Intel® Platform Innovation Framework for EFI Capsule 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

1 Introduction

Overview

This document describes the process by which data can be passed from runtime (operating system [OS]–present) applications back to the preboot phases (Pre-EFI Initialization [PEI], Driver Execution Environment [DXE], and Boot Device Selection [BDS]) in an implementation of the Intel® Platform Innovation Framework for EFI (hereafter referred to "the Framework"), using a formatted variable-length data structure known as a *capsule*. Capsules were designed for and are intended to be the major vehicle for delivering firmware volume changes. There is no requirement that capsules be limited to that function.

In many ways, the design of capsules builds on, and can be considered an extension of, the firmware volume design. The format of the capsule is very similar to the format of a firmware volume, allowing the same tools to create both formats and the same (or slightly upgraded versions of the same) protocols to manipulate both.

This document does not describe the mechanism for modifying system variables. These mechanisms are described in various EFI specifications. Capsules are, however, a mechanism by which variables that are specific to a single firmware volume (for example, policy data concerning an add-in device) can be manipulated.

Goals

The capsule mechanism was designed with the following goals:

- **Generic Solution:** Capsules provide a generic mechanism for transitioning data from the OS to the boot mechanism. While an update is the main target of capsules, other functionality may be implemented, particularly by passing drivers as part of the capsule.
- **Commonality:** The mechanism should be able to support all firmware devices from all vendors whose firmware is compliant with the requirements of the mechanism, thus providing a common means to update platforms over systems running Framework-based firmware.
- **Security:** Capsules allow the levels of security for the update to be tuned to the security requirements of the system.
- **Abstraction:** The various components do not have to be aware of each other's implementations. For example, an OS-present update application should not have to be aware of the technologies that are used to implement the firmware devices.
- **Distribution control:** Various types of systems have various support models. The mechanism should allow control over who decides what updates are allowed.
- **OS-Present Update:** The user should be able to perform the job of requesting the update in the user's most familiar environment.

- **Atomic Operation:** An update to a particular feature may require updates to several files. Allowing the files to be grouped into a single monolithic (from the end-user's point of view) entity allows testing to be focused on the entire update rather than on each individual piece.
- **Recovery Support:** Capsules should simplify the effort that is required to implement mechanisms to recover corrupted firmware storage.

Theory of Operation

Capsules provide a number of benefits, including the following:

- Firmware volumes that are provided by capsules are atomic from the point of view of the end-user and the capsule processor utility. This grouping means that several related actions can be performed by a single capsule. As a result, a user cannot pick and choose among the various partial updates. This constraint limits testing to manageable levels.
- Capsules are self describing and use a standard format to allow the following:
  - Common tools can be used to create them.
  - Common applications can be used to manipulate them for update.
  - Common DXE drivers can be used to pick them apart to perform the capsule's actions.
- Capsules provide a transport mechanism for drivers and other data that is not part of the update image. For example, this mechanism allows the code to write to particular devices that might only be part of the capsule and not take up valuable (costly) firmware space in firmware devices such as on add-in cards.
- Capsules can carry descriptive user information that would take up too much space to be tolerated in many firmware devices.
- Capsules provide a location for update-related security data to be carried. The update process localizes the places where changes to firmware volumes take place, and hence the places where validation of such changes is required.
- Capsules can carry update programs that would be too large to fit in firmware. For example, in most applications, the program that provides Pre-EFI Initialization Module (PEIM) relocation can be part of the capsule rather than stored as a part of the firmware itself. Also, in the case of add-in firmware volumes that do not store EFI variables, all firmware update code can be provided as drivers in the capsules.

Definitions of Terms

The following definitions are specific to this document or are not otherwise in common usage.

capsule

A formatted variable-length data structure that is used to pass data from runtime (OS-present) applications back to the preboot (PEI, DXE, BDS) phases.

capsule driver

The DXE driver that is responsible for manipulating capsules.

capsule update

A special boot path in which a capsule can be loaded into the system memory at OS runtime and then executed following a processor INIT or warm reset.

capsule volume

A data structure that is similar to a firmware volume and forms the “outer layer” of the capsule.

capsule volume HOB

The PEI Hand-Off Block (HOB) that describes a capsule or set of capsules in memory.

firmware volume file

A firmware volume that is contained inside a file. The capsule volume consists of a firmware volume file and, possibly, some files added by an OS-present application.

mailbox (or update mailbox)

A memory-resident data structure into which the update OS-present application writes an unedited capsule list.

Section Extraction Protocol

A driver-defined protocol that is used in coordination with the GUIDed section type to enable extensions to the types of sections that constitute files.

split capsule

A capsule that is divided into more than one piece. Split capsules are used in cases where the size of the equivalent nonsplit capsule is larger than the size of the media in which it is to be transported.

subcapsule

A constituent piece of a split capsule. Each subcapsule has the same format as a (nonsplit) capsule.

Impacts on Other EFI Specifications

This specification describes one impact to existing EFI specifications and how those specifications are to be used.

To pass application-derived parameters from the application into DXE, this design requires that OS-present applications be able to perform write operations to what amounts to a firmware volume. Consideration should be given to creating libraries that abstract the writes so that the implementation of the firmware volume is kept parallel between OS-present applications and DXE.

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

Capsule Overview

Introduction

This section describes capsules at a high level. Subsequent sections delve deeper into the various topics discussed here. Figure 2-1 shows the format of a sample capsule.

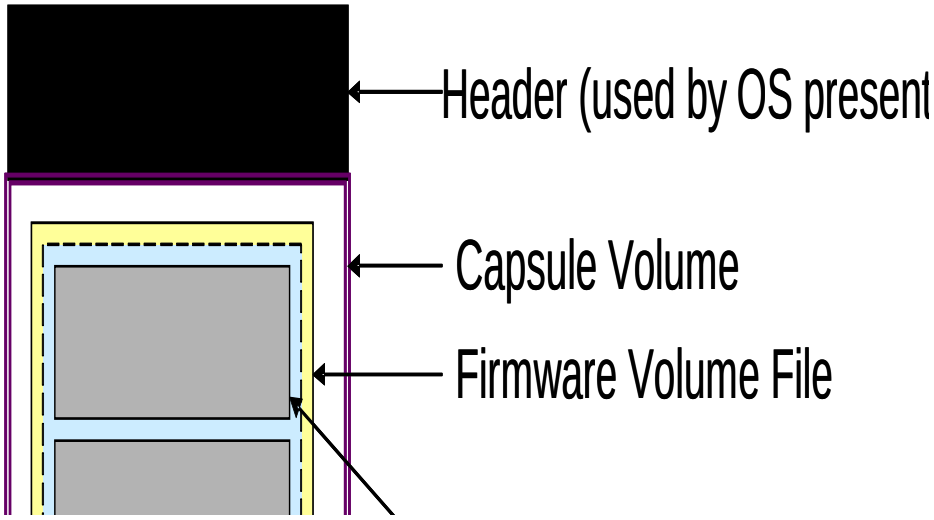

Figure 2-1. Sample Capsule Format

Capsule Contents

A capsule consists of a *capsule header* and a *capsule volume*, which are described in the following subsections.

Capsule Header

The capsule header is the part of the capsule that is used by an OS-present application. It contains the following types of information:

- **Informative text:** This information allows the OS-present application to describe the contents of the capsule to the end user. The text is in Unicode and can support several different languages.
- **Original equipment manufacturer (OEM)–specific information:** The capsule header may also contain an OEM-specific header.

See “Capsule Header Discussion” later in this chapter for additional discussion of the capsule header. See chapter 3, “Code Definitions,” for the definition of the capsule header.

Capsule Volume

A capsule volume has the same basic structure as a standard firmware volume. Capsule volumes, however, have more predictably defined formats than the more highly extensible firmware volume format.

A capsule volume consists of the following files:

- **Firmware volume file:** This file contains a firmware volume image section. The section contains a correctly formatted firmware volume. This file is the base part of the capsule. It is "encapsulated" inside a capsule volume so that the contents of the firmware volume can pass through whatever security "hoops" that are required by the platform's policy before the contents (the drivers in particular) are treated as equal to the other drivers in the firmware. The developer of the firmware volume file must leave space for the other files that might accompany it in the capsule volume.
- **Configuration results file:** This file is written by the OS-present application and provides the results from the Setup program. The file is a string of Internal Forms Representation (IFR) name, value pairs as defined in the *Intel® Platform Innovation Framework for EFI Human Interface Infrastructure Specification.* Flags in the capsule header indicate support for this file by the capsule.

See "Common File Formats" in chapter 3, "Code Definitions," for definitions of these file types.

Capsule Processing

Capsule processing is the effort that is expended by the OS-present application to allow its drivers to be dispatched by the DXE Dispatcher. The following subsections define the responsibilities at each "stop" along the path.

OS- Present Application

The OS-present application is responsible for locating the capsule and prompting the user for data such as path and setup responses. The OS-present application is then responsible for loading the capsule into nonswapped memory according to the data structure described in **`EFI_CAPSULE_HEADER`**. The structure supports spanning capsules across noncontiguous blocks and multiple capsules. The memory that is used to store the mailbox must follow several requirements as defined in "Mailbox Format" later in this chapter.

The OS-present application then causes the OS to cleanly shut down and call the EFI Runtime Service **`ResetSystem()`** with the pointer to the base of the capsule data structure as its parameter. On a multiprocessor system (MP), all processors must proceed to the shutdown. The following sections assume that all except one processor are inactive.

Note that, in some cases, multiple OS-present applications might need to cooperate to accomplish these tasks.

EFI Runtime Service ResetSystem()

The EFI Runtime Service **`ResetSystem()`** stores the base address in a platform-defined location. This storage requires cooperation between the **`ResetSystem()`** call and a counterpart platform PEIM.

`ResetSystem()` then causes a nonvolatile reset of the system. By nonvolatile, it is meant that the main system RAM is preserved over the reset. This memory is being used as a large mailbox to ferry the capsule from the OS-present application to DXE.

PEI

In the PEI phase, the capsule is processed by the capsule PEIM. The capsule PEIM's responsibility is to locate the capsule using the data that was stored by the EFI Runtime Service **`ResetSystem()`**. The capsule PEIM is then responsible for making the probably noncontiguous parts of the capsules into a contiguous memory block that does not collide with other blocks that are used by PEI and DXE. It describes this block by creating a capsule volume HOB that describes the block's location and length.

DXE and BDS

BDS is responsible for initiating and driving the process of extracting the capsule from the capsule volume HOB and turning that data into firmware volume(s) that can then be used by DXE and subsequent phases. The process sequence is as follows:

1. BDS tries to locate the capsule volume HOB.
2. If the capsule volume HOB is found, it processes each capsule in the HOB by using the DXE Service **`ProcessFirmwareVolumes()`** to cause the capsule volume to be made visible as a firmware volume.
3. **`ProcessFirmwareVolumes()`** also expands firmware volume files as their own firmware volumes.
4. BDS then attempts to ensure that the required devices have been started using a process that is identical to the process used to ensure that boot paths are enabled.
5. After completion, BDS returns to DXE to enable processing of the newly created firmware volumes.

Security Infrastructure

Although it is not required, there are good reasons to store the capsule's executables inside a firmware volume file inside the capsule.

The main reason to store the capsule's executables in a firmware volume file is to support various security implementations. In particular, the DXE Service **`ProcessFirmwareVolumes()`** uses section extraction services to extract firmware volumes. If GUIDed wrapper sections surround the firmware volume image section, the normal process of reading the image will cause the functions inside protocols with that GUID to be invoked. One use of those functions is to validate the images, which changes the security status of the new firmware volume accordingly.

The security status of firmware volumes is important because, among other reasons, that status can be used to control whether drivers contained in that firmware volume are dispatched.

An example implementation might, thus, implement a security scheme where BDS sets the security status of the capsule volume itself to untrusted and then uses GUIDed wrapper sections to validate the trustworthiness of internal images. If the image passes validation, it would be assigned trusted status. The Security Architectural Protocol might then be used to configure the DXE Dispatcher to dispatch only drivers from trusted volumes. Note that drivers that are resident in onboard firmware volumes would, in this example, be assumed to be trusted.

Firmware volumes that are created from firmware volume files may locate their parent volumes (the volumes from which they were extracted) using functions that are provided in their (the child volume's) Firmware Volume Block Protocol. Applications can use this scheme to locate data files such as configuration results files that reside outside the security wrapper.

The same protocols and infrastructure that are used for normal files are thus used for capsules, which makes this architecture efficient from the view of size and testing.

See these specifications for additional information on the following:

- Firmware Volume Block Protocol: *Intel® Platform Innovation Framework for EFI Firmware Volume Block Specification*
- GUIDed sections and section extraction protocols: *Intel® Platform Innovation Framework for EFI Firmware Volume Specification*
- Security Architectural Protocol: *Intel® Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification* (DXE CIS)

Capsule Header Discussion

The capsule header provides an interface between the capsule developers and the typically OS-present applications that process the capsule. The OS-present applications use the human-readable text to describe the capsule to the user.

The header is also used by the developer to indicate to the OS-present application the parameters that are required for the capsule to be useful in the subsequent preboot space. There are several parameters that may be used to update configuration information for devices on the platform. The update information is represented inside EFI in the form of IFR forms and their resultant name-value pair strings. The OS-present application uses the **`DEVDESC=`** name/value pairs in the IFR results to associate the results with the relevant firmware. The results are written as a file into the capsule volume.

Application Requirements

Introduction

The OS-present program that provides the capsule to the EFI Runtime Service **`ResetSystem()`** is known as the *update application* or simply the *application*, which is probably more appropriate because “update” presupposes a usage model.

This section describes the OS-present interfaces that are used to implement the process by which the capsule is delivered. It does not describe a particular implementation of an example update application because that implementation is correctly the province of developers familiar with the particular operating environment and, hence, is outside the scope of this document.

The requirements in the following sections do place some limitations and requirements on the operating environment. How those requirements are met is again up to the developer of the update utility and operating environment.

Split Capsules

For a variety of reasons, particularly media size constraints in certain environments, it is possible that a capsule might need to be broken up into component pieces or *subcapsules*. The application is then responsible for coalescing and reordering these subcapsules into a unified whole.

Each subcapsule has the same structure as a full capsule. The only difference is that the contents of the capsule body are shortened. Thus, the way to detect that a capsule is a subcapsule is to note that the capsule body is shorter than the header’s ***`CapsuleImageSize`***. The application determines that it has all of the subcapsules that are required to reconstitute the complete capsule when the sum of all of the lengths of the capsule bodies equals the ***`CapsuleImageSize`***. It is invalid for ***`CapsuleImageSize`*** to vary between component subcapsules of the same capsule.

The application determines if the subcapsules that are presented to it are components of the same capsule by validating that the ***`InstanceId`*** GUIDs in each of the subcapsules are identical and not equal to zero (a value that may be present only in a nonsplit capsule).

The application determines the order of the subcapsules by using the ***`SequenceNumber`*** in the capsule header. This value must start with zero and increase by ones (0, 1, 2, …), with lower numbers indicating earlier subcapsules.

NOTE

Because it is possible for files that contain subcapsules—particularly those with ***`SequenceNumber`*** *values other than 0—to be presented in any order, it is required that all subcapsules retain valid descriptive text fields. Depending on the application, it might be valid to reject subcapsules that do not start with* ***`SequenceNumber`*** *values in other than increasing order.*

The ***`OffsetToSplitInformation`*** string is intended to be used by the application to describe the format in which the capsule appears. In a common example, this string would include the names of the subcapsule files.

Mailbox Format

Capsule contents are communicated by the OS-present application to PEI/DXE using an *update mailbox* (or simply *mailbox*). The mailbox is a memory-resident data structure into which the update application writes an unedited capsule list.

The mailbox is a two-level structure. The lower level consists of blocks of data. The higher level is known as the *block directory* and consists of a series of mainly contiguous structures, each of which has the format defined in "Mailbox Format" in chapter 3, "Code Definitions."

When the EFI Runtime Service **`ResetSystem()`** is invoked, all parts of the mailbox must be in memory (not swapped out). In addition, the pointers in each block must be valid in the mode in which PEI and DXE run during their phases. This requirement usually means that they must be "physical" or "linear" pointers rather than the virtual pointers with which OSs more commonly work. It is expected that the mailbox memory will be generally allocated out of memory that the OS uses for memory-mapped I/O because this memory is physically addressed and nonswapped.

In most instruction sets and OS architectures, allocation of physical memory is possible only on a "page" granularity (which can range from 4 KB to at least 1 MB). The mailbox structure is defined to abstract the page size from DXE while still enabling the update utility and OS to allocate noncontiguous pages for both data blocks and the block directory. The following requirements ensure the safe and well-defined transmission of data:

- Each new capsule must start on a new page of memory.
- All pages except for the last must be completely filled by the capsule.
- The last page must have at least one byte of capsule in it.
- The mailbox "page size" for any particular instruction set must be a constant but can vary from instruction set to instruction set.
- Pages must be naturally aligned.
- Pages may not overlap one another.
- Pages may not address wrap.

Figure 2-2 below shows a capsule that is divided into 4 data blocks. The block directory is shown as consisting of two sections, with three data block pointers followed by a continuation pointer (the length of 1 is not shown) to a block directory with one block pointer and an end-of-list indicator (length of 0). Note that the blocks do not have to be in any particular order in the memory space.

Multiple capsules are supported by creating each new capsule separately. Once created, a continuation pointer record that points to the previous capsule is appended to the end of the new capsule's block descriptor list. That is, each new capsule must begin with a new block directory. Data inside the capsule volume, along with the page alignment requirements above, is sufficient to determine where one capsule ends and the next begins.

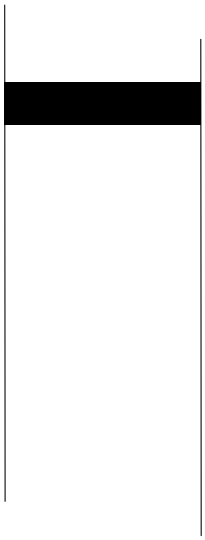

Figure 2-2. Capsule Continuation Pointer

EFI Runtime Service ResetSystem() Parameters

The following additions are made to the EFI Runtime Service **`ResetSystem()`**:

- A new reset type, **`EfiResetUpdate`** is added to the **`EFI_RESET_TYPE`** enumeration. The value assigned to **`EfiResetUpdate`** will be defined when this section is moved.
- *`DataSize`* is **`sizeof(VOID *)`** plus the size of the Unicode string.
- *`ResetData`* points to a Unicode string, which is ignored, followed by a (**`VOID *`**) to the beginning of the mailbox block directory.

The call to the **`ResetSystem()`** is responsible for the following:

- Saving the address of the base of the mailbox so that it can be discovered after the system transitions to PEI
- Causing the transition from the Afterlife (AL) to PEI modes such that the system's memory is not corrupted

Using **`ResetSystem()`** allows the mechanism by which the system re-enters PEI and DXE to be abstracted from the OS and capsule application. It also allows the mechanism by which the mailbox address is passed across the reset divide to be redefined as systems evolve.

Capsule PEIM and PEI Support Requirements

The capsule PEIM is responsible for the following:

- Turning the (probably) noncontiguous pieces of the mailbox into a single linear block of memory that is protected from corruption by other parts of PEI and DXE.
- Removing the excess data that was caused by page alignment between capsules. The actual capsule length can be determined by examining the capsule (firmware) volume header.
- Describing this block of memory to subsequent Framework phases via a capsule volume HOB. The capsule PEIM locates the base address of the mailbox using the platform-dependent mechanism defined by the EFI Runtime Service **`ResetSystem()`**.

Neither the PEI Foundation nor any of its PEIMs may modify the mailbox contents. In particular, the PEI Foundation must avoid placing its stack so that it overlaps the mailbox.

A PEIM must create a capsule volume HOB that describes the locations of the mailbox in memory. The capsule volume HOB is a standard memory allocation HOB type. Type **`EFI_HOB_CAPSULE_VOLUME`** is defined in the *Intel® Platform Innovation Framework for EFI Hand-Off Block (HOB) Specification.*

DXE and BDS Capsule Support

Introduction

Following the PEI phase, the capsule volume HOB is processed via a cooperation between DXE and BDS.

General Requirements

The DXE phase itself must not write on the capsule. The capsule memory should be reserved in the memory map by either the PEI phase or the DXE phase.

Capsule Processing

The capsule volume HOB must first be broken into its component capsules. This action may be done by either DXE or BDS.

BDS then uses the DXE Service **`ProcessFirmwareVolumes()`** to turn the capsule images into firmware volumes. BDS performs this task so that it can read these volumes before the DXE Dispatcher has a chance to do so.

BDS then uses the DXE Service **`Dispatch()`**to cause the DXE Dispatcher to see the new files.

When dispatched, the drivers are provided with the protocol of the firmware volume from which they were extracted. Data in that protocol can be used to discover the “parent” firmware volume so that, for example, the configuration results file can be accessed.

Usage Model

The following steps outline the general flow of typical events during a capsule update. This section assumes certain implementation details that are not, in fact, requirements of this specification.

1. The user obtains the capsule from the vendor.
2. The user invokes an OS-present firmware update application (hereafter referred to as “the application”).
3. The application searches for capsules.
4. The application extracts the header information from the valid capsules that it finds and displays the human-readable information so that the user can choose the capsule(s) with which to update the system.
5. The application obtains information from the user that is required by the capsule. This information can include configuration information. The required information is described in the capsule header, and the OS-present application stores this information in the capsule volume as firmware volume files. Because the capsule’s firmware volume file is typically hashed and signed, the data cannot be written into that volume without invalidating the hash.
6. The application writes the capsules into blocks of memory in its nonpaged address space using the physical addresses of the memory to link the blocks together. Nonpaged memory is commonly available because it is generally required for sourcing and sinking memory-mapped I/O device transactions.
7. The application then requests that the system be rebooted. To the user, this request would look very similar to the familiar “The system must be rebooted for the new driver to take effect” messages that are common in OSs.

NOTE

In high-availability systems, the capsules could instead be queued at this point either in memory or on other media (e.g., a hard disk). The rest of the process could then take place during the next reboot.

8. The system calls the EFI Runtime Service **`ResetSystem()`** with a pointer to the beginning of the capsule descriptors.
9. When **`ResetSystem()`** is called, this call indicates the end of the Runtime (RT) phase and the start of the Framework AL phase. In the AL phase, EFI again owns all of the system’s resources. When it encounters a capsule reset, **`ResetSystem()`** stores the base address of the capsule (the first **`EFI_CAPSULE_BLOCK_DESCRIPTOR`**) into a known platform-dependent location (either in system memory at a known reserved location or in another device such as Real Time Clock [RTC] CMOS battery-backed-up memory).
10. **`ResetSystem()`** then resets the system using a platform-dependent mechanism. The mechanism must preserve the state of RAM.
11. The SEC and PEI phases run.
12. Using the base address stored by **`ResetSystem()`**, the capsule PEIM coalesces the capsule data in memory that will not be used by the other parts of PEI.
13. The capsule PEIM then discards the internal fragmentation that was caused by page alignment.

14. The capsule PEIM next creates a capsule volume HOB to report the base address of the capsules to DXE and BDS.
15. BDS processes the capsules from the capsule volume HOB one at a time. It first uses the DXE Service **`ProcessFirmwareVolumes()`** to turn the capsule volume into a firmware volume.
16. BDS then invokes the DXE Dispatcher using the DXE Service **`ProcessFirmwareVolumes()`**.
17. The DXE Dispatcher processes the files as it would any other firmware volume.
18. The drivers in the new firmware volume are run as any other DXE drivers would be. They can then process the files that are written from the capsule volume as they would any other data files. Existing mechanisms already support locating the files in the same volume from which the drivers were loaded.
19. Using the images that are stored as files inside the capsule image firmware volume, the drivers then find the parent firmware volume and update the firmware devices.
20. The drivers return status via a GUID/data pair in the EFI System Table.

This scenario assumes that the capsule is being used for update. As noted, a capsule update is expected to be the most commonly used application for capsules.

3
Code Definitions

Introduction

This section provides code definitions for the capsule-specific data types and structures discussed in section 2, “Design Discussion.” The following data types and structures are defined in this section:

- **EFI_CAPSULE_HEADER**
- **EFI_CONFIG_FILE_NAME_GUID**
- **EFI_CAPSULE_BLOCK_DESCRIPTOR**

Capsule Header

EFI_CAPSULE_HEADER

Summary

Defines the start of a capsule.

GUID

```
#define EFI_CAPSULE_GUID \
  { 0x3b6686bd, 0xd76, 0x4030, 0xb7, 0xe, 0xb5, 0x51, \
    0x9e, 0x2f, 0xc5, 0xa0}
```

Prototype

```
typedef struct  {
  EFI_GUID            CapsuleGuid;
  UINT32              HeaderSize;
  UINT32              Flags;
  UINT32              CapsuleImageSize;
  UINT32              SequenceNumber;
  EFI_GUID            InstanceId;
  UINT32              OffsetToSplitInformation;
  UINT32              OffsetToCapsuleBody;
  UINT32              OffsetToOemDefinedHeader;
  UINT32              OffsetToAuthorInformation;
  UINT32              OffsetToRevisionInformation;
  UINT32              OffsetToShortDescription;
  UINT32              OffsetToLongDescription;
  UINT32              OffsetToApplicableDevices;
} EFI_CAPSULE_HEADER;
```

Parameters

CapsuleGuid

A defined GUID that indicates the start of a capsule. See “GUID” above for the capsule-specific GUID definition. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

HeaderSize

The size of the **EFI_CAPSULE_HEADER** structure.

Flags

A bit-mapped list describing the capsule’s attributes. All undefined bits should be written as zero (0). See “Related Definitions” below for defined bit values.

CapsuleImageSize

The length in bytes (27,415 for an image containing 27,415 bytes) of the entire image including all headers. If the this value is greater than the size of the data presented in the capsule body, this means that the image is separated across multiple media. If this value is less than the size of the data, it is an error.

SequenceNumber

A zero-based number that enables a capsule to be split into pieces and then recombined for easier transfer across media with limited size. The lower the *SequenceNumber*, the earlier in the final image that the part of the capsule is to appear. In capsules that are not split, this value shall be zero.

InstanceId

Used to group the various pieces of a split capsule to ensure that they comprise the same base image. It is valid for this item to be zero, in which case the capsule cannot be split into components.

OffsetToSplitInformation

The offset in bytes from the beginning of the header to the start of an EFI string that contains a description of the identity of the subcapsules that make up the capsule. If the capsule is not split, this value should be zero. The same string should be presented for all subcapsules that constitute the same capsule.

OffsetToCapsuleBody

The offset in bytes from the beginning of the header to the start of the part of the capsule that is to be transferred to DXE.

OffsetToOemDefinedHeader

The offset in bytes from the beginning of the header to the start of the OEM-defined header. This value must be less than *OffsetToCapsuleBody*. Type **EFI_CAPSULE_OEM_HEADER** is defined in “Related Definitions” below.

OffsetToAuthorInformation

The offset in bytes from the beginning of the header to the start of human-readable text that describes the entity that created the capsule. This value must be less than *OffsetToCapsuleBody*.

OffsetToRevisionInformation

The offset in bytes from the beginning of the header to the start of human-readable text that describes the revision of the capsule and/or the capsule’s contents. This value must be less than *OffsetToCapsuleBody*.

OffsetToShortDescription

The offset in bytes from the beginning of the header to the start of a one-line (less than 40 Unicode characters in any language) description of the capsule. It is intended to be used by OS-present applications when providing a list of capsules from which the user can choose. This value must be less than *OffsetToCapsuleBody*.

OffsetToLongDescription

The offset in bytes from the beginning of the header to the start of an EFI string that contains any or all of the following:

- A description of the capsule's contents
- The applicability of the capsule's contents
- Changes made since the previous revision of the firmware
- Any other information deemed useful by the developers of the capsule

The long description may include several paragraphs or pages of text. On the other hand, no rich formatting (for instance, bold, italics, or different fonts) is available. The capsule's developers must determine the display technologies in its market segments and tailor the format of the long description accordingly. This value must be less than *OffsetToCapsuleBody*.

OffsetToApplicableDevices

This field is reserved for future use by this specification. For future compatibility, this field must be set to zero (0x0000).

Description

The capsule header defines the start of a capsule. This header, and the optional OEM header, are removed by the application. The header is not passed on to the EFI firmware during the following reboot cycle. The OEM header **EFI_CAPSULE_OEM_HEADER** that follows the capsule header is defined in "Related Definitions" below. Note the following:

- The headers should look the same, except for the four parameters that are used to manage split capsules (see below). The strings in all but the first located will be ignored.
- The *InstanceId* parameter allows the application to match up the various pieces of a capsule image. The *SequenceNumber* parameter allows the order in which those pieces need to appear to be determined. The *CapsuleImageSize* parameter lets the application know when it has all of the pieces.

The following four parameters enable support for *split* capsules, which is the ability to break a capsule into multiple pieces and then have the capsule application join them back together:

- *CapsuleImageSize*
- *SequenceNumber*
- *InstanceId*
- *OffsetToSplitInformation*

The use of these four entries is described in greater detail in "Split Capsules" in chapter 2, "Design Discussion."

The following four human-readable strings support localization into multiple languages.

- *OffsetToAuthorInformation*
- *OffsetToRevisionInformation*
- *OffsetToShortDescription*
- *OffsetToLongDescription*

Each string consists of one or more sets of language/text pairs. The language descriptor is a Unicode representation of a 3-character ISO 696-2 language triple. Immediately following (no terminating zero) is the null-terminated Unicode text in that language. The string terminates with a null (leaving the null from the final language/text pair, followed by the null for the end of the strings). For example:

```
CHAR16 Text[] = L“eng One\0fra Un\0ger Einz\0”;
```

This simple string encodes the English (eng) “One,” the French (fra) “Un,” and the German (ger) “Einz.”

NOTE

C will add an implicit null terminator to the end of the string, so only one “\0” is required at the end of the string.

The ISO 696-2 language triples should be represented in lower case and are terminated by a required space (to allow for future expansion). Applications that access these strings should be written to accommodate language designators of other than three characters. They should treat any characters up to and including the space as a part of the designator.

Depending on policy, some types of security might need to be overridden. Consider, for example, recovery. In this case, it is probable that much of the system’s firmware has been rendered invalid or untrustworthy, including, in many cases, the security validation code. In this case, for example, implementers may—after careful analysis of security policy—define a special recovery device descriptor GUID that can be used in cases of recovery but that would not be valid in normal situations.

Related Definitions

```
//*******************************************
// Flags value
//*******************************************
#define EFI_CAPSULE_HEADER_FLAG_SETUP     0x0000000000000001
```

EFI_CAPSULE_HEADER_FLAG_SETUP

This flag is set if the capsule can support setup changes and clear if it cannot.

```
//*******************************************
// EFI_CAPSULE_OEM_HEADER
//*******************************************

typedef struct  {
  EFI_GUID            OemGuid;
  UINT32              HeaderSize;
  // UINT8            OemHeaderData[…];
} EFI_CAPSULE_OEM_HEADER;
```

OemGuid

An OEM-specific GUID that is used to uniquely identify the header and data. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

HeaderSize

The length in bytes of the **EFI_CAPSULE_OEM_HEADER**, starting with the first byte of the *OemGuid* field.

OemHeaderData

A variable-length array of OEM-specific data.

Common File Formats

Rather than inventing new formats for the contents of capsules, the standard volume, file, and section organization is used inside capsules as well. Some files and sections are used in specific ways inside capsules. The term *NameGuid* is defined in the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification*.

Firmware Volume Image Section

See the *Intel® Platform Innovation Framework for EFI Firmware Volume Specification* for a description of this section type.

Configuration Results File

EFI_CONFIG_FILE_NAME_GUID

Summary

The *configuration results file* is written by the OS-present application and provides the results from the Setup program. The file is a string that contains the configuration string. This string is defined in the *Intel® Platform Innovation Framework for EFI Human Interface Infrastructure Specification*.

The configuration results file must have the *NameGuid* listed below.

The following is the string format:

```
[<name>=<value][&<name>=<value>]*
```

where the name and value are defined in the IFR or translation thereof for the setup options of the device being configured. This format is also used by HTML and its successors.

GUID

```
// {98B8D59B-E8BA-48ee-98DD-C295392F1EDB}

#define EFI_CONFIG_FILE_NAME_GUID \
  { 0x98b8d59b, 0xe8ba, 0x48ee, 0x98, 0xdd, 0xc2, 0x95, \
  0x39, 0x2f, 0x1e, 0xdb }
```

Mailbox Format

EFI_CAPSULE_BLOCK_DESCRIPTOR

Summary

Defines the mailbox format.

Prototype

```
typedef struct {
  UINTN        Length;
  VOID         *DataBlock;
  UINT32       Signature;
  UINT32       CheckSum;
} EFI_CAPSULE_BLOCK_DESCRIPTOR;
```

Parameters

Length

Length in bytes of the data block.

DataBlock

Physical address of the data block.

Signature

The fixed-value Capsule Block Descriptor Signature (CBDS). Type **CAPSULE_BLOCK_DESCRIPTOR_SIGNATURE** is defined in "Related Definitions" below.

Checksum

To sum this structure to 0.

Description

This data structure defines the mailbox format. An *update mailbox*, or simply *mailbox*, is the mechanism by which the OS-present application communicates the capsule contents to PEI/DXE. The mailbox is a memory-resident data structure into which the update application writes an unedited capsule list. See "Mailbox Format" in chapter 2, "Design Discussion," for additional information.

The checksum is calculated by summing the first three entries when treated as **UINT32** values and negating the result.

The array of structures is terminated with a structure whose *Length* is 0. If the *DataBlock* pointer is **NULL**, the list is terminated. If the *DataBlock* pointer is not **NULL** but the *Length* is 0, the *DataBlock* pointer is known as a "continuation pointer" and points to a further list of **EFI_CAPSULE_BLOCK_DESCRIPTOR** structures.

Related Definitions

```
//*******************************************************
// CBDS Signature Value
//*******************************************************

#define CAPSULE_BLOCK_DESCRIPTOR_SIGNATURE \
                EFI_SIGNATURE_32('C','B','D','S')
```