Intel® Platform Innovation Framework for EFI Architecture Specification

Version 0.9

September 16, 2003

Intel order number xxxxxx-001

Revision History

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

Contents

Figures

Tables

1 Overview

1.1 Executive Summary

This document provides a technical overview of the architecture of the Intel® Platform Innovation Framework for EFI (hereafter referred to as "the Framework"). The Framework is a firmware infrastructure that may be used to initialize and configure systems and then load operating systems (OSs) or embedded operating environments for computers that use processors based on or that are compatible with Intel® Architecture (IA). The Framework differs from previous generations of firmware infrastructure used on IA systems in the following ways:

- It employs a modular component design.
- It uses high-level language coding wherever possible.
- It is designed from the outset to support long-term growth of capabilities in the preboot environment.

To promote interoperability of firmware building blocks in the horizontal industry that is based on IA, such a firmware design needs to be implemented in a high-level language with an open interface that allows other parties such as independent BIOS vendors (IBVs), original equipment manufacturers (OEMs), independent software vendors (ISVs), and independent hardware vendors (IHVs) to add platform innovation around a central framework in the form of plug-in components known as EFI drivers.

The task of boot firmware (whether the BIOS or firmware based on the Framework) is to make a collection of hardware before the boot look like a complete system after the boot. For the foreseeable future, it is less expensive to build chips and boards that power up uninitialized so that, when reset, systems built around these components are in a generally primitive state. They rely heavily on the boot firmware to prepare the system to boot the OS, provide services to the OS (particularly early in the boot process), and provide manageability data on the system.

The Framework architecture supports these requirements using a series of phases, each building on the preceding phase. Each phase is characterized by the resources available to it, the rules by which the code in the phase must abide, and the results of the phase.

The infrastructure available in each phase is provided by the central framework, while the platform-specific features are implemented using intercommunicating modules known as EFI drivers. EFI drivers are somewhat analogous to device drivers in OSs. They provide the Framework architecture with its extensibility and allow it to do the following:

- Meet requirements from a range of platforms
- Incorporate new initiatives and fixes, as well as new hardware
- Support modular software architecture

EFI drivers can be developed at different times by different organizations, which introduces issues that monolithic, traditional BIOSs did not face. The Framework defines powerful solutions for sequencing EFI driver execution, abstracting EFI driver interfaces, and managing shared resources. The Framework and EFI drivers may optionally be cryptographically validated before use to ensure that a chain of trust exists from power-on until the OS boots and beyond.

1.2 Purpose and Intended Audience

This document defines only the highest level of the Framework architecture. This scope is intended to provide readers with a conceptual understanding of the main elements of the Framework architecture design from a functional point of view.

The detail provided in this specification is insufficient to create a prototype. A series of specifications detailing the various programming interfaces at successively more detailed levels is available to expand on the architectural description presented here.

The audience focus for this document is primarily on developers who will be creating or modifying firmware code that is to be used on systems that implement Framework-style firmware. Such developers include the following:

- Silicon support developers intending to create modules and EFI drivers to support processor or chipset components
- IBV and OEM groups that create complete firmware packages for system-level board designs
- IHV developers creating code to support add-in cards or devices

In addition, this document may be of interest to developers who are working on OS loaders or other applications or test programs that are designed to function in the preboot environment.

1.3 Structure of the Document

The remainder of this document is divided into sections. Each section describes either the design of code that implements a phase of operation defined as part of the Framework or the design of a major subsystem of a firmware implementation based on Framework architecture.

Table 1-1 describes the organization of this specification.

Table 1-1. Specification Organization and Contents

Section	Description
1. Overview	Provides a general description of the Framework architecture, and describes the organization, goals, and target audience for the *Framework Architecture Specification.*
2. Security (SEC) Phase	Describes the initial operations after platform reset or power-on to ensure that firmware integrity is intact.
3. Pre-EFI Initialization (PEI) Phase	Provides a detailed description of low-level code to do minimal processor, chipset, and platform configuration to support memory discovery.
4. Driver Execution Environment (DXE) Phase	Provides a detailed description of the operating environment for the majority of firmware code, which is built as EFI drivers, that complete initialization of platform and devices.
5. Boot Device Selection (BDS) Phase	Provides a detailed description of the platform-centered policy engine to determine how an OS or runtime environment to boot is selected and loaded.
6. Runtime (RT) Phase	Describes the code that implements services available from the firmware to the runtime operating environment on the system, typically a shrink-wrap OS.

continued

Table 1-1. Specification Organization and Contents (continued)

Section	Description
7. Afterlife (AL) Phase	Describes the support for the platform retaking control after the normal runtime environment terminates by either user intervention or system failure.
8. Firmware Store	Describes the firmware store; its functions; the formats of modules; and the methods of accessing, updating, and recovering modules in the store.
9. User Interface	Describes the presentation services and infrastructure to support user interactions for operations such as platform or device configuration and BDS policy management.
10. Applied Security	Describes the required and optional security services available in the Framework environment and gives examples of their application in various systems.
11. Manageability	Describes services that support platform-centered management and system state reporting capabilities.
12. Legacy Compatibility	Describes the compatibility code that implements compatibility between the EFI OS and devices requiring a traditional option ROM (OpROM) or applications requiring services otherwise only available through a traditional BIOS.
13. Boot Paths	Describes routes through the Framework implementation that may be taken during power-on, power management events, and other system state transitions.
Glossary	An alphabetical list of terms and abbreviations used in this specification, along with their definitions.
References	Lists the additional documents and specifications that are required or suggested for interpreting the information presented in this document.
Index	Alphabetical index for the *Framework Architecture Specification.*

1.4 The Framework and EFI

The *Extensible Firmware Interface Specification* is a public industry specification that describes an abstract programmatic interface between platform firmware and shrink-wrap OSs or other custom application environments.

The Framework is designed to take advantage of the abstraction inherent in the EFI specification. Firmware based on the Framework architecture will deliver a complete implementation of platform firmware that exports a conforming implementation of the interfaces in the EFI specification.

There are many possible strategies for implementing the EFI specification. One strategy is to layer code that implements the EFI interfaces over a conventional BIOS or System Abstraction Layer (SAL), as demonstrated by the *EFI Sample Implementation* available from the Intel web site. In contrast, the Framework architecture describes a design for a complete replacement for conventional firmware stacks.

While the Framework is but one possible implementation of the EFI specification, Intel views the Framework as the implementation of choice for firmware that can support systems based on all members of the IA family.

1.5 Architectural and Operational Overview

Figure 1-1 shows a block diagram view of the Framework design. The architecture represents a structured implementation composed primarily of the following:

- Foundation code that binds the pieces together
- Modular elements of code that perform the functional job of enumerating and initializing the platform

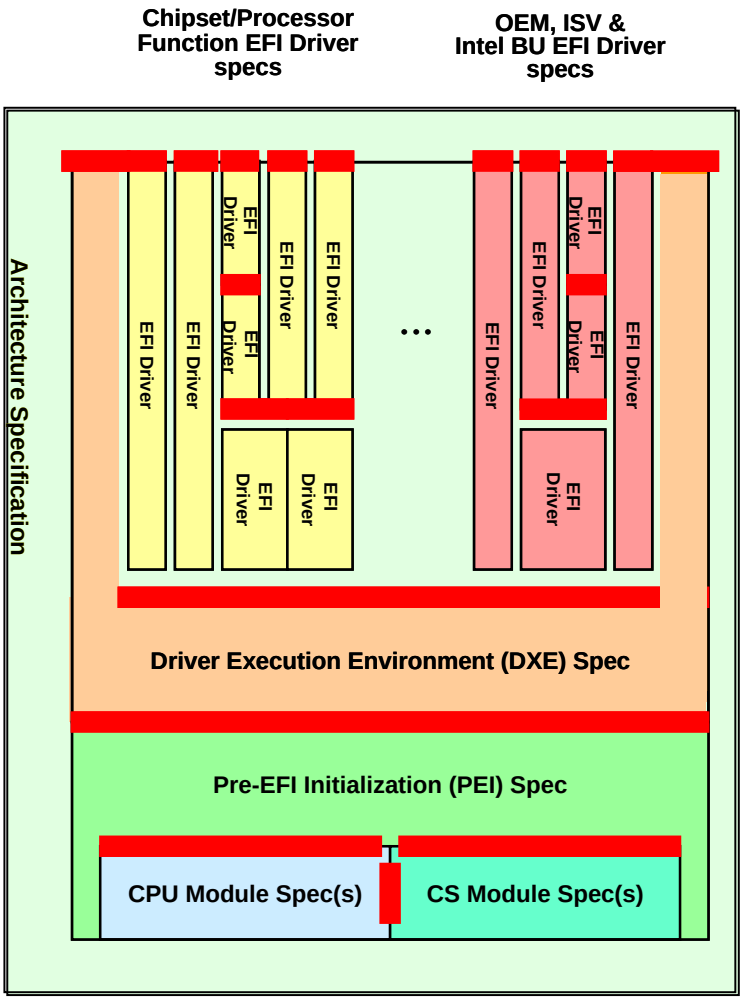

Figure 1-1. Framework Block Diagram Level Architecture

The Framework is further divided into two main parts that operate sequentially:

- Pre-EFI Initialization (PEI) phase
- Driver Execution Environment (DXE) phase

A primary design goal for the Framework architecture is to take advantage of contemporary computer science design principles for software. As a result, a key objective for the design is to place as much of the code as possible in structured, high-level language code and in an architectural structure that follows a coherent object model of the devices and services required in a platform.

Because the high-level language of choice, C, requires memory in the shape of a stack, the first task of the design is to find and initialize at least as much memory as is required to instantiate a stack. It is up to a given implementation based on the Framework whether the amount of memory initialized before DXE includes all or only some portion of the physical memory present in a system. The objective of memory discovery in PEI is to enable C code to be started as close to the reset vector as is practical on any given system. The division between PEI and DXE is hinged on this principle.

PEI represents the smallest possible amount of code that is able to find and initialize memory and other resources that are needed to switch execution into C language code. It consists of the following:

- Foundation "glue code" labeled as *Pre-EFI Initialization (PEI) Spec* in Figure 1-1 above
- A collection of modules that are specific to the processor, chipset, and board layout

In practice, the architecture allows for an arbitrary number of modules. Because the PEI code does the minimum work that is necessary to discover and initialize memory, much of the chipset and other component initialization is deferred until the DXE environment is up and running. Early PEI code is likely written in machine-native assembly code. As a result, it contains the least portable components of any implementation based on the Framework architecture. Clearly, keeping this code to a minimum supports the following goals:

- Maximize the potential to reuse code between platforms.
- Ease the maintenance of Framework code.

Once memory is discovered, PEI marshals a set of handoff state information describing the platform resource map and then initializes and jumps to the DXE Foundation code.

DXE consists of the high-level language portion of the Framework's Foundation code. The code is responsible for materializing intrinsic services such as memory allocation or timer tick that are needed to support generalized modular pieces of code known as *EFI drivers.*

NOTE

It is important to note that "EFI driver" in this context has no connection to OS-present drivers for Windows, Linux*, or any other conventional OS. In the Framework, the term "EFI driver" is used to refer to a modular piece of code that runs in the DXE phase.*

The EFI drivers contain most of the code in the platform that fully enumerates and initializes the platform components, as well as managing the process of handing control to the final operating environment for the system.

EFI drivers in DXE may manage hardware devices or produce services for the preboot environment. The driver model for the Framework is the same driver model that is defined as part of the *EFI 1.10 Specification*, although there are some additional packaging features available for managing EFI drivers that are used in an implementation based on the Framework.

Figure 1-1 shows a number of heavy red lines. These lines represent defined interfaces that are published to promote interoperability of components that may potentially be provided by multiple silicon or add-in component vendors. The top-most line represents the interface presented to OSs. Because firmware based on the Framework is an implementation of the EFI specification, this line therefore represents that specification where the underlying implementation of the services is

provided by a combination of drivers and intrinsic services that is materialized by the DXE portion of the Framework's Foundation code.

The diagram should not be viewed as "to scale" with respect the component areas. Typical implementations of the Framework architecture will contain relatively far more EFI driver code and module code than Foundation code. Over time, the Foundation code is expected to be stable code that binds the other components together and ensures interoperability for components. The Framework architecture is designed to promote innovation by providing a rich environment for writers of drivers that operate in the preboot DXE environment.

Figure 1-2 shows the phases that a platform with Framework-based firmware goes through on a cold boot. This chapter covers the transition from the PEI to the DXE phase, the DXE phase, and the DXE phase's interaction with the BDS phase.

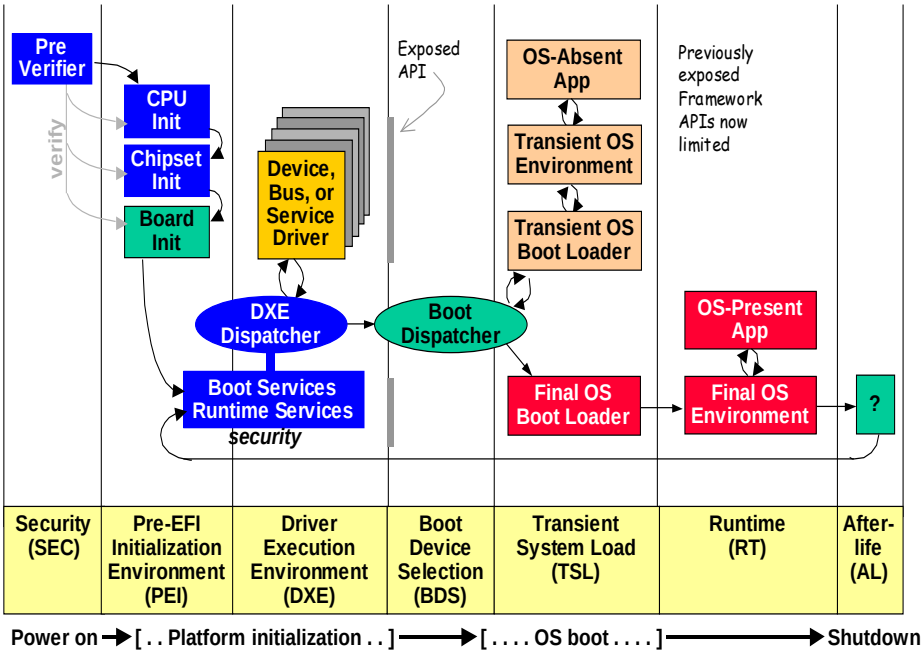

Figure 1-2. Framework Firmware Phases

The Security (SEC) phase supports inspection of the very first opcode that will be executed on the system to ensure that the chosen platform firmware image has not been tampered with. This capability will usually require hardware support that is not typically available in processor and chipset components at the time of writing. However, the architecture provides hooks to support this capability presuming that it may be added in future products.

The PEI phase discovers memory and prepares a resource map for the DXE phase. It is important to note that PEI-to-DXE handoff is a one-way transition—once the initial program loader for DXE is complete, the PEI code is no longer available for use on the platform and DXE becomes a self-contained operating environment.

One of the early tasks of DXE is to find and load the boot manager. This component is responsible for determining what OS (or equivalent) is to be booted. In doing so, it also identifies the required boot devices. Once it knows the intended boot devices, the DXE Dispatcher can find and load the appropriate EFI drivers for that set of devices. Implementations may choose to load all possible EFI drivers or to optimize boot time by loading only those needed for the current boot operation.

After DXE, the Framework has additional phases that cover interaction with the operating system in boot and runtime. The Transient System Load (TSL) phase allows service interfaces to be available to OS loaders before the platform is taken over completely by the OS kernel. The Runtime (RT) phase provides a means to have certain EFI drivers be present during OS execution to support the OS as required. The architecture also contains a "post OS" environment, the Afterlife (AL) phase that is designed to allow platform firmware to execute after the OS in the case of voluntary or involuntary termination of the OS.

The graphic shows one flow of execution from power-on through shutdown. It is important to note that there are several possible execution paths through the Framework design to cope with such things as power management transitions and the like. While these other paths are not illustrated in Figure 1-2, they are described in more detail in the chapters that follow.

1.6 The Framework and Industry Specifications

The Framework was designed in such a way that it could be a complete replacement for existing firmware designs such as BIOS (IA-32) and SAL (Intel® Itanium® processor family). As such, most of the functional capabilities of the resulting firmware need to be the same as those presented by existing firmware solutions. This requirement is realized by providing implementations of EFI drivers or framework interfaces that conform to the standards and specifications established for those capabilities.

For example, Framework-based implementations will completely comply with the ACPI specification, but the construction of that support will be fundamentally different from ACPI implementations in BIOS code.

In addition, as detailed in chapter 12, "Legacy Compatibility," the Framework design includes a provision for conforming implementations to deliver compatibility with a conventional BIOS. The compatibility is centered on supporting OSs and applications that would normally interact with the "runtime" portion of a conventional BIOS. An implementation of the Framework that includes this capability will therefore conform to industry specifications that define the operation of BIOS "runtime" services, table, and data area interfaces.

See the "References" section on page 113 for a complete list of the standards and specifications that are required for establishing these capabilities.

1.7 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 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.
`Italic Monospace`	In code or in text, words in *`Italic Monospace`* indicate placeholder names for variable information that must be supplied (i.e., arguments).

See the “Glossary” section on page 107 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.

2
Security (SEC) Phase

2.1 Introduction

The Security (SEC) phase in the Framework architecture represents the root of trust in a platform that had core root-of-trust maintenance from the reset vector. The SEC phase begins with power-on and includes the first few steps that are required to find and authenticate the PEI Foundation before launching it. The objective is to ensure that the first code executed by the processor is trustworthy and that this code has sufficient resources in and of itself to determine the trustworthiness of any subsequent code. What "authenticate" and "trustworthy" mean can evolve over time and vary across platforms.

2.2 Phase Prerequisites

There are no *software* prerequisites for the SEC phase. The SEC phase is the first code executed after power-on. However, the SEC code will require that the processor have some prior knowledge of the platform configuration and some minimum level of hardware support.

For Framework-conformant systems, this phase leverages a processor capability to produce some temporary memory. This memory might be the processor cache, system static RAM (SRAM), or some other system capability that allows early access to some minimum amount (at least 4 KB) of temporary memory. In addition, this phase will have the *a priori* knowledge of where this early memory can be mapped and the location of the Boot Firmware Volume (BFV). It is from this BFV that the SEC code will discover and possibly authenticate the PEI Foundation. This authentication can be done in the software only or entail the use of additional hardware.

2.3 Handoff Requirements

The SEC code is required to hand information to the PEI Foundation. This handoff information includes the results of the processor built-in self test (BIST) for platforms that collect this information, the address of the BFV, the size of the initial memory, and a possibly nonzero list of PEIM-to-PEIM descriptors. The latter services allow for passing services into the PEI Foundation, including but not limited to the verification service used by the SEC to authenticate the PEI Foundation, pointers to state information such as SAL-A hand-off, and so on.

2.3.1 IA-32

In 32-bit Intel architecture (IA-32), the SEC component is responsible for receiving the results of the processor BIST on platforms that perform BIST. The processor BIST is the integrity checking of the processor behavior by the microcode. In addition, the SEC phase is responsible for initializing some early memory store. On a processor that uses the processor cache as memory, the SEC needs to program the Memory Type Range Registers (MTRRs) to window a region of the address space and program the processor control registers to have the processor cache operate in no-eviction mode. This capability is a means of using the cache as a memory store for stack and data usage. An alternate SEC implementation could use SRAM on the motherboard, chipset, or another location for this store, but the invariant is that it needs to be memory that can be used in conjunction with the permanent memory.

The early memory region, size, and location needs to be passed into the PEI Foundation, along with the PEIM-to-PEIM Interfaces (PPIs) in SEC and BIST results, if any.

2.3.2 Intel® Itanium® Processor Family

The Itanium processor family is similar to IA-32 in that it is also responsible for ascertaining the processor BIST information and initializing the early memory store. To determine the BIST information, the Processor Abstraction Layer (PAL) hand-off state is used. To initialize the early memory store, if the platform uses the processor cache as memory, there is a distinguished PAL call that must be made to initialize the cache for this early, no-eviction mode usage. In the Itanium processor family, the no-eviction mode supports caching of the ROM in addition to the stack and data store. Also, the no-eviction mode on Itanium processors supports multiprocessor (MP) execution with each processor having a discontiguous mapped region for the no-eviction range of memory. The PAL-A code is also encapsulated in the Itanium® architecture variant of SEC. The PAL-A code is the equivalent of IA-32 processor microcode for the Itanium processor family.

2.4 Services

In addition, the SEC phase has a table that includes where the temporary memory is mapped. This table is important in that a given vendor may have a particular region of the address space that would be “safer” for this mode of execution. By “safe” we mean that the bus cycle would not cause errant behavior in the chipset if there is an unexpected write-back from the early-memory space that was using the processor cache in no-eviction mode.

As noted above, the SEC phase passes in a possibly non-NULL PEI PPI descriptor list. A set of services that might be conveyed into the PEI Foundation from the SEC phase could also include the Security PPI and a Trusted Computing Group (TCG) PPI for Hash-Extend and event logging, in addition to the information and verification services. These services allow the Core Root-of-Trust Module (CRTM) to have the necessary services to authenticate the PEI Foundation and allow the PEI Foundation to reuse these services using PPI invocations that call back into SEC, as opposed to duplicating them in the implementation of the PEI Foundation.

2.5 Security Use Models

The SEC component allows for an array of security use models, which are described in the following sections.

2.5.1 Core Root-of-Trust Module (CRTM)

The SEC component is packaged as a Firmware File System (FFS) file of type Volume Top File (VTF) to ensure that it is aligned at the top of the 4 GB address space to decode the reset vector on IA-32 and Itanium processor family class systems. For Intel® processors based on Intel® XScale™ technology, the file may be at address zero, but the key requirement is to be at the address where the initial processor code fetch occurs in response to a reset.

As a result of this initial execution, the code operating in the SEC phase represents the CRTM in the Framework firmware. This representation is in contrast to a traditional PC-AT* BIOS where the BIOS boot-block is often referred to as the CRTM. This CRTM distinction is important in that it represents the Trusted Computing Base (TCB), or the smallest component that would need to have third-party scrutiny and examination for purposes of a security audit, for example. The code will need this scrutiny if there is a chain-of-trust model for the platform, and this code is the first to execute therein. Other components that would be part of this TCB and are described later might include the PEI Foundation, DXE Foundation, and associated Security Architectural Protocol.

2.5.2 Attested Boot

For an attested boot scheme, the SEC component could provide the CRTM. Therein, the SEC phase would have knowledge about a security coprocessor, such as the Trusted Processing Module (TPM) in TCG. The SEC component would perform the Hash-Extend operation on both itself and the subsequent module to which it passes control, namely the PEI Foundation. In addition, a portion of the early memory can be used as the beginning of the event log to describe the attestation operations.

2.5.3 Processor Root-of-Trust Key and RAM

In the future, if a public/private key pair representing a “root of trust” is embedded in the processor, the SEC module would be the module signed with the private key so that the microcode could verify the SEC file against the associated public key on a given reset. This design would provide a true root-of-trust maintenance handoff in which the transitive trust relationship would extend from the processor, wherein microcode could be the CRTM and the associated Framework code in SEC would inherit the thread of execution. If this trust relationship were not satisfied by the SEC with respect to the processor, such as having the signature verification fail, the processor would engender some recovery condition or cease fetching the ensuing opcodes from the firmware store.

3
Pre-EFI Initialization (PEI) Phase

3.1 Introduction

The Pre-EFI Initialization (PEI) phase provides a standardized method of loading and invoking specific initial configuration routines for the processor, chipset, and motherboard. The PEI phase occurs after the Security (SEC) phase. The primary purpose of code operating in this phase is to initialize enough of the system to allow instantiation of the Driver Execution Environment (DXE) phase. At a minimum, the PEI phase is responsible for determining the system boot path and initializing and describing a minimum amount of system RAM and firmware volume(s) that contain the DXE Foundation and DXE Architectural Protocols.

3.1.1 Scope

The PEI phase is responsible for initializing enough of the system to provide a stable base for follow-on phases. It is also responsible for detecting and recovering from corruption of the firmware storage space.

A 2000-era PC generally starts execution in a very primitive state. Processors might need updates to their internal microcode; the chipset (the chips that provide the interface between processors and the other major components of the system) require considerable initialization; and RAM requires sizing, location, and other initialization. The PEI phase is responsible for initializing these basic subsystems. The PEI phase is intended to provide a simple infrastructure by which a limited set of tasks can easily be accomplished to transition to the more advanced DXE phase. The PEI phase is intended to be responsible for only a very small subset of tasks that are required to boot the platform; in other words, it should perform only the minimal tasks that are required to start DXE. As improvements in the hardware occur, some of these tasks may migrate out of the PEI phase of execution.

3.1.2 Rationale

The design for PEI is essentially a miniature version of DXE that addresses many of the same issues. The PEI phase consists of several parts:

- A PEI Foundation
- One or more Pre-EFI Initialization Modules (PEIMs)

The goal is for the PEI Foundation to remain relatively constant for a particular processor architecture and to support add-in modules from various vendors for particular processors, chipsets, platforms, and other components. These modules usually cannot be coded without some interaction between one another and, even if they could, it would be inefficient to do so.

PEI is unlike DXE in that DXE assumes that reasonable amounts of permanent system RAM are present and available for use. PEI instead assumes that only a limited amount of temporary RAM exists and that it could be reconfigured for other uses during the PEI phase after permanent system RAM has been initialized. As such, PEI does not have the rich feature set that DXE does. The following are the most obvious examples of this difference:

- DXE has a rich database of loaded images and protocols bound to those images.
- PEI lacks a rich module hierarchy such as the DXE driver model.

3.1.3 Overview

The PEI phase consists of some Foundation code and specialized drivers known as PEIMs that customize the PEI phase operations to the platform. It is the responsibility of the Foundation code to dispatch the plug-ins in a sequenced order and provide basic services. The PEIMs are analogous to DXE drivers and generally correspond to the components being initialized. It is expected that common practice will be that the vendor of the component will provide the PEIM, possibly in source form so the customer can quickly debug integration problems.

The implementation of the PEI phase is more dependent on the processor architecture than any other Framework phase. In particular, the more resources that the processor provides at its initial or near initial state, the richer the PEI environment will be. As such, there are several parts of the following discussion that note requirements for the architecture but are otherwise left less completely defined because they are processor architecture specific.

3.2 Phase Prerequisites

Sections 3.2.1 through 3.2.3 describe the prerequisites necessary for the successful completion of the PEI phase.

3.2.1 Temporary RAM

The PEI Foundation requires that the SEC phase initialize a minimum amount of scratch pad RAM that can be used by the PEI phase as a data store until system memory has been fully initialized. This scratch pad RAM should have access properties similar to normal system RAM—through memory cycles on the front side bus, for example. After system memory is fully initialized, the temporary RAM may be reconfigured for other uses. Typical provision for the temporary RAM is an architectural mode of the processor's internal caches.

3.2.2 Boot Firmware Volume

The Boot Firmware Volume (BFV) contains the PEI Foundation and PEIMs. It must appear in the memory address space of the system without prior firmware intervention and will typically contain the reset vector for the processor architecture.

The contents of the BFV follow the format of the EFI flash file system. The PEI Foundation will follow the EFI flash file system format to find PEIMs in the BFV. A platform-specific PEIM may inform the PEI Foundation of the location of other firmware volumes in the system, which allows the PEI Foundation to find PEIMs in other firmware volumes. The PEI Foundation and PEIMs are named by unique IDs in the EFI flash file system.

The PEI Foundation and some PEIMs required for recovery must either be locked into a nonupdateable BFV or be able to be updated using a fault-tolerant mechanism. The EFI flash file system provides error recovery; if the system halts at any point, either the old (preupdate) PEIM(s) or the newly updated PEIM(s) are entirely valid and the PEI Foundation can determine which is valid.

3.2.3 Security Primitives

The SEC phase provides an interface to the PEI Foundation to perform verification operations. To continue the root of trust, the PEI Foundation will use this mechanism to validate various PEIMs.

3.3 Concepts

Sections 3.3.1 through 3.3.6 describe the concepts in the PEI phase design.

3.3.1 PEI Foundation

The PEI Foundation is a single binary executable that is compiled to function with each processor architecture. It performs two main functions:

- Dispatching PEIMs
- Providing a set of common core services used by PEIMs

The PEI Dispatcher's job is to hand control to the PEIMs in an orderly manner. The common core services are provided through a service table referred to as the PEI Services Table. These services do the following:

- Assist in PEIM-to-PEIM communication.
- Abstract management of the temporary RAM.
- Provide common functions to assist the PEIMs in the following:
  - Finding other files in the FFS
  - Reporting status codes
  - Preparing the handoff state for the next phase of the Framework

When the SEC phase is complete, SEC invokes the PEI Foundation and provides the PEI Foundation with several parameters:

- The location and size of the BFV so that the PEI Foundation knows where to look for the initial set of PEIMs.
- A minimum amount of temporary RAM that the PEI phase can use
- A verification service callback to allow the PEI Foundation to verify that PEIMs that it discovers are authenticated to run before the PEI Foundation dispatches them

The PEI Foundation assists PEIMs in communicating with each other. The PEI Foundation maintains a database of registered interfaces for the PEIMs. These interfaces are called PEIM-to-PEIM Interfaces (PPIs). The PEI Foundation provides the interfaces to allow PEIMs to register PPIs and to be notified (called back) when another PEIM installs a PPI.

The PEI Dispatcher consists of a single phase. It is during this phase that the PEI Foundation examines each file in the firmware volumes that contain files of type PEIM. It examines the dependency expression (depex) within each firmware file to decide if a PEIM can run. A dependency expression is code associated with each driver that describes the dependencies that must be satisfied for that driver to run. The binary encoding of dependency expressions for PEIMs is the same as that of dependency expressions associated with a DXE driver.

3.3.2 Pre-EFI Initialization Modules (PEIMs)

Pre-EFI Initialization Modules (PEIMs) are executable binaries that encapsulate processor, chipset, device, or other platform-specific functionality. PEIMs may provide interface(s) that allow other PEIMs or the PEI Foundation to communicate with the PEIM or the hardware for which the PEIM abstracts. PEIMs are separately built binary modules that typically reside in ROM and are therefore uncompressed. A small subset of PEIMs exist that may run from RAM for performance reasons. These PEIMs will reside in ROM in a compressed format. PEIMs that reside in ROM are execute-in-place modules that may consist of either position-independent code or position-dependent code with relocation information.

3.3.3 PEI Services

The PEI Foundation establishes a system table named the PEI Services Table that is visible to all PEIMs in the system. A PEI service is defined as a function, command, or other capability that is manifested by the PEI Foundation when that service's initialization requirements are met. Because the PEI phase has no permanent memory available until nearly the end of the phase, the range of services created during the PEI phase cannot be as rich as those created during later phases. Because the location of the PEI Foundation and its temporary RAM is not known at build time, a pointer to the PEI Services Table is passed into each PEIM's entry point and also to part of each PPI. The PEI Foundation provides the following classes of services:

- **PPI Services:** Manages PPIs to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
- **Boot Mode Services:** Manages the boot mode (S3, S5, normal boot, diagnostics, etc.) of the system.
- **HOB Services:** Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the Framework.
- **Firmware Volume Services:** Walks the FFS in firmware volumes to find PEIMs and other firmware files in the flash device.
- **PEI Memory Services:** Provides a collection of memory management services for use both before and after permanent memory has been discovered.
- **Status Code Services:** Common progress and error code reporting services, i.e. port 080h or a serial port for simple text output for debug.
- **Reset Services:** Provides a common means by which to initiate a restart of the system.

3.3.4 PEIM-to-PEIM Interfaces (PPIs)

PEIMs may invoke other PEIMs through interfaces named PEIM-to-PEIM Interfaces (PPIs). The interfaces themselves are named using Globally Unique Identifiers (GUIDs) to allow the independent development of modules and their defined interfaces without naming collision. A GUID is a 128-bit value used to differentiate services and structures in the boot services environment (also see section 11.2.1). The PPIs are defined as structures that may contain functions, data, or a combination of the two. PEIMs must register their PPIs with the PEI Foundation, which manages a database of registered PPIs. A PEIM that wants to use a specific PPI can then query the PEI Foundation to find the interface it needs. There are two types of PPIs:

- Services
- Notifications

PPI services allow a PEIM to provide functions or data for another PEIM to use. PPI notifications allow a PEIM to register for a callback when another PPI is registered with the PEI Foundation.

3.3.5 Simple Heap

The PEI Foundation uses temporary RAM to provide a simple heap store before permanent system memory is installed. PEIMs may request allocations from the heap, but there is no mechanism to free memory from the heap. Once permanent memory is installed, the heap is relocated to permanent system memory, but the PEI Foundation does not fix up existing data within the heap. Therefore, a PEIM cannot store pointers in the heap when the target is other data within the heap (i.e. linked lists).

3.3.6 Hand-Off Blocks (HOBs)

Hand-Off Blocks (HOBs) are the architectural mechanism for passing system state information from the PEI phase to the DXE phase in the Framework architecture. A HOB is simply a data structure (cell) in memory that contains a header and data section. The header definition is common for all HOBs and allows any code using this definition to know two items:

- The format of the data section
- The total size of the HOB

HOBs are allocated sequentially in the memory that is available to PEIMs after permanent memory has been installed. There are a series of core services that facilitate HOB manipulation, as noted in section 3.3.3. This sequential list of HOBs in memory will be referred to as the *HOB list.* The first HOB in the HOB list must be the Phase Handoff Information Table (PHIT) HOB that describes the physical memory used by the PEI phase and the boot mode discovered during the PEI phase.

Only PEI components are allowed to make additions or changes to HOBs. Once the HOB list is passed into DXE, it is effectively read-only for DXE components. The ramifications of a read-only HOB list for DXE is that handoff information, such as boot mode, must be handled in a unique fashion; if DXE were to engender a recovery condition, it would not update the boot mode but instead would implement the action using a special type of reset call. The HOB list contains system state data at the time of PEI-to-DXE handoff and does not represent the current system state during DXE. DXE components should use services that are defined for DXE to get the current system state instead of parsing the HOB list.

As a guideline, it is expected that HOBs passed between PEI and DXE will follow a one producer–to–one consumer model. In other words, a PEIM will produce a HOB in PEI, and a DXE Driver will consume that HOB and pass information associated with that HOB to other DXE components that need the information. The methods that the DXE Driver uses to provide that information to other DXE components should follow mechanisms defined by the DXE architecture.

3.4 Operation

PEI phase operation consists of invoking the PEI Foundation, dispatching all PEIMs in an orderly manner, and discovering and invoking the next phase. During PEI Foundation initialization, the PEI Foundation initializes the internal data areas and functions that are needed to provide the common PEI services to PEIMs. During PEIM dispatch, the PEI Dispatcher traverses the firmware volume(s) and discovers PEIMs according to the flash file system definition. The PEI Dispatcher then dispatches PEIMs if the following criteria are met:

- The PEIM has not already been invoked.
- The PEIM file is correctly formatted.
- The PEIM is trustworthy.
- The PEIM's dependency requirements have been met.

After dispatching a PEIM, the PEI Dispatcher will continue traversing the firmware volume(s) until either all discovered PEIMs have been invoked or no more PEIMs can be invoked because the requirements listed above cannot be met for any PEIMs. Once this condition has been reached, the PEI Dispatcher's job is complete and it invokes an architectural PPI for starting the next phase of the Framework, the DXE Initial Program Load (IPL) PPI.

3.4.1 Dependency Expressions

The sequencing of PEIMs is determined by evaluating a *dependency expression* associated with each PEIM. This Boolean expression describes the requirements that are necessary for that PEIM to run, which imposes a weak ordering on the PEIMs. Within this weak ordering, the PEIMs may be initialized in any order. The GUIDs of PPIs and the GUIDs of file names are referenced in the dependency expression. The dependency expression is a representative syntax of operations that can be performed on a plurality of dependencies to determine whether the PEIM can be run. The PEI Foundation evaluates this dependency expression against an internal database of run PEIMs and registered PPIs. Operations that may be performed on dependencies are the logical operators **`AND`**, **`OR`**, and **`NOT`** and the sequencing operators **`BEFORE`** and **`AFTER`**.

3.4.2 Verification / Authentication

The PEI Foundation will be stateless with respect to security. Instead, security decisions are assigned to platform-specific components. The two components of interest that abstract security include the Security PPI and a Verification PPI. The purpose of the Verification PPI is to check the authentication status of a given PEIM. The mechanism used therein may include digital signature verification, a simple checksum, or some other OEM-specific mechanism. The result of this verification will be returned to the PEI Foundation, which in turn will convey the result to the Security PPI. The Security PPI will decide whether to defer execution of the PEIM or to let the execution occur. In addition, the Security PPI provider may choose to generate an attestation log entry of the dispatched PEIM or provide some other security exception.

3.4.3 PEIM Execution

PEIMs run to completion when invoked by the PEI Foundation. Each PEIM is invoked only once and must perform its job with that invocation and install other PPIs to allow other PEIMs to call it as necessary. PEIMs may also register for a notification callback if it is necessary for the PEIM to get control again after another PEIM has run.

3.4.4 Memory Discovery

Memory discovery is an important architectural event during the PEI phase. When a PEIM has successfully discovered, initialized, and tested a contiguous range of system RAM, it reports this RAM to the PEI Foundation. When that PEIM exits, the PEI Foundation will migrate PEI usage of the temporary RAM to real system RAM, which involves the following two tasks:

- The PEI Foundation must switch PEI stack usage from temporary RAM to permanent system memory.
- The PEI Foundation must migrate the simple heap allocated by PEIMs (including HOBs) to real system RAM.

Once this process is complete, the PEI Foundation installs an architectural PPI to notify any interested PEIMs that real system memory has been installed. This notification allows PEIMs that ran before memory was installed to be called back so that they can complete necessary tasks—such as building HOBs for the next phase of DXE—in real system memory.

3.4.5 Intel Itanium Processor MP Considerations

This section gives special consideration to the PEI phase operation in Itanium processor family multiprocessor (MP) systems. In Itanium®-based systems, all of the processors in the system start up simultaneously and execute the PAL initialization code that is provided by the processor vendor. Then all the processors call into the Framework start-up code with a request for recovery check. The start-up code allocates different chunks of temporary memory for each of the active processors and sets up stack and backing store pointers in the allocated temporary memory. The temporary memory could be a part of the processor cache (cache as RAM), which can be configured by invoking a PAL call. The start-up code then starts dispatching PEIMs on each of these processors. One of the early PEIMs that runs in MP mode is the PEIM that selects one of the processors as the boot-strap processor (BSP) for running the PEIM stage of the booting.

This BSP continues to run PEIMs until it finds permanent memory and installs the memory with the PEI Foundation. Then the BSP wakes up all the processors to determine their health and PAL compatibility status. If none of these checks warrants a recovery of the firmware, the processors are returned to the PAL for more processor initialization and a normal boot.

The Framework start-up code also gets triggered in an Itanium-based system whenever an INIT or a Machine Check Architecture (MCA) event occurs in the system. Under such conditions, the PAL code outputs status codes and a buffer called the *minimum state buffer*. A Framework-specific data pointer that points to the INIT and MCA code data area is attached to this minimum state buffer, which contains details of the code to be executed upon INIT and MCA events. The buffer also holds some important variables needed by the start-up code to make decisions during these special hardware events.

3.4.6 Recovery

Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes on nonvolatile storage devices are managed as blocks. If the system loses power while a block or semantically bound blocks are being updated, the storage might become invalid. On the other hand, the device might become corrupted by an errant program or by errant hardware. Assuming PEI lives in a fault-tolerant block, it can support a recovery mode dispatch.

A PEIM or the PEI Foundation itself can discover the need to do recovery. A PEIM can check a "force recovery" jumper, for example, to detect a need for recovery. The PEI Foundation might discover that a particular PEIM does not validate correctly or that an entire firmware volume has become corrupted.

The concept behind recovery is that enough of the system firmware is preserved so that the system can boot to a point that it can read a copy of the data that was lost from chosen peripherals and then reprogram the firmware volume with that data.

Preservation of the recovery firmware is a function of the way the firmware volume store is managed. In the EFI flash file system, PEIMs required for recovery will be marked as such. The firmware volume store architecture must then preserve marked items, either by making them unalterable (possibly with hardware support) or protect them using a fault-tolerant update process.

Until recovery mode has been discovered, the PEI Dispatcher proceeds as normal. If it encounters PEIMs that have been corrupted (for example, by receiving an incorrect hash value), it must change the boot mode to "recovery." Once set to recovery, other PEIMs must not change it to one of the other states. After the PEI Dispatcher has discovered that the system is in recovery mode, it will restart itself, dispatching only those PEIMs that are required for recovery. It is also possible for a PEIM to detect a catastrophic condition or to be a forced-recovery detect PEIM and to inform the PEI Dispatcher that it needs to proceed with a recovery dispatch. The recovery dispatch is completed when a PEIM finds a recovery firmware volume on a recovery media and the DXE Foundation is started from that firmware volume. Drivers within that DXE firmware volume can perform the recovery process.

3.4.7 S3 Resume

The PEI phase on S3 resume (save-to-RAM resume) differs in several fundamental ways from the PEI phase on a normal boot. The differences are as follows:

- The memory subsection is restored to its presleep state rather than initialized.
- System memory owned by the OS is not used by either the PEI Foundation or the PEIMs.
- The DXE phase is not dispatched on a resume because it would corrupt memory.
- The PEIM that would normally dispatch the DXE phase instead uses a special Hardware Save Table to restore fundamental hardware back to a boot configuration. After restoring the hardware, the PEIM passes control to the OS-supplied resume vector.
- The DXE and later phases during a normal boot save enough information in the Framework reserved memory or a firmware volume area for hardware to be restored to a state that the OS can use to restore devices. This saved information is located in the Hardware Save Table.

4
Driver Execution Environment (DXE) Phase

4.1 Introduction

The Driver Execution Environment (DXE) phase contains an implementation of EFI that is compliant with the *EFI 1.10 Specification*. As a result, both the DXE Foundation and DXE drivers share many of the attributes of EFI images. The DXE phase is the phase where most of the system initialization is performed. The Pre-EFI Initialization (PEI) phase is responsible for initializing permanent memory in the platform so the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position-independent data structures called Hand-Off Blocks (HOBs).

There are several components in the DXE phase, as follows:

- DXE Foundation
- DXE Dispatcher
- A set of DXE drivers

The DXE Foundation produces a set of Boot Services, Runtime Services, and DXE Services. The DXE Dispatcher is responsible for discovering and executing DXE drivers in the correct order. The DXE drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for console and boot devices. These components work together to initialize the platform and provide the services required to boot an OS. The DXE and Boot Device Selection (BDS) phases work together to establish consoles and attempt the booting of OSs. The DXE phase is terminated when an OS successfully begins its boot process (i.e., the BDS phase starts). Only the runtime services provided by the DXE Foundation and services provided by runtime DXE drivers are allowed to persist into the OS runtime environment. The result of DXE is the presentation of a fully formed EFI interface.

Figure 4-1 shows the phases that a platform with Framework firmware goes through on a cold boot. This chapter covers the following:

- Transition from the PEI to the DXE phase
- The DXE phase
- The DXE phase's interaction with the BDS phase

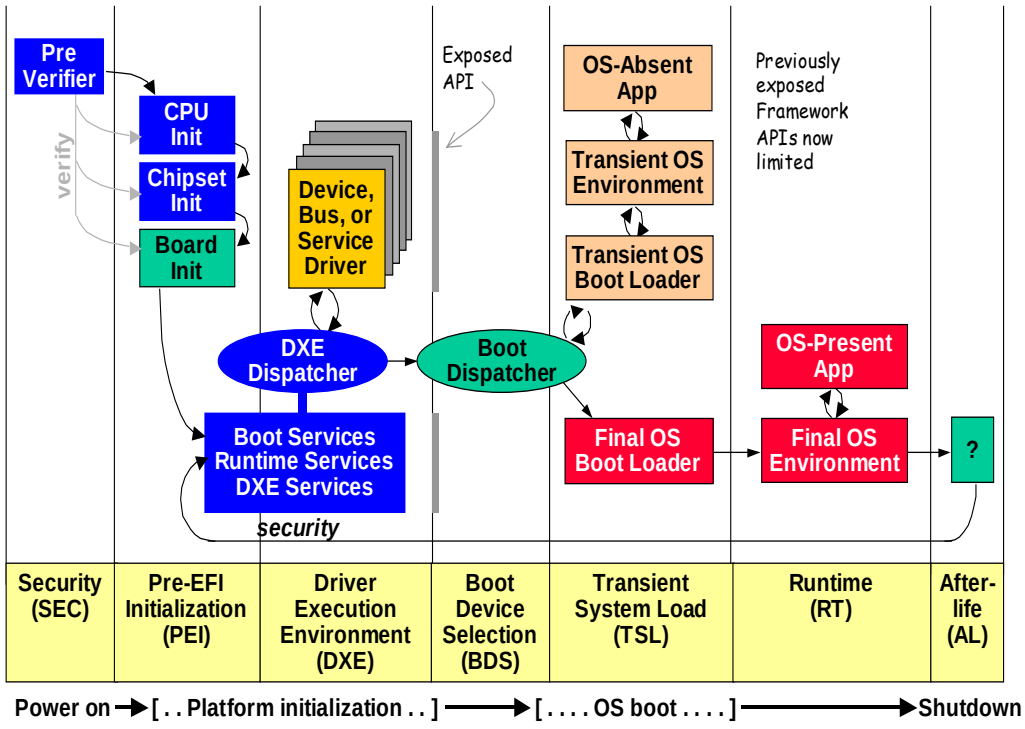

Figure 4-1. Framework Firmware Phases

4.2 DXE Foundation

The DXE Foundation is designed to be completely portable with no processor, chipset, or platform dependencies. This portability is accomplished by designing in several features:

- **The DXE Foundation depends only upon a HOB list for its initial state.** This single dependency means that the DXE Foundation does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE Foundation.
- **The DXE Foundation does not contain any hard-coded addresses.** As a result, the DXE Foundation can be loaded anywhere in physical memory, and it can function correctly no matter where physical memory or where firmware volumes are located in the processor's physical address space.
- **The DXE Foundation does not contain any processor-specific, chipset-specific, or platform-specific information.** Instead, the DXE Foundation is abstracted from the system hardware through a set of architectural protocol interfaces. These architectural protocol interfaces are produced by a set of DXE drivers that are invoked by the DXE Dispatcher.

The DXE Foundation must produce the EFI System Table and its associated set of EFI Boot Services and EFI Runtime Services. The DXE Foundation also contains the DXE Dispatcher, whose main purpose is to discover and execute DXE drivers stored in firmware volumes. The order in which DXE drivers are executed is determined by a combination of the optional *a priori* file (see section 4.3.1) and the set of dependency expressions that are associated with the DXE drivers. The firmware volume file format allows dependency expressions to be packaged with the executable DXE driver image. DXE drivers utilize a PE/COFF image format, so the DXE Dispatcher must also contain a PE/COFF loader to load and execute DXE drivers.

The DXE Foundation must also maintain a *handle database*. A handle database is a list of one or more handles, and a *handle* is a list of one or more unique protocol GUIDs. A *protocol* is a software abstraction for a set of services. Some protocols abstract I/O devices, and other protocols abstract a common set of system services. A protocol typically contains a set of APIs and some number of data fields. Every protocol is named by GUID, and the DXE Foundation produces services that allow protocols to be registered in the handle database. As the DXE Dispatcher executes DXE drivers, additional protocols will be added to the handle database including the DXE Architectural Protocols that are used to abstract the DXE Foundation from platform-specific details.

4.2.1 Hand-Off Block (HOB) List

The HOB list contains all the information that the DXE Foundation requires to produce its memory-based services. The HOB list contains information on the boot mode, the processor's instruction set, and the memory that was discovered in the PEI phase. It also contains a description of the system memory that was initialized by the PEI phase, along with information about the firmware devices that were discovered in the PEI phase. The firmware device information includes the system memory locations of the firmware devices and of the firmware volumes that are contained within those firmware devices. The firmware volumes may contain DXE drivers, and the DXE Dispatcher is responsible for loading and executing the DXE drivers that are discovered in those firmware volumes. Finally, the HOB list may contain the I/O resources and memory-mapped I/O resources that were discovered in the PEI phase.

Figure 4-2 shows an example HOB list. The first entry in the HOB list is always the Phase Handoff Information Table (PHIT) HOB that contains the boot mode. The rest of the HOB list entries can appear in any order. This example shows the different types of system resources that can be described in a HOB list. The most important ones to the DXE Foundation are the HOBs that describe system memory and the HOBs that describe firmware volumes. A HOB list is always terminated by an end-of-list HOB. The one additional HOB type that is not shown in the figure is the GUID extension HOB that allows a PEIM to pass private data to a DXE driver. Only the DXE driver that recognizes the GUID value in the GUID extension HOB will be able to understand the data in that HOB. The HOB entries are all designed to be position independent. This independence allows the DXE Foundation to relocate the HOB list to a different location if it is not suitable to the DXE Foundation.

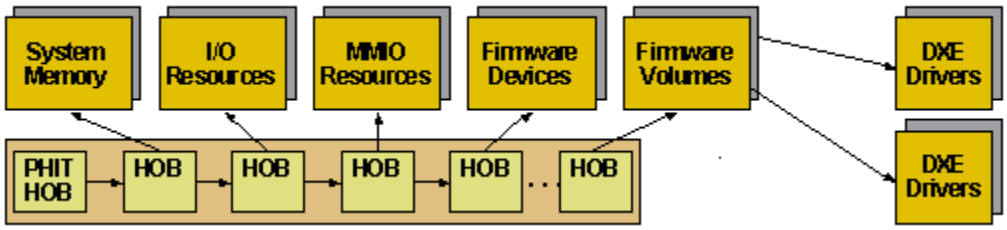

Figure 4-2. HOB List

4.2.2 DXE Architectural Protocols

The DXE Foundation is abstracted from the platform hardware through a set of DXE Architectural Protocols. These protocols function just like other protocols in every way, except that the DXE Foundation consumes these protocols to produce the EFI Boot Services and EFI Runtime Services. DXE drivers that are loaded from firmware volumes produce the DXE Architectural Protocols. This design means that the DXE Foundation must have enough services to load and start DXE drivers before even a single DXE driver is executed.

The DXE Foundation is passed a HOB list that must contain a description of some amount of system memory and at least one firmware volume. The system memory descriptors in the HOB list are used to initialize the EFI services that require only memory to function correctly. The system is also guaranteed to be running on only one processor in flat physical mode with interrupts disabled. The firmware volume is passed to the DXE Dispatcher, which must contain a read-only FFS driver

to search for the *a priori* file and any DXE drivers in the firmware volumes. When a driver is discovered that needs to be loaded and executed, the DXE Dispatcher will use a PE/COFF loader to load and invoke the DXE driver. The early DXE drivers will produce the DXE Architectural Protocols, so the DXE Foundation can produce the full complement of EFI Boot Services and EFI Runtime Services. Figure 4-3 shows the HOB list being passed to the DXE Foundation. The DXE Foundation consumes the services of the DXE Architectural Protocols shown in the figure and then produces the EFI System Table, EFI Boot Services Table, and the EFI Runtime Services Table.

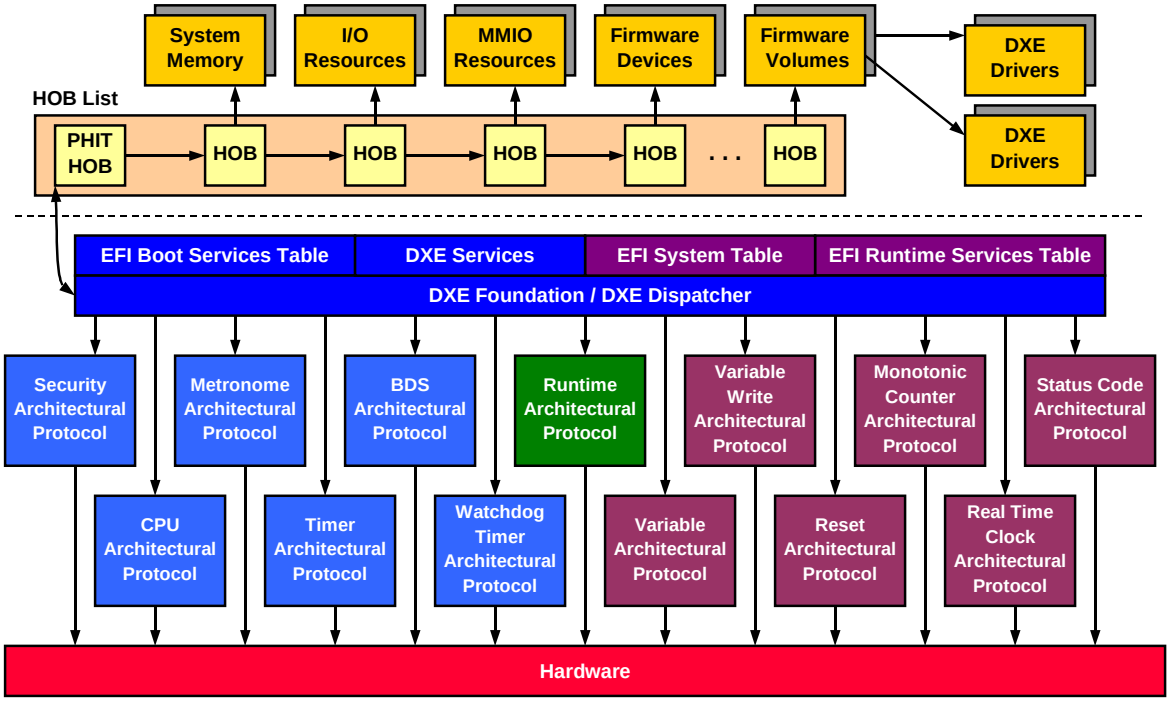

Figure 4-3. DXE Architectural Protocols

Figure 4-3 shows all the major components present in the DXE phase. The EFI Boot Services Table and DXE Services Table shown on the left are allocated from EFI boot services memory. This allocation means that the EFI Boot Services Table and DXE Services Table are freed when the OS runtime phase is entered. The EFI System Table and EFI Runtime Services Table on the right are allocated from EFI runtime services memory, and they do persist into the OS runtime phase.

The DXE Architectural Protocols shown on the left in the figure are used to produce the EFI Boot Services. The DXE Foundation, DXE Dispatcher, and the protocols shown on the left will be freed when the system transitions to the OS runtime phase. The DXE Architectural Protocols shown on the right are used to produce the EFI Runtime Services. These services will persist in the OS

runtime phase. The Runtime Architectural Protocol in the middle is special. This protocol provides the services that are required to transition the runtime services from physical mode to virtual mode under the direction of an OS. Once this transition is complete, these services can no longer be used.

The following is a brief summary of the DXE Architectural Protocols:

- **Security Architectural Protocol:** Allows the DXE Foundation to authenticate files stored in firmware volumes before they are used.
- **CPU Architectural Protocol:** Provides services to manage caches, manage interrupts, retrieve the processor's frequency, and query any processor-based timers.
- **Metronome Architectural Protocol:** Provides the services required to perform very short calibrated stalls.
- **Timer Architectural Protocol:** Provides the services required to install and enable the heartbeat timer interrupt required by the timer services in the DXE Foundation.
- **BDS Architectural Protocol:** Provides an entry point that the DXE Foundation calls once after all of the DXE drivers have been dispatched from all of the firmware volumes. This entry point is the transition from the DXE phase to the BDS phase, and it is responsible for establishing consoles and enabling the boot devices required to boot an OS.
- **Watchdog Timer Architectural Protocol:** Provides the services required to enable and disable a watchdog timer in the platform.
- **Runtime Architectural Protocol:** Provides the services required to convert all runtime services and runtime drivers from physical mappings to virtual mappings.
- **Variable Architectural Protocol:** Provides the services to retrieve environment variables and set volatile environment variables.
- **Variable Write Architectural Protocol:** Provides the services to set nonvolatile environment variables.
- **Monotonic Counter Architectural Protocol:** Provides the services required by the DXE Foundation to manage a 64-bit monotonic counter.
- **Reset Architectural Protocol:** Provides the services required to reset or shut down the platform.
- **Status Code Architectural Protocol:** Provides the services to send status codes from the DXE Foundation or DXE drivers to a log or device.
- **Real Time Clock Architectural Protocol:** Provides the services to retrieve and set the current time and date as well as the time and date of an optional wakeup timer.

4.2.3 EFI System Table

The DXE Foundation produces the EFI System Table, which is consumed by every DXE driver and executable image invoked by BDS. It contains all the information that is required for these components to use the services provided by the DXE Foundation and any previously loaded DXE driver. Figure 4-4 shows the various components that are available through the EFI System Table.

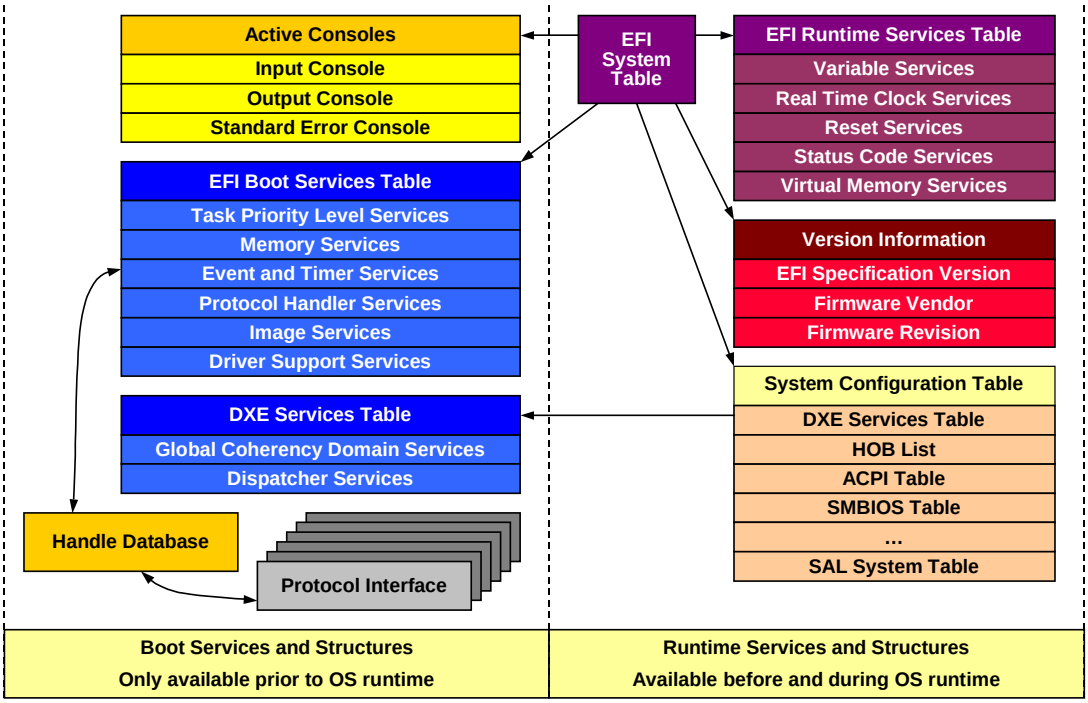

Figure 4-4. EFI System Table and Related Components

The DXE Foundation produces the EFI Boot Services, EFI Runtime Services, and DXE Services with the aid of the DXE Architectural Protocols. The EFI System Table provides access to all the active console devices in the platform and the set of EFI Configuration Tables. The EFI Configuration Tables are an extensible list of tables that describe the configuration of the platform including pointers to tables such as DXE Services, the HOB list, ACPI, System Management BIOS (SMBIOS), and the SAL System Table. This list may be expanded in the future as new table types are defined. Also, through the use of the Protocol Handle Services in the EFI Boot Services Table, any executable image can access the handle database and any of the protocol interfaces that have been registered by DXE drivers.

When the transition to the OS runtime is performed, the handle database, active consoles, EFI Boot Services, and services provided by boot service DXE drivers are terminated. This termination frees more memory for use by the OS and leaves the EFI System Table, EFI Runtime Services Table, and the system configuration tables available in the OS runtime environment. There is also the option of converting all of the EFI Runtime Services from a physical address space to an OS-specific virtual address space. This address space conversion may only be performed once.

4.2.4 EFI Boot Services Table

The following is a brief summary of the services that are available through the EFI Boot Services Table:

- **Task Priority Services:** Provides services to increase or decrease the current task priority level. This priority mechanism can be used to implement simple locks and to disable the timer interrupt for short periods of time. These services depend in the CPU Architectural Protocol.
- **Memory Services:** Provides services to allocate and free pages in 4 KB increments and allocate and free pool on byte boundaries. It also provides a service to retrieve a map of all the current physical memory usage in the platform.
- **Event and Timer Services:** Provides services to create events, signal events, check the status of events, wait for events, and close events. One class of events is timer events, which supports periodic timers with variable frequencies and one-shot timers with variable durations. These services depend on the CPU Architectural Protocol, Timer Architectural Protocol, Metronome Architectural Protocol, and Watchdog Timer Architectural Protocol.
- **Protocol Handler Services:** Provides services to add and remove handles from the handle database. It also provides services to add and remove protocols from the handles in the handle database. Additional services are available that allow any component to look up handles in the handle database and open and close protocols in the handle database.
- **Image Services:** Provides services to load, start, exit, and unload images using the PE/COFF image format. These services depend on the Security Architectural Protocol.
- **Driver Support Services:** Provides services to connect and disconnect drivers to devices in the platform. These services are used by the BDS phase to either connect all drivers to all devices, or to connect only the minimum number of drivers to devices required to establish the consoles and boot an OS. The minimal connect strategy is how a fast boot mechanism is provided.

4.2.5 EFI Runtime Services Table

The following is a brief summary of the services that are available through the EFI Runtime Services Table:

- **Variable Services:** Provides services to lookup, add, and remove environment variables from nonvolatile storage. These services depend on the Variable Architectural Protocol and the Variable Write Architectural Protocol.
- **Real Time Clock Services:** Provides services to get and set the current time and date. It also provides services to get and set the time and date of an optional wakeup timer. These services depend on the Real Time Clock Architectural Protocol.
- **Reset Services:** Provides services to shutdown or reset the platform. These services depend on the Reset Architectural Protocol.
- **Status Code Services:** Provides services to send status codes to a system log or a status code reporting device. These services depend on the Status Code Architectural Protocol.
- **Virtual Memory Services:** Provides services that allow the runtime DXE components to be converted from a physical memory map to a virtual memory map. These services can only be called once in physical mode. Once the physical to virtual conversion has been performed, these services cannot be called again. These services depend on the Runtime Architectural Protocol.

4.2.6 DXE Services Table

The following is a brief summary of the services that are available through the DXE Services Table:

- **Global Coherency Domain Services:** Provides services to manage I/O resources, memory-mapped I/O resources, and system memory resources in the platform. These services are used to dynamically add and remove these resources from the processor's Global Coherency Domain (GCD).
- **DXE Dispatcher Services:** Provides services to manage DXE drivers that are being dispatched by the DXE Dispatcher.

4.3 DXE Dispatcher

After the DXE Foundation is initialized, control is handed to the DXE Dispatcher. The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes. The DXE Dispatcher searches for drivers in the firmware volumes described by the HOB list. As execution continues, other firmware volumes might be located. When they are, the DXE Dispatcher searches them for drivers as well.

When a new firmware volume is discovered, a search is made for its *a priori* file. The *a priori* file has a fixed file name and contains the list of DXE drivers that should be loaded and executed first. There can be at most one *a priori* file per firmware volume, and it is legal to not have an *a priori* file at all. Once the DXE drivers from the *a priori* file have been loaded and executed, the dependency expressions of the remaining DXE drivers in the firmware volumes are evaluated to determine the order in which they will be loaded and executed. The *a priori* file provides a strongly ordered list of DXE drivers that are not required to use dependency expressions. The dependency expressions provide a weakly ordered execution of the remaining DXE drivers. Before each DXE driver is executed, it must be authenticated with the Security Architectural Protocol. This authentication prevents DXE drivers with unknown origins from being executed.

Control is transferred from the DXE Dispatcher to the BDS Architectural Protocol after the DXE drivers in the *a priori* file and all the DXE drivers whose dependency expressions evaluate to **TRUE** have been loaded and executed. The BDS Architectural Protocol is responsible for establishing the console devices and attempting the boot of OSs. As the console devices are established and access to boot devices is established, additional firmware volumes may be discovered. If the BDS Architectural Protocol is unable to start a console device or gain access to a boot device, it will reinvoke the DXE Dispatcher. This invocation will allow the DXE Dispatcher to load and execute DXE drivers from firmware volumes that have been discovered since the last time the DXE Dispatcher was invoked. Once the DXE Dispatcher has loaded and executed all the DXE drivers it can, control is once again returned to the BDS Architectural Protocol to continue the OS boot process.

4.3.1 The *a priori* File

The *a priori* file is a special file that may be present in a firmware volume. The rule is that there may be at most one *a priori* file per firmware volume present in a platform. The *a priori* file has a known GUID file name, so the DXE Dispatcher can always find the *a priori* file. Every time the DXE Dispatcher discovers a firmware volume, it first looks for the *a priori* file. The *a priori* file contains the list of DXE drivers that should be loaded and executed before any other DXE drivers are discovered. The DXE drivers listed in the *a priori* file are executed in the order that they appear. If any of those DXE drivers have an associated dependency expression, then those dependency expressions are ignored.

The purpose of the *a priori* file is to provide a deterministic execution order of DXE drivers. DXE drivers that are executed solely based on their dependency expression are weakly ordered, which means that the execution order is not completely deterministic between boots or between platforms. There are cases, however, that require a deterministic execution order. One example would be to list the DXE drivers that are required to debug the rest of the DXE phase in the *a priori* file. These DXE drivers that provide debug services might have been loaded much later if only their dependency expressions were considered. By loading them earlier, more of the DXE Foundation and DXE drivers can be debugged. Another example is to use the *a priori* file to eliminate the need

for dependency expressions. Some embedded platforms may require only a few DXE drivers with a highly deterministic execution order. The *a priori* file can provide this ordering, and none of the DXE drivers would require dependency expressions. The dependency expressions do have some amount of firmware device overhead, so this method might actually conserve firmware space. The main purpose of the *a priori* file is to provide a greater degree of flexibility in the firmware design of a platform.

4.3.2 Dependency Grammar

A DXE driver is stored in a firmware volume as a file with one or more sections. One of the sections must be a PE/COFF image. If a DXE driver has a dependency expression, then it is stored in a dependency section. A DXE driver may contain additional sections for compression and security wrappers. The DXE Dispatcher can identify the DXE drivers by their file type. In addition, the DXE Dispatcher can look up the dependency expression for a DXE driver by looking for a dependency section in a DXE driver file. The dependency section contains a section header followed by the actual dependency expression that is composed of a packed byte stream of opcodes and operands.

Dependency expressions stored in dependency sections are designed to be small to conserve space. In addition, they are designed to be simple and quick to evaluate to reduce execution overhead. These two goals are met by designing a small, stack-based instruction set to encode the dependency expressions. The DXE Dispatcher must implement an interpreter for this instruction set to evaluate dependency expressions. Table 4-1 contains a summary of the supported opcodes in the dependency expression instruction set.

Table 4-1. Dependency Expression Opcode Summary

Opcode	Description
0x00	BEFORE <File Name GUID>
0x01	AFTER <File Name GUID>
0x02	PUSH <Protocol GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END
0x09	SOR

Because multiple dependency expressions may evaluate to **TRUE** at the same time, the order in which the DXE drivers are loaded and executed may vary between boots and between platforms even though the contents of their firmware volumes are identical. This variation is why there is a weak ordering for the execution of DXE drivers in a platform when dependency expressions are used.

4.4 DXE Drivers

There are two subclasses of DXE drivers:

- DXE drivers that execute very early in the DXE phase
- DXE drivers that comply with the EFI 1.10 Driver Model

The execution order of the first subclass, the early DXE drivers, depends on the presence and contents of an *a priori* file and the evaluation of dependency expressions. These early DXE drivers will typically contain processor, chipset, and platform initialization code. They will also typically produce the DXE Architectural Protocols that are required for the DXE Foundation to produce its full complement of EFI Boot Services and EFI Runtime Services. To support the fastest possible boot time, as much initialization as possible should be deferred to the second subclass of DXE drivers, those that comply with the EFI 1.10 Driver Model.

The DXE drivers that comply with the EFI 1.10 Driver Model do not perform any hardware initialization when they are executed by the DXE Dispatcher. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the BDS phase to connect the drivers to the devices required to establish consoles and provide access to boot devices. The DXE Drivers that comply with the EFI 1.10 Driver Model ultimately provide software abstractions for console devices and boot devices but only when they are explicitly asked to do so.

All DXE drivers may consume the EFI Boot Services and EFI Runtime Services to perform their functions. However, the early DXE drivers need to be aware that not all of these services may be available when they execute because not all of the DXE Architectural Protocols might have been registered yet. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed.

The DXE drivers that comply with the EFI 1.10 Driver Model do not need to be concerned with this possibility. These drivers simply register the Driver Binding Protocol in the handle database when they are executed. This operation can be performed without the use of any DXE Architectural Protocols. The BDS phase will not be entered until all of the DXE Architectural Protocols are registered. If the DXE Dispatcher does not have any more DXE drivers to execute but not all of the DXE Architectural Protocols have been registered, then a fatal error has occurred and the system will be halted.

5
Boot Device Selection (BDS) Phase

5.1 Introduction

The Boot Device Selection (BDS) Architectural Protocol executes during the BDS phase. The BDS Architectural Protocol is discovered in the DXE phase, and it is executed when two conditions are met:

- **All of the DXE Architectural Protocols have been registered in the handle database.** This condition is required for the DXE Foundation to produce the full complement of EFI Boot Services and EFI Runtime Services.
- **The DXE Dispatcher does not have any more DXE drivers to load and execute.** This condition occurs only when all the *a priori* files from all the firmware volumes have been processed and all the DXE drivers whose dependency expression have evaluated to **TRUE** have been loaded and executed.

The BDS Architectural Protocol locates and loads various applications that execute in the preboot services environment. Such applications might represent a traditional OS boot loader or extended services that might run instead of or prior to loading the final OS. Such extended preboot services might include setup configuration, extended diagnostics, flash update support, OEM value-adds, or the OS boot code.

Vendors such as IBVs, OEMs, and ISVs may choose to use a reference implementation, develop their own implementation based on the reference, or develop an implementation from scratch.

The BDS phase performs a well-defined set of tasks. The user interface and user interaction that occurs on different boots and different platforms may vary, but the boot policy that the BDS phase follows is very rigid. This boot policy is required so OS installations will behave predictably from platform to platform. The tasks include the following:

- Initialize console devices based on the **ConIn**, **ConOut**, and **StdErr** environment variables.
- Attempt to load all drivers listed in the **Driver####** and **DriverOrder** environment variables.
- Attempt to boot from the boot selections listed in the **Boot####** and **BootOrder** environment variables.

If the BDS phase is unable to connect a console device, load a driver, or boot a boot selection, it is required to reinvoke the DXE Dispatcher. This invocation is required because additional firmware volumes may have been discovered while attempting to perform these operations. These additional firmware volumes may contain the DXE drivers required to manage the console devices or boot devices. Once all of the DXE drivers have been dispatched from any newly discovered firmware volumes, control is returned to the BDS phase. If the BDS phase is unable to make any additional forward progress in connecting the console device or the boot device, then the connection of that console device or boot selection fails. When a failure occurs, the BDS phase moves on to the next console device, driver load, or boot selection.

5.2 Console Devices

Console devices are abstracted through the Simple Text Output and Simple Input Protocols. Any device that produces one or both of these protocols may be used as a console device on a Framework-based platform. There are several types of devices that are capable of producing these protocols, including the following:

- **VGA Adapters:** These adapters can produce a text based display that is abstracted with the Simple Text Output Protocol.
- **Universal Graphics Adapters (UGA):** These adapters produce a graphical interface that supports Block Transfer (BLT) operations. A text-based display that produces the Simple Text Output Protocol can be simulated on top of a UGA display by BLTing Unicode glyphs into the frame buffer.
- **Serial Terminal:** A serial terminal device can produce both the Simple Input and Simple Text Output Protocols. Serial terminals are very flexible, and they can support a variety of wire protocols such as PC-ANSI, VT-100, VT-100+, and VTUTF8.
- **Telnet:** A telnet session can produce both the Simple Input and Simple Text Output Protocols. Like the serial terminal, a variety of wire protocols can be supported including PC-ANSI, VT-100, VT-100+, and VTUTF8.
- **Remote Graphical Displays (HTTP):** A remote graphical display can produce both the Simple Input and Simple Text Output Protocols. One possible implementation could use HTTP, so standard Internet browsers could be used to manage a Framework-based platform.

5.3 Boot Devices

There are several types of boot devices supported in the Framework:

- Devices that produce the Block I/O Protocol and are formatted with a FAT file system
- Devices that directly produce the File System Protocol
- Devices that directly produce the Load File Protocol

Disk devices will typically produce the Block I/O Protocol, and network devices will typically produce the Load File Protocol.

A Framework implementation may also choose to include legacy compatibility drivers. These drivers provide the services required to boot a traditional OS, and the BDS phase must also support booting a traditional OS.

5.4 Boot Services Terminate

The BDS phase is terminated when an OS loader is executed and an OS is successfully booted. There is a single service called **`ExitBootServices()`** that an OS loader or an OS kernel may call to terminate the BDS phase. Once this call is made, all of the boot service components are freed and their resources are available for use by the OS. When the call to **`ExitBootServices()`** returns, the Runtime (RT) phase has been entered.

6
Runtime (RT) Phase

6.1 Introduction

The Runtime (RT) phase is entered when the OS makes an EFI function call that signifies the platform firmware must relinquish control of the system hardware required by the OS for it to function. This relinquishing requires that a majority of the services started in DXE be terminated. The Runtime phase is defined by the termination of DXE services and not the availability of the EFI Runtime Services. The Runtime phase transitions to the Afterlife (AL) phase when the OS context is no longer relevant.

6.2 Overview

The Runtime phase is entered when an OS invokes the EFI Boot Service **`ExitBootServices()`**. After the **`ExitBootServices()`** call is invoked, the DXE Foundation and all other Boot Services code are terminated. As a result, only the EFI System Table and the EFI Runtime Services, which became available in the DXE phase, remain in the Runtime phase. Figure 6-1 below shows the Framework Runtime architecture.

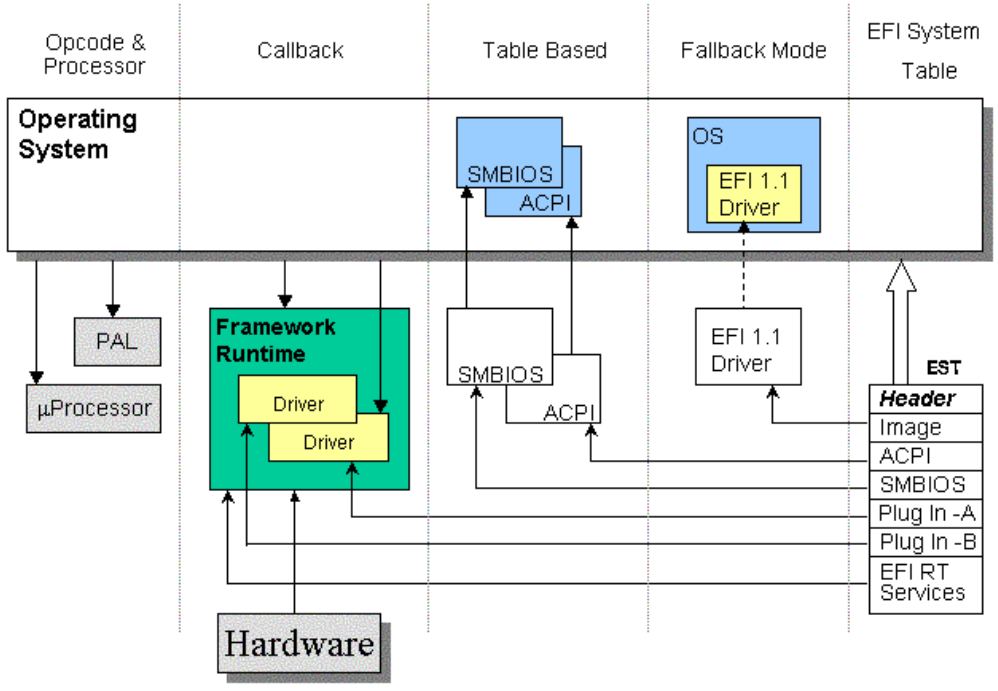

Figure 6-1. Framework Runtime (RT) Architecture

The OS is handed a pointer to the EFI System Table, and it uses this table to access platform firmware runtime services and information. As shown in Figure 6-1, the EFI System Table can point to EFI Runtime Services, C-callable runtime interfaces, interface tables, and safe-mode drivers. Table 6-1 lists examples of different kinds of Framework runtime interfaces; see the sections listed in the table for details.

Table 6-1. Framework Runtime Interfaces

Runtime Interface	Examples	For details, see section…
Table-based	ACPI, SMBIOS, SAL System Table	6.4
Callback	EFI Runtime Services, Itanium architecture SAL, Itanium architecture Floating Point Software Assist (FPSWA)	6.5
Opcode and processor	Itanium architecture PAL, Streaming SIMD Extensions (SSE) Instruction, I/O Instruction	6.6
Fallback mode	UGA, Universal Network Driver Interface (UNDI)	6.7

It is also possible for runtime code to be executed due to a hardware event; see section 6.5.2 for more information.

6.3 EFI System Table

The Runtime phase is entered when the OS or OS loader invokes the EFI Boot Service call **`ExitBootServices()`**. The call to **`ExitBootServices()`** changes ownership of processor and platform resources from the platform firmware to the OS. The first stage OS loader is an EFI application and thus will be passed a pointer to the EFI System Table on its invocation (see the *EFI 1.10 Specification* for more information).

The EFI System Table contains a pointer to the EFI Runtime Services and a pointer and number of entries in the EFI Configuration Table. The EFI Runtime Services represent the DXE intrinsic services that persist into runtime. The EFI Configuration Table contains GUID pointer pairs that can point to table-based interfaces or runtime services.

6.4 Table-Based Interfaces

Tables are passed to the OS using a GUID pointer pair in the EFI System Table. A DXE driver can add a table to the EFI System Table by calling the EFI Boot Service **`InstallConfigurationTable()`**.

Table-based interfaces consist of data that is handed directly to the OS. SMBIOS and ACPI are two examples of table-based interfaces that exist in current Intel architecture systems. The address in the EFI System Table points to the root of a tree of tables that can contain data and or an interpreted language. SMBIOS defines a table that contains only data, while ACPI defines tables that contain data and tables that contain an interpreted language, ACPI Machine Language (AML).

A table-based runtime interface must be constructed in the DXE phase. The interface must be valid and function in all subsequent phases. The BDS will signal an event to indicate that the platform is about to attempt to boot an OS. The “ready to boot” event allows multiple independent entities to construct a single table. The table is free to define its own unique requirements and to document them as required to support the table. The GUID in the EFI System Table defines the special sandbox rules for the Runtime phase that are unique to this table.

Because the tables will be used by the OS in the Runtime phase, it is very desirable to make the information in the table self-relative. Current interfaces such as SMBIOS and ACPI follow these rules (see the ACPI 2.0 and SMBIOS 2.3 specifications for more information).

6.5 Callback

The callback mode of runtime services is the class of services that are equivalent to the mechanism provided by the PC-AT BIOS runtime interfaces on an original Intel architecture personal computer. Callback implies that the OS is physically transferring control to the platform by calling into platform firmware.

The problem with callback interfaces is that the platform firmware is not coded to the driver model of the OS, which forces the OS to mask all interrupts when calling the platform firmware. Because the driver models for any given OS change over time and because every OS has a different driver model, it is difficult to define in which mode the platform firmware runtime code should be.

The other problem with runtime code is the ownership of platform resources. In MP systems, the runtime code cannot arbitrarily touch platform resources because another processor might be accessing these resources concurrently with the runtime code.

6.5.1 EFI Runtime Services

The EFI Runtime Services are constructed during the DXE phase. The constructors for the Runtime Services may access any resource or service that is available in DXE. Outside their constructors, however, the Runtime Services cannot depend on any DXE services or interfaces. In general, it should be assumed that an arbitrary protocol is valid only during the DXE phase, because any protocol that persists into the Runtime phase would have to explicitly state this capability in its specification.

The DXE Foundation contains no runtime code, and all Runtime Services are provided directly through the Runtime Architectural Protocol. The DXE code constructs the EFI Runtime Services Table out of runtime data, but the Runtime Services entry points are filled in directly by the Runtime Architectural Protocols. This design allows the DXE Foundation to be implemented with no runtime code or services.

6.5.1.1 Transition to Virtual Mode

EFI runtime services can be called only in physical mode at boot service time, before the OS loader calls **`ExitBootServices()`** and transitions to runtime. The OS may call the runtime services in virtual mode, but before doing so, it must call the **`SetVirtualAddressMap()`** runtime function to register the virtual addresses that are required by the EFI firmware in the system memory map. The call to **`SetVirtualAddressMap()`** then signals an EFI driver to convert to virtual mode, and the EFI implementation and runtime EFI drivers will use the **`ConvertPointer()`** runtime function to pass in physical addresses and get the virtual address returned.

The OS can call **`SetVirtualAddressMap()`** only one single time and only in physical mode. After calling **`SetVirtualAddressMap()`**, it may call EFI runtime functions only in virtual mode. It is a one-way gate to virtual mode. EFI addresses this issue by leveraging the infrastructure that exists in the compiler, linker, and image loader. The compiler and linker generate an image that can run at specific link addresses, but the image also contains a fix-up table that includes all the information that is needed to relocate the image to run at any address. Thus, the one-way gate to switch to virtual mode runtime calls exists to allow the code to transition all of its internal pointers by calling **`ConvertPointer()`** in physical mode.

The transition to virtual mode is very complex and can be safely done only one time. A runtime driver is responsible only for fixing up any pointer that the code initialized. The transition to virtual mode is so complex because some data structures will contain virtual addresses and others will still contain physical addresses while the code is being transitioned. The complexity of the transition can vary considerably depending on the data structures that must be converted. Consider, for example, a linked list. If you converted the next pointer in the list to a virtual address and then tried to traverse to the target of the next pointer, it would fail because it is a virtual mode pointer and your code is still executing in physical mode. The Runtime Architectural Protocol will be responsible for reapplying PE32 fix-ups to all runtime images using the new virtual mappings.

6.5.1.2 Virtual Mode Calling

This section highlights the issues with calling firmware in virtual mode. There are two basic areas of concern:

- Code generated by a high-level language compiler
- Code generated by the programmer

In this context, the term *virtual mode* is the same as commonly used in the OS. When we talk about *physical mode* and *virtual mode*, we are really talking about what address space a program is using for its code and data. Physical mode is what goes out to the hardware, and virtual mode is what each process in the OS uses. Each process in an OS could share the same virtual address space. The OS kernel also runs in the virtual space that is used by firmware runtime calls, which is generally fixed. Thus, when firmware is called in virtual mode, the address of the firmware code and any data it is using is different than if it was called in physical mode.

All references to image formats in this document are relative to PE32. PE32 is a relocatable image format derived from Common Object File Format (COFF). A relocatable image format implies that an image can be linked at a fixed address, but it contains the information required to load it at another address. It is the responsibility of the "loader of the images" to "fix up" the image to match the loaded address. The image load does this by processing the fix-up (also known as relocation) table in the PE32 image.

Fix-ups pose a very large challenge in code that can be called in physical or virtual mode. Fix-ups are designed to be run only once, at the time a program is loaded into memory. EFI solves this problem by requiring a one-time transition to virtual mode, which allows a complex, controlled reapplication of the fix-up information. As a comparison, the legacy SAL runtime convention allows the mixing of virtual and physical mode calls. This SAL convention makes code that requires fix-ups illegal.

6.5.1.3 SAL Runtime Legacy

The SAL has runtime conventions that allow the OS to call in both virtual and physical mode. This convention is accomplished by two simple, but restrictive, coding rules:

1. Always write position-independent code (PIC).
2. Never use an absolute or constant address.

The next assumption is that the code is written to all be PIC. This assumption is straightforward if you are writing assembly code but much more complex if you are writing C code. It is quite difficult to write PIC in C because any reference to a linkage table implies addresses that will need to be fixed up. Any runtime architecture that supports code for Itanium architecture must solve the issues that are created because of the different calling conventions for SAL and EFI code.

6.5.2 Hardware

Hardware events can cause runtime platform firmware code to begin execution. This section discusses the following hardware events:

- System Management Mode (SMM)
- Machine Check Architecture (MCA)

6.5.2.1 System Management Mode (SMM)

System Management Mode (SMM) on IA-32 processors is a mode of operation distinct from the flat-model, protected-mode operation of the DXE and PEI phases. SMM is defined to be a real-mode environment with 32-bit data access and is activated in response to an interrupt type or using the System Management Interrupt (SMI) pin. The interesting point about SMM is that it is an OS-transparent mode of operation and is a distinct operational mode. It can coexist within an OS runtime—thus its inclusion in the Runtime phase chapter of this specification.

Having stated that clarification, however, the Framework SMM design provides a mechanism to load DXE runtime drivers into SMM. The SMM infrastructure code will be loaded by a Boot Service driver and then does the following:

- Prepares an execution environment that relocates itself to the appropriate SMRAM location.
- Trampolines into flat-model protected mode.
- Supports receiving image loading requests from Boot Service agents. The SMM infrastructure code also supports receiving messages from both Boot Service and Runtime agents.

The model for loading drivers into SMM is that the DXE SMM runtime driver will have a dependency expression that includes at least the **`EFI_SMM_BASE_PROTOCOL`**. This dependency is necessary because the DXE runtime driver that is intended for SMM will use the **`EFI_SMM_BASE_PROTOCOL`** to reload itself into SMM and rerun its entry point in SMM. In addition, other SMM-loaded protocols can be placed in the dependency expression of a given SMM DXE runtime driver. The logic of the DXE Dispatcher—namely, checking if the GUIDs for the protocols are present in the protocol database—can then be used to determine if the driver can be loaded.

Once loaded into SMM, the DXE SMM runtime driver can use a very limited set of services. The driver can use EFI Boot Services while in its constructor entry point that runs in boot service space and SMM. In this second entry point in SMM, the driver can do several things:

- Register an interface in the conventional protocol database to name the SMM-resident interfaces to future-loaded SMM drivers
- Register with the SMM infrastructure code for a callback in response to an SMI-pin activation or an SMI-based message from a boot-service or runtime agent (i.e., outside-of-SMM code).

After this “constructor” phase in SMM, however, the environmental constraints are the same as other runtime drivers. Specifically, the SMM driver should not rely upon any other boot services because the operational mode of execution can migrate away from these services (the **`ExitBootServices()`** call is asynchronous to invoking the SMM infrastructure code). Several EFI Runtime Services can have the bulk of their processing migrated into SMM, and the runtime-visible portion would simply be a proxy that uses the **`EFI_SMM_BASE_PROTOCOL`** to “thunk” or call back into SMM to implement the services. Having a proxy allows for a model of sharing error-handling code, such as flash access services, with runtime code, such as the EFI Runtime Services **`GetVariable()`** or **`SetVariable()`**. Figure 6-2 shows the Framework SMM architecture.

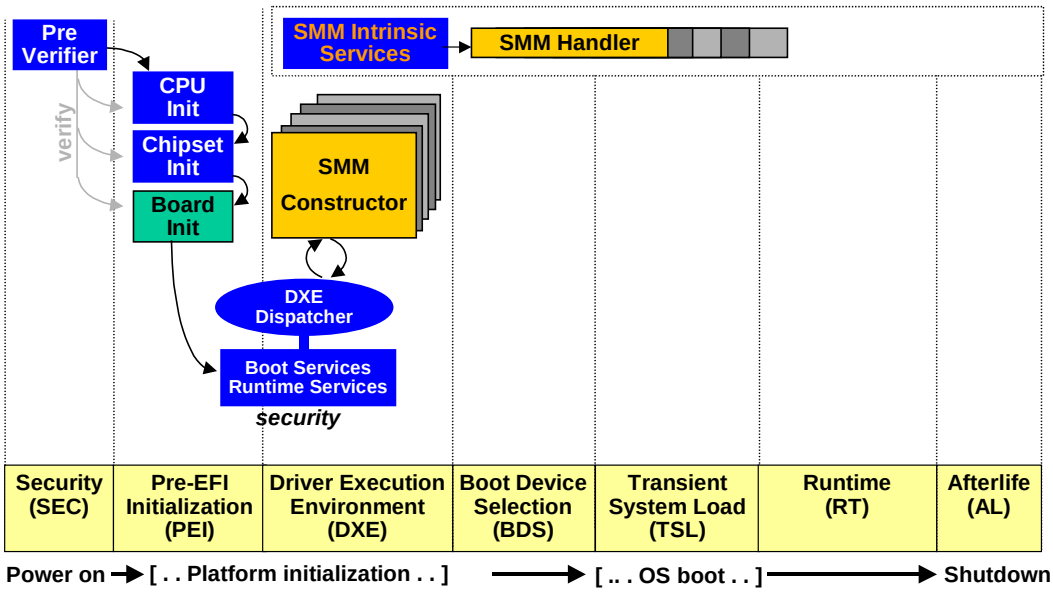

Figure 6-2. Framework SMM Architecture

6.5.2.2 Machine Check Architecture (MCA)

The Itanium processor family supports a Machine Check Architecture (MCA) that allows the platform and OS to interact to manage faults and errors on a system. The OS can register to be notified when a critical failure or event happens on a platform. A fault is detected by the PAL, and the PAL can automatically fix any processor faults. The PAL hands off control to the SAL to log information about the fault and to allow the SAL to try and use its unique system knowledge to fix the fault. The SAL then hands control back to the OS so it can log error information or try and fix a failure.

To the OS, MCA looks like a set of SAL APIs that can be called in either physical or virtual mode. The PAL always calls the SAL in physical mode.

6.6 Processor Opcode

While processor opcodes are not traditionally part of the system firmware, it is possible that system-firmware-like features could be implemented using processor opcodes in the future. A hypothetical example of this implementation could be adding a PCI configuration cycle instruction into the processor. This section defines how firmware-like features can be implemented using a processor opcode.

The processor offers the OS an architectural set of resources that is duplicated across all processor elements in a system. This architectural set differs from typical system resources that are globally accessible to all processors. These processor resources are well documented and their behavior is deterministic. The maximum latency and all side effects to the system are documented. The opcode will explicitly imply any access to a resource and an OS will be able to maintain coherency in the system.

Given that an opcode must explicitly declare all side effects, it cannot be used to abstract accesses to system resources. For example, if an opcode existed to do PCI configuration cycles, this opcode could use registers in the chipset. If the OS were also using the same chipset registers for other purposes, coherency problems would result. To emulate the opcode model, instructions must alter only visible processor resources or resources in the processor/system that are not visible to the OS.

The processor opcode model is not the same thing as the SMI mode on Intel architecture platforms. The SMI mode is typically implemented as a global system state. On MP systems, the SMI is wired to all the processors in the system. When an SMI happens, all processors must enter the SMI and check in before any access can be made to shared system resources. The latency of entering and exiting SMI mode is greatly increased by the synchronization of all the processors in the system. The side effect of the SMI mode is that all processors are taken away from their current context, which does not make it a processor opcode-like model.

The operation of the processor's opcodes is defined by the processor's architectural definition. The processor can implement opcodes as native instructions, microcode, or PAL. The PAL instructions are implemented as microcode that is stored external to the processor. This microcode has an architectural entry point that is external to the processor.

It should be noted that a key attribute of the Runtime phase is the modification of defined processor or system resources. Other resources may be altered or consumed by the opcode, but these resources must be private to the opcode. The mechanism to extend system features into opcodes is to allow private access to system resources by the processor. The processor must be capable of maintaining system coherency without the intervention of the OS on its private resources (see the *Intel® Itanium® Architecture Software Developer's Manual*).

6.7 Fallback Mode

A fallback mode driver is defined as an EFI driver that can be used by the OS. The OS must produce an environment, or *virtual machine*, that allows a typical preboot EFI driver to function in the OS. The OS can use the fallback mode driver in a case where the OS performance driver is not present. There are also situations in an OS when the performance driver cannot be used and a fallback mode driver can be used instead. The following are examples of OS situations where a fallback mode driver would be required:

- Displaying an early graphics screen in the OS before the OS loads its performance driver
- Printing crash information after the OS has crashed

It is possible that a fallback mode EFI driver could be used in special situations, even on a normal OS boot.

The fallback mode driver interface involves an EFI System Table pointing to an EFI driver image. This driver is a standard EFI driver and is coded to run in the EFI Boot Services environment. The EFI driver handed to the OS is treated as data and is not loaded, relocated, or initialized by the Framework code. The OS must produce a virtual machine that produces an EFI Boot Services environment. The methodology that an OS uses to produce an EFI virtual machine is OS dependent and not defined by this specification.

The OS can discover the EFI drivers using several mechanisms. The OS can use its own bus drivers to extract EFI drivers from a device. An example of this would be the OS PCI bus driver extracting the EFI driver from a PCI device OpROM. A GUID pointer pair can also be used in the EFI System Table to pass up EFI drivers to the OS.

This specification defines only the mechanism by which the OS discovers EFI drivers. It does not define the EFI drivers that must exist in the OS or in the platform firmware. Various tables and rules could be defined for different OS virtual machines. While the Framework does not define how the OS EFI virtual machine should function, certain basic guidelines will be followed. Because the purpose of the OS virtual machine is to ensure that the EFI driver follows the OS device driver model, the virtual machine must produce the EFI System Table, including the EFI Boot and Runtime Services. The OS virtual machine will also likely produce low-level I/O protocols.

Figure 6-3 shows an OS EFI virtual machine that is used to support a UGA (replacement for VGA) EFI 1.10 driver. The OS virtual machine produces the EFI services and a programmatic abstraction for PCI. The OS exports an interface from the UGA driver to the OS device driver stack.

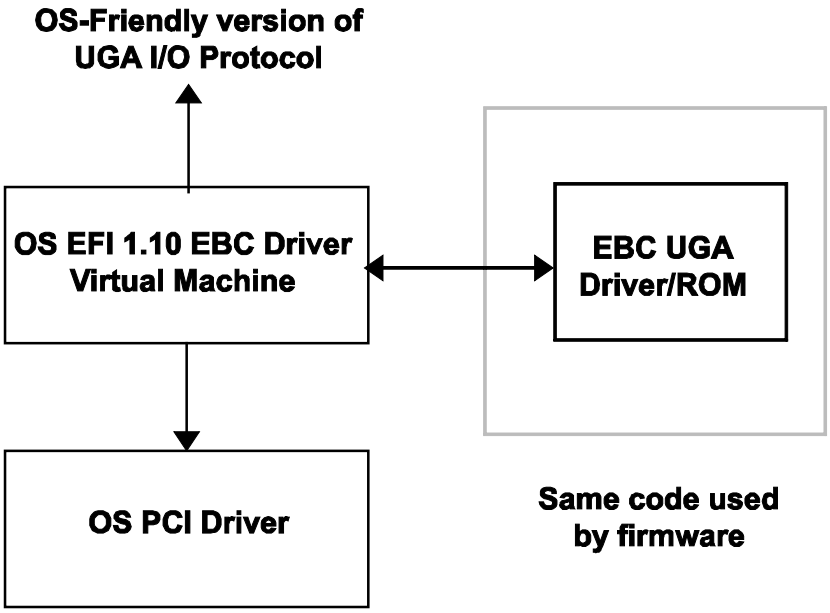

Figure 6-3. Fallback Mode Driver

It is most likely that various specifications will be used to specify the OS support for various types of OS-present EFI virtual machines. The contents of these specifications are outside the scope of this document. A fallback mode EFI driver can be used in lieu of an OS performance driver or in conjunction with the performance driver. An example of this implementation is UGA. The UGA EFI driver can be used in cases where an OS performance driver is not available or needed. The UGA EFI driver can also be called from the OS in an emergency situation such as an OS crash.

7
Afterlife (AL) Phase

7.1 Introduction

The Afterlife (AL) phase represents a transition of platform control from the OS back to the Framework firmware. The Afterlife phase is a continuation of the Runtime phase in which the Framework firmware regains control, for example either by the OS calling the EFI Runtime Service **`ResetSystem()`** or by invoking an ACPI sleep state. It is also possible to enter the Afterlife phase through an asynchronous event. The Afterlife phase can terminate with the system being reset or with the OS waking back up from a sleep state.

The Afterlife phase may be invoked in cooperation with the OS. Some firmware security schemes require that blocks in the flash devices be locked to prevent software write-access and unlocked only with a processor reset. The possibility for a firmware update to require a processor reset requires the OS to hand a firmware capsule to the Framework firmware and invoke the Afterlife phase. It may be possible to suspend the OS using an ACPI sleep state and to resume back to the OS context upon completion of an Afterlife phase task.

The OS can also theoretically be interrupted through some platform-specific hardware mechanism to force a transition to the Afterlife phase. A forced transition to the Afterlife phase could potentially be used to recover from OS or hardware faults in a system.

7.2 Overview

The Afterlife phase can be invoked by calling the EFI Runtime Service **`ResetSystem()`**, by entering an ACPI sleep state, or through some form of asynchronous entry.

7.2.1 Reset

The OS can call the EFI Runtime Service **`ResetSystem()`** to enter the Afterlife phase. The OS can pass information back to the Framework firmware by passing an optional data buffer, *`ResetData`*, to the **`ResetSystem()`** call. The *`ResetData`* parameter can represent OS panic information about why the OS needed to terminate. It can also, for example, represent a Framework capsule that will be used to update a Framework firmware volume.

7.2.2 ACPI Sleep States

The ACPI specification defines global system sleep states. The S3 global sleep state implies that the system processors and system cache context are lost and must be stored by the OS. In the S3 state, the chipset context is also lost and must be restored by the Framework firmware before the OS can wake from the S3 state.

The S4 sleep state implies that all system context is saved by the OS and that the platform is powered off. The S5 sleep state is a soft off and requires a complete system reset to wake the system up. The main difference between an S4 and S5 state is that the OS loader will restore the system context in S4 and the system will boot normally in S5.

It would be possible to use ACPI sleep states to transfer control from the OS to the Afterlife phase to perform a Framework function such as a capsule update. After the capsule has been updated, the OS context could be recovered.

7.2.3 Asynchronous Entry

It is possible to enter the Afterlife phase through an asynchronous event such as an SMI or nonmaskable interrupt (NMI). The most likely reason to enter the Afterlife phase would be to take control of the system if the OS should crash. This feature is common today on some classes of server and workstation-like systems.

Although not defined today, a future industry standard specification could define asynchronous entry points to the Afterlife phase that cooperate with the OS.

7.3 Capsule Update

A *capsule update* is a mechanism in which a preformed update image can be loaded into the system memory at OS runtime and then executed following a processor INIT or warm reset. The form of the reset is similar to an ACPI S3 transition, in that the processor begins execution at the reset vector but the contents in memory remain intact. The capsule is then extracted/reconstructed from system memory and executed. Because the reset unlocks the firmware hub, writing the flash is now possible.

7.4 Operating System Cooperation

Most of the potential value of the Afterlife phase would be derived from cooperation between the Framework firmware and the OS. The exact mechanism used to cooperate between the OS and the Framework firmware has not been defined yet, because it requires working closely with OS vendors.

If the Afterlife phase is required to update a firmware volume, it may be possible to do so without shutting down the OS. It may be possible to place the OS into an ACPI sleep state, transition to the Afterlife phase, and then restore the OS context.

In the future, the Afterlife phase might also be used to produce a “hang free” PC. The OS can cooperate with the Framework firmware to let the platform take control after an OS crash for diagnostic or debugging purposes, which might allow the platform to try to recover automatically from an OS crash. It could also be possible to use an asynchronous Afterlife entry to recover from a hung system. The asynchronous entry to the Afterlife phase can be initiated though platform-specific means that may vary by system vendor.

While the Afterlife phase holds a lot of promise to implement features typically found only on vertically integrated platforms, it should be noted that it will require a significant, coordinated, horizontal effort in the computer industry to implement advanced features in the Afterlife phase.

8
Firmware Store

8.1 Overview

Unlike a traditional legacy BIOS, which is generally built monolithically and contains few independent components, Framework-based firmware is highly modular and consists of many small, independently linked components. Each of these components may place additional requirements on the image format for data organization, authentication, compression, and so on. A new approach to firmware storage is needed to ensure the following:

- Firmware devices are used efficiently.
- The storage strategy is flexible and allows various components to be found and retrieved without *a priori* knowledge of exactly where to find them or the methods required to retrieve them

This chapter provides a high-level description of the architectural mechanisms that the Framework provides to enable these capabilities. Subordinate specifications will describe these features in detail. The remainder of this chapter is broken into the following subtopics:

- **Firmware Volumes:** Defines firmware volumes and the protocols that are used to access firmware volumes
- **Image Format:** Defines the binary encoding of files that are accessed through the Firmware Volume Protocol.
- **Firmware File System:** Defines the binary storage format for block-based devices (i.e., flash) that is required for PEI. It is also typically used for DXE but is not strictly required.
- **Update:** Covers a variety of topics related to updating firmware, including authentication, flash hardening, fault tolerance, platform-specific policy, and crisis recovery

8.2 Firmware Volumes

A firmware device is a persistent physical repository containing firmware code and/or data. There can be a single firmware device or multiple firmware devices on a platform. To use firmware devices more efficiently, BIOS developers may divide a single firmware device into multiple *logical* firmware devices or aggregate multiple firmware devices into a single logical firmware device. It is also possible to layer a logical firmware device on a block device. The Framework defines a logical firmware device as a *firmware volume.*

Because a firmware volume that is built on top of a physical firmware device can be viewed as a block device, the Framework defines the Firmware Volume Block Protocol to provide block-level access to a firmware volume. On the other hand, in many cases a firmware volume is used to store firmware files, where a file is an arbitrary collection of data that might be a driver, data file, PEIM, or virtually any other collection of data. The Firmware Volume Protocol is used to provide file-level access to a firmware volume. The following subsections discuss the Firmware Volume Protocol and Firmware Volume Block Protocol.

8.2.1 Firmware Volume Protocol

The Firmware Volume Protocol contains a minimal API set that enables normal read/write operations on files and the ability to locate files within the firmware volume. Additionally, there are APIs to manage attributes of the firmware volume hardware. The attributes APIs allow implementation of a wide variety of read/write protection and firmware volume hardening strategies as required to protect system firmware from inadvertent or malicious corruption.

Generally speaking, the actual storage media that is abstracted by the Firmware Volume Protocol is not architectural and can vary greatly in implementation. There is a great deal of flexibility in how the capabilities of the underlying firmware volume hardware can be represented and manipulated using the Firmware Volume Protocol APIs. Whenever possible, code should not make assumptions about how or where data is actually stored and should exclusively use the firmware volume interface. Using this interface enables storage hardware to be implemented to solve various platform requirements and to evolve over time as new technologies become available.

8.2.2 Firmware Volume Block Protocol

Firmware Volume Block Protocol provides block-level access to a firmware volume. There are some circumstances when more definition of the media and format is required because a software abstraction of the firmware volume is not possible. For example, during the PEI phase, code is generally expected to be stored in some sort of flash ROM device and is “execute in place.” The PEI Foundation and PEIMs that comprise the PEI phase are stored as files within a firmware volume that is manifested in a flash ROM. Because the code is “execute in place” and the PEI Foundation must be able to parse a firmware volume to locate PEIMs for execution, more constraints are put on the firmware volume storage and format. To that end, the Framework also specifies a binary format for the firmware volume header and a file system (known as the Firmware File System, or FFS) for use in block devices, such as flash.

Using an analogy from the disk drive world, the firmware volume header is like a disk partition record, and the FFS is like the format of a partition, i.e. FAT32.

8.3 Firmware File System

The Firmware File System (FFS) is a flat file system because there is no provision for a directory hierarchy. Files are stored one after another so the end of one file also marks the beginning of the next. Each file consists of a FFS file header, followed by an arbitrary amount of data, which comprises the contents of the file. The FFS file header contains all meta-data about the file. This meta-data includes things like the file name (a GUID), the file type (file types are enumerated as part of the EFI image format definition), size, and other information.

Again making the analogy to a disk drive, a disk file system—FAT32 for example—stores the location of a file on a disk as well as its size, name, and other attributes. The actual contents of the file are not part of this meta-data. Assume an application file named FOO.EXE. When the file FOO.EXE is written to a FAT32 partition, how the file is organized on the disk itself is defined by the FAT32 format. The fact that it is an executable file is captured in the file name extension. If the same file were stored in a FAT16 partition, the file would be organized differently on the disk itself, yet executing the program would yield exactly the same result. Thus, the contents of the file are separate from the file's meta-data. In this manner, the FFS is responsible for defining the binary format of the file itself and managing the meta-data.

FFS files are stored consecutively at the bottom of a firmware volume, with free space extending down from the top of the firmware volume. Each file is named with a GUID in the FFS header and there may never be two valid files with the same file name GUID in the same firmware volume.

Both IA-32 and Itanium architectures fix addresses near 4 GB, which are used in boot-strapping the processor from reset. Therefore, it must be possible to locate data at a specific address near 4 GB to enable boot-strapping the processor. Placing data at this address is done by justifying a file to the top of a firmware volume. FFS does this justification by defining a specific file name GUID that has a special meaning. A file with this GUID is known as a *Volume Top File* and has the required property of being justified against the end of the firmware volume. In this way, the code and data that are required at processor reset can still reside within a firmware volume that is formatted with FFS.

In the DXE space, a driver that manages a FFS firmware volume and produces the Firmware Volume Protocol is known as the *FFS driver*. The FFS driver is responsible for maintaining all state information in the firmware volume device. Depending on the feature set for the particular firmware volume, this information may include advanced features such as fault tolerance, RAID, special alignment requirements, and so on.

To reiterate, the Firmware Volume Protocol interface provides for interoperability, and the FFS defines one possible binary storage format that may underlie an instance of the Firmware Volume Protocol.

8.4 Framework Image Format

The Framework further defines the format of file contents to do the following:

- Simplify searching through a firmware volume.
- Provide for advanced features like compression and authentication.
- Enable aggregating separate but related data into a single file.

The following sections describe the Framework image format in more detail.

8.4.1 File Type

Returning to the FOO.EXE example above, the file system used to store FOO.EXE has no bearing on the actual contents of FOO.EXE. The FAT32 or FAT16 driver abstracts the contents of FOO.EXE from how it is physically stored. Similarly, the Firmware Volume Protocol interface returns the contents (or partial contents as the case may be) of a file without exposing the underlying file system format to the caller.

The format of the contents of FOO.EXE is implied by the ".EXE" in the file name. Depending on the operating environment, this extension typically indicates that the contents of FOO.EXE are a PE/COFF image and follow the PE/COFF image format. Similarly, the Framework image format defines the contents of a file returned by the Firmware Volume Interface.

The Framework image format defines an enumeration of file types. For example, the type **`EFI_FV_FILETYPE_DRIVER`** indicates that the file is a DXE driver and is interesting to the DXE Dispatcher. In the same way, files with the type **`EFI_FV_FILETYPE_PEIM`** are interesting to the PEI Dispatcher. In a FFS firmware volume, the file type is captured in a field of the FFS file header. At this time there are ten defined file types.

8.4.2 File Sections

In addition to file type, the Framework image format provides for flexible and varied internal organization of the file's contents. The basis for this internal organization is known as *file sections*.

A file section can be thought of as a "mini-file." It has a very small header that declares a type and size of the section followed by an arbitrary amount of data that comprises the contents of the section. There are two broad categories of sections:

- Encapsulation sections
- Leaf sections

Each file section type falls into one of these two categories.

While there is no provision for a directory hierarchy in FFS, the organization of individual file contents can be very hierarchical. Making a similar analogy between a FAT32 file system and the section organization of a Framework file's contents, encapsulation sections are similar to directories, and leaf sections are similar to files.

There are two types of encapsulation sections:

- **A general purpose encapsulation section with a GUID in its header.** Any form of encapsulation can be created simply be generating a new GUID and defining the associated rules and format.
- **An encapsulation section dedicated to compression.**

A leaf section is always a data entity and terminates any hierarchy that is formed by encapsulation sections. There are many types of leaf sections, the most common of which is **`EFI_SECTION_PE32`**, which is a complete PE/COFF image. There are several other leaf section types to handle various types of images—for example, **`EFI_SECTION_PEI_DEPEX`** and **`EFI_SECTION_DXE_DEPEX`** are dependency expressions for a PEIM or DXE driver, respectively.

8.5 Update

Firmware updates can come in many various forms from simple nonvolatile variable storage to a fault-tolerant update of everything that comprises the system firmware. The capabilities present in any given platform are highly dependent on the class of the platform and the features and platform update policy that are implemented. The Framework architecture has several features that make it well suited to fit a wide variety of update capabilities within its framework. The following are the three most prominent features:

- FFS
- Fault Tolerant Write Protocol
- A boot path known as *capsule update*

The FFS is designed with robust updates in mind. A catastrophic system failure (i.e., a power failure) during a FFS write is recoverable because the written file is tracked using single-bit writes to the FSS file header. The next time that the FFS volume is mounted, the failed write is detected and the file system can be brought back to a stable state. It is also possible to do a lock-step atomic update of multiple files with the FFS.

Consider the case of a FFS file system where deleted files are sprinkled through the firmware volume image. It is necessary to recover the space consumed by the deleted files and return that space to the firmware volume. In essence, this action is “defragmenting” the firmware volume to “reclaim” space consumed by previously deleted files. A problem occurs when one block contains both good data and deleted data. Erasing the entire block puts the good data at risk of loss due to a catastrophic system failure. The solution to this problem is the Fault Tolerant Write Protocol. It is an abstraction used to manage a block of private storage that can be used to continue a supplied storage area. If a catastrophic system failure occurs during a reclaim operation, the Fault Tolerant Write Protocol provides the necessary hooks to continue the operation after the system has been rebooted.

Another aspect of firmware updates revolves around platform security and hardening the firmware storage against accidental or malicious corruption. By definition, accomplishing this hardening requires support from hardware. While other firmware storage hardening solutions could certainly be implemented, the model for the Framework is to use a “block-locking” feature that is supported by various flash technologies. This block-locking feature enables specific blocks to be secured with a one-way switch. Writes to nonvolatile storage would then be disabled until the next reset occurs.

A capsule update is a mechanism in which a preformed update can be loaded into the system memory at OS runtime and then executed following a processor INIT or warm reset. The form of the reset is similar to an ACPI S3 transition, in that the processor begins execution at the reset vector but the contents in memory remain intact. The capsule is then extracted/reconstructed from system memory and executed. Because the reset unlocked the “block-locking” feature, writing to the nonvolatile storage block is now possible.

Lastly, if the firmware has become corrupted and is unbootable, a “crisis recovery” is required. It is like a capsule update, except that instead of recognizing an update capsule in memory following a soft reset, a crisis update firmware volume is loaded from an I/O device (i.e., CD-ROM). The drivers dispatched from this “recovery” firmware volume have full access to the normal firmware store and can take appropriate recovery actions.

9
User Interface

9.1 Introduction

9.1.1 Continuing Evolution

The three basic devices that are required for a boot are similar to the three device types that are required for basic OS application functionality, as follows:

- **Boot path:** The path of the logical device from which the OS is to be read to initiate the OS boot-strap process. The analogy in application functionality is the current directory.
- **Standard input device:** The logical device from which instructions and data can be read.
- **Standard output device:** The logical device where results can be reported.

Two of the three required devices have much to do with the actual human sitting in front of the system or attached remotely.

In the early history of computers, standard input typically took the form of a card reader while standard output took the form of a line printer. Later, standard input took the form of a keyboard and standard output the form of a CRT, still with the same 80 columns as cards. Graphics-based systems followed, with the advent of higher resolution monitors and pointing devices such as mice.

The history of user-interface standards parallels this history of hardware. Initially, each system or vendor provided its own interfaces for how to represent input and output characters and for I/O control. Over time these interfaces evolved into standards such as ASCII, EBCDIC, and VT100. With the advent of the PC, different standards appeared, such as extended ASCII and VGA. As PCs became more ubiquitous, these standards proved less useful. Newer standards such as Unicode and those for language and country designators indicate a more global effort.

User interfaces themselves also evolved along similar paths. With cards, columnar alignment was important. As CRTs proliferated, character-based menuing systems became popular. As mice and graphics support became available, windowing systems came to predominate. The look and feel of these interfaces continued to evolve, with the Internet becoming a driving force in continuing efforts to make increasingly complex systems easier to use and accessible by more people from more cultures.

9.1.2 Goals

There is no way for the architecture to force good user interfaces because there is only a vague definition of what a good interface is. The architecture cannot know a particular application's requirements, and it is possible for a determined developer to circumvent the interfaces. The architecture is left to assist the developer in successfully providing designs with the following properties:

- **Positive end-user experience:**
  The interfaces supported should lend themselves to creating user interfaces that target the type of user that will be using the system. What is easy to use to an occasional user of setup may be irritating and tedious for an IT manager who deals with the same interfaces many times each day.
- **Proper levels of abstraction:**
  The vicissitudes of the device interfaces should be hidden from applications so that the application developer can develop rich applications that can be retargeted as required by the environment in which the application finds itself. In general, applications should be able to easily be written so that they can be run directly on the system in which they reside while the user I/O is routed remotely (for example, through wiring concentrators serially or over LANs).
- **Small size:** The devices in which firmware is stored are likely to have significant cost per bit for the foreseeable future. As such, reducing size is a recurring requirement. Areas such as string and graphical image storage can consume large amounts of space.
- **Consistent:** The user interfaces that are developed should have similar basic features so as to reduce the learning curve and frustration levels of users.

As with most sets of goals, at times the implications align. For example, a consistent interface, if shared, provides for a smaller overall firmware device footprint. There are other times, such as implementing ease-of-use by providing familiar (large) graphical interfaces, that two goals may conflict.

9.1.3 Hierarchy

The support provided by the architecture forms a rough hierarchy. The following provide a basic environment for the drivers making up the hierarchy:

- Data types and structures such as Unicode with defined limitations
- Architecturally specified representations for fonts, strings, and graphical images

At the base of the hierarchy is the support for devices that sit on different buses (such as USB keyboards and mice, PS/2 keyboards and mice, serial ports, and PCI and AGP video cards but also hierarchies on top of LANs providing Telnet support, for example). Drivers can then be provided that map the device/bus interface requirements into a series of device abstractions for the basic classes of user-interface devices—keyboards, mice, consoles, and video. These device class abstractions are defined so that applications are shielded from the specifics each of the possible devices within a class.

The interfaces then provide support for multiple languages to do the following:

- Add to the default set of font characters (glyphs).
- Manipulate tokenized strings so that the application is less aware of the language it is displaying.
- Install keyboard abstractions so that a country/language keyboard layout can be comprehended for general string input.

Menuing is enabled using the definition of a forms language that is loosely based on popular Internet forms languages, including HTML and XML, but is modified to suit the preboot environment. The traditional firmware Setup program then becomes a browser for this sort of form. Setup also provides interfaces so it can be used by other applications and drivers. These interfaces are rich enough to provide dynamic forms manipulation so that they can assist in providing more traditional windowing features such as necessarily dynamic file open menus. The output of the forms can then be stored in variables or routed to drivers for further processing.

9.1.4 Design Considerations

The author of an OS-present application must consider many factors when determining a user interface. A considerable body of information has accumulated on the proper ways to create these user interfaces.

Authors of a pre-OS application must consider a number of factors that are different from those of their OS-present counterparts. The most telling difference is that the pre-OS application likely will not be used as frequently as the OS-present application. The goal of the type of user interface that is typically presented by the pre-OS is to be "walk up and use." In other words, the user should require little or no coaching to navigate around the interface and provide input—think ATM machine rather than spreadsheet when thinking of the pre-OS application.

A "walk up and use" interface is characterized by several features that the user of a typical OS-present application (for example, a word processor or paint program) would find irritating, including the following:

- Input is limited.
- Input choices are always described.
- Help is provided without request.

While the architectural support for user interfaces is focused on "walk up and use," it is possible to write more complex applications that require more learning but are easier to use for frequent users.

9.1.5 Terminology

The following terms apply to the descriptions of the user interface presented in this section. See the "Glossary" section on page 107 for complete definitions for all terms in this specification.

VT-100	A terminal and serial protocol.
VT-UTF8	A serial protocol definition that is mostly compatible with VT-100.
Unicode	A standard defining an association between numeric values and characters from the majority of the world's currently used languages.
ISO 3166	An association between a country or region and a two- or three-character ASCII string.

ISO 639-2	An association between a language or dialect and a three-character ASCII string.
internationalization	Concepts by which an interface is made useful to users speaking different languages and from various cultures by adapting the interfaces to the user. "STOP" in English would be "ALTO" in Spanish and "СТОП" in Russian. Alphabetic characters on keyboards are local to the language and might be local to the country the keyboard is localized for. For example, a French keyboard in France is different from a French keyboard in Canada.
localization	Concepts by which an interface is made useful to users speaking different languages and from various cultures by adapting the interfaces to the user.
BLT	Block Transfer (pronounced "blit" as in "slit" or "flit").
glyph	The graphical representation of a single Unicode weight (class).
font	A translation between Unicode weights and glyphs.

9.2 Data Types and Structures

9.2.1 Unicode

Characters form a basic data type for user interfaces as well as for most programming languages. Different mappings from values to characters have been common in programming from the days of EBCDIC and ASCII. The mapping used in EFI and throughout the Framework is the 16-bit version of Unicode. Because of the way Unicode is defined, the 16-bit version provides for a representation of about 37,000 different standard characters.

The remaining 28,000 or so characters that are possible with 16-bit coding fall into several classes (weights). Some weights are user defined and intended for representing specialized scripts such as Mayan. Some of the weights are used as prefixes. The weight following the prefix encodes 64,000 more characters. The Framework architecture, along with much of the rest of the community (including XML), excludes these uses for weights.

The Framework architecture does support nonspacing weights. These weights are intended to be OR-ed in to preceding characters to create otherwise nonencoded combinations. This method is most commonly used in languages such as Vietnamese, where not all sets of accent marks have specific Unicode weights.

Unicode is extended slightly by allocating a few weights in the specialized area to indicate attributes that would otherwise require large applications to support, particularly a nonbreaking space character.

9.2.2 Fonts

Font glyphs in the architecture are represented in two different widths (8 and 16 pixels wide) and are 19 pixels high. The two widths are required to represent logographic characters (for example, Chinese and Japanese Kanji) while making good use of text space for alphabetic languages (English, Cyrillic, Arabic). The default font set is known as Latin 1 and covers the characters required for the common Latin-based European languages, as well as common symbols such as box drawing characters, circles, and triangles. Support is provided to augment the Latin-1 set with glyphs corresponding to other Unicode weights. To achieve the goal of consistency, it is important for all of the characters that are provided to be from the same font.

The actual representation also contains the Unicode weight and attributes. One of these attributes allows the associated drivers to determine if the character is nonspacing.

9.2.3 Keyboard Mappings

Attached keyboards generate scan codes, which are a mapping of the key position, not equivalent weight. The mapping from scan code to weight must be done internally. Remote text input is normally already translated into Unicode. For example, although the scan code for the character next to the tab key is the same for all keyboards, the corresponding weight the user expects may change if the user is in the United States (Q) or in France (A). The situation becomes even more complicated in ideographic languages (such as Kanji in Japan), where thousands of characters must be represented. Intelligent applications known as Input Method Editors (IMEs) watch the input stream of phonetic characters, suggesting corresponding ideographic characters when appropriate.

The mechanism provided here is to support a table of scan code–to–weight entries. The format is complicated by required support for modifier keys such as shift and control. The intent for this support is to provide support in alignment with the requirements of the preboot environment. For example, in Japanese only the Kana (phonetic) characters are supported. For the few cases where text is input (such as passwords and modem control strings), this compromise is sufficient.

9.2.4 Strings

Strings are represented as null-terminated ordered arrays of Unicode weights. Because of Unicode features such as nonspacing characters, there are several types of “length” for the same string.

Strings can be abstracted by preprocessing into string tokens, which are 16-bit values that provide the basis for localization. The same token may be applied to different language-specific data structures to provide localized strings.

9.2.5 Graphical Images

Graphical images are manipulated in the architecture in two basic formats:

- Raw bit map
- Sectioned Graphics Format

The lower-level format is a raw bit map: a rectangle of pixels, in which each pixel is represented by a 32-bit value encoding the red, green, and blue magnitude of the color value of the pixel.

The higher-level format is the Sectioned Graphics Format. This format provides a common compact mechanism for storing the image. The format is unique in that it allows multiple resolutions of the image to be stored. This allowance provides an ability to create a single file that defines an image that can be displayed on graphics devices of various resolutions and through interfaces that might further limit the display.

As the Sectioned Graphics Format name implies, the format consists of a series of sections. Each section can add more colors to an image or might add resolution (X, Y, and/or color depth) to the image that already is described. Each section consists of a palette of 24-bit color values followed by an array of pixels. The pixels can be 1, 4, or 8 bits wide depending on the number of palette entries. The default compression scheme for images is the same as for code and uses the existing compression wrapper section. Colors are built up in the image 255 colors at a time by using the last palette value plus one to be transparent.

Special versions are defined for legacy VGA compatibility with a color depth of 16 colors including two locations reserved for black and white (so that monochrome text may be displayed).

9.2.6 Forms

Forms are represented using a byte stream of defined operations known as Internal Forms Representation (IFR), which are operators followed by their associated operands. IFR operations are the equivalent of the tags found in form descriptions such as HTML. IFR operations provide a subset of the features of HTML tags, including titles, subtitles, radio buttons, and check boxes. IFR allows the developer to associate help text with the questions to facilitate the browser's display of context-sensitive help.

IFR does not support scripting languages but it does support operations to control the visibility of other operations and support the validation of results. IFR represents strings using string tokens. IFR is designed to be easily translated into the XML version of HTML. (XML supports Unicode whereas HTML uses ASCII.)

IFR defines output using a triple of an ID, a size, and a value per question. A mapping from these attributes to the standard HTML name=value[&name=value]* format is defined. This text-based mapping forms the basis of forms results processing. A higher-level language representation for IFR is defined and is known as Visual Forms Representation (VFR).

9.2.7 Human Interface Packs

Collections of strings, fonts, keyboard tables, or forms are known as *packages*. Packages can then be grouped together to form *packs*. The font and keyboard parts are treated as generic (not associated with any particular driver or application), whereas the strings and forms are grouped and may be referred to using a handle.

9.2.8 Configuration

Support for the user interface is configured through the use of nonvolatile environment variables that contain the currently selected language, country or region code, and paths to standard input and output devices.

9.3 Console Support

9.3.1 Text I/O

Basic CRT-terminal-like text support is provided through the Simple Input Protocol and Simple Text Output Protocol.

The minimal Simple Input Protocol supports only cursor control and editing functions. These functions are the basic standard functions that are required for internationalized user interfaces and are in common locations on keyboard devices. More advanced functionality requires keyboard mappings as described in section 9.2.3. Devices translate their scan codes or encoding schemes into this subset of Unicode. The Simple Input Protocol provides services to reset the input devices, to read a key stroke, and to wait for the next key stroke.

The Simple Text Output Protocol provides basic services to use a text output device. The device can be configured and that configuration checked. Unicode strings can be displayed and the color of the output text can be controlled using the 16 standard colors (bright and dark versions of black, white, cyan, magenta, yellow, red, green, and blue). The protocol also provides a mechanism to test a string for the availability of fonts to display it. The required minimum support is Latin-1 but glyphs may be added using other protocols to extend support.

9.3.2 Graphical I/O

The Universal Graphics Adapter (UGA) Protocol defines a set of interfaces that abstract video output devices. UGA also defines the minimum attributes required to claim support:

- 800 pixels wide
- 600 pixels high
- 32 bits of color at 60 Hz

Other resolutions may optionally be supported.

UGA is represented by the following two protocols, which focus on different environments:

- **UGA Draw Protocol:** Focuses on the preboot environment and supports mode control and basic BLT functions.
- **UGA I/O Protocol:** Focuses on the OS-present environment and provides for public abstractions such as power management, channel I/O, and mode control, as well as private abstraction of silicon-specific interfaces.

For the UGA driver to function in both environments, it must be presented in EFI Byte Code (EBC) format and written using a defined but limited set of other protocols and functions.

The corresponding input protocol is the Simple Pointer Protocol, which abstracts mice, roller balls, and other similar pointing devices. Interfaces report resolution and relative motion. Other devices may also "sign up" events to be triggered when a change in the device's state (motion and/or a change in button status) is reported.

9.3.3 Example: Console Splitter

The "console splitter" is an architectural example of the use of the basic user I/O abstractions. It serves the purpose of grouping a configurable number of devices behind what appears to outside drivers and applications as a single device. The driver publishes a single Simple Input Protocol and Simple Text Output Protocol. The driver is responsible for determining the "least common denominator" attributes for the logical device so requests that callers make will fit on all of the abstracted devices.

9.4 Human Interface Infrastructure

Many of the higher-order user interfaces are provided by the Human Interface Infrastructure (HII) Protocol. This protocol can be viewed as providing access to a database that manages fonts, keyboard mappings, strings, and forms. The protocol accepts packs consisting of descriptions of any of these elements and disperses them accordingly.

Fonts and keyboard mapping are managed as shared resources. Any driver that attempts to display characters may use the fonts contributed by any other driver, for example. Any application using the Simple Input Protocol may receive Unicode characters that are decoded from scan codes by using a keyboard map.

Strings and forms are unique to the driver(s) or application that submitted them. The HII Protocol returns a handle to be used to query the strings and forms. A string is thus referred to by a triple: a handle, a country code, and a string token. Because forms use strings in the same package, they receive the same handle. All parts of the package are considered optional. For example, the package submitted by a purely text-based application might contain fonts and strings but is unlikely to contain forms.

9.4.1 Fonts

Fonts are submitted using the standard pack submission function.

One font-based function in the HII Protocol provides a bridge between the Simple Text Output Protocol and the functions of the UGA BLT-level protocols by translating strings to their pixel equivalents, one output character at a time. Note that, due to the definition of Unicode, this translation might cause several characters in the input string to be consumed.

9.4.2 Strings

Strings are submitted using the standard pack submission function. Separate string packages must be submitted in the same pack for each language (or set of languages) to be supported.

A string package supports one or more languages consisting of a single primary language and, optionally, a number of secondary languages. A user might designate himself to be Catalan, a separate language type in the ISO 639-2 language scheme. For the purposes of configuring a computer, Catalan is close enough to Spanish. A developer would therefore indicate a primary language of Spanish with a secondary language of Catalan.

Some strings are not known at build time. For example, the driver that configures memory will generally determine the amount of memory that is installed as a part of that configuration and then be expected to report that amount to the user. The simplest way to report it is to use the HII Protocol function to write over a dummy string with a dynamically created string.

A separate Unicode Collation Protocol provides for changing case in Unicode strings (the equivalent of C's **`toupper()`** and **`tolower()`**) and for comparing alphabets.

9.4.3 Keyboard Mapping

Keyboard mapping is submitted using the standard pack submission function. Keyboard maps vary by both country and language.

9.4.4 Forms

Forms are submitted using the standard pack submission function.

Designated forms may be extracted from the HII database along with the associated handles (so that the accompanying strings may also be extracted). This mechanism provides the basis for the ability of these forms to be made available to OS-present browsers either as they are or after conversion to a standard forms language such as HTML.

Forms can be updated after they are submitted using the HII Protocol. An IFR form tag (known as a *label tag*) identifies areas where it is possible to make modifications within a form. The driver that is modifying the form provides an IFR tag stream, a handle, and the value associated with the label tag. The new IFR is inserted after the label tag. IFR is designed to be completely position independent to enable this and other features.

9.5 Setup

The EFI Form Browser Protocol provides browser capabilities for forms. The forms can have already been submitted using the HII Protocol or can be handed in when the EFI Form Browser Protocol is invoked.

If the forms were previously submitted, Setup performs as any browser would by mapping the tags into the equivalent user interface, accepting input, and reporting the results. The IFR tags are less rich at constraining the visual aspects of the user interface than their HTML equivalents. For example, the IFR one-of tag may be translated into several HTML tags including radio buttons and drop-down combo boxes.

If Setup is invoked when a form is passed in, Setup processes that form and returns the results to the caller in a manner analogous to Common Gateway Interface (CGI) network-based organizations. Setup will also support certain callback tag modifiers that it would not support when processing previously submitted forms. If it encounters these tags, it calls the routine that invoked it with the parameters describing the location of the tag, so that the driver can dynamically manipulate the data in the forms. This feature allows Setup to act more dynamically than it would with previously submitted forms. On the other hand, the callback mechanism is more complex for the driver to support and such functionality is lost if the form is designated to be passed on to an OS-present browser.

Setup is the closest interface to the sort of windowing interfaces that are found in modern OSs. The Setup Protocol reflects the leaner nature of the preboot space.

9.6 Processing Forms Results

When the system has finished processing forms that were extracted from the HII database and is about to execute the boot target, the firmware exports the contents of the HII database and the resultant configuration changes for the O/S to use during runtime. The data that is exported is in a form that is easily used by an OS-present browser and can provide the ability to configure the system during the OS runtime. An OS-present browser might configure a system from the data that had previously been exported and then use a capsule to pass back the resulting configuration changes so that the changes can be processed during the next preboot phase of operation.

10
Applied Security

10.1 Applied Security within the Framework Boot Phases

Security impacts several aspects of the Framework infrastructure. The security infrastructure involves the following:

- The PEI Security PPI and the DXE Security Architectural Protocol in the PEI and DXE foundation provide the platform security policy.
- The SEC code establishes the root of trust by being the first Framework code to run when the reset vector is received.

One central security requirement is the ability to perform an authenticated update to avoid the platform becoming hijacked by rogue code or becoming inoperable (i.e., the “doorstop” phenomena).

10.1.1 Power-on Security

This section describes a range of capabilities that can be optionally applied depending on system requirements. These technologies are optional because a deeply embedded system might have scant requirements, while an enterprise server might require all power-on security items. What is central to the architecture, however, is to detail the surface areas wherein the architecture would be applied and the components interact.

The SEC phase begins with power-on and includes the first few steps that are required to find and authenticate the PEIMs before they are launched. The objective is to ensure the following:

- The first code executed by the processor is trustworthy.
- This code has sufficient resources in and of itself to determine the trustworthiness of any subsequent code.

During the SEC phase, a trusted boot scenario requires chain-of-trust maintenance throughout the firmware preboot execution. However, there are alternate schemes (which are not covered in this document) that allow the trust perimeter to be erected late in the platform evolution. This late trust perimeter can be accomplished with Framework firmware through the use of trusted core components and security policy components. These core components include the PEI and DXE; the policy components include the security PEIM and security driver.

As the PEIMs initially configure the processor, chipset, and platform, any code executed before these PEIMs (i.e., the SEC code) is extremely constrained in the resources that it can assume are available for its use. The SEC infrastructure code will optionally authenticate the PEI Foundation, and the PEI Dispatcher will optionally authenticate any PEIM before dispatching it. Valid responses to a failed authentication include deferring dispatch of the PEIM and/or logging the failure.

Because of these constraints, the specific form of image authentication is highly dependent on the characteristics of the specific processor, chipset, and platform combination and cannot currently be specified as part of the Framework architecture. However, we may illustrate several possible approaches. Note that all of the proposed solutions require that the processor have some prior knowledge of the platform configuration.

Also note that all of the proposed solutions require at least a minimum level of hardware support. At a minimum, the platform must be able to hold data objects that are used to anchor the verification (for example, a public key or a list of hashes) in a way that prevents tampering or deletion. Such protected storage cannot be supported solely by software.

10.1.2 Security Services

At the end of the PEI phase, the initial processor, chipset and platform PEIMs have been executed. As the first step in the transition to the DXE phase, a set of services are loaded and a DXE Dispatcher is launched from the *a priori* list. The services include the following:

- Security services in support of standards such as Boot Integrity Services (BIS) and Trusted Computing Group (TCG)
- OEM-defined platform security (Security Architecture Protocol)
- Technology-specific image signing (GUIDed authentication sections in FFS)

The security services have a richer operating environment than the power-on security services, such as the TCG PPI. They might assume a small amount of system memory and the availability of basic chipset and platform features. If security features are included in the processor or chipset, they can be accessed by these services.

The security services provide the following capabilities:

- Retrieving the boot authorization key
- Verifying an object's manifest against the boot authorization key
- Logging the verification and launch of each object
- Updating the boot authorization key
- Exposing the underlying cryptographic functions used in the above four capabilities

The services are used by the callers of the Security Architecture Protocol and the BDS phase boot manager. Both allow drivers to be discovered, loaded, and executed in a sandbox environment. Both dispatchers use the services to perform integrity and authenticity tests before they accept any discovered driver.

Each PEIM, driver, or application must come with an associated manifest that is signed by the boot authorization key to indicate that the system owner has approved it for use on this platform. If an image does not have an appropriate manifest, or if the manifest verification test returns an error, then the dispatcher must decide how to proceed. There are two alternatives:

- Reject the driver, log the failure, and proceed. Any subsequent phase or application can examine the failure log, including the final OS and/or the user. This solution produces a clean (untainted) execution environment but without the services that the suspect driver would have provided.
- Accept the driver anyway but log the failure. Any subsequent phase or application can examine the failure log, including the final OS and/or the user. Note that this solution results in an execution environment that is considered tainted, as no assertions of integrity or authorization can be made. This alternative is not recommended.

Note that the dispatcher may ask the user to accept or reject the driver. These decisions are subsumed in the security driver that publishes the Security Architectural Protocol or PPI.

It is strongly recommended to define an additional hardware extension somewhere in the platform to make it easier to identify tainted boots. This extension can be as simple as a "clean boot flag" that is set on power-up and then may only be read or cleared. Once cleared (due to accepting a failed driver), the flag remains cleared until a power cycle.

The services defined above are themselves based on combinations of underlying cryptographic functions (for example, hash and verify signature). Beyond the basic image integrity and authorization checks of the services, these underlying functions can also be useful to the PEIMs and applications. Therefore, the underlying cryptographic functions are exposed. However, as part of the services, these functions are limited to supporting algorithms and standards that are required for verifying and authenticating images.

10.1.3 Digital Certificates

A *digital certificate* is a method of exposing a public key in a way that the key's integrity can be verified. The most common standard for digital certificates is the X.509v3 open standard for certificate content encoding. This standard is complex and provides for a variety of data encoding and an open-ended set of attributes to be included in the certificate. While these certificates are quite flexible, software to interpret them can be quite complex, far more than is typically required in the Framework environment, particularly for image verification.

Therefore, digital certificates to be used in the Framework environment require only a small subset of the full X.509v3 standard. The TCG specification describes this subset. Specifically, Framework certificates must include at least the following fields:

- "Subject" public key
- "Issuer" public key
- Signature of the certificate, generated with the issuer's private key

10.1.4 Signed Manifests

A *signed manifest* is a verifiable "bill of materials" representing a collection of objects, such as an executable image. The manifest does *not* contain the objects themselves, only references to the objects. Key characteristics include the following:

- Multiple data objects can be represented within a single manifest.
- The manifest contains a hash for each object, which allows each object's integrity to be tested independently.
- Each object may have optional information to facilitate locating the object or expressing related attributes.
- A digital signature covers the manifest, ensuring that the manifest itself has integrity.

Each driver vendor produces a signed manifest representing its plug-in and signs the manifest with a private key held by the vendor. A corresponding public key is released in a certificate, along with a certificate chain that traces the authenticity of the vendor's key back to some trusted root. Intel has tool suites for the OS-present environment to aid in the creation of key pairs, public-key certificates, and signed manifests.

10.1.5 Boot Authorization Key

The EFI BIS Protocol has the concept of a Boot Authorization (BA) public key that represents the system owner (an IT department in the corporate environment or the user in the consumer environment). The system owner also holds a corresponding private key, which is held separately and is used in the OS-present environment. The key pair is used to authorize drivers and applications for use in the platform, by countersigning their manifests.

The BA public key must be held in persistent storage that is accessible to the Framework BIS driver. Ideally, the BA public key is held in protected persistent storage, where updates to this key are cryptographically protected.

The general process is as follows:

1. At install time, the system owner examines the supplied plug-in or application and determines if this image will be authorized to run on the platform. This test is only partly automated—the basic image verification can be performed by software, tracing the manifest signature by the vendor back through certificate chains to a trusted root. However, this method handles only the integrity test.
2. The system owner must still determine if this image is suitable for use on the platform. If the system owner decides it is, the owner oversigns (appends his or her signature to) the manifest.
3. At this point, the plug-in or application and its oversigned manifest may be installed in the system.
4. On a subsequent boot, the plug-in or application is discovered and its manifest retrieved. For all such modules or applications, the manifest is verified, with the signature verification based on the BA key embedded in the platform. If this test fails, either of the following occurred:
   - The image does not match the manifest (bill of materials).
   - The image's manifest was not oversigned by the BA key and therefore has not been explicitly approved for use on this platform.

5. If no BA key is installed in the platform, then all images are presumed to be approved for use in the platform.
6. Optionally, image integrity testing might still occur, with the verification using the vendor's signature rather than the added BA signature.

A system may be delivered from the OEM with no BA key installed. However, each installed image should still come with a manifest signed by the vendor, and the manifest storage should contain sufficient empty space to accept an additional signature. The system owner then has the option of reviewing each boot image and selectively authorizing the use of each image by oversigning with the BA key.

OS-present tools can be provided to do the following:

- Enable the generation of a BA key pair.
- Display an image's manifest information and the results of the vendor signature verification so the system owner can approve or reject the image for use.
- Oversign the image manifest.
- Install the oversigned image and its manifest in the platform.
- Update the BA key in the platform.

10.2 Security Usage Models

This section provides selected examples for common tasks in support of the Framework security model.

10.2.1 Secure Boot

A secure boot involves the PEI and DXE Dispatchers having each PEIM and EFI driver wrapped in an authentication section. The authentication results are passed to the Security PPI and Security Architectural Protocol respectively in response to the extraction. The authentication status dictates whether the PEIM or driver is dispatched. This decision is left up to the Security PEIM or driver. The behavior can be to skip the PEIM in PEI, set the EFI driver to the Schedule on Request (SOR) queue, or lock down the flash and dispatch the untrusted module in either phase.

10.2.2 Attested Boot

An attested boot includes creating a record of all of the PEIMs or EFI drivers that have been launched. The record will involve an event log in unsecure memory of all of the PEIMs and EFI drivers that have been launched, the platform state, and other environment information. In addition to this log, the relevant information will be logged into a write-only resource, such as the Platform Configuration Registers (PCRs) in TCG using the Hash-Extend operation. This event logging will occur from the SEC phase, through PEI, and into DXE. The challenger can replay the event log if it wishes to make a trust decision about the firmware. A common challenger might include the OS loader in a trusted boot environment.

10.2.3 Approving Boot Images

The EFI BIS Protocol can be used to challenge the authenticity of a launched application. This same technique can be used to store the BA key for additional components. The use of the EFI BIS Protocol is important because the origin of images cannot necessarily be trusted, especially images loaded across the network using the EFI PXE Base Code Protocol. As such, an EFI implementation can challenge any image that is loaded across the network or locally by using the EFI BIS Protocol. This challenge entails having a set of credentials associated with the image, which is not limited to a signed manifest describing the image. If the BA key (i.e., certificate) allows sufficient corroboration of the manifest, then the loaded image can be deemed trustable.

10.2.4 Authenticated Update

In addition to challenging images before they are invoked, updating the firmware is a key aspect of the firmware operational model that has security implications. As such, the capsule update mechanism of the Framework needs to have a means by which to qualify the capsule. Before a capsule's contents become exposed as executable or as content that can be dispatched by DXE, the associated firmware volume from the capsule needs to have its authentication state assessed. This assessment can be from the DXE Foundation firmware volume read-only driver that is used in concert with the Security Architectural Protocol driver. The DXE Foundation would discover a new possible firmware volume and then query the Security Architectural Protocol about the authentication state of the candidate volume. Because the Security Architectural Protocol driver has *a priori* platform knowledge, it can dictate which volumes were part of the initial trusted set, which ones are newly introduced through a capsule restart and need additional scrutiny, and so on. As mentioned with the earlier technology, the means by which this trust assertion can be made may entail cryptographic techniques or aspects of the technology mentioned above, but the policy and associated means are under the control of this platform agent and invisible to the DXE Foundation.

10.3 Security Technology

As a result of the component-based design of the Framework firmware, many future possibilities can be implemented. As mentioned in the SEC description, the Framework firmware has the concept of core components, such as the PEI Foundation, SEC, DXE Foundation, and some set of DXE Architectural Protocols. A security taxonomy (model) can be defined such that these components constitute the Trusted Computing Base (TCB) for the system.

This security model has many ramifications. The first result is that it provides a distinction for the code and resources that can be "armored" against other unknown or unsigned components such as drivers and applications. This armoring can include use of software techniques, such as running untrusted code as software-contained EBC drivers. The armoring can also extend to hardware. Hardware techniques could leverage protection facilities, such as paging, in current processors and extend to more exotic techniques such as software-only or future hardware-based virtual machine monitors. Also, because these TCB components qualify any subsequent components that get loaded, they are the only ones that need to be "measured" in the sense of an attested boot. A TCG Platform Configuration Register (PCR) could be reserved for each of the cores and then a final PCR for the rest of the PEIMs and drivers. These PCRs would then act as a bill of materials that describes the platform in an "attested boot" scenario whenever the trust of the firmware is challenged or needs to be tracked.

11 Manageability

11.1 Introduction

Manageability is a term describing the features of a system that combine to enable the system to be configured, supported, and diagnosed. For a home user, manageability might take a very different form than for a corporate user whose notebook is hooked to a LAN infrastructure and who is supported by an IT organization.

Many of the features that lead to a more manageable system are features that are also used for other purposes or are extensions to existing features. The following are three major classes of functionality:

- Features that allow the drivers accomplishing other tasks to report progress and results and to accept configuration
- Features that allow access to accumulated data and allow management applications to perform configuration
- Features that provide an infrastructure to assist the first two classes

Manageability is an area of continuing innovation. At the time that the Framework architecture was being defined, the state of the art for managing computer systems was still somewhat primitive. As time passes, the situation hardly improves. Systems are managed using serial ports that emulate terminals that are no longer made by companies that no longer exist. The existing standards for reporting information seem out of date even before they are released. For example, memory speed was reported for several years in terms of nanoseconds. It was then reported in megahertz. Still later, simple numeric "names" (equivalent to airplane types such as Boeing 707* and Airbus A300*) were in vogue. This constant evolution of the things being managed, as well as the technology used to manage them makes flexibility a necessity.

11.2 Data Structures

11.2.1 GUIDs

EFI's use of GUIDs to identify drivers, firmware volumes, and just about everything else makes it simpler to track the software equivalent of assets inside systems.

11.2.2 Firmware Volumes

With well-defined firmware volume formats, manageability applications can be written to locate drivers and other files in a variety of systems that are created by different vendors to determine their revisions and status.

11.2.3 Device Paths and Device Descriptions

A device path is a path from the root of the system (roughly corresponding to the processor in most architectures) to devices. This path is similar to a directory path for a file in an OS. The device path describes the buses that the device is "behind" and the connections (such as USB ports) that identify that device. This type of device path has been implicitly common in many OSs but has been lacking in the preboot space. The standard description of device paths enables a consistent description of the hierarchy of devices in a system, thus allowing management applications to better describe the assets contained in systems and where those assets are located.

Device descriptions are the counterpart to device paths. They describe the device without describing its location. For add-in devices, the developer cannot have *a priori* knowledge where the device will be installed (what USB hub port or what PCI slot, for example). These descriptions can be used to determine if, for example, a firmware update will work in a system.

11.3 Progress Codes

Progress codes are used to indicate that the system has reached a particular state. They are not error codes—that is, they do not indicate a failure. Instead, they are used as gross measurements of the duration of activities and to determine the context of the system when it unexpectedly crashed. Progress codes replace the existing POST codes, which are simple 2-hex-digit values.

This specification uses the term *progress code protocol* to refer generically to a protocol that the dispatcher invokes before invoking drivers and that includes the ideas of progress, status, and error codes. The actual name of the protocol differs based on the phase of operation in which it is invoked. The following are the progress code protocols invoked in each phase of the Framework:

- PEI phase:
  - — Status Code Service (PEI Service)
  - — Status Code PPI (optional architectural PPI)
- DXE phase: Status Code Architectural Protocol

The driver that publishes the progress code protocol is defined by the architecture as platform specific. In particular, the Framework architecture does not define what this protocol is to do with the information it is provided. Some obvious alternatives include the following:

- Mapping the data provided by the caller into a series of numerical fields for output on a debug card
- Logging the data for later analysis

11.4 Data Hub

The data hub is a volatile database intended as the major focus for the accumulation of manageability data. The hub is fed by "producers" with chunks of data in a defined format. Consumers may then extract the data in temporal "log" order. As an example, progress codes might be recorded in the data hub for future processing. Other data contributed to the data hub might include, for example, statistics on enumerated items such as memory, add-in buses, and add-in cards and data on errors encountered during boot (for example, the system did not boot off the network because the cable was not plugged in).

Some classes of data have defined formats. For example, the amount of memory in the system is reported in a standard format so that consumers can be written to extract the data. Other data is system specific. For example, additional detail on errors might be specific to the driver that discovered the error. The consumer might be a driver that tabularizes data from the data hub, providing a mechanism for the raw data to be made available to the OS for post-processing by OS-based applications.

The intent of the data hub is for drivers that enumerate and configure parts of the system to report their discoveries to the data hub. This data can then be extracted by other drivers that report those discoveries using standard manageability interfaces such as SMBIOS and Intelligent Platform Management Interface (IPMI). The alternative to a data-hub-like architecture is to require all drivers to be aware of all reporting formats.

11.5 Remote Manageability

11.5.1 Console Redirection

An IT support person might be required to support systems in several sites spread around a country, region, or the world. In the past, the person would be required to visit the system or rely on an inexperienced user to make changes required to resolve an issue. The trend has been for manageability applications to use the existing network infrastructure to gain remote access to the system. Much of this management can take place under the auspices of the system but actions such as remotely installing the OS and reconfiguring Setup must take place under the control of the firmware.

To enable remote management, the architecture supports abstractions of consoles that look to the drivers and applications running in the system as if they were little removed from hardware when, in fact, they are the tip of a rich network or serial stack.

Protocols are defined for serial, Telnet, and LAN redirection. These protocols define the basic infrastructure but leave the higher-level aspects open for the system designers to decide on trade-offs of rich content versus size.

11.5.2 User Interface Design Considerations

Different market segments and different users have different requirements for manageability. Remote management has traditionally been through a command line interface rather than menuing. There are several reasons for this difference:

- The first manageability efforts were mainframe based so many of the standards evolved around mainframe technology, particularly the use of serial ports and command interfaces.
- The bandwidth across manageability channels has traditionally been slow enough that graphical performance was not reasonable.
- Command line interfaces are easier to control with scripts (batch files) than with menuing interfaces.

The designer should consider supporting both menuing and command line interfaces to address the requirements of end users and managers.

11.5.3 The Future of Technical Support

With the advent of higher speed communication channels such as ISDN, DSL, and broadband (cable modems), it is possible to imagine the day when the home user's system is managed in a similar way. The user would request technical support over e-mail or the phone, providing the technical support access to the system using some security mechanism. The technical support person could then run diagnostics and change configuration to resolve the issue. Manageability has traditionally been viewed as the domain of IT teams managing multitudes of systems on the corporate intranet. How long it will take that technology to filter down to the other types of systems is a function of a combination of factors, including the following:

- Proliferation of higher-speed communications to more regions of the world
- Manageability of the system
- Existence of sufficiently secure channels
- Creation of the technical support applications

The firmware architecture defined here is intended to enable this type of scenario.

11.6 Add-in Card Manageability

An option ROM (OpROM) is a piece of firmware that resides on an add-in card or its equivalent. When the functionality associated with an add-in ROM is integrated onto the system board, the OpROM firmware is typically stored as separate drivers in the system firmware volumes.

OpROMs perform the equivalent of OS-present drivers—they translate between hardware with unique interfaces and the interfaces of the rest of the software. Because the preboot environment uses only certain types of devices—namely those involved in providing boot images and user output devices—OpROMs are typically provided only on these classes of devices. User-input devices have interfaces that are standard enough that they rarely require OpROMs. The EFI OpROM format supports several manageability features.

The most fundamental feature of the architecture is that the code that is extracted from OpROMs is treated the same as if it were from the system firmware. The architecturally required protocols are available for the drivers extracted from the OpROM the same as they are from drivers carried on the system. The same abstractions that allow the system configuration to be remotely managed allow the OpROMs to be managed. Further, using the same interfaces provides a level of consistency in the user interfaces and reduces code size on the OpROMs.

The OpROMs have specialized interfaces to support configuration and diagnostics. The configuration interface is intended to allow the OpROM to provide specialized configuration "applications" for general-purpose configuration. Further, the interface may be extended to provide device-specific utilities, for example, providing an OS-neutral mechanism to configure a RAID controller. The diagnostics interface is useful for enabling the manager to perform preboot or post-crash diagnostics on the hardware without requiring a boot to the OS. Diagnostics may also become customers of the data hub, scanning for and processing data previously generated by the drivers extracted from the same OpROM.

11.7 Manageability in the Phases

11.7.1 BDS

11.7.1.1 File System Drivers

The *EFI 1.10 Specification* defines the Simple File System Protocol, which provides a file system layer over disklike devices. The Framework uses this protocol's definition and will typically include an implementation of it if the platform requires access to mass storage devices. This protocol can be used to access disk partitions that are formatted in a standard format. This functionality assists system manageability in several ways.

Most disk formats today provide for hidden disk partitions. These partitions are reserved early when formatting the disk and are effectively hidden from the OS during normal operation. These logical drives can be used to store everything from diagnostics, to disk partitions, to extended font tables. Disks that are formatted in the format defined by the Simple File System Protocol do not require booting off the disk to access data on that disk. Instead, the disk can be used as an extension of the system's firmware volume space.

11.7.1.2 Boot Control

BDS provides areas for innovation for boot ordering selection. The system attempts to boot off of devices in a prescribed order. That order is defined in system variables.

The change from BIOS Boot Specification (BBS) to BDS is also being used as an inflection point to enhance and improve the PXE-to-UNDI interface that defines the low-level standard interfaces between a network adapter and the rest of the firmware. These improvements enable simpler LAN boot and the use of the Simple Network Protocol as the basis for the preboot LAN stack. Drivers on top of that stack can then be written to front for console I/O, UGA, and other protocols to provide even graphical support remotely.

11.7.2 Runtime

Usually systems that are powered up are running the OS, not the preboot environment. The preboot environment, on the other hand, has visibility into parts of the system that the OS is not aware of but that are important to managing the system. For example, although the OS is aware of the amount of RAM in the system, it typically does not know the following:

- Number of RAM modules in the system
- Amount of RAM that each memory module is contributing
- Presence of bad RAM modules
- Number of free RAM slots that are available.
- Types of RAM modules that are required by the system.

On the other hand, upgrading RAM is one of the most typical upgrades performed by IT. To make the system useful to all of its customers, it behooves the preboot environment and OS to work together to provide seamless schemes for reporting configuration and effecting changes. The Framework architecture defines several schemes for doing this.

Standard manageability interfaces such as IPMI and SMBIOS are made available to the OS using services when the OS is loaded. These interfaces replace schemes that required the OS, or its applications, to search for interfaces through memory.

The representation of both system and add-in card setups using a common forms-based notation enables OS-present utilities to be written that browse the forms. These utilities provide several benefits:

- Allow OS-present configuration of the system without requiring applications specially written for each platform.
- Allow the OS-present configuration to use interfaces similar to the rest of the OS.

Scripting tools may also be written to process the same forms data.

Once the configuration is created, it may be reported back to the system firmware through two architectural mechanisms:

- Runtime services that provide both read and write access to variables
- Capsules

In the first mechanism, variables are intended to operate in a manner analogous to environment variables in the OS space and store configuration information that is useful both by the preboot space and the OS. Defined variables include the country code, language code, and boot paths. Other variables that are used by PEI and DXE modules to change their operations may be encoded in the configuration results. Again, the management application can easily be written so that it can process results from any form.

In the second mechanism, capsules provide a method by which drivers may be loaded in the OS space and handed off to the pre-OS space through a rebootlike operation. The most common use for this facility is for updating system configuration, either by providing results back into the preboot space from an OS configuration or by updating parts of the firmware that is resident in the system. The capsule mechanism is flexible enough to be adapted to a wide variety of systems. For high-availability systems, the OS-present capsule mechanism can be defined to queue updates so that the system can remain functional for long periods of time. The update can then take place the next time the system is rebooted for other reasons. From the user's point of view, it is common for the system to reboot to load drivers so the required reboot is not unexpected.

11.7.3 Afterlife

The "last" phase in the Framework architecture is a place for innovation. The OS can be viewed as a very large and complicated EFI application. If care is taken, the OS can return to EFI once its time is complete or EFI can be reloaded without corrupting vital parts of the OS memory. This transfer can occur either voluntarily (through a shutdown request) or involuntarily (through a crash). The equivalent of this phase has been in place in higher-end systems, particularly mainframes, from very early in the history of computers, as exemplified by the post-mortem core dump. With the considerably larger amount of memory in modern systems, more intelligent post-crash analysis tools are required.

The EFI interface for entrance into the Afterlife phase is through the EFI Runtime Service **`ResetSystem()`**. Parameters that are provided to this runtime service are used to control the type of reset, including possible directions for reload of EFI.

The environment provided by EFI and the Framework in this phase is simple enough and rich enough to provide a useful resource to analyze memory and the state of peripherals in the preboot environment using, for example, the diagnostics provided by the OpROMs.

12
Legacy Compatibility

12.1 Introduction

There is always a transitional period between the introduction of a new technology and the phase-out of the technology that it replaces. The addition of compatibility code to the Framework bridges this transitional period from BIOS to EFI. The compatibility code allows a traditional OS or an EFI OS to be booted off a device that requires a traditional OpROM. The compatibility code consists of the following components:

- A series of EFI compatibility drivers and other EFI code that is added to EFI (referred to as EfiCompatibility code)
- A non-EFI, stripped-down, traditional 16-bit BIOS (referred to as Compatibility16 code)

Together, the Compatibility16 code and the EfiCompatibility code constitute the compatibility code that is described in this chapter.

12.2 Compatibility Environment

The compatibility environment that the Framework and EFI code have been designed to support assumes external and internal interfaces. The following sections define those assumptions.

The degree of support for legacy "application" code—i.e., code that expects a conventional BIOS on the platform—depends on the amount of Compatibility16 code included in the platform. A platform vendor can choose how much of the Compatibility16 runtime compatibility to provide. For example, the initial Framework implementation included support for only traditional OSs that were ACPI aware. In this particular example, MS-DOS* will boot but there is no guarantee that all DOS programs will work.

12.2.1 External Assumptions

The external assumptions include assumptions about both external hardware and software, as follows:

- When they are unloaded, EFI device drivers that have an EFI OpROM leave the hardware in a neutral state that allows the equivalent traditional OpROM to be invoked without any adverse device interaction.
- Traditional OpROMs cannot be unloaded and thus leave the hardware in a non-neutral state.
- The UGA hardware is bi-modal, which also supports a VGA emulation mode.
- The UGA device provides both a UGA OpROM and a traditional VGA OpROM.
- Legacy device programming is done either by EFI, EfiCompatibility, or ACPI. EfiCompatibility is the EFI code that corresponds to EFI compatibility drivers, code that generates data for compatibility interfaces, or code that invokes compatibility services.

Table 12-1 lists the OS and OpROM combinations that were assumed to be authorized (legal) when the Compatibility16 code was designed.

Table 12-1. OS and OpROM Combinations

Video OpROM	Other OpROM	Boot Device OpROM	OS	Valid
EFI	EFI	EFI	EFI	Compatibility not required
EFI	EFI	EFI	Traditional	No
EFI [Note]	EFI	Traditional	EFI	Yes
EFI	EFI	Traditional	Traditional	No
EFI [Note]	Traditional	EFI	EFI	Yes
EFI	Traditional	EFI	Traditional	No
EFI [Note]	Traditional	Traditional	EFI	Yes
EFI [Note]	Traditional	Traditional	Traditional	Yes
Traditional	EFI	EFI	EFI	Yes
Traditional	EFI	EFI	Traditional	No
Traditional	EFI	Traditional	EFI	Yes
Traditional	EFI	Traditional	Traditional	No
Traditional	Traditional	EFI	EFI	Yes
Traditional	Traditional	EFI	Traditional	No
Traditional	Traditional	Traditional	EFI	Yes
Traditional	Traditional	Traditional	Traditional	Yes

Note: The EFI UGA video driver must be unloaded and reinvoked in VGA mode with a VGA OpROM.

12.2.2 Internal Assumptions

The Compatibility16 code consists of a traditional runtime BIOS, INT18, and INT19. The POST code is removed. To present a minimal space footprint, the EFI code functions as the traditional POST equivalent. The following assumptions pertain to the Compatibility16 bit code except where indicated:

- **There are no runtime text messages.** It is considered too expensive, space wise, to carry a display engine for a few messages and to ensure that EFI and Compatibility16 are localized coherently. Compatibility16 is a traditional legacy BIOS with the POST and BIOS Setup code removed. Compatibility16 BIOS code generally executes in real mode.
- **There is no need for cache control.** It is assumed that cache is always enabled or controlled by the OS.
- **There are no flash or nonvolatile RAM (NVRAM) updates.** There are several reasons for this assumption:
  - — Compatibility16 code knows nothing of EFI firmware volumes.
  - — Having multiple independent entities trying to maintain flash or NVRAM will introduce system instability and/or security problems.

This assumption does have the following ramifications to the Compatibility16 code:
  — No Extended System Configuration Data (ESCD)
  — No processor patches
  — No update of SMBIOS structures
  — No CMOS save to flash
- **There is no BIOS Setup.** EFI provides this functionality.
- **There is no POST.** EFI provides this functionality.
- **SMBIOS 2.3 is supported in a limited manner,** as follows:
  — Table entry only.
  — No Plug and Play interface.
  — Static information only; no flash updates.
  — All SMBIOS functions are read-only and both OEMs and manufacturing must use EFI utilities to write asset tags.
- **The boot hard disk drive needs to be an INT13 drive 0x80.** Other drives can be assigned numbers in any order.
- **USB legacy is supported from INT19 on.** Pre-INT19 is EFI using any required drivers, including keyboard and mouse.
- **EFI is sufficient for S3.** No compatibility code required.
- **EFI drivers, EfiCompatibility drivers, or ACPI Source Language (ASL) are used to program traditional devices.** There are no Plug and Play device nodes.
- **EFI provides the ASL code.**

12.3 Framework EfiCompatibility Code

The Framework EfiCompatibility code is a component of BDS. Using EfiCompatibility code changes the operation of the BDS phase in the following ways:

- Additional drivers are loaded to emulate traditional software interrupts.
- Traditional OpROMs are dispatched as required.
- A special path is required to boot a traditional OS.

The boot device and the OS being booted determine the amount of compatibility code to execute. The minimal amount exists in the case where an EFI-aware OS is booted off the onboard IDE. This scenario requires only the core EfiCompatibility code, Compatibility16, and the INT13 driver. Because the EFI-aware OS expects to use EFI protocols, the console-out driver can be an EFI driver such as UGA. The traditional equivalent of INT19 never occurs because the OS is not loaded off the boot sector. The maximum amount of compatibility code to execute exists in the case of booting a traditional OS off an OpROM-controlled device. This scenario requires that all OpROMs are traditional. In addition to the minimal set of compatibility code, all other components must be loaded. A traditional INT19 occurs with the OS being loaded off the boot sector.

Figure 12-1 below shows the Framework support for compatibility code.

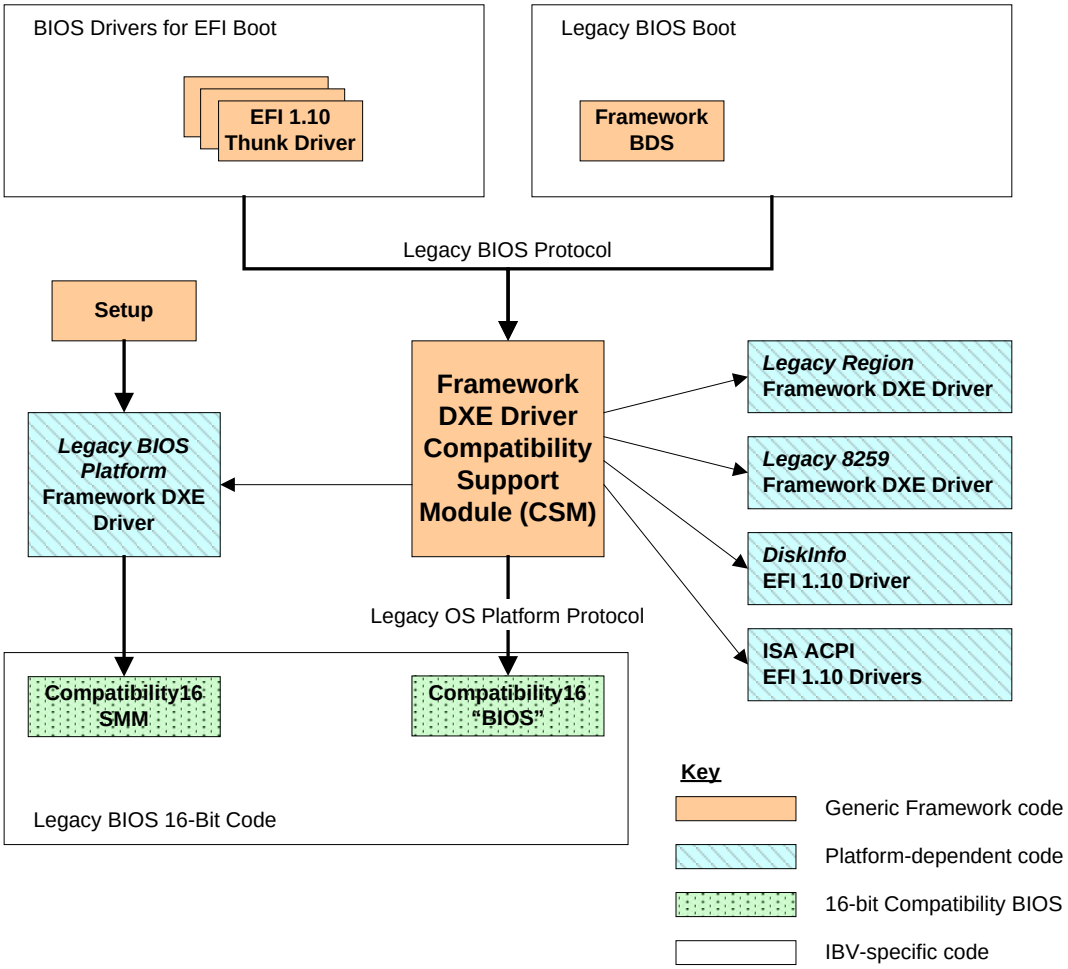

Figure 12-1. Compatibility Overview

12.3.1 Major Components

The Framework EfiCompatibility code incorporates the following features and capabilities, which are described in the following sections:

- BDS and compatibility interfaces
- A stripped-down IBV 16-bit real-mode runtime BIOS
- The ability to “thunk” and “reverse thunk” between 16-bit compatibility code and EFI
- Setup capability through EFI Setup
- UGA-to-VGA INT10 driver
- INT13 driver
- INT16 driver

12.3.1.1 BDS and Compatibility Interfaces

When the boot manager component running in the BDS phase determines that the selected boot path requires compatibility code to be executed, it will initiate some or all of the following operations to complete the path:

- Dispatch compatibility interfaces
- Determine which OpROMs to invoke
- Determine boot devices
- Launch the traditional OS boot process

12.3.1.2 IBV 16-Bit Real-Mode Runtime BIOS

A 16-bit real-mode runtime BIOS is a stripped-down version of a traditional BIOS. It is required only to contain the following:

- EFI/Compatibility16 communications
- INT18
- INT19
- Runtime code

EFI provides the equivalents of POST and BIOS Setup; see section 12.2.2 for a complete list of what is not included in the IBV 16-bit real-mode runtime BIOS. A series of functions is added so information can be passed from EfiCompatibility to the Compatibility16 code. These functions are accessed using a new table, the Compatibility16 Table, that is located in the Compatibility16 code.

The Compatibility16 Table resides in the IBV 16-bit BIOS and is used to provide communications between EfiCompatibility and Compatibility16 code. This communication is through one of two mechanisms:

- EfiCompatibility thunks to an entry point, which is provided by the table, with a function number and a set of information.
- The table provides addresses and lengths for specific data storage locations within the Compatibility16 code. An example is the location where SMBIOS data structures are stored.

12.3.1.3 Thunk and Reverse Thunk

The thunk provides a mechanism to transfer control from native execution mode (EFI) to 16-bit legacy code. The reverse thunk provides a mechanism to transfer from 16-bit legacy code to native execution mode. The thunk provides an interface to emulate a Far Call into 16-bit code and an interface to emulate a software interrupt.

12.3.1.4 Setup

The EFI Setup displays the traditional devices.

12.3.1.5 UGA-to-VGA INT10 Driver

This situation occurs when traditional OpROMs are to be invoked. The UGA controller is to be placed in VGA emulation mode and the equivalent VGA OpROM invoked. This driver must translate EFI console-out data and requests into their VGA equivalents. Requirements for legacy VGA support include the following:

- All INT10 functions, both character and dot, must be supported.
- The OpROMs may directly access both VGA registers and video memory buffers.
- UGA hardware supports a VGA mode and can be switched between UGA/VGA modes multiple times.

12.3.1.6 INT13 Driver

This driver is used when EFI needs to access legacy-supported rotating media such as legacy floppy or hard disk drives. The driver translates EFI block I/O requests into the equivalent INT13 requests.

12.3.1.7 INT16 Driver

The Compatibility16 BIOS does not take over USB emulation until a traditional OS is booted. Until that time, Compatibility16 INT16 must cause a reverse thunk to an INT16 emulator and thunk back with the data. OpROMs that directly access the keyboard controller may fail to give the desired results when USB keyboards are used.

12.3.2 Traditional OS Boot

Booting a traditional OS requires the following additional code:

- **SMM:** EFI provides most SMM functionality. If any unique Compatibility16 SMM code is required, that code must be registered with the EFI SMM code.
- **SMBIOS:** The traditional BIOS uses the SMBIOS data structures that are generated by EFI.
- **Programming devices in traditional mode:**
  A traditional OS requires devices to be programmed differently than EFI. For example, EFI is polled while legacy devices are allocated interrupts, even if they are not normally used. This difference requires the devices to be reprogrammed when booting a traditional OS. As stated in section 12.2.2, Compatibility16 code does not contain code to program legacy devices. ACPI or EfiCompatibility might be used to perform these changes in configuration using a defined protocol. The traditional BIOS converts EFI data into an EfiCompatibility equivalent. An example is converting a polled serial port into an interrupt-driven serial port.
  Miscellaneous: An INT15 E820 table must be created and the ACPI Root System Description Pointer (RSD_PTR) programmed.
- **Boot process:** After the EfiCompatibility code has completed what processing it can, then the Compatibility16 code is invoked through the special table located in the Compatibility16 code, with the boot function containing a list of boot devices. This invocation allows the Compatibility16 code to perform any last-minute bookkeeping before issuing the INT19.

13
Boot Paths

13.1 Introduction

The normal code flow in the Framework passes through a succession of phases, in the following order:

1. SEC
2. PEI
3. DXE
4. BDS
5. Runtime
6. Afterlife

This section describes alternatives to this ordering.

13.2 Reset Boot Paths

The following sections describe the boot paths that are followed when a system encounters several different types of reset.

13.2.1 Intel Itanium Processor Reset

Itanium architecture contains enough hooks to authenticate PAL-A and PAL-B code that is distributed by the processor vendor. The internal microcode on the processor silicon, which starts up on a PowerGood reset, finds the first layer of processor abstraction code (called PAL-A) that is located in the BFV using architecturally defined pointers in the BFV. It is the responsibility of this microcode to authenticate that the PAL-A code layer from the processor vendor has not been tampered. If the authentication of the PAL-A layer passes, control then passes to the PAL-A layer, which then authenticates the next layer of processor abstraction code (called PAL-B) before passing control to it. In addition to this microarchitecture-specific authentication, the SEC phase of EFI is still responsible for locating the PEI Foundation and verifying its authenticity.

In an Itanium-based system, it is also imperative that the firmware modules in the BFV be organized such that at least the PAL-A is contained in the fault-tolerant regions. This processor-specific PAL-A authenticates the PAL-B code, which usually is contained in the non-fault-tolerant regions of the firmware system. The PAL-A and PAL-B binary components are always visible to all the processors in a node at the time of power-on; the system fabric should not need to be initialized.

13.2.2 Non-Power-on Resets

Non-power-on resets can occur for many reasons. There are PEI and DXE system services that reset and reboot the entire platform, including all processors and devices. It is important to have a standard variant of this boot path for cases such as the following:

- Resetting the processor to change frequency settings
- Restarting hardware to complete chipset initialization
- Responding to an exception from a catastrophic error

This reset is also used for Configuration Values Driven through Reset (CVDR) configuration.

13.3 Normal Boot Paths

A traditional BIOS executes POST from a cold boot (G3 to S0 state), on resumes, or in special cases like INIT. EFI covers all those cases but provides a richer and more standardized operating environment

The basic code flow of the system needs to be changeable due to different circumstances. The boot path variable satisfies this need. The initial value of the boot mode is defined by some early PEIMs, but it can be altered by other, later PEIM(s). All systems must support a basic S0 boot path. Typically a system has a more rich set of boot paths, including S0 variations, S-state boot paths, and one or more special boot paths.

The architecture for multiple boot paths presented here has several benefits, as follows:

- The PEI Foundation is not required to be aware of system-specific requirements such as MP and various power states. This lack of awareness allows for scalability and headroom for future expansion.
- Supporting the various paths only minimally impacts the size of the PEI Foundation.
- The PEIMs that are required to support the paths scale with the complexity of the system.

Note that the Boot Mode Register becomes a variable upon transition to the DXE phase. The DXE phase can have additional modifiers that affect the boot path more than the PEI phase. These additional modifiers can indicate if the system is in manufacturing mode, chassis intrusion, or AC power loss or if silent boot is enabled.

In addition to the boot path types, modifier bits might be present. The “recovery-needed” modifier is set if any PEIM detects that it has become corrupted.

13.3.1 Basic G0-to-S0 and S0 Variation Boot Paths

The basic S0 boot path is “boot with full configuration.” This path setting informs all PEIMs to do a full configuration. The basic S0 boot path must be supported.

The Framework architecture also defines several optional variations to the basic S0 boot path. The variations that are supported depend on the following:

- Richness of supported features
- If the platform is open or closed
- Platform hardware

For example, a closed system or one that has detected a chassis intrusion could support a boot path that assumes no configuration changes from last boot option, thus allowing a very rapid boot time. Unsupported variations default to basic S0 operation. The following are the defined variations to the basic boot path:

- **Boot with minimal configuration:**
  This path is for configuring the minimal amount of hardware to boot the system.
- **Boot assuming no configuration changes:**
  This path uses the last configuration data.
- **Boot with full configuration plus diagnostics:**
  This path also causes any diagnostics to be executed.
- **Boot with default settings:** This path uses a known set of safe values for programming hardware.

13.3.2 S-State Boot Paths

The following optional boot paths allow for different operation for a resume from S3, S4, and S5:

- **S3** (Save to RAM Resume): Platforms that support S3 resume must take special care to preserve/restore memory and critical hardware.
- **S4** (Save to Disk): Some platforms may want to perform an abbreviated PEI and DXE phase on a S4 resume.
- **S5** (Soft Off): Some platforms may want an S5 system state boot to be differentiated from a normal boot—for example, if buttons other than the power button can wake the system.

An S3 resume needs to be explained in more detail because it requires cooperation between a G0-to-S0 boot path and an S3 resume boot path. The G0-to-S0 boot path needs to save hardware programming information that the S3 resume path needs to retrieve. This information is saved in the Hardware Save Table using predefined data structures to perform I/O or memory writes. The data is stored in an EFI equivalent of the INT15 E820 type 4 (firmware reserved memory) area or a firmware device area that is reserved for use by EFI. The S3 resume boot path code can access this region after memory has been restored.

13.4 Recovery Paths

All of the above boot paths can be modified or aborted if the system detects that recovery is needed. Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes on nonvolatile storage devices (flash, disk) are managed as blocks. If the system loses power while a block, or semantically bound blocks, are being updated, the storage might become invalid. On the other hand, the device might become corrupted by an errant program or by errant hardware. The system designers must determine the level of support for recovery based on their perceptions of the probabilities of these events occurring and their consequences.

The following are some reasons why system designers may choose to not support recovery:

- A system's firmware volume storage media might not support modification after being manufactured. It might be the functional equivalent of a ROM.
- Most mechanisms of implementing recovery require additional firmware volume space, which might be too expensive for a particular application.
- A system may have enough firmware volume space and hardware features that the firmware volume can be made sufficiently fault tolerant to make recovery unnecessary.

13.4.1 Discovery

Discovering that recovery is required may be done using a PEIM (for example, by checking a "force recovery" jumper) or the PEI Foundation itself. The PEI Foundation might discover that a particular PEIM has not validated correctly or that an entire firmware has become corrupted.

13.4.2 General Recovery Architecture

The concept behind recovery is to preserve enough of the system firmware so that the system can boot to a point where it can do the following:

- Read a copy of the data that was lost from chosen peripherals.
- Reprogram the firmware volume with that data.

Preserving the recovery firmware is a function of the way the firmware volume store is managed, which is generally beyond the scope of this document. For the purpose of this description, it is expected that the PEIMs and other contents of the firmware volumes that are required for recovery will be marked. The architecture of the firmware volume store must then preserve marked items, either by making them unalterable (possibly with hardware support) or protect them using a fault-tolerant update process. Note that a PEIM is required to be in a fault-tolerant area if it indicates it is required for recovery or if a PEIM that is required for recovery depends on it. This architecture also assumes that it is fairly easy to determine that firmware volumes have become corrupted.

The PEI Dispatcher then proceeds as normal. If it encounters PEIMs that have been corrupted (for example, by receiving an incorrect hash value), it itself must change the boot mode to "recovery." Once set to recovery, other PEIMs must not change it to one of the other states. After the PEI Dispatcher has discovered that the system is in recovery mode, it will restart itself, dispatching only those PEIMs that are required for recovery. A PEIM can also detect a catastrophic condition or a forced-recovery event and inform the PEI Dispatcher that it needs to proceed with a recovery dispatch. A PEIM can alert the PEI Foundation to start recovery by OR-ing the **`BOOT_IN_RECOVERY_MODE_MASK`** bit onto the present boot mode. The PEI Foundation will then reset the boot mode to **`BOOT_IN_RECOVERY_MODE`** and start the dispatch from the beginning with **`BOOT_IN_RECOVERY_MODE`** as the sole value for the mode.

NOTE

At this point, a physical reset of the system has not occurred. The PEI Dispatcher has only cleared all state information and restarted itself.

It is possible that a PEIM could be built to handle the portion of the recovery that would initialize the recovery peripherals (and the buses they reside on) and then to read the new images from the peripherals and update the firmware volumes.

It is considered far more likely that the PEI will transition to DXE because DXE is designed to handle access to peripherals. This transition has the additional benefit that, if DXE then discovers that a device has become corrupted, it may institute recovery without transferring control back to the PEI.

Because the PEI Foundation does not have a list of what it is to dispatch, how does it know if an area of invalid space in a firmware volume should have contained a PEIM or not? It seems that the PEI Foundation may discover most corruption as an incidental result of its search for PEIMs. In this case, if the PEI Foundation completes its dispatch process without discovering enough static system memory to start DXE, then it should go into recovery mode.

13.5 Special Boot Path Topics

The remaining sections in this chapter discuss special boot paths that might be available to all processors or specific considerations that apply only for Itanium processors.

13.5.1 Special Boot Paths

The following are special boot paths in the Framework architecture. Some of these paths are optional and others are processor-family specific.

- **Forced recovery boot:** A jumper or an equivalent mechanism indicates a forced recovery.
- **Itanium architecture boot paths:** See the next section..
- **Capsule update:** This boot mode can be an INIT, S3, or some other means by which to restart the machine. If it is an S3, for example, the capsule cause will supersede the S3 restart. It is incumbent upon platform code, such as a memory initialization PEIM, to determine the exact cause and perform the correct behavior (i.e., S3 state restoration versus INIT behavior).

13.5.2 Special Itanium® Architecture Boot Paths

Itanium architecture requires the following special boot paths:

- Boot after INIT: An INIT has occurred.
- Boot after MCA: A Machine Check Architecture (MCA) event has occurred.

Itanium processors possess several unique boot paths that also invoke the dispatcher located at the System Abstraction Layer entry point (SALE_ENTRY). The processor INIT and MCA are two asynchronous events that start up the SEC code/dispatcher in an Itanium-based system. The EFI security module is transparent during all the code paths except for the recovery check call that happens during a cold boot. The PEIMs or DXE drivers that handle these events are architecture aware and do not return the control to the core dispatcher. They call their respective architectural handlers in the OS.

13.5.3 Itanium Architecture Access to the Boot Firmware Volume

Figure 13-1 shows the reset boot path that an Itanium processor follows.

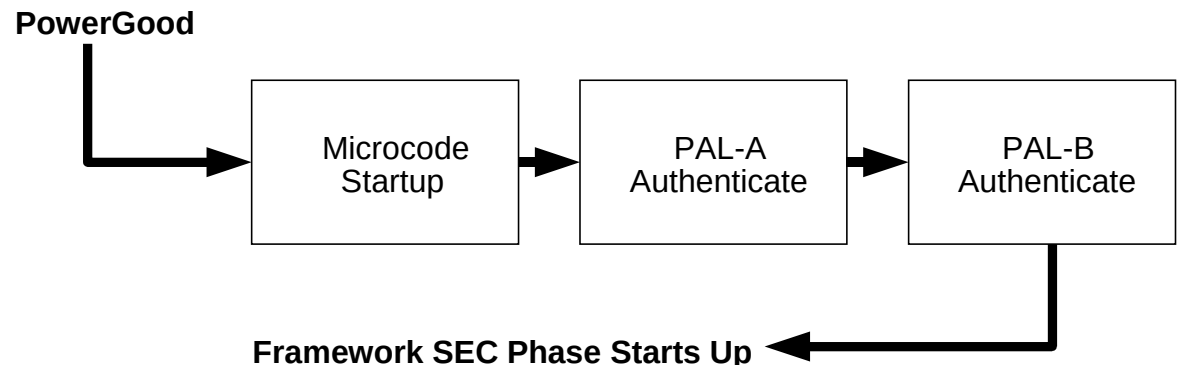

Figure 13-1. Itanium Architecture Resets

In Itanium architecture, the microcode starts up the first layer of the PAL code (which is provided by the processor vendor) that resides in the Boot Firmware Volume (BFV). This code minimally initializes the processor and then finds and authenticates the second layer of PAL code (called PAL-B). The location of both PAL-A and PAL-B can be found by consulting either the architected pointers in the ROM (near the 4 GB region) or by consulting the Firmware Interface Table (FIT) pointer in the ROM. The PAL layer communicates with the OEM boot firmware using a single entry point called SALE_ENTRY.

The Itanium architecture defines the initialization described above. In addition, however, Itanium-based systems that use the Framework architecture must do the following:

- **A "special" PEIM must be resident in the BFV to provide information about the location of the other firmware volumes.**

  The PEI Foundation will be located at the SALE_ENTRY point on the BFV. The Itanium architecture PEIMs may reside in the BFV or other firmware volumes, but a "special" PEIM must be resident in the BFV to provide information about the location of the other firmware volumes.
- **The BFV of a particular node must be accessible by all the processors running in that node.**

  All the processors in each node start up and execute the PAL code and subsequently enter the PEI Foundation. The BFV of a particular node must be accessible by all the processors running in that node. This distinction also means that some of the PEIMs in the Itanium architecture boot path will be MP aware.
- **Firmware modules in a BFV must be organized such that PAL-A, PAL-B, and FIT binaries are always visible to all the processors in a node at the time of power-on.**

  These binaries must be visible without any initialization of the system fabric.

Appendix A
Glossary

ACPI	Advanced Configuration and Power Interface. See http://www.acpi.info/ to view the specification.
AL	Afterlife phase. Power down phase. Refer to Chapter 7.
AML	ACPI Machine Language.
API	Application Program Interface. Programmatic interfaces for the firmware (as opposed to Win32-type OS-level APIs).
APM	Advanced Power Management. See http://www.microsoft.com/hwdev/archive/BUSBIOS/amp_12.asp to view the specification.
a priori file	A file with a known GUID that contains the list of DXE drivers that are loaded and executed in the listed order before any other DXE drivers are discovered.
ASCII	American Standard Code for Information Interchange.
ASL	ACPI Source Language.
BA	Boot Authorization.
BBS	BIOS Boot Specification.
BDS	Boot Device Selection phase. Refer to Chapter 5.
BFV	Boot Firmware Volume. Code (i.e., PEI and PEIM code) that appears in the memory address space of the system without prior firmware intervention. *See also* FV.
BIOS	Basic Input/Output System.
BIS	Boot Integrity Services.
BIST	Built-in self test.
BLT	Block Transfer (pronounced “blit” as in “slit” or “flit”). A series of functions that form the basis of manipulation graphical data. The operation used to draw a rectangle of pixels on the screen.
BNF	Backus-Naur Form. A metasyntactic notation used to specify the syntax of programming languages, command sets, and the like.
boot device	The device handle that corresponds to the device from which the currently executing image was loaded.

boot manager — The part of the firmware implementation that is responsible for implementing system boot policy. Although a particular boot manager implementation is not specified in this document, such code is generally expected to be able to enumerate and handle transfers of control to the available OS loaders as well as EFI applications and drivers on a given system. The boot manager would typically be responsible for interacting with the system user, where applicable, to determine what to load during system startup. In cases where user interaction is not indicated, the boot manager would determine what to load and, if multiple items are to be loaded, what the sequencing of such loads would be.

Boot Services — The collection of interfaces and protocols that are present in the boot environment. The services minimally provide an OS loader with access to platform capabilities required to complete OS boot. Services are also available to drivers and applications that need access to platform capability. Boot services are terminated once the OS takes control of the platform.

CGI — Common Gateway Interface.

COFF — Common Object File Format. An (originally) Unix*-based file format that is now recognized under several OSs. The format uses one or more header fields followed by the section data for the file.

Compatibility16 — A traditional legacy BIOS with the POST and BIOS Setup code removed. Compatibility16 BIOS code executes in real mode.

compatibility BIOS — The combination of both EfiCompatibility and Compatibility16.

CompatibilitySmm — Any IBV-provided SMM code to perform traditional functions that are not provided by EFI.

CRC — Cyclic Redundancy Check. A fixed-size error checking code appended to the end of a block of data (file) that is based on the content of the file.

CRTM — Core Root-of-Trust Module.

CSM — Compatibility Support Module. The combination of EfiCompatibility, CompatibilitySmm, and Compatibility16.

CVDR — Configuration Values Driven through Reset.

depex — Dependency expression. Code associated with each driver that describes the dependencies that must be satisfied in order for that driver to run.

dispatch entry point — The entry point that the dispatcher invokes.

driver — Modular chunk of firmware code that supports chipset or platform features. Reusable in multiple system contexts.

DSL — Digital Subscriber Line.

DXE — Driver Execution Environment phase. Refer to Chapter 4.

DXE Foundation	A set of intrinsic services and an execution mechanism for sequenced control of driver modules.
DXE Services	Services, such as security services and driver services, that are usable by DXE drivers.
EBCDIC	Extended Binary Coded Decimal Interchange Code.
EfiCompatibility	EFI code that corresponds to EFI compatibility drivers, code that generates data for compatibility interfaces, or code that invokes compatibility services.
ESCD	Extended System Configuration Data.
EXE	Executable file format. An (originally) MS DOS-based file format that consists of three different parts, the header, the relocation table, and the binary code.
FAT	File Allocation Table.
FD	Firmware Device. A persistent physical repository that contains firmware code and/or data and that may provide NVS. For the purposes of this architecture specification, the topology of FDs should be abstracted via FVs.
FFS	Firmware File System. A binary storage format that is well suited to firmware volumes. The abstracted model of the FFS is a flat file system.
firmware device	*See* FD.
firmware volume	*See* FV.
FIT	Firmware Interface Table.
font	A translation between Unicode weights and glyphs. This “M” and this “M” and this “M” represent the same weight but in different fonts.
FPSWA	Floating Point Software Assist.
Framework	Intel® Platform Innovation Framework for EFI.
FS	Firmware Store. The abstracted model of the FS is a flat “file system” where individual files are SUMs.
FV	Firmware volume. There are one or more FVs in the FS. The FV containing the “reset vector” is known as the Boot Firmware Volume (BFV).
GCD	Global coherency domain. The address resources of a system as seen by a processor. It consists of both system memory and I/O space.
glyph	The graphical representation of a single Unicode weight.
GUID	Globally Unique Identifier. A 128-bit value used to differentiate services and structures in the boot services environment.
HII	Human Interface Infrastructure. Repository of configuration and translation information for localization.
HOB	Hand-Off Block. A structure used to pass information from one boot phase to another (i.e., from the PEI phase to the DXE phase).

HTML	Hypertext Markup Language.
IA-32	Intel processors based 32-bit Intel architecture.
IBV	Independent BIOS vendor.
IFR	Internal Forms Representation. A binary encoding of forms-based display content and configuration information.
IME	Input Method Editor.
internationalization	Concepts by which an interface is made useful to users speaking different languages and from various cultures by the use of accepted neutral symbols: a red octagon means “stop.” A mouse is international because the actions are the same no matter the language or culture. Certain keys on the keyboard (space, shift, cursor controls, functions, numbers, etc.) are international as well.
intrinsic services	Services, such as security services and driver services, that remain available after the phase during which they are instantiated.
IPL	Initial Program Load. An architectural PPI that starts the DXE phase.
IPMI	Intelligent Platform Management Interface.
ISDN	Integrated Services Digital Network.
ISO 3166	An association between a country or region and a two or three character ASCII string.
ISO 639-2	An association between a language or dialect and a three character ASCII string.
IT	Information Technology.
localization	Concepts by which an interface is made useful to users speaking different languages and from various cultures by adapting the interfaces to the user. “STOP” in English would be “ALTO” in Spanish and “СТОП” in Russian. Alphabetic on keyboards are local to the language and may be local to the country the keyboard is localized for. For example, a French keyboard in France is different from a French keyboard in Canada.
MCA	Machine Check Architecture.
NMI	Nonmaskable interrupt.
NVRAM	Nonvolatile Random Access Memory.
NVS	Nonvolatile storage. Flash, EPROM, ROM, or other persistent store that will not go away once system power is removed.
OEM	Original equipment manufacturer.
OpROM	Option ROM.
PAL	Processor Abstraction Layer. A binary distributed by Intel that is used by the 64-bit Itanium processor family.

PCI	Peripheral Component Interconnect. See http://www.pcisig.com/ for more information.
PCR	Platform Configuration Register.
PE/COFF	PE32, PE32+, or Common Object File Format. A defined standard file format for binary images.
PEI	Pre-EFI Initialization phase. Refer to Chapter 3.
PEI Foundation	A set of intrinsic services and an execution mechanism for sequenced control of PEIMs.
PEIM	Pre-EFI Initialization Module. Modular chunk of firmware code running in PEI that supports chipset or platform features. Reusable in multiple system contexts.
PEI Services	Common services that are usable by PEIMs.
PHIT	Phase Handoff Information Table. A HOB that describes the physical memory used by the PEI phase and the boot mode discovered during the PEI phase.
PIC	Position-independent code. Code that can be executed at any address without relocation.
POST	Power On Self Test.
PPI	PEIM-to-PEIM Interface.
RAM	Random Access Memory.
reverse thunk	The code to transition from 16-bit real mode to native execution mode.
ROM	Read-Only Memory.
root of trust	The base executable image of the system that has integrity and is authorized (by the system owner) to run on the platform.
RSD_PTR	Root System Description Pointer.
RT	Runtime phase. Refer to Chapter 6.
Runtime Services	Interfaces that provide access to underlying platform-specific hardware that may be useful during OS runtime, such as time and date services. These services become active during the boot process but also persist after the OS loader terminates boot services.
SAL	System Abstraction Layer.
SALE_ENTRY	System Abstraction Layer entry point.
sandbox	The common properties of a driver or preboot environment that allow applications to run. These properties include a defined load image format and services that can run in the sandbox.
SIMD	Single Instruction, Multiple Data.
SMI	System Management Interrupt.

SMM	System Management Mode.
SOR	Schedule on Request.
SRAM	Static Random Access Memory.
SSE	Streaming SIMD Extensions.
SUM	Separately Updateable Module. A portion of the BFV that is treated as a separate module that can be updated without affecting the other SUMs in the BFV.
TCB	Trusted Computing Base.
TCG	Trusted Computing Group.
TE image	Terse Executable image. An executable image format that is specific to the Framework. This format is used only in PEI and is used for storing executable images in a smaller amount of space than would be required by a full PE32+ image.
thunk	The code to transition from native execution mode to 16-bit real mode.
UNDI	Universal Network Driver Interface.
Unicode	A standard defining an association between numeric values known as “weights” and characters from the majority of the worlds currently used languages. See the Unicode specification for more information.
USB	Universal Serial Bus. See http://www.usb.org/ for more information.
VFR	Visual Forms Representation. A high-level language representation of IFR.
VM	Virtual Machine.
VT-100	A terminal and serial protocol originally defined by Digital Equipment Corporation. Limited to 7-bit ASCII.
VTF	Volume Top File. A file in a firmware volume that must be located such that the last byte of the file is also the last byte of the firmware volume.
VT-UTF8	A serial protocol definition that extends VT-100 to support Unicode.
watchdog timer	An alarm timer that may be set to go off. This can be used to regain control in cases where a code path in the boot services environment fails to or is unable to return control by the expected path.
XIP	Execute In Place. PEI code that is executed from its storage location in a firmware volume.
XML	Extensible Markup Language.

Appendix B
References

The following publications and sources of information may be useful to you or are referred to by this specification:

- ACPI / Power Management. Microsoft Corporation. Windows Platform Development Web site. http://www.microsoft.com/whdc/hwdev/tech/onnow/default.mspx
- *Advanced Configuration and Power Interface Specification.* Rev. 2.0. 2000. http://www.acpi.info/
- *BIOS Boot Specification.* Ver. 1.01. Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation. 1996. http://www.phoenix.com/en/customer+services/white+papers-specs/
- Data Encryption Standard. National Institute of Standards and Technology (NIST). FIPS Publication 46-1: 22 January 1988. Originally issued by National Bureau of Standards.
- *"El Torito" Bootable CD-ROM Format Specification.* Ver. 1.0. Phoenix Technologies, Ltd., IBM Corporation. 1995. http://www.phoenix.com/en/customer+services/white+papers-specs/
- *Hardware Design Guide Version 3.0 for Microsoft Windows 2000 Server*. Intel Corporation, Microsoft Corporation. 2000. http://www.intel.com/design/servers/desguide/hdgv3.htm
- *Itanium® Processor Family System Abstraction Layer Specification.* Intel Corporation. 2002. http://developer.intel.com/design/itanium/family/
- *Itanium® Software Conventions and Runtime Architecture Guide.* Intel Corporation. 2001. http://developer.intel.com/design/itanium/family/
- Intelligent Platform Management Interface (IPMI): http://www.intel.com/design/servers/ipmi/index.htm
- ISO 639-2 specification. *Codes for the representation of names of languages.* http://www.iso.ch/
- ISO 3166 specification. *Codes for the representation of names of countries and their subdivisions.* http://www.iso.ch/
- *Microsoft Portable Executable and Common Object File Format Specification.* Rev. 6.0. Microsoft Corporation. 1999. http://www.microsoft.com/hwdev/hardware/PECOFF.asp
- *PCI BIOS Specification.* Rev. 2.1. PCI Special Interest Group. http://www.pcisig.com/specifications
- PCI Special Interest Group (SIG): http://www.pcisig.com/
- *PKCS #7: Cryptographic Message Syntax Standard.* Ver. 1.5. RSA Laboratories. November 1993.
- *Preboot Execution Environment (PXE) Specification.* Ver. 2.1. Intel Corporation. 1999. http://www.intel.com/labs/manage/wfm/wfmspecs.htm.

- *System Management BIOS Reference Specification.* Ver. 2.3. American Megatrends Inc., Award Software International Inc., Compaq Computer Corporation, Dell Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, Phoenix Technologies Limited, and SystemSoft Corporation. 1998. http://www.dmtf.org/
- *Trusted Computing Group (TCG) Main Specification.* Ver. 1.1a. 2001. http://www.trustedcomputinggroup.org/
- *The Unicode Standard.* Ver. 4.0.0. The Unicode Consortium. 2003. http://www.unicode.org/
- Universal Serial Bus (USB) Implementers Forum: http://www.usb.org/
- *Universal Serial Bus PC Legacy Compatibility Specification.* Ver. 0.9. USB Implementers Forum. 1996. http://www.usb.org/
- *Wired for Management Baseline.* Ver. 2.0. Intel Corporation. 1998. http://www.intel.com/labs/manage/wfm/wfmspecs.htm

Index