Intel® Platform Innovation Framework for EFI Recovery Specification

Version 0.9

September 16, 2003

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE. Except for a limited copyright license to copy this specification for internal use only, no license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information in this specification. Intel does not warrant or represent that such implementation(s) will not infringe such rights.

Designers must not rely on the absence or characteristics of any features or instructions marked “reserved” or “undefined.” Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Intel, the Intel logo, and Itanium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

* Other names and brands may be claimed as the property of others.

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 recovery architecture and the core code and services that are required for an implementation of the Intel® Platform Innovation Framework for EFI (hereafter referred to as the “Framework”). *Recovery* is the automatic process of updating a “bad” version of a Framework firmware with a good version. This specification does the following:

- Describes the Framework recovery philosophy and the generic high-level recovery flow
- Explains the architectural PEIM-to-PEIM Interfaces (PPIs) for recovery that are published by PEIMs
- Provides code definitions for the platform-independent PPIs that are architecturally required by the *Intel® Platform Innovation Framework for EFI Architecture Specification* and that are used during the Loading the Recovery DXE Image phase

Target Audience

This document is targeted at system software developers who are evaluating the Framework for their product. It is also intended for developers who need to port or leverage this infrastructure for their platform.

This document is platform independent.

Scope

Note the following limitations in the scope of this specification:

- This document is hardware neutral and does not discuss platform-specific hardware or their associated modules.
- This document describes only the platform-independent recovery modules that any Framework PEI interface would need to provide. It does not describe all of the possible PEIMs that might be required to produce this functionality.
- This document addresses only the PEIM-to-PEIM Interfaces (PPIs) used during the Loading the Recovery DXE Image phase.

Goals

The goal of this design is to allow the creation of portable modules that subscribe to standard-based interfaces. The recovery PEIM contains platform-specific policy decisions, such as where to look for the recovery capsule, but these decisions are internal to an implementation and opaque to the architecture.

Required Features

This architecture requires the existence of some platform modules to initialize the system fabric so that the recovery code can run. The minimal requirements include the following:

- Useable system memory
- A means by which to enable the hardware interface to the device(s) containing a recovery capsule

These devices may have other requirements, which are outside the scope of this document.

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.

Procedure Descriptions

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

ProcedureName(): The formal name of the procedure.

Summary: A brief description of the procedure.

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

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

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

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

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

PPI Descriptions

A PEIM-to-PEIM Interface (PPI) description generally has the following format:

PPI Name: The formal name of the PPI.

Summary: A brief description of the PPI.

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

PPI Interface Structure: A “C-style” procedure template defining the PPI calling structure.

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

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

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

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

Pseudo-Code Conventions

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

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

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

Typographic Conventions

This document uses the typographic and illustrative conventions described below:

Plain text

The normal text typeface is used for the vast majority of the descriptive text in a specification.

Plain text (blue)

In the online help version of this specification, any plain text that is underlined and in blue indicates an active link to the cross-reference. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification.

Bold

In text, a **Bold** typeface identifies a processor register name. In other instances, a **Bold** typeface can be used as a running head within a paragraph.

Italic

In text, an *Italic* typeface can be used as emphasis to introduce a new term or to indicate a manual or specification name.

`BOLD Monospace`

Computer code, example code segments, and all prototype code segments use a **`BOLD Monospace`** typeface with a dark red color. These code listings normally appear in one or more separate paragraphs, though words or segments can also be embedded in a normal text paragraph.

`Bold Monospace`

In the online help version of this specification, words in a **`Bold Monospace`** typeface that is underlined and in blue indicate an active hyperlink to the code definition for that function or type definition. Click on the word to follow the hyperlink. Note that these links are *not* active in the PDF of the specification. Also, these inactive links in the PDF may instead have a **`Bold Monospace`** appearance that is underlined but in dark red. Again, these links are not active in the PDF of the specification.

`Italic Monospace`

In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).

`Plain Monospace`

In code, words in a `Plain Monospace` typeface that is a dark red color but is not bold or italicized indicate pseudo code or example code. These code segments typically occur in one or more separate paragraphs.

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

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

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

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

2
Design Discussion

Terms

Following are definitions of the basic terms that are used throughout this specification:

crisis recovery

Setting the boot mode to recovery based on the detection of some illegal state.

forced recovery

Setting the boot mode to recovery based on the detection of some user-initiated signal, such as an installed jumper.

MBR

Master Boot Record. The data structure that resides on the first sector of a hard disk and defines the partitions on the disk.

recovery

The automatic process of updating a "bad" version of a Framework firmware with a good version. See the next topic, Definition of "Recovery", for more details.

Definition of "Recovery"

By way of introduction, the concept of *recovery* should first be described. For most firmware systems, the code located in the persistent baseboard storage, such as flash memory, is decomposed into the following two regimes:

- The recovery block
- The nonrecovery block

The recovery block code is usually stored in some hardware-protected region of the baseboard flash part and is responsible for receiving the reset vector when a system is restarted.

Traditionally, legacy BIOSs have taken one of two approaches:

- The recovery block is a complete, stripped-down BIOS used only during recovery.
- The recovery block has a dual personality: normal and recovery operation.

Both approaches leave noncritical hardware in a benign state and critical hardware programmed with "safe" values. The Framework architecture allows both approaches. Recovery mode requires that all code accesses must remain within the recovery block, while normal mode allows accesses to the nonrecovery block. Additionally, the recovery block can be updated in a fault-tolerant manner or hardware protected from ever being updated. Again the Framework architecture allows both approaches.

Steps in the Recovery Sequence

Steps in the Recovery Sequence

The global Framework recovery follows a series of architectural steps, which are flexible to accommodate original equipment manufacturer (OEM) design criteria or preferences. The table below lists the steps in the recovery sequence.

Table 2-1. Steps in the Recovery Sequence

Step	Description	See Section…
1	Detect that recovery is needed	Detecting That Recovery Is Needed
2	Complete the Pre-EFI Initialization (PEI) recovery phase	PEI Recovery Phase
3	Find the recovery Driver Execution Environment (DXE) image	Finding and Loading the Recovery DXE Image: Overview
4	Load the recovery DXE image	Finding and Loading the Recovery DXE Image: Recovery Sequence
5	Complete the DXE recovery phase	Finding and Loading the Recovery DXE Image and DXE Recovery Phase and After
6	Load the "recovery operating system"	DXE Recovery Phase and After
7	Load the recovery application	DXE Recovery Phase and After
8	Recover the Framework firmware	Recovering the Framework Firmware

Detecting That Recovery Is Needed

The PEI Dispatcher starts dispatching PEIMs in the normal manner. During this process, a PEIM determines that recovery is warranted. This detection may be due to a platform-specific module detecting whether a recovery condition should be engendered or "forced" based upon some user input, such as a jumper setting detected through a General Purpose I/O (GPIO) input.

There are also architectural reasons for going into recovery, such as environmental factors. In the case of the Intel® Itanium® processor family, an environmental factor for engendering a recovery would occur if there were no Processor Abstraction Layer B (PAL-B) revisions that matched the present processor revisions.

In the case of a flash update, a recovery would need to occur if a power failure occurs while the nonrecovery portion of the flash part is in the process of being updated and does not finish. These latter cases are characterized under the classification *crisis recovery*. Other "crisis recovery" instances include the following:

- Modules not passing some integrity checks, such as signature or checksum
- A series of errors when initializing hardware

The hardware case would also engender a crisis recovery possibly because the recovery boot is usually a simple, conservative boot-strap much in the same sense as the "safe mode" boot of operating systems. Regardless of the reason, the PEIM initiates the recovery action by setting the **`BOOT_IN_RECOVERY_MODE`** bit (see the PEI CIS for the definition).

PEI Recovery Phase

NOTE

This section describes only the platform-independent recovery modules that any Framework PEI implementation would need to provide. It does not describe all of the possible PEIMs that might be required to affect this functionality.

The setting of the **BOOT_IN_RECOVERY_MODE** bit notifies the PEI Dispatcher to transition to recovery mode. The PEI Dispatcher clears out the list of PEIMs that have been dispatched and restarts the dispatch of PEIMs.

The following PEIMs need to have the **FFS_ATTRIBUTE_RECOVERY** bit (Bit 0) set in the PEIM's Firmware File System (FFS) header (**EFI_FFS_FILE_HEADER**):

- PEIMs that require different actions in recovery mode than in normal mode
- PEIMs that are known to be required for recovery mode

These modules, along with the Security (SEC) start-up code and the PEI Foundation, make up the recovery block.

The PEI Dispatcher starts searching for PEIMs to invoke and invokes only those PEIMs that are actually marked as being for *recovery dispatch*. These PEIMs might serve a dual purpose for the normal boot, such as having just one PEIM for memory initialization. However, when these PEIMs detect that the boot mode variable is set to recovery, they should perform only the minimal behavior. This sharing is not architectural—it is an implementation artifact to save flash storage space. Unshared recovery PEIM-to-PEIM Interfaces (PPIs) return without any action taken in nonrecovery environments. Following is the intent of the platform-specific PEIMs:

- Alerting the PEI Foundation of the recovery condition
- Initializing enough of the platform I/O complex and memory such that the PPIs designated in this document and related documents can run

~~***Review Note: After IDF, check if Interface Import Table (in the following sentence) is still a current, accurate term to use and fix as needed.***~~

The modules marked *recovery dispatch* can have Interface Import Table references to nonrecovery PPIs. The PEI Dispatcher automatically includes the PEIMs containing these PPIs as recovery PEIMs. These referenced nonrecovery PEIMs need to be stored in the boot block to ensure that all required modules are in a secure, uncorruptible area. This requirement implies that, during a normal update, a PEIM that is newly referenced by recovery code and that is in a non-boot-block area must migrate to the boot block.

Note that PEIMs can alter their behavior in several ways based on the **BOOT_IN_RECOVERY_MODE** bit:

- Perform normally or no behavior change. This behavior is typical of support modules.
- Perform no action. This behavior is typical for noncritical hardware. Examples are nonrecovery modules.
- Perform minimal configuration. This behavior is typical of critical hardware. Memory initialization is an example.
- Enable functionality. This behavior is typical of modules that are required only for recovery.

The actions taken by the PEIMs are both platform and module specific and are outside the scope of this document.

The figure below shows how platform modules are dispatched in the PEI phase.

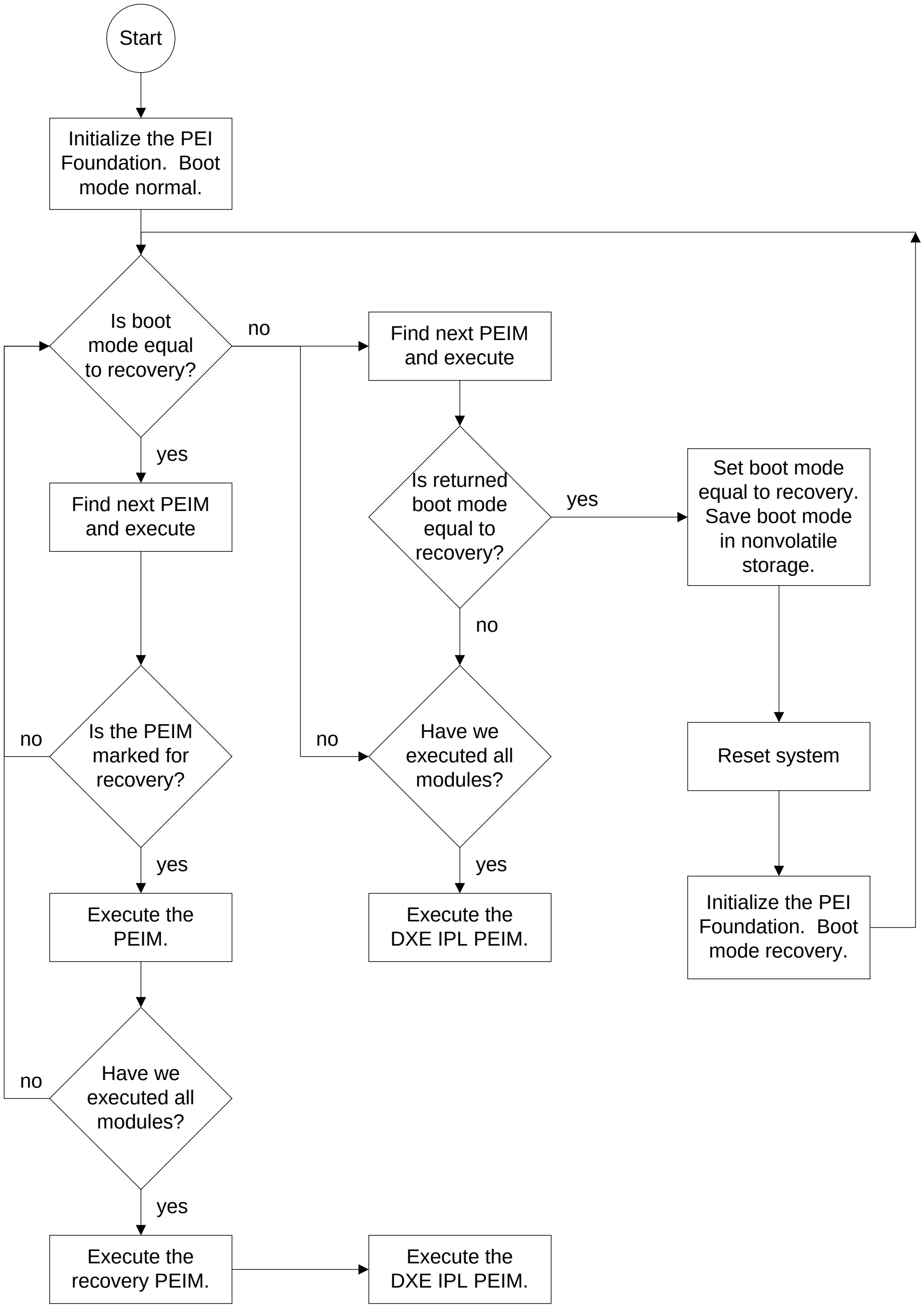
Start
Initialize the PEI Foundation. Boot mode normal.
Is boot mode equal to recovery?
no
Find next PEIM and execute
yes
Is returned boot mode equal to recovery?
yes
Set boot mode equal to recovery. Save boot mode in nonvolatile storage.
Find next PEIM and execute
no
no
Is the PEIM marked for recovery?
no
Have we executed all modules?
Reset system
yes
yes
Execute the PEIM.
Execute the DXE IPL PEIM.
Initialize the PEI Foundation. Boot mode recovery.
no
Have we executed all modules?
yes
Execute the recovery PEIM.
Execute the DXE IPL PEIM.

Figure 2-1. Platform Module Dispatch in PEI

See the indicated specifications for code definitions of the following:

- **BOOT_IN_RECOVERY_MODE**: Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification (PEI CIS)
- **EFI_FFS_FILE_HEADER**: Intel® Platform Innovation Framework for EFI Firmware File System Specification

Finding and Loading the Recovery DXE Image

Finding the Recovery DXE Image: Overview

The PEI Dispatcher specifically invokes the DXE Initial Program Load (IPL) PEIM, regardless of normal or recovery mode. The DXE IPL PEIM detects that a recovery is in process and invokes a recovery-specific PPI, the Recovery Module PPI. The Recovery Module PPI, **EFI_PEI_RECOVERY_MODULE_PPI**, does the following:

- Loads a binary capsule that includes a recovery DXE image into memory
- Updates the Hand-Off Block (HOB) table with the DXE firmware volume

See Code Definitions for the PPIs that are needed to load the DXE phase.

Note that the Recovery Module PPI is device and content neutral. The DXE IPL PEIM uses the Recovery Module PPI to load a DXE image and invokes the DXE image normally. The DXE IPL PEIM does not know or care about the capsule’s internal structure or from which device the capsule was loaded.

The internals of the recovery PEIM normally fall within four phases:

- Searching the supported devices for recovery capsules
- Deciding which capsule to load
- Loading the capsule into memory
- Loading the resulting DXE firmware volume

The Recovery Module PPI encompasses the first three phases and the DXE IPL PEIM encompasses the last phase. See the next topic, Recovery Sequence: Detailed Steps, for the details of these four phases.

Recovery Sequence

The normal, nonrecovery sequence is that after completion of the PEI phase, the PEI Dispatcher specifically invokes the DXE Initial Program Load (IPL) PEIM. The recovery sequence is identical to the nonrecovery sequence in that the PEI Dispatcher also specifically invokes the DXE IPL PEIM. After invoking the DXE IPL PEIM, the recovery sequence is as follows:

1. The DXE IPL PEIM detects that a recovery is in process, searches for the Recovery Module PPI, and invokes the recovery function **EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()**.
2. **EFI_PEI_RECOVERY_MODULE_PPI** searches for one or more instances of the Device Recovery Module PPI, **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI**. For each instance found, the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetNumberRecoveryCapsules()** function is invoked to determine the following:
    - The number of recovery DXE capsules detected by the specified device
    - The maximum buffer size required to load a capsule
3. **EFI_PEI_RECOVERY_MODULE_PPI** then decides the following:
    - The device search order, if more than one Device Recovery Module PPI was discovered
    - The individual search order, if the device reported more than one recovery DXE capsule was found generating a search order list
4. **EFI_PEI_RECOVERY_MODULE_PPI** invokes the device recovery function **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** to load a capsule that includes a recovery DXE image into memory. The capsule that is returned from the device recovery module is a capsule that contains the recovery DXE image.
5. The **EFI_PEI_RECOVERY_MODULE_PPI** security does the following:
    - Verifies the capsule
    - Generates a data Hand-Off Block (HOB) entry for a security failure
    - Tries the next entry in the search order list
6. Once a valid capsule has been loaded, **EFI_PEI_RECOVERY_MODULE_PPI** does the following:
    - Decomposes the capsule and updates the HOB table with the recovery DXE firmware volume information. The path parameters are assumed to be redundant for recovery. The Setup parameters are either redundant or fixed.
    - Invalidates all HOB entries for updateable firmware volume entries.

The DXE capsule that is loaded by the Device Recovery Module PPI makes no assumptions about contents or format other than assuming that the recovery DXE image is somewhere in the returned capsule.

The following subsections describe the different recovery PPIs.

Recovery PPIs

Recovery Module PPI

The Recovery Module PPI, **`EFI_PEI_RECOVERY_MODULE_PPI`**, invokes the Device Recovery Module PPI **`EFI_PEI_DEVICE_RECOVERY_MODULE_PPI`** to do the following:

- Determine the number of DXE recovery capsules found by each device
- Determine capsule information
- Load a specific DXE recovery capsule from the indicated device
- Determine the device load order

The capsule is security verified and decomposed and the HOB table is updated with the DXE recovery firmware volume.

There are two general categories of recovery PPIs:

- Device recovery PPI
- Device recovery block I/O PPI

The Device Recovery Module PPI is device neutral. The Device Recovery Block I/O PPI is device specific and used to access the physical media. The following subsections describe the PPI associated with each category. See Code Definitions for the definitions of these PPIs.

Device Recovery Module PPI

The table below lists the device recovery functions in the Device Recovery Module PPI, **`EFI_PEI_DEVICE_RECOVERY_MODULE_PPI`**.

Table 2-2. Device Recovery Module Functions

Function	Description
GetNumberRecoveryCapsules()	Scans the devices that are supported by the PPI for DXE recovery capsules and reports the number found. The internal ordering should reflect the priority in the load order, with the highest priority capsule number set to one and the lowest priority number set to *N.*
GetRecoveryCapsuleInfo()	Provides the size of the indicated capsule and a *`CapsuleType`* Globally Unique Identifier (GUID). The recovery module uses this information to allow an alternate priority scheme based on the *`CapsuleType`* information.
LoadRecoveryCapsule()	Loads the indicated DXE recovery capsule instance and returns a capsule with the actual number of bytes loaded.

Device Recovery Block I/O PPI

The Device Recovery Block I/O PPI, **`EFI_PEI_RECOVERY_BLOCK_IO_PPI`**, differs from the Device Recovery Module PPI in that the Device Recovery Block I/O PPI is used for physical media access. The Device Recovery Module PPI uses this PPI to search for capsules. This PPI is included with the recovery PEIMs because a block I/O is the most common recovery media.

The table below lists the functions in the Device Recovery Block I/O PPI.

Table 2-3. Device Recovery Block I/O Functions

Function	Description
GetNumberOfBlockDevices()	Returns the number of block I/O devices supported. There is no ordering priority.
GetBlockDeviceMediaInfo()	Indicates the type of block I/O device found, such as a legacy floppy or CD-ROM. The block size and last block number are also returned.
ReadBlocks()	Reads the indicated block I/O device starting at the given logical block address (LBA) and for buffer size/block size.

DXE Recovery Phase and After

DXE Recovery Phase and After

The recovery capsule that is shadowed to memory contains one of the following:

- The DXE Foundation and a complement of DXE drivers sufficient to initialize enough of the system to invoke a second stage loader
- A full set of the DXE Foundation and DXE drivers sufficient to initialize enough of the system to invoke the operating system and/or a Framework application
- A full set of the DXE Foundation and DXE drivers and an EFI application to update firmware volumes

The operating system or the Framework application may reside on a device other than the one from which the recovery DXE was loaded.

The DXE IPL PEIM invokes the DXE Dispatcher that was loaded from the recovery capsule in the normal fashion. The DXE Dispatcher starts processing the DXE modules in the recovery capsule. Control is passed to the boot loader module in a normal fashion. There are two architectural paths that can be implemented at this point:

- Booting an operating system
- Booting to a Framework application

Both architectures are valid but the associated modules are mutually exclusive. Booting an operating system and executing a recovery application under the operating system require that any platform-specific protocols be runtime interfaces. Booting a Framework application and not an operating system requires the platform-specific protocols to be boot service interfaces.

The following subsections list additional requirements for recovering the Framework firmware or completing other post-recovery operations.

Recovering the Framework Firmware

The nonrecovery blocks need to be updated in such a manner that the recovery application “calls back” to the recovery DXE modules for any platform-specific hardware manipulation. This callback ensures that the recovery application is platform neutral. Note that the recovery may process many firmware volumes to update both the baseboard platform and add-in cards.

Post-Recovery Operations

After a new Framework firmware has been installed, the system needs to be rebooted for the recovered firmware to execute. However, prior to the reboot, other cleanup operations may be required.

3
Code Definitions

Introduction

This section contains the definitions of the platform-independent PPIs that are required for all implementations of the Framework. The table below explains the organization of this section and lists the PPIs that are defined in this section.

Table 3-1. Organization of the Code Definitions Section

Section	Summary	PPI Definition
Recovery Module PPI	Describes the main Recovery Module PPI.	EFI_PEI_RECOVERY_MODULE_PPI
Device Recovery Module PPI	Describes the Device Recovery Module PPI.	EFI_PEI_DEVICE_RECOVERY_MODULE_PPI
Device Recovery Block I/O PPI	Describes the Device Recovery Block I/O PPI. This section is device specific and addresses the most common form of recovery media—block I/O devices such as legacy floppy, CD-ROM, or IDE devices.	EFI_PEI_RECOVERY_BLOCK_IO_PPI

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

- **`EFI_PEI_BLOCK_IO_MEDIA`**
- **`EFI_PEI_BLOCK_DEVICE_TYPE`**
- **`EFI_PEI_LBA`**

Recovery Module PPI

EFI_PEI_RECOVERY_MODULE_PPI

Summary

Finds and loads the recovery files.

GUID

```
#define EFI_PEI_RECOVERY_MODULE_PPI \
{0xFB6D9542, 0x612D, 0x4f45, 0x87, 0x2F, 0x5C, 0xFF, 0x52, 0xE9,
0x3D, 0xCF}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_RECOVERY_MODULE_PPI {
  EFI_PEI_LOAD_RECOVERY_CAPSULE          LoadRecoveryCapsule;
} EFI_PEI_RECOVERY_MODULE_PPI;
```

Parameters

LoadRecoveryCapsule

Loads a DXE binary capsule into memory.

Description

This module has many roles and is responsible for the following:

1. Calling the driver recovery PPI **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetNumberRecoveryCapsules()** to determine if one or more DXE recovery entities exist.
2. If no capsules exist, then performing appropriate error handling.
3. Allocating a buffer of *MaxRecoveryCapsuleSize* as determined by **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.GetNumberRecoveryCapsules()** or larger.
4. Determining the policy in which DXE recovery capsules are loaded.
5. Calling the driver recovery PPI **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** for capsule number *x*.
6. If the load failed, performing appropriate error handling.
7. Performing security checks for a loaded DXE recovery capsule.
8. If the security checks failed, then logging the failure in a data HOB. ~~The mechanism is TBD.~~
9. If the security checks failed, then determining the next **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()** capsule number; otherwise, go to step 11.
10. If more DXE recovery capsules exist, then go to step 5; otherwise, perform error handling.

11. Decomposing the capsule loaded by **`EFI_PEI_DEVICE_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()`** into its components. It is assumed that the path parameters are redundant for recovery and Setup parameters are either redundant or canned.
12. Invalidating all HOB entries for updateable firmware volume entries. This invalidation prevents possible errant drivers from being executed.
13. Updating the HOB table with the recovery DXE firmware volume information generated from the capsule decomposition.
14. Returning to the PEI Dispatcher.

EFI_PEI_RECOVERY_MODULE_PPI.LoadRecoveryCapsule()

Summary

Loads a DXE capsule from some media into memory and updates the HOB table with the DXE firmware volume information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_LOAD_RECOVERY_CAPSULE) (
  IN EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_RECOVERY_MODULE_PPI      *This
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_MODULE_PPI** instance.

Description

This function, by whatever mechanism, retrieves a DXE capsule from some device and loads it into memory. Note that the published interface is device neutral.

Status Codes Returned

EFI_SUCCESS	The capsule was loaded correctly.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

Device Recovery Module PPI

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI

Summary

Presents a standard interface to **EFI_PEI_RECOVERY_MODULE_PPI**, regardless of the underlying device(s).

GUID

```
#define EFI_PEI_DEVICE_RECOVERY_MODULE_PPI   \
{ 0x0DE2CE25, 0x446A, 0x45a7, 0xBF, 0xC9, 0x37, 0xDA, 0x26, 0x34,
0x4B, 0x37}
```

PPI Interface Structure

```
typedef struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI {
  EFI_PEI_DEVICE_GET_NUMBER_RECOVERY_CAPSULE
                                GetNumberRecoveryCapsules;
  EFI_PEI_DEVICE_GET_RECOVERY_CAPSULE_INFO
                                GetRecoveryCapsuleInfo;
  EFI_PEI_DEVICE_LOAD_RECOVERY_CAPSULE
                                LoadRecoveryCapsule;
} EFI_PEI_DEVICE_RECOVERY_MODULE_PPI;
```

Parameters

GetNumberRecoveryCapsules

Returns the number of DXE capsules that were found. See the **GetNumberRecoveryCapsules()** function description.

GetRecoveryCapsuleInfo

Returns the capsule image type and the size of a given image. See the **GetRecoveryCapsuleInfo()** function description.

LoadRecoveryCapsule

Loads a DXE capsule into memory. See the **LoadRecoveryCapsule()** function description.

Description

The role of this module is to present a standard interface to **EFI_PEI_RECOVERY_MODULE_PPI**, regardless of the underlying device(s). The interface does the following:

- Reports the number of recovery DXE capsules that exist on the associated device(s)
- Finds the requested firmware binary capsule
- Loads that capsule into memory

A device can be either a group of devices, such as a block device, or an individual device. The module determines the internal search order, with capsule number 1 as the highest load priority and number *N* as the lowest priority.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. GetNumberRecoveryCapsules()

Summary

Returns the number of DXE capsules residing on the device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_GET_NUMBER_RECOVERY_CAPSULE) (
  IN EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI   *This,
  OUT UINTN                                  *NumberRecoveryCapsules
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

NumberRecoveryCapsules

Pointer to a caller-allocated **UINTN**. On output, **NumberRecoveryCapsules* contains the number of recovery capsule images available for retrieval from this PEIM instance.

Description

This function, by whatever mechanism, searches for DXE capsules from the associated device and returns the number and maximum size in bytes of the capsules discovered. Entry 1 is assumed to be the highest load priority and entry *N* is assumed to be the lowest priority.

Status Codes Returned

EFI_SUCCESS	One or more capsules were discovered.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. GetRecoveryCapsuleInfo()

Summary

Returns the size and type of the requested recovery capsule.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_GET_RECOVERY_CAPSULE_INFO) (
  IN  EFI_PEI_SERVICES                            **PeiServices,
  IN  struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI  *This,
  IN  UINTN                                       CapsuleInstance,
  OUT UINTN                                       *Size,
  OUT EFI_GUID                                    *CapsuleType
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

CapsuleInstance

Specifies for which capsule instance to retrieve the information. This parameter must be between one and the value returned by **GetNumberRecoveryCapsules()** in *NumberRecoveryCapsules*.

Size

A pointer to a caller-allocated **UINTN** in which the size of the requested recovery module is returned.

CapsuleType

A pointer to a caller-allocated **EFI_GUID** in which the type of the requested recovery capsule is returned. The semantic meaning of the value returned is defined by the implementation. Type **EFI_GUID** is defined in **InstallProtocolInterface()** in the *EFI 1.10 Specification.*

Description

This function returns the size and type of the capsule specified by *`CapsuleInstance`*.

Status Codes Returned

EFI_SUCCESS	One or more capsules were discovered.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	A recovery DXE capsule cannot be found.

EFI_PEI_DEVICE_RECOVERY_MODULE_PPI. LoadRecoveryCapsule()

Summary

Loads a DXE capsule from some media into memory.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_DEVICE_LOAD_RECOVERY_CAPSULE) (
  IN OUT EFI_PEI_SERVICES                         **PeiServices,
  IN struct _EFI_PEI_DEVICE_RECOVERY_MODULE_PPI   *This,
  IN UINTN                                        CapsuleInstance,
  OUT VOID                                        *Buffer
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_DEVICE_RECOVERY_MODULE_PPI** instance.

CapsuleInstance

Specifies which capsule instance to retrieve.

Buffer

Specifies a caller-allocated buffer in which the requested recovery capsule will be returned.

Description

This function, by whatever mechanism, retrieves a DXE capsule from some device and loads it into memory. Note that the published interface is device neutral.

Status Codes Returned

EFI_SUCCESS	The capsule was loaded correctly.
EFI_DEVICE_ERROR	A device error occurred.
EFI_NOT_FOUND	The requested recovery DXE capsule cannot be found.

Device Recovery Block I/O PPI

Device Recovery Block I/O PPI

The Recovery Module PPI and the Device Recovery Module PPI subsections earlier in Code Definitions are device neutral. This section is device specific and addresses the most common form of recovery media—block I/O devices such as legacy floppy, CD-ROM, or IDE devices.

The Recovery Block I/O PPI is used to access block devices. Because the Recovery Block I/O PPIs that are provided by the PEI ATAPI driver and PEI legacy floppy driver are the same, here we define a set of general PPIs for both drivers to use.

EFI_PEI_RECOVERY_BLOCK_IO_PPI

Summary

Provides the services required to access a block I/O device during PEI recovery boot mode.

GUID

```
#define EFI_PEI_IDE_BLOCK_IO_PPI \
  { 0x0964e5b22, 0x6459, 0x11d2, 0x8e, 0x39, 0x00, 0xa0, 0xc9,
0x69, 0x72, 0x3b }

#define EFI_PEI_144_FLOPPY_BLOCK_IO_PPI \
  { 0xda6855bd, 0x07b7, 0x4c05, 0x9e, 0xd8, 0xe2, 0x59, 0xfd,
0x36, 0x0e, 0x22 }
```

These GUIDs are hardware-device class GUIDs that would be imported only by the Virtual Block I/O PEIM. This virtual PEIM imports only actual Block I/O PPIs from the device-class ones listed above and publishes a single instance of the Block I/O PPI for consumption by the File System PEIM. In the parlance of the Framework DXE software stack, this Virtual Block I/O PEIM is actually embodying the functionality of the partition driver. This Virtual Block I/O PEIM has to multiplex the multiple possible instances of block I/O and also know how to parse at least El Torito for CD-ROM and perhaps Master Boot Record (MBR) and GUID Partition Table (GPT) in the future.

```
#define EFI_PEI_VIRTUAL_BLOCK_IO_PPI \
   { 0x695d8aa1, 0x42ee, 0x4c46, 0x80, 0x5c,0x6e, 0xa6, 0xbc,
0xe7, 0x99, 0xe3 }
```

Two classes of devices are supported, each of which can publish this PPI. This *a priori* segregation is performed so that the Virtual Block I/O PEIM can import multiple possible interfaces. The only way to achieve this end in the present implementation of the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS) is by using daisy chaining. The daisy chain model permits only one of the *N* chained PPIs to return, whereas here we would like to have the possibility of more than one module return information to the File System PEIM.

Again, it is important to stress that there is only one architectural definition of this Block I/O interface, with the two classes of interface providers. The first class abstracts actual device instances, and the second consumes these virtual interfaces. This consumption is implementation dependent inasmuch as it can use the Import Table mechanism or the PEI Service **LocatePpi()** intrinsic for discovering and invoking the services.

PPI Interface Structure

```
typedef struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI {
  EFI_PEI_GET_NUMBER_BLOCK_DEVICES   GetNumberOfBlockDevices;
  EFI_PEI_GET_DEVICE_MEDIA_INFORMATION
                                     GetBlockDeviceMediaInfo;
  EFI_PEI_READ_BLOCKS                ReadBlocks;
} EFI_PEI_RECOVERY_BLOCK_IO_PPI;
```

Parameters

GetNumberOfBlockDevices

Gets the number of block I/O devices that the specific block driver manages. See the **GetNumberOfBlockDevices()** function description.

GetBlockDeviceMediaInfo

Gets the specified media information. See the **GetBlockDeviceMediaInfo()** function description.

ReadBlocks

Reads the requested number of blocks from the specified block device. See the **ReadBlocks()** function description.

Description

This function provides the services that are required to access a block I/O device during PEI recovery boot mode.

EFI_PEI_RECOVERY_BLOCK_IO_PPI. GetNumberOfBlockDevices()

Summary

Gets the count of block I/O devices that one specific block driver detects.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_NUMBER_BLOCK_DEVICES) (
  IN  EFI_PEI_SERVICES                       **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI   *This,
  OUT UINTN                                  *NumberBlockDevices
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

NumberBlockDevices

The number of block I/O devices discovered.

Description

This function is used for getting the count of block I/O devices that one specific block driver detects. To the PEI ATAPI driver, it returns the number of all the detected ATAPI devices it detects during the enumeration process. To the PEI legacy floppy driver, it returns the number of all the legacy devices it finds during its enumeration process. If no device is detected, then the function will return zero.

Status Codes Returned

None.

EFI_PEI_RECOVERY_BLOCK_IO_PPI.GetBlockDeviceMediaInfo()

Summary

Gets a block device's media information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_GET_DEVICE_MEDIA_INFORMATION) (
  IN  EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI    *This,
  IN UINTN                                    DeviceIndex,
  OUT EFI_PEI_BLOCK_IO_MEDIA                  *MediaInfo
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

MediaInfo

The media information of the specified block media. Type **EFI_PEI_BLOCK_IO_MEDIA** is defined in "Related Definitions" below. The caller is responsible for the ownership of this data structure.

Note that this structure describes an enumeration of possible block device types. This enumeration exists because no device paths are actually passed across interfaces that describe the type or class of hardware that is publishing the block I/O interface. This enumeration will allow for policy decisions in the Recovery PEIM, such as "Try to recover from legacy floppy first, LS-120 second, CD-ROM third." If there are multiple partitions abstracted by a given device type, they should be reported in ascending order; this order also applies to nested partitions, such as legacy MBR, where the outermost partitions would have precedence in the reporting order. The same logic applies to systems such as IDE that have precedence relationships like "Master/Slave" or "Primary/Secondary"; the master device should be reported first, the slave second.

Description

This function will provide the caller with the specified block device's media information. If the media changes, calling this function will update the media information accordingly.

Related Definitions

```
//**************************************************
// EFI_PEI_BLOCK_IO_MEDIA
//**************************************************

typedef struct {
  EFI_PEI_BLOCK_DEVICE_TYPE   DeviceType;
  BOOLEAN                     MediaPresent;
  UINTN                       LastBlock;
  UINTN                       BlockSize;
} PEI_BLOCK_IO_MEDIA;
```

DevType

The type of media device being referenced by *DeviceIndex*. Type **EFI_PEI_BLOCK_DEVICE_TYPE** is defined below.

MediaPresent

A flag that indicates if media is present. This flag is always set for nonremovable media devices.

LastBlock

The last logical block that the device supports.

BlockSize

The size of a logical block in bytes.

```
//************************************************************
// EFI_PEI_BLOCK_DEVICE_TYPE
//************************************************************
typedef enum {
  LegacyFloppy    =   0,
  IdeCDROM        =   1,
  IdeLS120        =   2,
  UsbMassStorage  =   3,
  MaxDeviceType
} EFI_PEI_BLOCK_DEVICE_TYPE;
```

Status Codes Returned

EFI_SUCCESS	Media information about the specified block device was obtained successfully.
EFI_DEVICE_ERROR	Cannot get the media information due to a hardware error.

EFI_PEI_RECOVERY_BLOCK_IO_PPI.ReadBlocks()

Summary

Reads the requested number of blocks from the specified block device.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_PEI_READ_BLOCKS) (
  IN  EFI_PEI_SERVICES                        **PeiServices,
  IN struct _EFI_PEI_RECOVERY_BLOCK_IO_PPI     *This,
  IN UINTN                                     DeviceIndex,
  IN EFI_PEI_LBA                               StartLBA,
  IN UINTN                                     BufferSize,
  OUT VOID                                     *Buffer
);
```

Parameters

PeiServices

General-purpose services that are available to every PEIM. Type **EFI_PEI_SERVICES** is defined in the *Intel® Platform Innovation Framework for EFI Pre-EFI Initialization Core Interface Specification* (PEI CIS).

This

Indicates the **EFI_PEI_RECOVERY_BLOCK_IO_PPI** instance.

DeviceIndex

Specifies the block device to which the function wants to talk. Because the driver that implements Block I/O PPIs will manage multiple block devices, the PPIs that want to talk to a single device must specify the device index that was assigned during the enumeration process. This index is a number from one to *NumberBlockDevices*.

StartLBA

The starting logical block address (LBA) to read from on the device. Type **EFI_PEI_LBA** is defined in "Related Definitions" below.

BufferSize

The size of the *Buffer* in bytes. This number must be a multiple of the intrinsic block size of the device.

Buffer

A pointer to the destination buffer for the data. The caller is responsible for the ownership of the buffer.

Description

The function reads the requested number of blocks from the device. All the blocks are read, or an error is returned. If there is no media in the device, the function returns **EFI_NO_MEDIA**.

Related Definitions

```
//*******************************************************
// EFI_PEI_LBA
//*******************************************************

typedef UINT64             EFI_PEI_LBA;
```

EFI_PEI_LBA is the **UINT64** LBA number.

Status Codes Returned

EFI_SUCCESS	The data was read correctly from the device.
EFI_DEVICE_ERROR	The device reported an error while attempting to perform the read operation.
EFI_INVALID_PARAMETER	The read request contains LBAs that are not valid, or the buffer is not properly aligned.
EFI_NO_MEDIA	There is no media in the device.
EFI_BAD_BUFFER_SIZE	The *BufferSize* parameter is not a multiple of the intrinsic block size of the device.