be unperturbed free space.
2. Create a completely new file in the pad file's data area (free space) at the lowest address. If the new file has special alignment requirements, it must be handled in the same manner as in Reclaiming Pad Free Space. This new file must be written completely, including the file header and data. The *State* bit of this file is written such that *State* = **00000111b**. Because it is really part of the pad file's data area, it is not yet visible as part of the FFS.
3. If the new file created in step 2 does not completely fill the pad file's data area, another pad file must be created to fill this space. This file is created in the same manner as in step 2, except the beginning of the new pad file's header follows the data for the file created in step 2.
4. Set the **EFI_FILE_MARKED_FOR_UPDATE** bit to **TRUE** in the original file that is targeted for update. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made.
5. Set the **EFI_FILE_HEADER_INVALID** bit in the original pad file to **TRUE**. The transition of this bit to the **TRUE** state must be atomic and fully complete before any additional writes to the firmware volume are made. This transition yields *State* = **00101111b**, which indicates the pad file's header is invalid. Because the pad file's header is now invalid, the *Length* field in the pad file's header is also no longer valid. The effect of making the header invalid is to skip only the pad file's header and look for another file header in what was the pad file's data area. Because the new file's header exists at this location, it is correctly interpreted as a valid file.
6. Delete the original file that was targeted for update following the steps described in Deleting a File in File State Transitions.

Updating Multiple Files in Lockstep

It is possible to update multiple files in a single firmware volume in lockstep using the technique described in Updating a File Using a Pad File's Free Space. To update multiple files, write multiple files to the pad file's data area in step 2. Then mark all of the corresponding original files in step 5 and delete them in step 6. A pad file can be created explicitly for this purpose.

Volume Top File

A Volume Top File (VTF) is a file that must be located such that the last byte of the file is also the last byte of the firmware volume. Regardless of the file type, a VTF must have the file name GUID of **EFI_FFS_VOLUME_TOP_FILE_GUID**. See **EFI_FFS_VOLUME_TOP_FILE_GUID** in "Code Definitions" for the GUID definition.

FFS driver code must be aware of this GUID and insert a pad file as necessary to guarantee the VTF is located correctly at the top of the firmware volume on write and update operations. File length and alignment requirements must be consistent with the top of volume. Otherwise, a write error occurs and the firmware volume is not modified.

3
Code Definitions

Introduction

This section provides the code definitions for the following data types and structures for the FFS. Some type definitions are not in their own section and can be found in “Related Definitions” of the parent data structure definition.

- **EFI_FIRMWARE_FILE_SYSTEM_GUID**
- **EFI_FFS_FILE_HEADER**
- **EFI_FFS_INTEGRITY_CHECK**
- **EFI_FFS_FILE_ATTRIBUTES**
- **EFI_FFS_FILE_STATE**
- **EFI_FFS_FILE_TAIL**
- **EFI_FV_FILETYPE_FFS_PAD**
- **EFI_FFS_VOLUME_TOP_FILE_GUID**

File Format

EFI_FIRMWARE_FILE_SYSTEM_GUID

Summary

The firmware volume header contains a data field for the file system Globally Unique Identifier (GUID). See the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification* for more information on the firmware volume header. For the FFS file system, the GUID is defined below.

GUID

```
// 7A9354D9-0468-444a-81CE-0BF617D890DF

#define EFI_FIRMWARE_FILE_SYSTEM_GUID \
{ 0x7A9354D9, 0x0468, 0x444a, 0x81, 0xCE, 0x0B, 0xF6  \
  0x17, 0xD8, 0x90, 0xDF  }
```

EFI_FFS_FILE_HEADER

Summary

Each file begins with a header that describes the state and contents of the file. The header is 8 bytes aligned with respect to the beginning of the firmware volume.

Prototype

```
typedef struct {
  EFI_GUID                   Name;
  EFI_FFS_INTEGRITY_CHECK    IntegrityCheck;
  EFI_FV_FILETYPE            Type;
  EFI_FFS_FILE_ATTRIBUTES    Attributes;
  UINT8                      Size[3];
  EFI_FFS_FILE_STATE         State;
} EFI_FFS_FILE_HEADER;
```

Parameters

Name

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

IntegrityCheck

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

Type

Identifies the type of file. Type **EFI_FV_FILETYPE** is defined in the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification.* FFS-specific file types are defined in **EFI_FV_FILETYPE_FFS_PAD**.

Attributes

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

Size

The length of the file in bytes, including the FFS header and file tail if it exists. The length of the file data is either **(*Size* – sizeof(EFI_FFS_FILE_HEADER))** or **(*Size* – sizeof(EFI_FFS_FILE_HEADER) – sizeof(EFI_FFS_FILE_TAIL))** depending on the existence of the file tail. This calculation means a zero-length file has a *Size* of 24 bytes, which is **sizeof(EFI_FFS_FILE_HEADER)**.

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

State

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

Related Definitions

```
//*******************************************************
// EFI_FFS_INTEGRITY_CHECK
//*******************************************************
typedef union {
  struct {
    UINT8             Header;
    UINT8             File;
  }                   Checksum;
  UINT16              TailReference;
} EFI_FFS_INTEGRITY_CHECK;
```

Header

The *IntegrityCheck.Checksum.Header* field is an 8-bit checksum of the file header. The *State* and *IntegrityCheck.Checksum.File* fields are assumed to be zero and the checksum is calculated such that the entire header sums to zero. The *IntegrityCheck.Checksum.Header* field is valid anytime the **EFI_FILE_HEADER_VALID** bit is set in the *State* field. See FFS File Integrity and State for more details.

File

If the **FFS_ATTRIB_CHECKSUM** (see definition below) bit of the *Attributes* field is set to one, the *IntegrityCheck.Checksum.File* field is an 8-bit checksum of the entire file The *State* field and the file tail are assumed to be zero and the checksum is calculated such that the entire file sums to zero.

If the **FFS_ATTRIB_CHECKSUM** bit of the *Attributes* field is cleared to zero, the *IntegrityCheck.Checksum.File* field must be initialized with a value of 0x55AA.

The *IntegrityCheck.Checksum.File* field is valid any time the **EFI_FILE_DATA_VALID** bit is set in the *State* field. See FFS File Integrity and State for more details.

TailReference

IntegrityCheck.TailReference is the full 16 bits of the *IntegrityCheck* field. It is used in calculating the value for the file tail if the **FFS_ATTRIB_TAIL_PRESENT** bit in the *Attributes* field is set. See **EFI_FFS_FILE_TAIL** for more details.

```
//*******************************************************************
// EFI_FFS_FILE_ATTRIBUTES
//*******************************************************************

typedef UINT8 EFI_FFS_FILE_ATTRIBUTES;

// FFS File Attributes
#define FFS_ATTRIB_TAIL_PRESENT         0x01
#define FFS_ATTRIB_RECOVERY             0x02
#define FFS_ATTRIB_HEADER_EXTENSION     0x04
#define FFS_ATTRIB_DATA_ALIGNMENT       0x38
#define FFS_ATTRIB_CHECKSUM             0x40
```

The figure below depicts the bit allocation of the `Attributes` field in an FFS file's header.

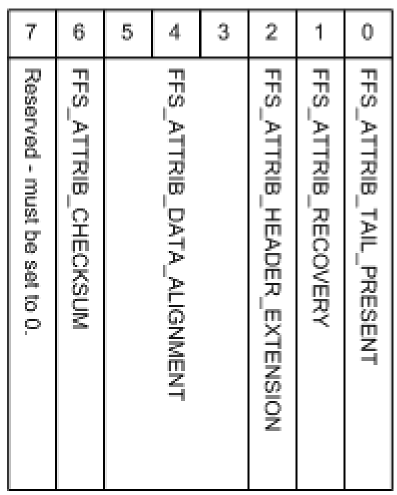

Figure 3-1. Bit Allocation of FFS `Attributes`

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

FFS_ATTRIB_TAIL_PRESENT	Indicates the 16-bit file tail at the end of the file exists. See **EFI_FFS_FILE_TAIL** for details.
FFS_ATTRIB_RECOVERY	Indicates this file is required to execute a crisis recovery.
FFS_ATTRIB_HEADER_EXTENSION	Reserved for use by future revisions of this specification. It must be set to zero.

FFS_ATTRIB_DATA_ALIGNMENT	Indicates that the beginning of the data must be aligned on a particular boundary relative to the firmware volume base. The three bits in this field are an enumeration of alignment possibilities. The firmware volume interface allows alignments based on powers of two from byte alignment to 64 KB alignment. FFS does not support this full range. The table below maps all FFS supported alignments to **`FFS_ATTRIB_DATA_ALIGNMENT`** values and firmware volume interface alignment values. No other alignments are supported by FFS. When a file with an alignment requirement is created, a pad file may need to be created before it to ensure proper data alignment. See Pad Files (File Type 0xF0) for more information regarding pad files.
FFS_ATTRIB_CHECKSUM	Determines the interpretation of *`IntegrityCheck.Checksum.File.`* See the *`IntegrityCheck`* definition above for specific usage.

The table below maps all FFS-supported alignments to **`FFS_ATTRIB_DATA_ALIGNMENT`** values and firmware volume interface alignment values.

Table 3-1. Supported FFS Alignments

Required Alignment (bytes)	Alignment Value in FFS *`Attributes`* Field	Alignment Value in Firmware Volume Interfaces
1	0	0
2	0	1
4	0	2
8	0	3
16	1	4
128	2	7
512	3	9
1 KB	4	10
4 KB	5	12
32 KB	6	15
64 KB	7	16

```
//*******************************************************************
// EFI_FFS_FILE_STATE
//*******************************************************************

typedef UINT8 EFI_FFS_FILE_STATE;

// FFS File State Bits
#define EFI_FILE_HEADER_CONSTRUCTION      0x01
#define EFI_FILE_HEADER_VALID             0x02
#define EFI_FILE_DATA_VALID               0x04
#define EFI_FILE_MARKED_FOR_UPDATE        0x08
#define EFI_FILE_DELETED                  0x10
#define EFI_FILE_HEADER_INVALID           0x20
```

All other *State* bits are reserved and must be set to **EFI_FVB_ERASE_POLARITY**. See FFS File Integrity and State for an explanation of how these bits are used. Type **EFI_FVB_ERASE_POLARITY** is defined in **EFI_FIRMWARE_VOLUME_HEADER** in the *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification.*

EFI_FFS_FILE_TAIL

Summary

The tail follows the data and is the last two bytes of the file's image in the storage media. The tail is used for file integrity checking and is present only when the **FFS_ATTRIB_TAIL_PRESENT** bit in the *Attributes* field of the file's header is set. The file tail is optional and is never required. It must never be present in zero-length files and pad files.

Prototype

```
typedef UINT16    EFI_FFS_FILE_TAIL;
```

Description

If the **FFS_ATTRIB_TAIL_PRESENT** bit is set, the tail is initialized to the bit-wise **NOT** of the header's *IntegrityCheck.TailReference* field.

Pad Files

EFI_FV_FILETYPE_FFS_PAD

Summary

A pad file is an FFS-defined file type that is used to pad the location of the file that follows it in the storage file.

Prototype

```
#define EFI_FV_FILETYPE_FFS_PAD        0xF0
```

Description

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

The **FFS_ATTRIB_TAIL_PRESENT** bit in the *Attributes* field must be clear for pad files. This restriction is because if the **FFS_ATTRIB_TAIL_PRESENT** bit were set, it would not be possible to reclaim the free space from the pad file (see Reclaiming Pad Free Space). Because the file is free space, an extended check of the file is simply a check for any nonfree data.

Volume Top File

EFI_FFS_VOLUME_TOP_FILE_GUID

Summary

A Volume Top File (VTF) is a file that must be located such that the last byte of the file is also the last byte of the firmware volume. Regardless of the file type, a VTF must have the file name GUID of **EFI_FFS_VOLUME_TOP_FILE_GUID** as defined below.

GUID

```
// {1BA0062E-C779-4582-8566-336AE8F78F09}

#define EFI_FFS_VOLUME_TOP_FILE_GUID \
{ 0x1BA0062E, 0xC779, 0x4582, 0x85, 0x66, 0x33, 0x6A, \
  0xE8, 0xF7, 0x8F, 0x9 };
```

4 Pseudo Code

FFS Initialization

The algorithm below describes a method of FFS initialization that ensures FFS file corruption can be detected regardless of the cause.

The `State` byte of each file must be correctly managed to ensure the integrity of the file system is not compromised in the event of a power failure during any FFS operation. It is expected that an FFS driver will produce an instance of the Firmware Volume Protocol and that all normal file operations will take place in that context. All file operations must follow all the creation, update, and deletion rules described in this specification to avoid file system corruption. See the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification* for the definition of the Firmware Volume Protocol.

The following **`FvCheck()`** pseudo code must be executed during FFS initialization to avoid file system corruption. If at any point a failure condition is reached, then the firmware volume is corrupted and a crisis recovery is initiated.

```
// Firmware volume initialization entry point – returns TRUE
// if FFS driver can use this firmware volume.
BOOLEAN FvCheck(Fv)
{
  // first check out firmware volume header
  if (FvHeaderCheck(Fv) == FALSE) {
    FAILURE();// corrupted firmware volume header
  }
  if (Fv->FvFileSystemId != EFI_FIRMWARE_FILE_SYSTEM_GUID) {
    return (FALSE);  // This firmware volume is not
                     // formatted with FFS
  }

  // next walk files and verify the FFS is in good shape
  for (FilePtr = FirstFile; Exists(Fv, FilePtr);
          FilePtr = NextFile(Fv, FilePtr)) {
    if (FileCheck (Fv, FilePtr) != 0) {
      FAILURE(); // inconsistent file system
    }
  }
  if (CheckFreeSpace (Fv, FilePtr) != 0) {
    FAILURE();
  }
  return (TRUE);     // this firmware volume can be used by the FFS
                     // driver and the file system is OK
}
```

```
// FvHeaderCheck – returns TRUE if FvHeader checksum is OK.
BOOLEAN FvHeaderCheck (Fv)
{
  return (Checksum (Fv.FvHeader) == 0);
}

// Exists – returns TRUE if any bits are set in the file header
BOOLEAN Exists(Fv, FilePtr)
{
  return (BufferErased (Fv.ErasePolarity,
              FilePtr, sizeof (EFI_FIRMWARE_VOLUME_HEADER) == FALSE);
}

// BufferErased – returns TRUE if no bits are set in buffer
BOOLEAN BufferErased (ErasePolarity, BufferPtr, BufferSize)
{
  UINTN  Count;

  if (Fv.ErasePolarity == 1) {
    ErasedByte = 0xff;
  } else {
    ErasedByte = 0;
  }
  for (Count = 0; Count < BufferSize; Count++) {
    if (BufferPtr[Count] != ErasedByte) {
      return FALSE;
    }
  }
  return TRUE;
}

// GetFileState – returns high bit set of state field.
  UINT8 GetFileState (Fv, FilePtr) {
  UINT8 FileState;
  UINT8 HighBit;

  FileState = FilePtr->State;
  if (Fv.ErasePolarity != 0) {
    FileState = ~FileState;
  }
  HighBit = 0x80;
  while (HighBit != 0 && (HighBit & FileState) == 0) {
    HighBit = HighBit >> 1;
  }
  return HighBit;
}
```

```
// FileCheck – returns TRUE if the file is OK
BOOLEAN FileCheck (Fv, FilePtr) {
  switch (GetFileState (Fv, FilePtr)) {
    case EFI_FILE_HEADER_CONSTRUCTION:
      SetHeaderBit (Fv, FilePtr, EFI_FILE_HEADER_INVALID);
      break;
    case EFI_FILE_HEADER_VALID:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
      Break;
    case EFI_FILE_DATA_VALID:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (DuplicateFileExists (Fv, FilePtr,
                      EFI_FILE_DATA_VALID) != NULL) {
        return (FALSE);
      }
      break;
    case EFI_FILE_MARKED_FOR_UPDATE:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (FilePtr->State & EFI_FILE_DATA_VALID) == 0) {
        return (FALSE);
      }
      if (FilePtr->Type == EFI_FV_FILETYPE_FFS_PAD) {
        SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
      }
      else {
        if (DuplicateFileExists (Fv, FilePtr, EFI_FILE_DATA_VALID)) {
          SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
        }
        else {
          if (Fv->Attributes & EFI_FVB_STICKY_WRITE) {
            CopyFile (Fv, FilePtr);
            SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
          }
          else {
            ClearHeaderBit (Fv, FilePtr, EFI_FILE_MARKED_FOR_UPDATE);
          }
        }
      }
      break;
```

```
    case EFI_FILE_DELETED:
      if (VerifyHeaderChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      if (VerifyFileChecksum (FilePtr) != TRUE) {
        return (FALSE);
      }
      break;
    case EFI_FILE_HEADER_INVALID:
      break;
  }
  return (TRUE);
}

// FFS_FILE_PTR * DuplicateFileExists (Fv, FilePtr, StateBit)
//    This function searches the firmware volume for another occurrence
//    of the file described by FilePtr in which the duplicate files
//    high state bit that is set is defined by the parameter StateBit.
//    It returns a pointer to a duplicate file if it exists and NULL
//    if it does not.

// CopyFile (Fv, FilePtr)
//    This real purpose of this function is to clear the
//    EFI_FILE_MARKED_FOR_UPDATE bit from FilePtr->State
//    in firmware volumes that have EFI_FVB_STICKY_WRITE == TRUE.
//    The file is copied exactly header and all, except the
//    EFI_FILE_MARKED_FOR_UPDATE bit in the file header of the
//    new file is clear.

// VerifyHeaderChecksum (FilePtr)
//    This purpose of this function is to verify the file header
//    sums to zero. See IntegrityCheck.Checksum.Header definition
//    for details.

// VerifyFileChecksum (FilePtr)
//    This purpose of this function is to verify the file integrity
//    check. See IntegrityCheck.Checksum.File definition for details.
//    It also verifies the file tail.
```

Pre-FFS Initialization Access to Files

The Security (SEC), Pre-EFI Initialization (PEI), and early Driver Execution Environment (DXE) code must be able to traverse the FFS and read and execute files before a write-enabled DXE FFS driver is initialized. Because the FFS may have inconsistencies due to a previous power failure or other system failure, it is necessary to follow a set of rules to verify the validity of files prior to using them. It is not incumbent on SEC, PEI, or the early read-only DXE FFS services to make any attempt to recover or modify the file system. If any situation exists where execution cannot continue due to file system inconsistencies, a recovery boot is initiated.

There is one inconsistency that the SEC, PEI, and early DXE code can deal with without initiating a recovery boot. This condition is created by a power failure or other system failure that occurs during a file update on a previous boot. Such a failure will cause two files with the same file name GUID to exist within the firmware volume. One of them will have the **`EFI_FILE_MARKED_FOR_UPDATE`** bit set in its *`State`* field but will be otherwise a completely valid file. The other one may be in any state of construction up to and including **`EFI_FILE_DATA_VALID`**. All files used prior to the initialization of the write-enabled DXE FFS driver *must* be screened with this test prior to their use. If this condition is discovered, it is permissible to initiate a recovery boot and allow the recovery DXE to complete the update.

The following pseudo code describes the method for determining which of these two files to use. The inconsistency is corrected during the write-enabled initialization of the DXE FFS driver.

```
// Screen files to ensure we get the right one in case
// of an inconsistency.
FFS_FILE_PTR EarlyFfsUpdateCheck(FFS_FILE_PTR * FilePtr) {
  FFS_FILE_PTR * FilePtr2;

  if (VerifyHeaderChecksum (FilePtr) != TRUE) {
    return (FALSE);
  }
  if (VerifyFileChecksum (FilePtr) != TRUE) {
    return (FALSE);
  }
  switch (GetFileState (Fv, FilePtr)) {
    case EFI_FILE_DATA_VALID:
      return (FilePtr);
      break;
    case EFI_FILE_MARKED_FOR_UPDATE:
      FilePtr2 = DuplicateFileExists (Fv, FilePtr,
                                      EFI_FILE_DATA_VALID);
      if (FilePtr2 != NULL) {
        if (VerifyHeaderChecksum (FilePtr) != TRUE) {
          return (FALSE);
        }
        if (VerifyFileChecksum (FilePtr) != TRUE) {
          return (FALSE);
        }
        return (FilePtr2);
      } else {
        return (FilePtr);
      }
      break;
  }
}
```

NOTE

There is no check for duplicate files once a file in the **EFI_FILE_DATA_VALID** *state is located. The condition where two files in a single firmware volume have the same file name GUID and are both in the* **EFI_FILE_DATA_VALID** *state cannot occur if the creation and update rules that are defined in this specification are followed.*