



# Generic IP independent BIOS Signing and Parsing

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Technology

in

Computer Science & Engineering

with specialization in Computer Science & Engineering

by

Gahan Saraiya

18MCEC10



Department of Computer Science & Engineering,

Institute of Technology,

Nirma University, Ahmedabad,

Gujarat - 382481, India.

December, 2019

---



# Declaration

I hereby declare that the dissertation ***Generic IP independent BIOS Signing and Parsing*** submitted by me to the Institute of Technology, Nirma University, Ahmedabad, 382481 in partial fulfillment of the requirements for the award of **Master of Technology in Computer Science & Engineering with specialization in Computer Science & Engineering** is a bona-fide record of the work carried out by me under the supervision of ***Prof. Dvijesh Bhatt***.

I further declare that the work reported in this dissertation, has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or University.

Sign:

Name & Roll. No.:

Date:



## Computer Science & Engineering

### Certificate

This is to certify that the dissertation entitled ***Generic IP independent BIOS Signing and Parsing*** submitted by ***Gahan Saraiya*** (Roll No. 18MCEC10) to Nirma University Ahmedabad, in partial fulfillment of the requirement for the award of the degree of **Master of Technology in Computer Science & Engineering with specialization in Computer Science & Engineering** is a bona-fide work carried out under my supervision. The dissertation fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this dissertation have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

Prof. Dvijesh Bhatt  
Guide & Assistant Professor,  
CSE Department,  
Institute of Technology,  
Nirma University, Ahmedabad.

Dr. Priyanka Sharma  
Professor,  
Coordinator M.Tech - CSE (CSE)  
Institute of Technology,  
Nirma University, Ahmedabad

Dr. Madhuri Bhavsar  
Professor and Head,  
CSE Department,  
Institute of Technology,  
Nirma University, Ahmedabad.

Dr. Alka Mahajan  
Director,  
Institute of Technology,  
Nirma University, Ahmedabad

# *Abstract*

Here goes the BOOM... aka abstract..

# *Acknowledgements*

is this page needed ??????

# Contents

|   |            |
|---|------------|
| <b>Declaration</b>  | <b>i</b>   |
| <b>Certificate</b>  | <b>ii</b>  |
| <b>Abstract</b>   | <b>iii</b> |
| <b>Acknowledgements</b>   | <b>iv</b>  |
| <b>List of Figures</b>  | <b>vii</b> |
| <b>List of Figures</b>  | <b>1</b>   |
| 1. Introduction . . . . .                                       | 2          |
| 1.1 Uncore Intellectual Properties . . . . .                    | 2          |
| 1.2 Legacy BIOS and UEFI . . . . .                              | 2          |
| BIOS . . . . .  | 2          |
| 1.2.1 Background of Legacy BIOS . . . . .                       | 2          |
| 1.2.2 Limitations of legacy BIOS . . . . .                      | 3          |
| 1.3 Unified Extensible Firmware Interface (UEFI) . . . . .      | 3          |
| 1.3.1 UEFI Driver Model Extension . . . . .                     | 4          |
| 1.3.2 UEFI's Role in boot process . . . . .                     | 5          |
| 1.4 Comparing of Legacy BIOS and UEFI . . . . .                 | 5          |
| 1.5 Advanced Configuration and Power Interface (ACPI) . . . . . | 5          |
| 1.5.1 Overview of ACPICA Subsystem . . . . .                    | 7          |
| 1.5.2 OS-independent ACPICA Subsystem . . . . .                 | 8          |
| 1.5.3 Operating System Services Layer . . . . .                 | 8          |
| 1.5.4 ACPICA Subsystem Interaction . . . . .                    | 9          |
| 2. Design . . . . .   | 10         |
| 2.1 UEFI Design Overview . . . . .                              | 10         |
| 2.1.1 UEFI Driver Goals . . . . .                               | 11         |
| 2.2 UEFI/PI Firmware Images . . . . .                           | 12         |
| 2.3 Platform Initialization PI Boot Sequence . . . . .          | 14         |
| 2.4 Security (SEC) . . . . .                                    | 14         |
| 2.5 Pre-EFI Initialization (PEI) . . . . .                      | 15         |

---

|        |  |    |
|--------|--|----|
| 2.5.1  | PEI Services . . . . .                                 | 16 |
| 2.5.2  | PEI Foundation . . . . .                               | 17 |
| 2.6    | PEI Dispatcher . . . . .                               | 18 |
| 2.7    | Drive Execution Environment (DXE) . . . . .            | 19 |
| 2.8    | Boot Device Selection (BDS) . . . . .                  | 19 |
| 2.9    | Transient System Load (TSL) and Runtime (RT) . . . . . | 19 |
| 2.10   | After Life (AL) . . . . .                              | 20 |
| 2.11   | Generic Build Process . . . . .                        | 20 |
| 2.11.1 | EFI Section Files . . . . .                            | 20 |
| 3.     | Architecture of BIOS Firmware . . . . .                | 22 |
| 3.1    | Overview . . . . .                                     | 22 |
| 3.2    | Design of Firmware Storage . . . . .                   | 22 |
|        | Firmware Device . . . . .                              | 22 |
|        | Flash . . . . .  | 22 |
| 3.3    | Firmware Volumes (FV) . . . . .                        | 23 |
| 3.4    | Firmware File System (FFS) . . . . .                   | 23 |
| 3.4.1  | Firmware File Types . . . . .                          | 24 |
| 3.5    | Firmware File Sections . . . . .                       | 24 |

---



# List of Figures

|    |  |    |
|----|--|----|
| 1  | Board of Directors of UEFI Forum . . . . .                                       | 4  |
| 2  | The ACPI Component Architecture . . . . .  | 7  |
| 3  | ACPICA Subsystem Architecture . . . . .  | 9  |
| 4  | Interaction between the Architectural Components . . . . .                       | 9  |
| 5  | UEFI Conceptual Overview . . . . .   | 10 |
| 6  | UEFI/PI Firmware Image Creation . . . . .  | 13 |
| 7  | UEFI/PI Firmware Image Creation . . . . .  | 14 |
| 8  | PI Boot Phases . . . . .   | 15 |
| 9  | Diagram of PI Operations . . . . .   | 17 |
| 10 | General EFI Section Format for large size Sections(greater than 16 MB) . . . . . | 21 |
| 11 | General EFI Section Format (less than 16 MB) . . . . .                           | 21 |
| 12 | Example File System Image . . . . .  | 25 |

architecture

---

# 1. Introduction

## 1.1 Uncore Intellectual Properties

Intel System on a Chip (SoC) features a new set of Intel Uncore Intellectual Property (IP) for every generation. The Uncore encompasses system agent (SA), memory and Uncore agents such as graphics controller, display controller, memory controller and Input Output (IO). The Uncore IPs are Peripheral Component Interface Express (PCIe), Graphics Processing Engine (GPE), Thunderbolt, Imaging Processing Agent (IPU), North Peak (NPK), Virtualization Technology for directed-IO (Vt-d), Volume Management Device (VMD).

PCI Express abbreviated as PCIe or PCI-E, is designed to replace the older PCI standards. A data communication system is developed for use the transfer data between the host and the peripheral devices via PCIe. Thunderbolt is the brand name of a hardware interface developed by Intel that allows the connection of external peripherals to a computer. Thunderbolt combines PCI Express (PCIe) and DisplayPort (DP) into two serial signals, and additionally provides DC power, all in one cable. Graphics Processing Engine (GPE), Integrated graphics, shared graphics solutions, integrated graphics processors (IGP) or unified memory architecture (UMA) utilize a portion of a computer's system RAM rather than dedicated graphics memory. GPEs can be integrated onto the motherboard as part of the chipset. Virtual Technology for Directed-IO (Vt-d) is an input/output memory management unit (IOMMU) allows guest virtual machines to directly use peripheral devices, such as Ethernet, accelerated graphics cards, and hard-drive controllers, through DMA and interrupt remapping.

## 1.2 Legacy BIOS and UEFI

**BIOS** is the dominant standard which defines a firmware interface.

"Legacy" (as in Legacy BIOS), in the context of firmware specifications, refer to an older, widely used specification. Major responsibility of BIOS is to set up the hardware, load and start an Operating System (OS). When the system boots, the BIOS initializes and identifies system devices including video display card, mouse, hard disk drive, keyboard, solid state drive and other hardware followed by locating software held on a boot device i.e. a hard disk or removable storage such as CD/DVD or USB and loads and executes that software, giving it control of the computer. This process is also referred to as "booting" or "boot strapping".

### 1.2.1 Background of Legacy BIOS

In 1980s, IBM developed the personal computer with a 16-bit BIOS with the aim of ending the BIOS after the first 250,000 products. Legacy BIOS is based upon

---

Intel's original 16-bit architecture, ordinarily referred to as "8086" architecture. And as technology advanced, Intel extended that 8086 architecture from 16 to 32-bit. Legacy BIOS is able to run different Operating System (OS), such as MS-DOS, equally well on systems other than IBM. Additionally, Legacy BIOS has a defined OS-independent interface for hardware that enables interrupts to communicate with video, disk and keyboard services along with the BIOS ROM loader and bootstrap loader, to name a few.

Use of legacy BIOS is diminishing and is expected to be phased out in new systems by the year 2020.

### **1.2.2 Limitations of legacy BIOS**

Over the years, many new configuration and power management technologies were integrated into BIOS implementations as well as support for many generations of Intel® architecture hardware. However certain limitations of BIOS implementations such as 16-bit addressing mode, 1 MB addressable space, PC AT hardware dependencies and upper memory block (UMB) dependencies persisted throughout the years. The industry also began to have need for methods to ensure quality of individual firmware modules as well as the ability to quickly integrate libraries of third-party firmware modules into a single platform solution across multiple product lines. These inherent limitations and existing market demands opened the opportunity for a fresh BIOS architecture to be developed and introduced to the market. The UEFI specifications and resulting implementations have begun to effectively address these persisting market needs.

One of the critical maintenance challenges for BIOS is that each implementation has tended to be highly customized for the specific motherboard on which it is deployed. Moving component modules across designs typically requires significant porting, integration, testing and debug work. This is one of the markets challenges the UEFI architecture promises to address.

## **1.3 Unified Extensible Firmware Interface (UEFI)**

UEFI was developed as a replacement for legacy BIOS to streamline the booting process, and act as the interface between a operating system and its platform firmware. It not only replaces most BIOS functions, but also offers a rich extensible pre-OS environment with advanced boot and runtime services. Unified Extensible Firmware Interface (UEFI) is grounded in Intel's initial Extensible Firmware Interface (EFI) specification 1.10, which defines a software interface between an operating system and platform firmware. The UEFI architecture allows users to execute applications on a command line interface. It has intrinsic networking capabilities and is designed to work with multi-processors (MP) systems.

---














|   |  |  |   |
|---|--|--|---|
| <b>MARK DORAN</b><br>President<br>Intel<br>              | <b>DONG WEI</b><br>Vice President<br>ARM<br>          | <b>JEFF BOBZIN</b><br>Secretary<br>Insyde Software<br> | <b>BILL KEOWN</b><br>Treasurer<br>Lenovo<br> |
| <b>WILLIAM MOYES</b><br>Advanced Micro Devices, Inc.<br> | <b>STEFANO RIGHI</b><br>American Megatrends, Inc.<br> | <b>ANDREW FISH</b><br>Apple<br>                         | <b>ANAND JOSHI</b><br>Dell<br>               |
| <b>KEVIN DEPEW</b><br>Hewlett Packard Enterprise<br>     | <b>RICK BRAMLEY</b><br>HP, Inc.<br>                   | <b>JEREMY KERR</b><br>IBM<br>                           | <b>TOBY NIXON</b><br>Microsoft<br>           |
| <b>DICK WILKINS</b><br>Phoenix Technologies<br>          |  |  |   |

FIGURE 1: Board of Directors of UEFI Forum

The UEFI Forum board of directors consists of representatives from 11 industry leaders as described in Figure 1. These involved organizations work to ensure that the UEFI specifications meet industry needs.

UEFI uses a different interface for boot services and runtime services but UEFI does not specify how "Power On Self Test" (POST) and Setup are implemented - those are BIOS' primary functions.

### 1.3.1 UEFI Driver Model Extension

Access to boot devices is provided through a set of protocol interfaces. One purpose of the UEFI Driver Model is to provide a replacement for PC-AT-style option ROMs. It is important to point out that drivers written to the UEFI Driver Model are designed to access boot devices in the pre-boot environment. They are not designed to replace the high-performance, OS-specific drivers.

The UEFI Driver Model is designed to support the execution of modular pieces of code, also known as drivers, that run in the pre-boot environment. These drivers may manage or control hardware buses and devices on the platform, or they may provide some software-derived, platform specific service. The UEFI Driver Model also contains information required by UEFI driver writers to design and implement any combination of bus drivers and device drivers that a platform might need to boot a UEFI-compliant OS.

The UEFI Driver Model is designed to be generic and can be adapted to any type of bus or device. The UEFI Specification describes how to write PCI bus drivers, PCI device drivers, USB bus drivers, USB device drivers, and SCSI drivers. Additional details are provided that allow UEFI drivers to be stored in PCI option ROMs, while maintaining compatibility with legacy option ROM images.

One of the design goals in the UEFI Specification is keeping the driver images as small as possible. However, if a driver is required to support multiple processor architectures, a driver object file would also be required to be shipped for each supported processor architecture. To address this space issue, this specification also defines the EFI Byte Code Virtual Machine. A UEFI driver can be compiled into a single EFI Byte Code object file. UEFI Specification compliant firmware must contain an EFI Byte Code interpreter. This allows a single EFI Byte Code object file that supports multiple processor architectures to be shipped. Another space saving technique is the use of compression. This specification defines compression and decompression algorithms that may be used to reduce the size of UEFI Drivers, and thus reduce the overhead when UEFI Drivers are stored in ROM devices.

The information contained in the UEFI Specification can be used by OSVs, IHVs, OEMs, and firmware vendors to design and implement firmware conforming to this specification, drivers that produce standard protocol interfaces, and operating system loaders that can be used to boot UEFI compliant operating systems.

### **1.3.2 UEFI's Role in boot process**

During the boot process, UEFI speaks to the operating system loader and acts as the interface between the operating system and the BIOS.

The PC-AT boot environment presents significant challenges to innovation within the industry. Each new platform capability or hardware innovation requires firmware developers to craft increasingly complex solutions, and often requires OS developers to make changes to their boot code before customers can benefit from the innovation. This can be a time-consuming process requiring a significant investment of resources. The primary goal of the UEFI specification is to define an alternative boot environment that can alleviate some of these considerations. In this goal, the specification is like other existing boot specifications.

## **1.4 Comparing of Legacy BIOS and UEFI**

### **1.5 Advanced Configuration and Power Interface (ACPI)**

The ACPI Component Architecture (ACPIA) defines and implements a group of software components that together create an implementation of the ACPI specification. A major goal of the architecture is to isolate all operating system dependencies to a relatively small translation or conversion layer (the OS Services

---

TABLE 1: Legacy BIOS v/s UEFI

|                                 | Legacy BIOS   | EFI                                      |
|---------------------------------|---|--|
| Language                        | Assembly  | C (99%)                                  |
| Resource                        | Interrupt Hardcode Memory Access<br>hardcore I/O Access | Diver, Protocols                         |
| Processor                       | x86 16-bit  | CPU Protects Mode (Flat Mode)            |
| Expand                          | Hook Interrupt  | Load Driver                              |
| OS Bridge                       | ACPI  | Run Time Driver Software                 |
| 3 <sup>rd</sup> Party ISV & IHV | Bas for Support   | Easy for Support and for Multi Platforms |

Layer) so that the bulk of the ACPICA code is independent of any individual operating system. Therefore, hosting the ACPICA code on new operating systems requires no source changes within the ACPICA code itself.

The components of the architecture include:

- An OS-independent, kernel-resident ACPICA Subsystem component that provides the fundamental ACPI services such as the AML interpreter and namespace management.
- An OS-dependent OS Services Layer for each host operating system to provide OS support for the OS-independent ACPICA Subsystem.
- An ASL compiler-disassembler for translating ASL code to AML byte code and for disassembling existing binary ACPI tables back to ASL source code.
- Several ACPI utilities for executing the interpreter in ring 3 user space, extracting binary ACPI tables from the output of the ACPI Dump utility, and translating the ACPICA source code to Linux/Unix format.

In Figure 2, the ACPICA subsystem is shown in relation to the host operating system, device driver, OSPM software, and the ACPI hardware

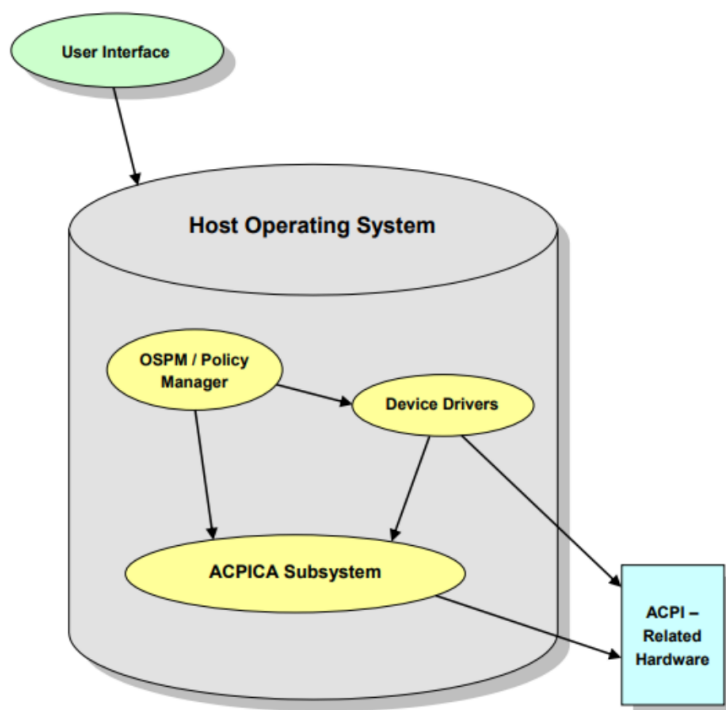


FIGURE 2: The ACPI Component Architecture

### 1.5.1 Overview of ACPICA Subsystem

The ACPICA Subsystem implements the low level or fundamental aspects of the ACPI specification. Included are an AML parser/interpreter, ACPI namespace management, ACPI table and device support, and event handling. Since the ACPICA subsystem provides low-level system services, it also requires low-level operating system services such as memory management, synchronization, scheduling, and I/O.

To allow the ACPICA Subsystem to easily interface to any operating system that provides such services, an Operating System Services Layer translates ACPICA-to-OS requests into the system calls provided by the host operating system. The OS Services Layer is the only component of the ACPICA that contains code that is specific to a host operating system.

Thus, the ACPICA Subsystem consists of two major software components:

- The basic kernel-resident ACPICA Subsystem provides the fundamental ACPI services that are independent of any particular operating system.
- The OS Services Layer (OSL) provides the conversion layer that interfaces the OS independent ACPICA Subsystem to a host operating system.

When combined into a single static or loadable software module such as a device driver or kernel subsystem, these two major components form the ACPICA



Subsystem. Throughout this document, the term "ACPICA Subsystem" refers to the combination of the OS-independent ACPICA Subsystem with an OS Services Layer components combined into a single module, driver, or load unit.

### **1.5.2 OS-independent ACPICA Subsystem**

The OS-independent ACPICA Subsystem supplies the major building blocks or subcomponents that are required for all ACPI implementations — including an AML interpreter, a namespace manager, ACPI event and resource management, and ACPI hardware support.

One of the goals of the ACPICA Subsystem is to provide an abstraction level high enough such that the host operating system does not need to understand or know about the very low-level ACPI details. For example, all AML code is hidden from the host. Also, the details of the ACPI hardware are abstracted to higher-level software interfaces.

The ACPICA Subsystem implementation makes no assumptions about the host operating system or environment. The only way it can request operating system services is via interfaces provided by the OS Services Layer.

The primary user of the services provided by the ACPICA Subsystem are the host OS device drivers and power/thermal management software.

### **1.5.3 Operating System Services Layer**

The OS Services Layer (or OSL) operates as a translation service for requests from the OS independent ACPICA subsystem back to the host OS. The OSL implements a generic set of OS service interfaces by using the primitives available from the host OS. Because of its nature.

The OS Services Layer must be implemented anew for each supported host operating system. There is a single OS-independent ACPICA Subsystem, but there must be an OS Services Layer for each operating system supported by the ACPI component architecture.

The primary function of the OSL in the ACPI Component Architecture is to be the small glue layer that binds the much larger ACPICA Subsystem to the host operating system. Because of the nature of ACPI itself — such as the requirement for an AML interpreter and management of a large namespace data structure — most of the implementation of the ACPI specification is independent of any operating system services. Therefore, the OS-independent ACPICA Subsystem is the larger of the two components.

The overall ACPI Component Architecture in relation to the host operating system is Figure

---

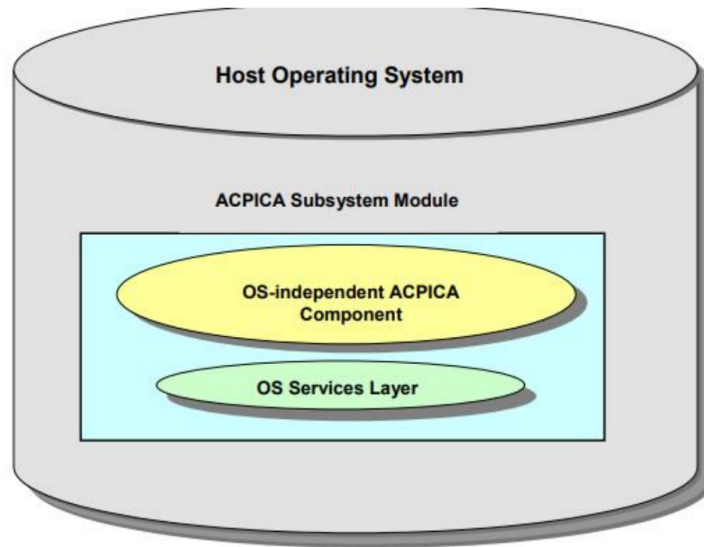


FIGURE 3: ACPICA Subsystem Architecture

#### 1.5.4 ACPICA Subsystem Interaction

The ACPICA Subsystem implements a set of external interfaces that can be directly called from the host OS. These Acpi\* interfaces provide the actual ACPI services for the host. When operating system services are required during the servicing of an ACPI request, the Subsystem makes requests to the host OS indirectly via the fixed AcpiOs\* interfaces. The diagram below illustrates the relationships and interaction between the various architectural elements by showing the flow of control between them. Note that the OS-independent ACPICA Subsystem never calls the host directly instead it makes calls to the AcpiOs \* interfaces in the OSL. This provides the ACPICA code with OS-independence.

The Interaction between the Architectural Components Is shown in Figure 4

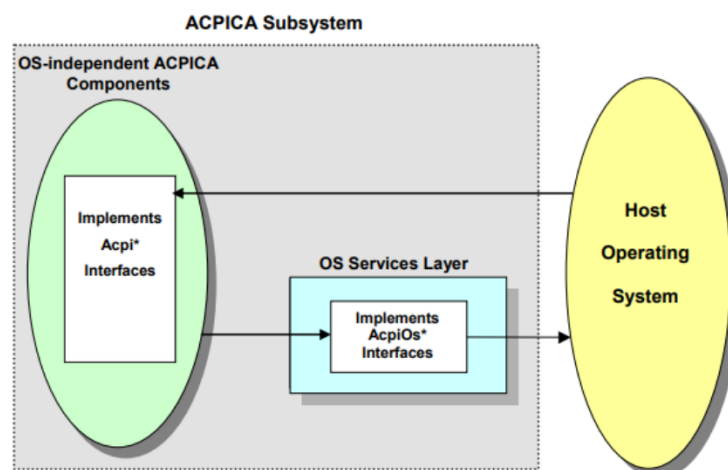


FIGURE 4: Interaction between the Architectural Components

## 2. Design

### 2.1 UEFI Design Overview

The design of UEFI is based on the following fundamental elements:

- **Reuse of existing table-based interfaces** - In order to preserve investment in existing infrastructure support code, both in the OS and firmware, a number of existing specifications that are commonly implemented on platforms compatible with supported processor specifications must be implemented on platforms wishing to comply with the UEFI specification.
- **System partition** defines a partition and file system that are designed to allow safe sharing between multiple vendors, and for different purposes. The ability to include a separate, shareable system partition presents an opportunity to increase platform value-add without significantly growing the need for nonvolatile platform memory
- **Boot services** provide interfaces for devices and system functionality that can be used during boot time. Device access is abstracted through "handles" and "protocols". This facilitates reuse of investment in existing BIOS code by keeping underlying implementation requirements out of the specification without burdening the consumer accessing the device.
- **Runtime services** - A minimal set of runtime services is presented to ensure appropriate abstraction of base platform hardware resources that may be needed by the OS during its normal operations.

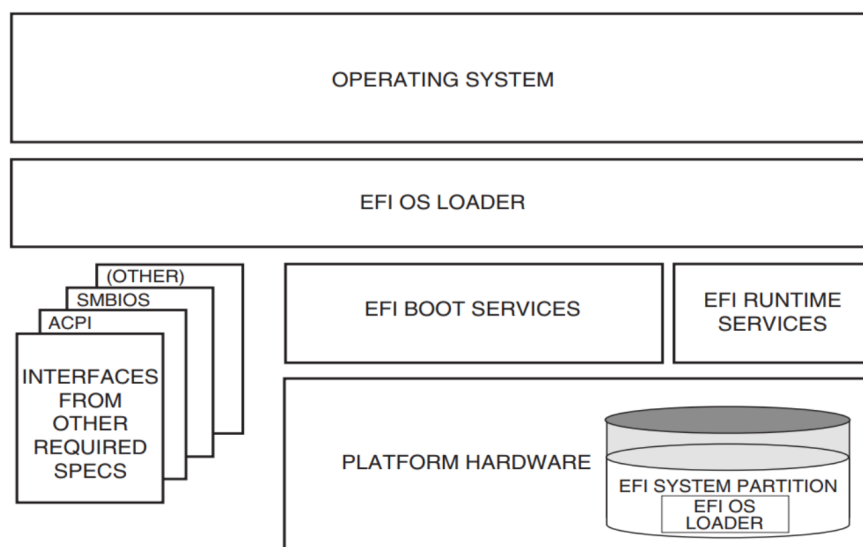


FIGURE 5: UEFI Conceptual Overview

**Error! Reference source not found** illustrates the interactions of the various components of an UEFI specification-compliant system that are used to accomplish platform and OS boot.

The platform firmware can retrieve the OS loader image from the System Partition. The specification provides for a variety of mass storage device types including disk, CD-ROM, and DVD as well as remote boot via a network. Through the extensible protocol interfaces, it is possible to add other boot media types, although these may require OS loader modifications if they require use of protocols other than those defined in this document.

Once started, the OS loader continues to boot the complete operating system. To do so, it may use the EFI boot services and interfaces defined by this or other required specifications to survey, comprehend, and initialize the various platform components and the OS software that manages them. EFI runtime services are also available to the OS loader during the boot phase.

### 2.1.1 UEFI Driver Goals

The UEFI Driver Model has the following goals:

- **Compatible** - Drivers conforming to this specification must maintain compatibility with the EFI and the UEFI Specification. This means that the UEFI Driver Model takes advantage of the extensibility mechanisms in the UEFI Specification to add the required functionality.
  - **Simple** - Drivers that conform to this specification must be simple to implement and simple to maintain. The UEFI Driver Model must allow a driver writer to concentrate on the specific device for which the driver is being developed. A driver should not be concerned with platform policy or platform management issues. These considerations should be left to the system firmware.
  - **Scalable** - The UEFI Driver Model must be able to adapt to all types of platforms. These platforms include embedded systems, mobile, and desktop systems, as well as workstations and servers.
  - **Flexible** - The UEFI Driver Model must support the ability to enumerate all the devices, or to enumerate only those devices required to boot the required OS. The minimum device enumeration provides support for more rapid boot capability, and the full device enumeration provides the ability to perform OS installations, system maintenance, or system diagnostics on any boot device present in the system.
  - **Extensible** - The UEFI Driver Model must be able to extend to future bus types as they are defined.
-

- **Portable** - Drivers written to the UEFI Driver Model must be portable between platforms and between supported processor architectures.
- **Interoperable** - Drivers must coexist with other drivers and system firmware and must do so without generating resource conflicts.
- **Describe complex bus hierarchies** - The UEFI Driver Model must be able to describe a variety of bus topologies from very simple single bus platforms to very complex platforms containing many buses of various types.
- **Small driver footprint** - The size of executables produced by the UEFI Driver Model must be minimized to reduce the overall platform cost. While flexibility and extensibility are goals, the additional overhead required to support these must be kept to a minimum to prevent the size of firmware components from becoming unmanageable.
- **Address legacy option rom issues** - The UEFI Driver Model must directly address and solve the constraints and limitations of legacy option ROMs. Specifically, it must be possible to build add-in cards that support both UEFI drivers and legacy option ROMs, where such cards can execute in both legacy BIOS systems and UEFI-conforming platforms, without modifications to the code carried on the card. The solution must provide an evolutionary path to migrate from legacy option ROMs driver to UEFI drivers.

## 2.2 UEFI/PI Firmware Images

UEFI and PI specifications define the standardized format for EFI firmware storage devices (FLASH or other non-volatile storage) which are abstracted into "Firmware Volumes". Build systems must be capable of processing files to create the file formats described by the UEFI and PI specifications. The tools provided as part of the EDK II BaseTools package process files compiled by third party tools, as well as text and Unicode files in order to create UEFI or PI compliant binary image files. In some instances, where UEFI or PI specifications do not have an applicable input file format, such as the Visual Forms Representation (VFR) files used to create PI compliant IFR content, tools and documentation have been provided that allows the user to write text files that are processed into formats specified by UEFI or PI specifications.

A Firmware Volume (FV) is a file level interface to firmware storage. Multiple FVs may be present in a single FLASH device, or a single FV may span multiple FLASH devices. An FV may be produced to support some other type of storage entirely, such as a disk partition or network device. For more information consult the Platform Initialization Specification, Volume 3. In all cases, an FV is formatted with a binary file system. The file system used is typically the Firmware File System (FFS), but other file systems may be possible in some cases. Hence, all modules are stored as "files" in the FV. Some modules may be "execute in place" (linked at a fixed address and executed from the ROM), while others are relocated when they are loaded into memory and some modules may be able to run

---

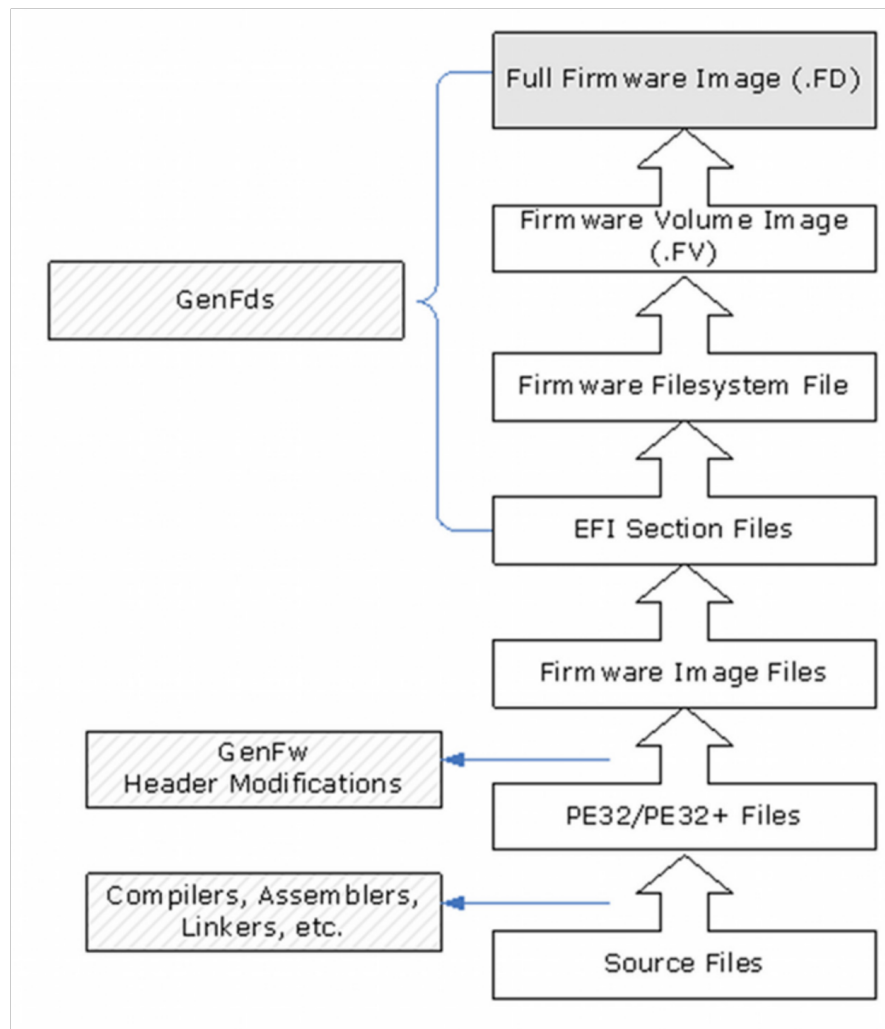


FIGURE 6: UEFI/PI Firmware Image Creation

from ROM if memory is not present (at the time of the module load) or run from memory if it is available. Files themselves have an internally defined binary format. This format allows for implementation of security, compression, signing, etc. Within this format, there are one or more "leaf" images. A leaf image could be, for example, a PE32 image for a DXE driver.

Therefore, there are several layers of organization to a full UEFI/PI firmware image. These layers are illustrated below in Figure 6. Each transition between layers implies a processing step that transforms or combines previously processed files into the next higher level. Also shown in Figure 6 are the reference implementation tools that process the files to move them between the different layers.

In addition to creating images that initialize a complete platform, the build process also supports creation of stand-alone UEFI applications (including OS Loaders) and Option ROM images containing driver code. Figure 7, below, shows the reference implementation tools and creation processes for both of these image types

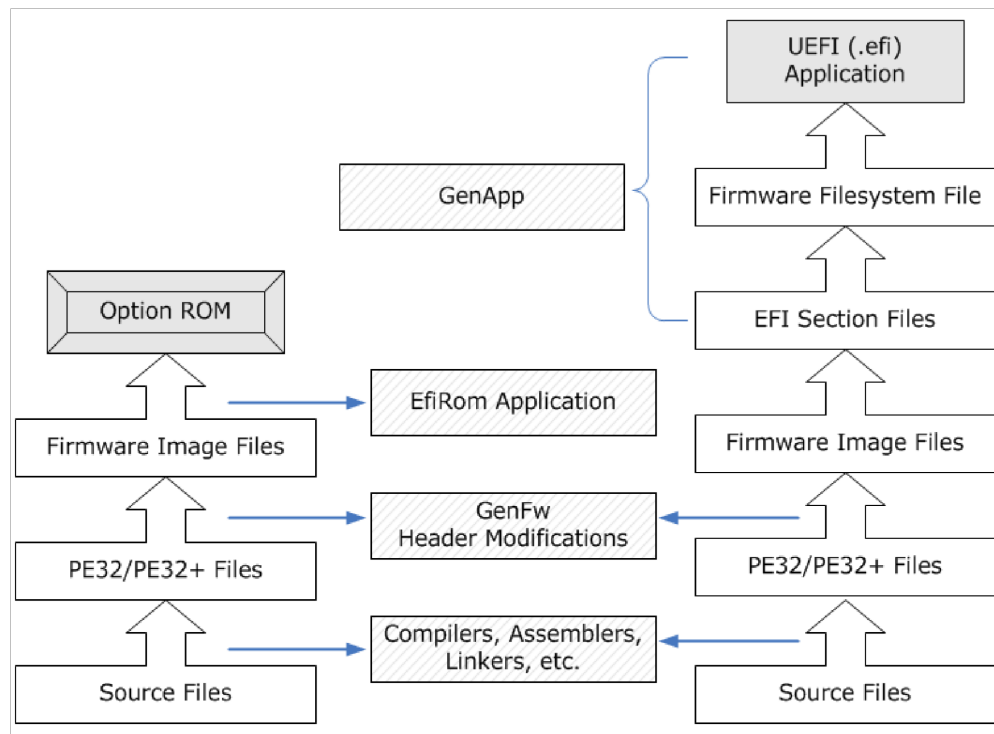


FIGURE 7: UEFI/PI Firmware Image Creation

The final feature that is supported by the EDK II build process is the creation of Binary Modules that can be packaged and distributed for use by other organizations. Binary modules do not require distribution of the source code. This will permit vendors to distribute UEFI images without having to release proprietary source code.

This packaging process permits creation of an archive file containing one or more binary files that are either Firmware Image files or higher (EFI Section files, Firmware File system files, etc.). The build process will permit inserting these binary files into the appropriate level in the build stages.

## 2.3 Platform Initialization PI Boot Sequence

PI compliant system firmware must support the six phases: security (SEC), pre-efi initialization (PEI), driver execution environment (DXE), boot device selection (BDS), run time (RT) services and After Life (transition from the OS back to the firmware) of system. Refer to Figure 8 below.

## 2.4 Security (SEC)

The Security (SEC) phase is the first phase in the PI Architecture and is responsible for the following:

# Platform Initialization (PI) Boot Phases

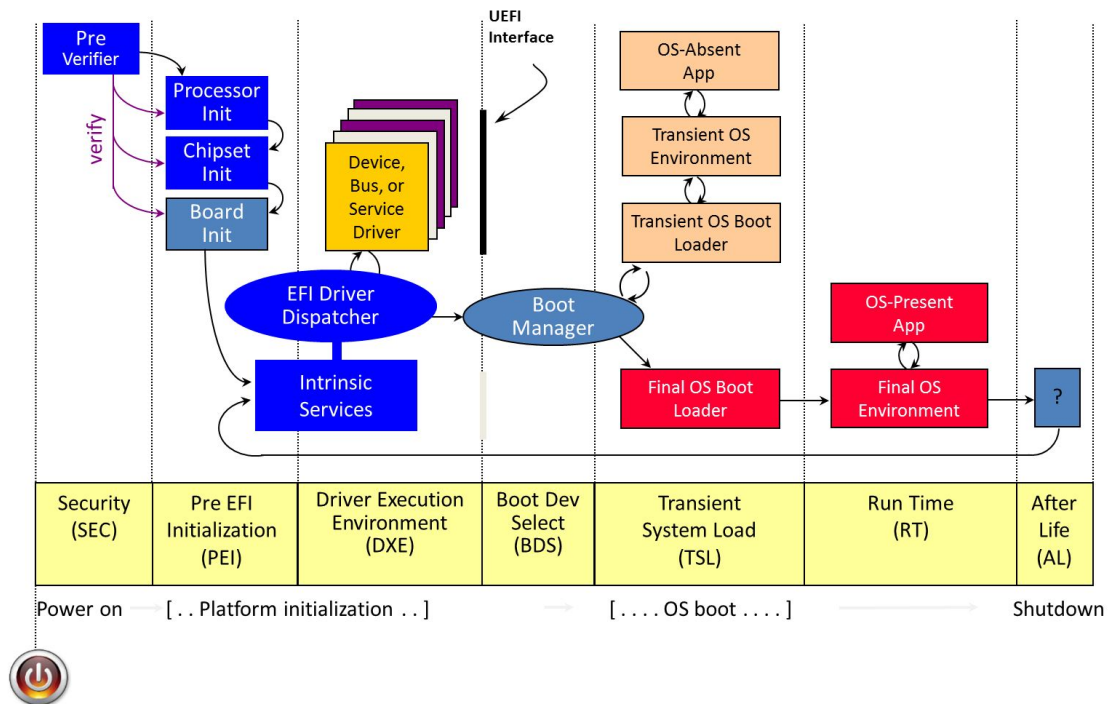


FIGURE 8: PI Boot Phases

- Handling all platform restart events
- Creating a temporary memory store
- Serving as the root of trust in the system
- Passing handoff information to the PEI Foundation

The security section may contain modules with code written in assembly. Therefore, some EDK II module development environment (MDE) modules may contain assembly code. Where this occurs, both Windows and GCC versions of assembly code are provided in different files

## 2.5 Pre-EFI Initialization (PEI)

The Pre-EFI Initialization (PEI) phase described in the PI Architecture specifications is invoked quite betimes in the boot period. Specifically, after about preliminary processing in the Security (SEC) phase, any machine restart event will invoke the PEI phase. The PEI phase is designed to be developed in many parts and consists of:

- PEI Foundation (core code)



- Pre-EFI Initialization Modules (specialized plug-ins)

The PEI phase initially operates with the platform in a nascent state, leveraging only on-processor resources, such as the processor cache as a call stack, to dispatch Pre-EFI Initialization Modules (PEIMs).

The PEI phase cannot assume the availability of amounts of memory (RAM) as DXE and hence PEI phase limits its support to the following:

- Locating and validating PEIMs
- Dispatching PEIMs
- Facilitating communication between PEIMs
- Providing handoff data to later phases

These PEIMs are responsible for the following:

- Initializing some permanent memory complement
- Describing the memory in Hand-Off Blocks (HOBs)
- Describing the firmware volume locations in HOBs
- Passing control into the Driver Execution Environment (DXE) phase

Figure 9 shows a diagram describes the action carried out during the PEI phase

### **2.5.1 PEI Services**

The PEI Foundation establishes a system table named the PEI Services Table that is visible to all Pre-EFI Initialization Modules (PEIMs) in the system. A PEI Service is defined as a function, command, or other capability 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 PEIM-to-PEIM Interface (PPI).

The PEI Foundation provides the classes of services listed in Table 2

---

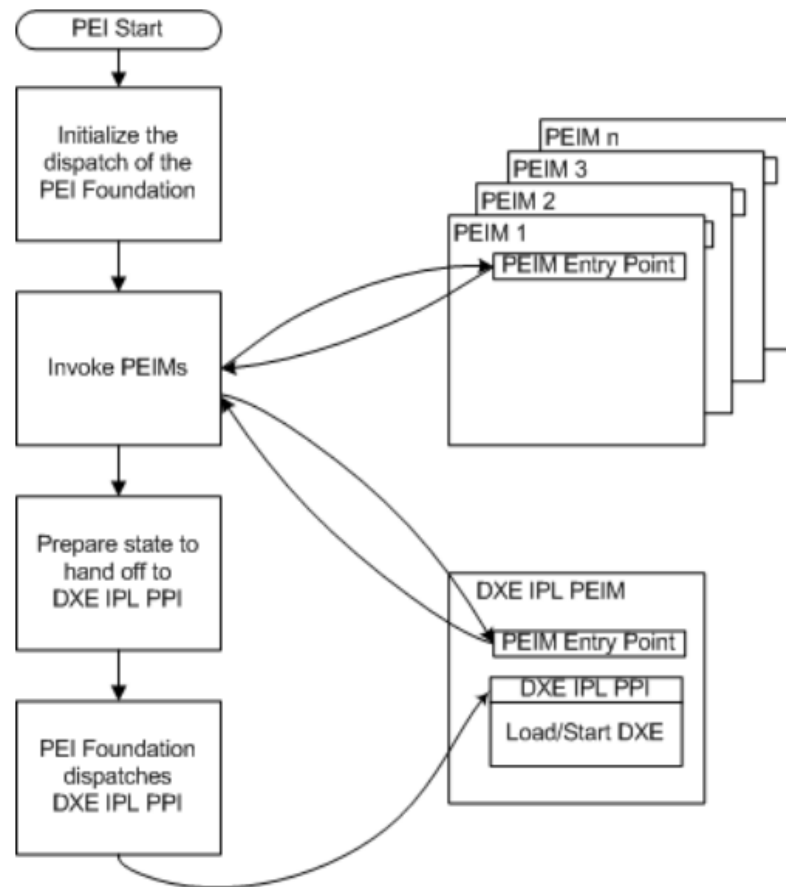


FIGURE 9: Diagram of PI Operations

### 2.5.2 PEI Foundation

The PEI Foundation is the entity that carried out the following activity:

- Dispatching of Pre-EFI initialization modules (PEIMs)
- Maintaining the boot mode
- Initialization of permanent memory
- Invoking the DXE loader

The PEI Foundation is written to be portable across all the various platform architectures of a given instruction-set. i.e. A binary for IA-32 (32-bit Intel architecture) works across all Pentium processors and similarly Itanium processor family work across all Itanium processors.

Irrespective of the processor micro architecture, the set of services uncovered by the PEI Foundation should be the same. This consistent surface area around the PEI Foundation allows PEIMs to be written in the C programming language and compiled across any micro architecture.

TABLE 2: Services provided by PEI Foundation Classes

| Service                  | Details   |
|--------------------------|---|
| PPI Services             | Manages PPIs to ease inter-module method calls between PEIMs. A database maintained in temporary RAM to track installed interfaces. |
| Boot Mode Services       | Manages the boot mode (S3, S5, diagnostics, normal boot, etc.)  |
| HOB Services             | Creates data structures (Hand-off-blocks) that are used to convey information to the next phase                                     |
| Firmware Volume Services | Finds PEIMs and along with that other firmware files in the firmware volumes  |
| PEI Memory Services      | provides a collection of memory management services (to be used before and after permanent memory to discovered)                    |
| Status Code Services     | Provides general progress and error code reporting services (i.e. port 080h or a serial port for text output for debug)             |
| Reset Services           | Provides a common means to aid initializing warm or cold restart of the system  |

## 2.6 PEI Dispatcher

The PEI Dispatcher is basically a state machine which is implemented in the PEI Foundation. The PEI Dispatcher evaluates the dependency expressions in Pre-EFI initialization modules (PEIMs) that are lying in the FVs being examined.

Dependency expressions are coherent combinations of PEIM-to-PEIM Interfaces (PPIs). These expressions distinguish the PPIs that must be available for use before a given PEIM can be invoked. The PEI Dispatcher references the PPI database in the PEI Foundation to conclude which PPIs have to be installed and evaluate the dependency expression for the PEIM. If PPI has already been installed then dependency expression will evaluate to TRUE, which notifies PEI Dispatcher it can run PEIM. At this stage, the PEI Foundation handovers control to the PEIM with TRUE dependency expression.

The PEI Dispatcher will exit Once the PEI Dispatcher has examined and evaluated all of the PEIMs in all of the uncovered firmware volumes and no more PEIMs can be dispatched (i.e. the dependency expressions do not evaluate from

FALSE to TRUE). At this stage, the PEI Dispatcher cannot invoke any additional PEIMs. The PEI Foundation then takes back control from the PEI Dispatcher and invokes the DXE IPL PPI to pass control to the DXE phase of execution.

## **2.7 Drive Execution Environment (DXE)**

Prior to the DXE phase, the Pre-EFI Initialization (PEI) phase is responsible for initializing permanent memory in the platform so that 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). HOBs are described in detail in the Platform Initialization Specification. There are several components in the DXE phase:

- DXE Foundation
- DXE Dispatcher
- A set of DXE Drivers

## **2.8 Boot Device Selection (BDS)**

The Boot Device Selection (BDS) phase is implemented as part of the BDS Architectural Protocol. The DXE Foundation will hand control to the BDS Architectural Protocol after all of the DXE drivers whose dependencies have been satisfied have been loaded and executed by the DXE Dispatcher. The BDS phase is responsible for the following:

- Initializing console devices
- Loading device drivers
- Attempting to load and execute boot selections

## **2.9 Transient System Load (TSL) and Runtime (RT)**

The Transient System Load (TSL) is primarily the OS vendor provided boot loader. Both the TSL and the Runtime Services (RT) phases may allow access to persistent content, via UEFI drivers and UEFI applications. Drivers in this category include PCI Option ROMs.

---

## 2.10 After Life (AL)

The After Life (AL) phase consists of persistent UEFI drivers used for storing the state of the system during the OS orderly shutdown, sleep, hibernate or restart processes.

## 2.11 Generic Build Process

All code starts out as either C sources and header files, assembly sources and header files, UCS-2 HII strings in Unicode files, Virtual Forms Representation files or binary data (native instructions, such as microcode) files. Per the UEFI and PI specifications, the C and Assembly files must be compiled and linked into PE32/PE32+ images. While some code is designed to execute only from ROM, most UEFI/PI modules are written to be relocate-able. These are written and built different. For example, Execute In Place (XIP) module code is written and compiled to run from ROM, while the majority of the code is written and compiled to execute from memory, which requires that the code be relocate able. Some modules may also permit dual mode, where it will execute from memory only if memory is available, otherwise it will execute from ROM. Additionally, modules may permit dual access, such as a driver that contains both PEI and DXE implementation code. Code is assembled or compiled, then linked into PE32/PE32+ images, the relocation section may or may not be stripped and an appropriate header will replace the PE32/PE32+ header. Additional processing may remove more non-essential information, generating a Terse (TE) image. The binary executables are converted into EFI firmware file sections. Each module is converted into an EFI Section consisting of an Section header followed by the section data (driver binary).

### 2.11.1 EFI Section Files

The general section format for sections less than 16MB in size is shown in Figure 11. Figure 10 shows the section format for sections 16MB or larger in size using the extended length field.

---

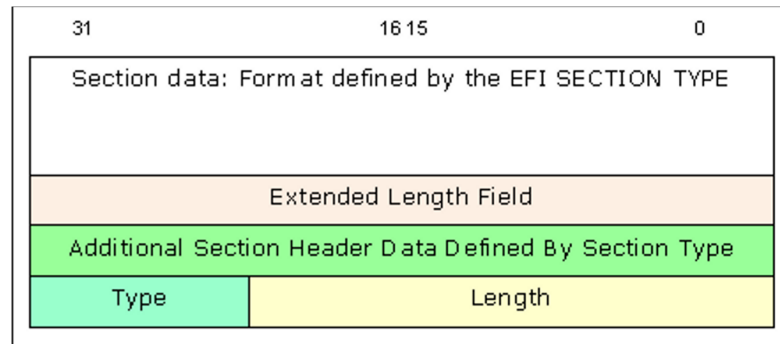


FIGURE 10: General EFI Section Format for large size Sections(greater than 16 MB)

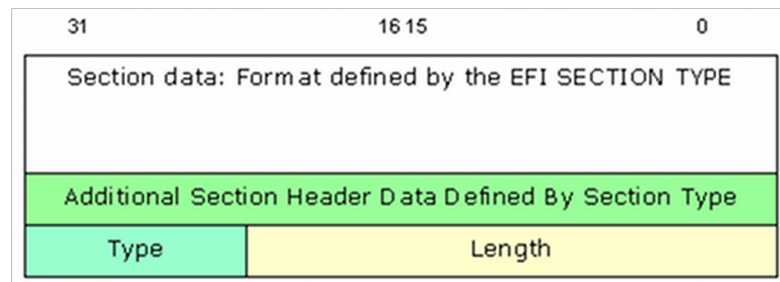


FIGURE 11: General EFI Section Format (less than 16 MB)

---

## 3. Architecture of BIOS Firmware

### 3.1 Overview

The basic Platform Initialization firmware storage concepts include:

- Firmware Volumes (FV)
- Firmware File Systems (FFS)
- Firmware Files
- Standard Binary Layout
- Pre-EFI Initialization (PEI) PEIM-to-PEIM Interfaces (PPIs)
- Driver Execution Environment (DXE) Protocols

### 3.2 Design of Firmware Storage

Design of firmware storage describes how files should be stored and accessed in non-volatile storage. Any firmware implementations must support and follow the standard PI Firmware Volume and Firmware File System Format.

**Firmware Device** is a persistent physical repository containing data and/or firmware code. Typically it is a flash component but may also be some other type of persistent storage. A single physical firmware device may be partitioned into multiple smaller pieces to form multiple logical firmware devices. Likewise, multiple firmware devices may be aggregated into one larger logical firmware device.

**Flash** devices are most usual non-volatile repository for firmware volumes. Often, flash devices are partitioned into sectors or blocks of possibly differing sizes, each with various run-time characteristics.

In the design of Firmware File System (FFS), several observed unique qualities of flash devices are listed below:

- Can be erased on a sector-by-sector basis. After ensuring, all bits of sector return their erase value<sup>1</sup>.
- Can be written on a bit-by-bit basis. i.e. if erase value is 0 then bit value 0 can be changed to 1.

---

<sup>1</sup>either all 0 or all 1

---

- Only by performing erase operation on an entire flash sector, non-erase value can change to erase value.
- Capable of enable/disable reads and writes to individual flash sectors or the entire flash.
- Writes and erases are longer operations than reads.
- Many times place restrictions on the trading operations that can be performed while a write or erase is occurring.

### **3.3 Firmware Volumes (FV)**

A Firmware Volume (FV) is a logical firmware device. Firmware Volume is the basic storage repository for data and/or code. Each and every firmware volume is organized into a file system. As such, the file is the base unit of storage for firmware. Table 3 describes attributes in each firmware volume.

Firmware volumes may also contain additional information describing the mapping between OEM file types and a GUID.

### **3.4 Firmware File System (FFS)**

A Firmware File System (FFS) describes the structure of files and (optional) free space within the firmware volume. Each firmware file systems has a unique GUID, which is used by the firmware to associate a driver with a newly exposed firmware volume.

Firmware files are code and/or data stored in firmware volumes. Attributes of files are described in Table 4.

Specific firmware volume formats may support additional attributes, such as integrity verification and staged file creation. The file data of certain file types is sub-divided in a standardized fashion into Firmware File Sections.

Non-standard file types are supported through the use of the OEM file types (described in detail in Table 5).

In the PEI phase, file-related services are provided through the PEI Services Table, using `FfsFindNextFile`, `FfsFindFileByName` and `FfsGetFileInfo`. In the DXE phase, file related services are provided through the `EFI_FIRMWARE_VOLUME2_PROTOCOL` services attached to a volume's handle (`ReadFile`, `ReadSection`, `WriteFile` and `GetNextFile`).

---



### 3.4.1 Firmware File Types

Consider an application file named FOO.EXE. 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 PI Firmware File System defines the contents of a file that is returned by the firmware volume interface.

The PI Firmware File System 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.

## 3.5 Firmware File Sections

Firmware file sections are separate discrete “parts” within certain file types. Each section has the following attributes:

While there are many types of sections, they fall into the following two broad categories:

- **Encapsulation sections** - containers that hold other sections. The sections contained within an encapsulation section are known as child sections, and the encapsulation section is known as the parent section are known as the parent section. An encapsulation section’s children may be leaves and/or more encapsulation sections and are called peers relative to each other. An encapsulation section does not contain data directly; instead it is just a vessel that ultimately terminates in leaf sections. Files that are built with sections can be thought of as a tree, with encapsulation sections as nodes and leaf sections as the leaves. The file image itself can be thought of as the root and may contain an arbitrary number of sections. Sections that exist in the root have no parent section but are still considered peers.
- **Leaf Sections** - Unlike encapsulation sections, leaf sections directly contain data and do not contain other sections. The format of the data contained within a leaf section is defined by the type of the section.

In the example shown in Figure 12, the file image root contains two encapsulation sections (E0 and E1) and one leaf section (L3). The first encapsulation section (E0) contains children, all of which are leaves (L0, L1, and L2). The second encapsulation section (E1) contains two children, one that is an encapsulation (E2) and the other that is a leaf (L6). The last encapsulation section (E2) has two children that are both leaves (L4 and L5).

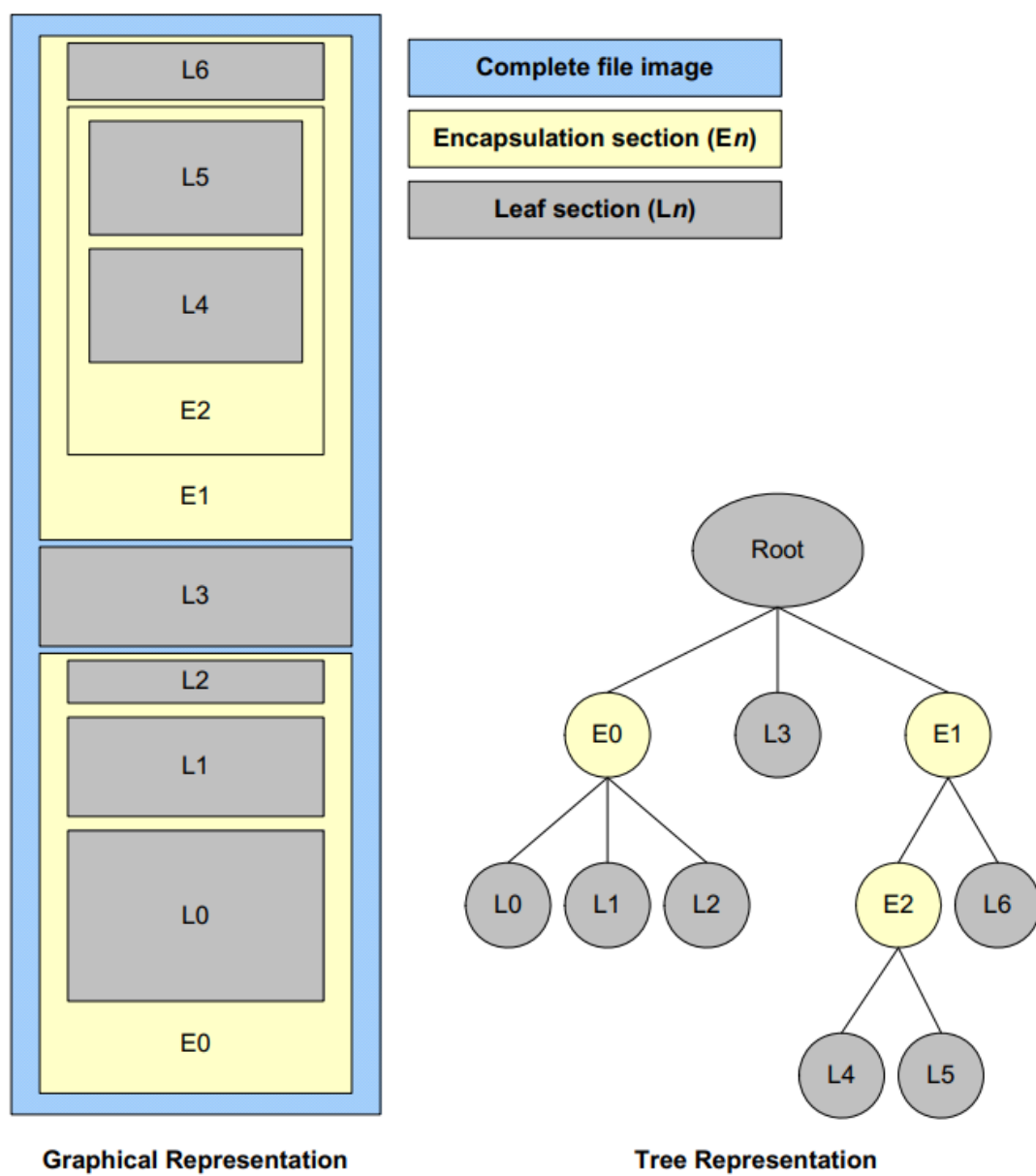


FIGURE 12: Example File System Image

TABLE 3: Firmware Volume Attributes

| Attribute                           | Description  |
|-------------------------------------|--|
| Name                                | each volume has a unique identifier name having UEFI Globally Unique Identifier (GUID).  |
| Size                                | describes total size of all volume data (including any header, files and free space)   |
| Format                              | describes Firmware File System (FFS) used in the body of the volume.   |
| Memory Mapped?                      | some volumes may be memory-mapped, indicates that the entire contents of the volume appear at once in the memory address space of the processor.   |
| Sticky Write?                       | Some volumes may require special erase cycles in order to change bits from a non-erase value to an erase value   |
| Erase Polarity                      | If a volume supports <i>Sticky Write</i> , then all bits within the volume will return to this value (0 or 1) after an erase cycle   |
| Alignment                           | The first byte of a volume is required to be aligned on some power-of-two boundary. At a minimum, this must be greater than or equal to the highest file alignment value. If <code>EFI_FVB2_WEAK_ALIGNMENT</code> is set in the volume header then the first byte of the volume can be aligned on any power-of-two boundary. A weakly aligned volume can not be moved from its initial linked location and maintain its alignment. |
| Read Enable/Disable Capable/Status  | Volumes may have the ability to change from readable to hidden   |
| Write Enable/Disable Capable/Status | Volumes may have the ability to change from writable to write protected  |
| Lock Capable/Status                 | Volumes may be able to have their capabilities locked  |
| Read-Lock Capable/Status            | Volumes may have the ability to lock their read status   |
| Write-Lock Capable/Status           | Volumes may have the ability to lock their write status  |

TABLE 4: Firmware Files Attributes

| Attribute | Description   |
|-----------|---|
| Name      | each volume has a unique identifier name having UEFI Globally Unique Identifier (GUID). File names must be unique within a firmware volume. Some file names have special significance.  |
| Type      | Each file has a type. There are four ranges of file types: Normal (0x00-0xBF), OEM (0xC0-0xDF), Debug (0xE0-0xEF) and Firmware Volume Specific (0xF0-0xFF). More file types information are described in Table 5.   |
| Alignment | Each file's data can be aligned on some power-of-two boundary. The specific boundaries that are supported depend on the alignment and format of the firmware volume. If <code>EFI_FVB2_WEAK_ALIGNMENT</code> is set in the volume header then file alignment does not |
| Size      | Describes size of each file, each file's data is zero or more bytes   |

TABLE 5: Firmware File Types

| Name                               | Value | Description  |
|------------------------------------|-------|--|
| FV_FILETYPE_RAW                    | 0x1   | Binary Data  |
| FV_FILETYPE_FREEFORM               | 0x2   | Sectioned Data   |
| FV_FILETYPE_SECURITY_CORE          | 0x3   | Platform core code used during the SEC phase   |
| FV_FILETYPE_PEI_CORE               | 0x4   | PEI Foundation   |
| FV_FILETYPE_DXE_CORE               | 0x5   | DXE Foundation   |
| FV_FILETYPE_PEIM                   | 0x6   | PEI Module (PEIM)  |
| FV_FILETYPE_DRIVER                 | 0x7   | DXE Driver   |
| FV_FILETYPE_COMBINED_PEIM_DRIVER   | 0x8   | Combined PEIM/DXE Driver   |
| FV_FILETYPE_APPLICATION            | 0x9   | Application  |
| FV_FILETYPE_SMM                    | 0xa   | Contains a PE32+ image that will be loaded into MMRAM in MM Traditional Mode   |
| FV_FILETYPE_FIRMWARE_VOLUME_IMAGE  | 0xb   | Firmware Volume Image  |
| FV_FILETYPE_COMBINED_SMM_DXE       | 0xc   | Contains PE32+ image that will be dispatched by the DXE Dispatcher and will also be loaded into MMRAM in MM Tradition Mode |
| FV_FILETYPE_SMM_CORE               | 0xd   | MM Foundation that support MM Traditional Mode   |
| EFI_FV_FILETYPE_MM_STANDALONE      | 0xe   | Contains PE32+ image that will be loaded into MMRAM in MM Standalone Mode  |
| EFI_FV_FILETYPE_MM_CORE_STANDALONE | 0xf   | Contains PE32+ image that support MM Tradition Mode and MM Standalone Mode   |
| FV_FILETYPE_OEM_MIN                | 0xc0  | OEM File Type  |
| FV_FILETYPE_OEM_MAX                | 0xdf  | OEM File Type  |
| FV_FILETYPE_DEBUG_MIN              | 0xe0  | Debug/Test File Type   |
| FV_FILETYPE_DEBUG_MAX              | 0xef  | Debug/Test File Type   |
| FV_FILETYPE_FFS_MIN                | 0xf0  | Firmware File System Specific File Type  |
| FV_FILETYPE_FFS_MAX                | 0xff  | Firmware File System Specific File Type  |
| FV_FILETYPE_FFS_PAD                | 0xf0  | Pad file for FFS   |

| Attribute | Description                   |
|-----------|-------------------------------|
| Type      | Each section has type         |
| Size      | describes size of the section |

---

# Glossary

**ACPI** Advanced Configuration and Power Interface . v, vii, 5, 7

**BDS** Boot Device Selection . vi, 14, 19

**BIOS** Basic Input Output System . v, 2, 5

**DXE** Driver Execution Environment . vi, 14, 19, 22

**EDK II** Extensible Firmware Interface Developer Kit II . 12

**FFS** Firmware File System . vi, 22, 23

**FV** Firmware Volume . vi, 18, 22, 23

**Operating System (OS)** Operating System . 2, 3

**PEI** Pre-EFI Initialization . v, 14, 15, 22

**PI** Platform Initialization . v, vii, 12, 14, 15, 22

**SEC** Security . v, 14

**UEFI** Unified Extensible Firmware Interface . v, 2–5, 10, 12