Generic IP independent BIOS Signing and Parsing

Submitted by

Gahan Saraiya

18MCEC10

Department of Computer Science & Engineering,

Institute of Technology,

Nirma University, Ahmedabad,

Gujarat - 382481, India.

April, 2020

Generic IP independent BIOS Signing and Parsing

Major Project

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technologyin Computer Science & Engineering

with specialization in Computer Science & Engineering

Submitted by

Gahan Saraiya

18MCEC10

Department of Computer Science & Engineering,

Institute of Technology,

Nirma University, Ahmedabad,

Gujarat - 382481, India.

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 & Engineeringwith 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 UniversityAhmedabad, 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 bonafide 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

Intel System on a Chip (SoC) features a new set of Intel Intellectual Property (IP) for every generation. BIOS involves development of major individual components such as Processor, Graphics/Memory Controller, Input/Output Controller hub, System Monitor/Management Bus, Direct Media Interface, SATA/IDE/USB, Peripheral Component Interconnect (PCI), Voltage Regulator and Advanced Configuration and Power Interface (ACPI) for every Intel System on a Chip (SoC). Duration of every iteration of the development of any such components takes a much longer duration to build the BIOS binary and executing on hardware/SoC and verifying the functional work flow from logs. Even for some minor changes such as changing setup option value takes a high amount of time for iteration which is indeed slowing down the process of release and development of the Intel Intellectual Property (IP). Major aim of this project will be to introduce a framework which can be used to reduce the cost of each build and test iteration for product development.

Acknowledgements

It gives me immense pleasure in expressing thanks and profound gratitude to Prof. Dvijesh Bhatt, Assistant Professor, Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout this work. The appreciation and continual support he has imparted has been a great motivation to me in reaching a higher goal. His guidance has triggered and nourished my intellectual maturity that I will benefit from, for a long time to come.

It gives me an immense pleasure to thank Dr. Madhuri Bhavsar, Honorable Head of Computer Science And Engineering Department, Institute of Technology, Nirma University, Ahmedabad for her kind support and providing basic infrastructure and healthy research environment.

A special thank you is expressed whole heartedly to Dr. Alka Mahajan, Honorable Director, Institute of Technology, Nirma University, Ahmedabad for the unmentionable motivation she has extended throughout course of this work. I would also thank the Institution, all faculty members of Computer Engineering Department, Nirma University, Ahmedabad for their special attention and suggestions towards the project work.

Gahan Saraiya

18MCEC10

Contents

List of Figures

1. Introduction

Intel System on a Chip (SoC) features a new set of Intel Uncore Intellectual Property (IP) for every generation. Section 1. covers the introduction and overview of BIOS, UEFI and it's role and major components - Advanced Configuration and Power Interface (ACPI), Peripheral Component Interconnect Express (PCIe) and Graphics Controller. Section 2. describes the design of UEFI and the boot phases in detail. The study of the BIOS binary structure and mapping of each components byte and alignment is described in Section 3.. Proposed work to reducing the process of build iteration described in Section 5..

1.1 Uncore Intellectual Properties

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 PCI or PCIe, 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 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 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.

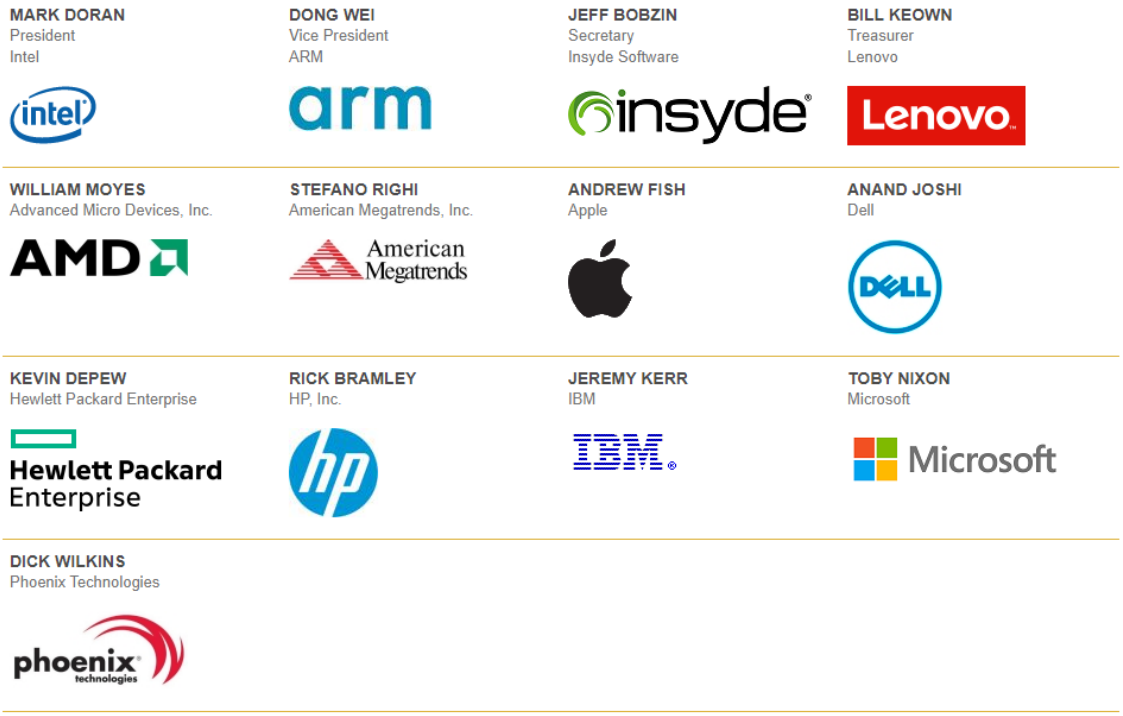

FIGURE 1: Board of Directors of UEFI Forum

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 Specificationcomplaint 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.

TABLE 1: Legacy BIOS v/s UEFI

	Legacy BIOS	EFI
Language	Assembly	C (99%)
Resource	Interrupt Hardcode Memory Access 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^{rd} Party ISV & IHV	Bas for Support	Easy for Support and for Multi Platforms

1.4 Comparing of Legacy BIOS and UEFI

1.5 Advanced Configuration and Power Interface (ACPI)

The ACPI Component Architecture (ACPICA) 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 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

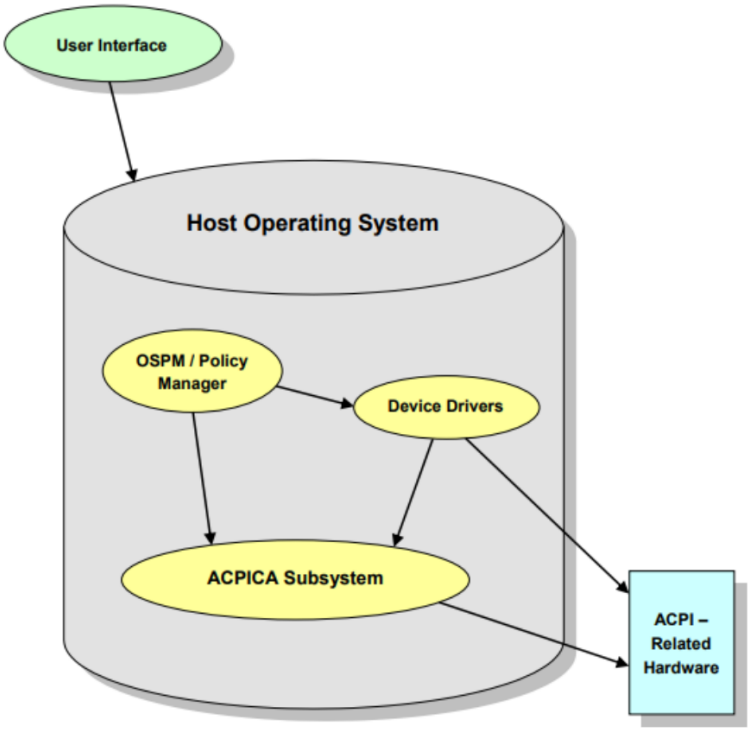

FIGURE 2: The ACPI Component Architecture

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

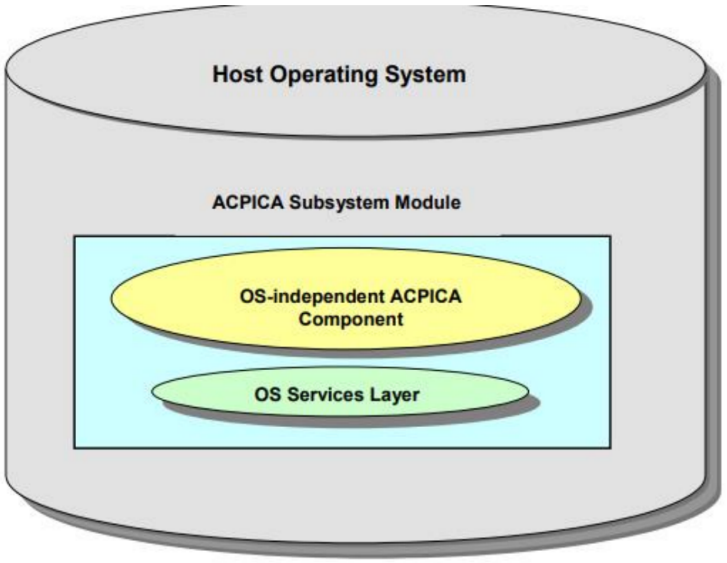

FIGURE 3: ACPICA Subsystem Architecture

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

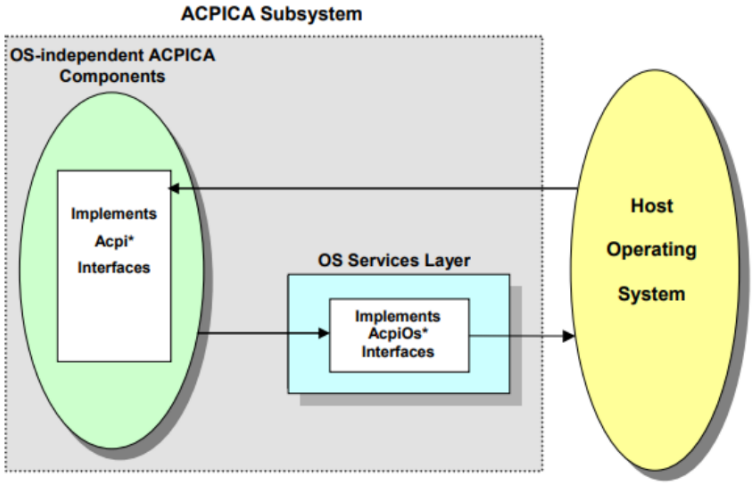

FIGURE 4: Interaction between the Architectural Components

1.6 Peripheral Component Interconnect Express (PCIe)

The PCI architecture has proven to be successful beyond even the most optimistic expectations. Today nearly every new computer platform comes outfitted with multiple PCI slots. In addition to the unprecedented number of PCI slots being shipped, there are also hundreds of PCI adapter cards that are available to satisfy virtually every conceivable application. This enormous momentum is difficult to ignore.

Today there is also a need for a new, higher-performance I/O interface to support emerging, ultrahigh-bandwidth technologies such as 10 Gigabit Ethernet, 10 Gigabit FibreChannel, 4X and 12X InfiniBand, and others. A standard that can meet these performance objectives, while maintaining backward compatibility to previous generations of PCI would undoubtedly provide the ideal solution.

To meet these objectives, the PCI-X 2.0 standard has been developed. PCI-X 2.0 has the performance to feed the most bandwidth-hungry applications while at the same time maintaining complete hardware and software backward compatibility to previous generations of PCI and PCI-X. The PCI-X 2.0 standard introduces two new speed grades: PCI-X 266 and PCI-X 533. These speed grades offer bandwidths that are two times and four times that of PCI-X 133 – ultimately providing bandwidths that are more than 32 times faster than the original version of PCI that was introduced eight years ago. It achieves the additional performance via time-proven DDR (Double Data Rate) and QDR (Quad Data Rate) techniques that transmit data at either 2-times or 4-times the base clock frequency. Because PCI-X 2.0 preserves so many elements from previous generations of PCI it is the beneficiary of a tremendous amount of prior development work. The operating systems, connector, device drivers, form factor, protocols, BIOS, electrical signaling, BFM (bus functional model), and other original PCI elements, are all heavily leveraged in the PCI-X 2.0 specification.

In fact, many of these elements remain identical in PCI-X 2.0. These similarities make implementation easy because these elements have already been designed and engineers are already familiar with them. As a result, the time-to-market is short, and risk is dramatically reduced.

The market migration to PCI-X 2.0 will also be easy because there are so many previous-generation PCI adapter cards already on the market. There are already hundreds of PCI adapter cards that are available today that can be utilized by every PCI-X 266 and PCI-X 533 slot. In addition, new PCI-X 266 and PCI-X 533 adapter cards have ready homes in any of the millions of PCI and PCI-X slots in existing systems. Because of these factors, PCI-X 2.0 provides the ideal next-generation, local I/O solution for high-bandwidth applications. It offers the performance needed for today's and tomorrow's applications in an easy-to-adopt, backward-compatible standard.

1.6.1 Functional Description

PCI BIOS functions provide a software interface to the hardware used to implement a PCI based system. Its primary usage is for generating operations in PCI specific address spaces (configuration space and Special Cycles). PCI BIOS functions are specified to operate in the following modes of the X86 architecture. The modes are: real-mode, 16:16 protected mode (also known as 286 protected mode), 16:32 protected mode (introduced with the 386), and 0:32 protected mode (also known as "flat" mode, wherein all segments start at linear address 0 and span the entire 4-GB address space).

Access to the PCI BIOS functions for 16-bit callers is provided through Interrupt 1Ah. 32-bit (i.e., protected mode) access is provided by calling through a 32-bit protected mode entry point. The PCI BIOS function code is B1h. Specific BIOS functions are invoked using a sub-function code. A user simply sets the host processors registers for the function and sub-function desired and calls the PCI BIOS software. Status is returned using the CARRY FLAG ([CF]) and registers specific to the function invoked.

1.6.2 UEFI PCI Services

UEFI stands for Unified Extensible Firmware Interface. The UEFI Specification, Version 2.3 or later describes an interface between the operating system and the platform firmware. The interface is in the form of data tables that contain platform-related information and boot and run-time services calls that are available to the operating system and its loader. Together, these provide a standard environment for booting an operating system.

The following sections provide an overview of the EFI Services relevant to PCI (including Conventional PCI, PCI-X, and PCI Express). For details, refer to the UEFI Specification. UEFI is processor-agnostic.

1.6.3 UEFI Driver Model

The UEFI Driver Model is designed to support the execution of drivers that run in the pre-boot environment present on systems that implement the UEFI firmware. These drivers may manage and control hardware buses and devices on the platform, or they may provide some software derived platform specific services.

The UEFI Driver Model is designed to extend the UEFI Specification in a way that supports device drivers and bus drivers. It contains information required by UEFI driver writers to design and implement any combination of bus drivers and device drivers that a platform may need to boot an UEFI-compliant operating system.

Applying the UEFI Driver Model to PCI, the UEFI Specification defines the PCI Root Bridge Protocol and the PCI Driver Model and describes how to write PCI bus drivers and PCI devices drivers in the UEFI environment. For details, refer to the UEFI Specification.

- **PCI Root Bridge Protocol** - A PCI Root Bridge is represented in UEFI as a device handle that contains a Device Path Protocol instance and a PCI Root Bridge Protocol instance. PCI Root Bridge Protocol provides an I/O abstraction for a PCI Root Bridge that the host bus can perform. This protocol is used by a PCI Bus Driver to perform PCI Memory, PCI I/O, and PCI Configuration cycles on a PCI Bus. It also provides services to perform different types of bus mastering DMA on a PCI bus. PCI Root Bridge Protocol abstracts device specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform independent code that is consuming basic system resources. An example of such system memory map changes is a system that provides non-identity memory mapped I/O (MMIO) mapping between the host processor view and the PCI device view.
- **PCI Driver Model** - The PCI Driver Model is designed to extend the UEFI Driver Model in a way that supports PCI Bus Drivers and PCI Device Drivers. This applies to Conventional PCI, PCI-X, and PCI Express. PCI Bus Drivers manage PCI buses present in a system. The PCI Bus Driver creates child device handles that must contain a Device Path Protocol instance and a PCI I/O Protocol instance. The PCI I/O Protocol is used by the PCI Device Driver to access memory and I/O on a PCI controller. PCI Device Drivers manage PCI controllers present on PCI buses. The PCI Device Drivers produce an I/O abstraction that may be used to boot an UEFI compliant operating system.

1.6.4 Graphics Output Protocol (GOP)

Graphics Output Protocol (GOP) is defined in the UEFI Specification to remove the hardware requirement to support legacy VGA and INT 10h BIOS. GOP provides a software abstraction to draw on the video screen.

1.6.5 BUS Performances and Number of Slots Compared

The various architectures defined by the PCISIG. The table shows the evolution of bus frequencies and bandwidths., as it obvious, increasing bus frequency compromises the number of electrical loads or number of connectors allowable on a bus at that frequencies. At some point for a given bus architecture there is an upper limit beyond which one cannot further increase the bus frequency, hence requiring the definition of a new bus architecture. A PCI express (PCIe) Interconnect that connects two devices is referred to as a Link. A link consists if either x1, x2, x4, x8, x12, x16 or x32 signal pairs in each direction. These signals are referred to as Lanes. A designer determines how many Lanes to implement based on the targeted performance benchmark required on a given Link.

The Table 2 shows aggregate bandwidth numbers for various Link width implementations, as is apparent from this table, the peak bandwidth achievable with PCIe is significantly higher than any existing bus today.

1.7 Graphics Controller

Most graphics controllers are PCI controllers. The graphics drivers managing those controllers are also PCI drivers. However, while most graphics controllers are PCI controllers, graphics controllers can make use of other buses, such as USB buses. Graphics drivers have these characteristics:

- UEFI graphics drivers follow the UEFI driver model.
- Depending on the adapter that the driver manages, a graphics driver can be categorized as either a single or a multiple output adapter.
- The graphics driver must create child handles for each output.
- Graphics drivers must create child handles for some of the graphics output ports and attach the Graphics Output Protocol (GOP Protocol), EDID Discovered Protocol, and EDID Active Protocol to each active handle that the driver produced.
- Graphics drivers are chip-specific because of the requirement to initialize and manage the graphics device. A UEFI driver is required for any PC hardware device needed for the boot process to complete. Hardware devices can be categorized into the following:

TABLE 2: Comparison of Bus Frequency, Bandwidth and Number of Slots

Bus Type	Clock Frequency	Peak Bandwidth	Number of Card Slots per Bus
PCI $32-bit$	33 MHz	133 $MBps$	$4-5$
PCI $64-bit$	33 MHz	266 $MBps$	$4-5$
PCI $32-bit$	66 MHz	266 $MBps$	$1-2$
PCI $64-bit$	66 MHz	533 $MBps$	$1-2$
PCI-X $32-bit$	66 MHz	266 $MBps$	4
PCI-X $64-bit$	66 MHz	533 $MBps$	4
PCI-X $32-bit$	133 MHz	533 $MBps$	$1-2$
PCI-X $64-bit$	133 MHz	1066 $MBps$	$1-2$
PCI-X $32-bit$	266 MHz effective	1066 $MBps$	1
PCI-X $64-bit$	266 MHz effective	2131 $MBps$	1
PCI-X $32-bit$	533 MHz effective	2131 $MBps$	1
PCI-X $64-bit$	533 MHz effective	4262 $MBps$	1

- Graphic output devices: Simple text, graphics output.
- Console devices: Simple input provider, simple input ex, simple pointer — mice, serial I/O protocol (remote consoles)

Note that independent hardware vendors (IHVs) can choose not to implement all of the required elements of the UEFI specification. For example, all elements might not be implemented on a specialized system configuration that does not support all the services and functionality implied by the required elements. Also, some elements are required depending on a specific platform's features. Some elements are required depending on the features that a specific driver requires. Other elements are recommended based on coding experience, for reasons of portability, and/or for other considerations. It is recommended that you implement all required and recommended elements in your drivers.

1.7.1 Graphics Output Protocol

The GOP (Graphics Output Protocol) Driver is part of the UEFI boot time drivers responsible for bringing up the display during bios boot. This driver enables logo display during bios boot time. This paper has a GOP device driver written for an Intel's IoT Android platform which is responsible for display control until the operating system and in turn the display controller of the system gains the control.

1.7.2 GOP Overview

The GOP driver is a replacement for legacy video BIOS and enables the use of UEFI pre-boot firmware without CSM. The GOP driver can be 32-bit, 64-bit, or IA-64 with no binary compatibility. UEFI pre-boot firmware architecture (32/64-bit) must match the GOP driver architecture (32/64-bit). The Intel Embedded Graphics Drivers' GOP driver can either be fast boot (speed optimized and platform specific) or generic (platform agnostic for selective platforms).

EFI defines two types of services: boot services and runtime services. Boot services are available only while the firmware owns the platform (i.e., before the ExitBootServices call), and they include text and graphical consoles on various devices, and bus, block and file services. Runtime services are still accessible while the operating system is running; they include services such as date, time and NVRAM access. In addition, the Graphics Output Protocol (GOP) provides limited runtime services support. The operating system is permitted to directly write to the frame buffer provided by GOP during runtime mode. However, the ability to change video modes is lost after transitioning to runtime services mode until the OS graphics driver is loaded. This paper includes a GOP driver written for an IoT's platform using the development kit EDK II which is responsible for the display during booting process until the operating system gains control of the display and invoke display devices.

1.7.3 GOP DRIVER

The EFI specification defined a UGA (Universal Graphic Adapter) protocol to support device-independent graphics. UEFI did not include UGA and replaced it with GOP (Graphics Output Protocol), with the explicit goal of removing VGA hardware dependencies. The two are similar.

Table 3 gives a quick comparison of GOP and video BIOS.

1.7.4 GOP Integration

The platform firmware must meet the following requirements for GOP Driver integration:

TABLE 3: Difference between Video BIOS and GOP

Bus Type	Clock Frequency
64 KB limit	execution No 64 KB limit. $32-bit$ protected mode
CSM is needed with UEFI system firmware	No need for CSM. Speed optimized (fast boot)
$16-bit$ The VBIOS works with both $32-bit$ and $64-bit$ architectures	The UEFI pre-boot firmware architecture must match the GOP driver.

TABLE 4: GOP Driver files

File Name	Description	Format
`GopDriver.efi`	The GOP driver binary	Uncompressed PE/COFF image
`Vbt.bin`	Contains Video BIOS Table (VBT) data	Raw Binary
`Vbt.bsf`	BMP script file. Required for modifying Vbt.bin using BMP tool	Text

- Platform firmware must be compliant to UEFI 2.1 or later.
- Platform must enumerate and initialize the graphics device.
- Platform must allocate enough graphics frame buffer memory required to support the native mode resolution of the integrated display.
- The platform must produce the standard `EFI_PCI_IO_PROTOCOL` and as well as the `EFI_DEVICE_PATH_PROTOCOL` on the graphics device handle. Additionally, the platform must produce `PLATFORM_GOP_POLICY_PROTOCOL`.
- The platform firmware must not launch the legacy Video BIOS.

The GOP Driver solution comprises the following files shown in Table 4 GOP driver files.

Customize the VBT data file `Vbt.bin` as per platform requirements and the corresponding BSF file. Integrate `Vbt.bin` and `GopDriver.efi` files into the platform firmware image. The process of accomplishing this step is determined by the platform implementer, specific to the platform firmware implementation.

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.

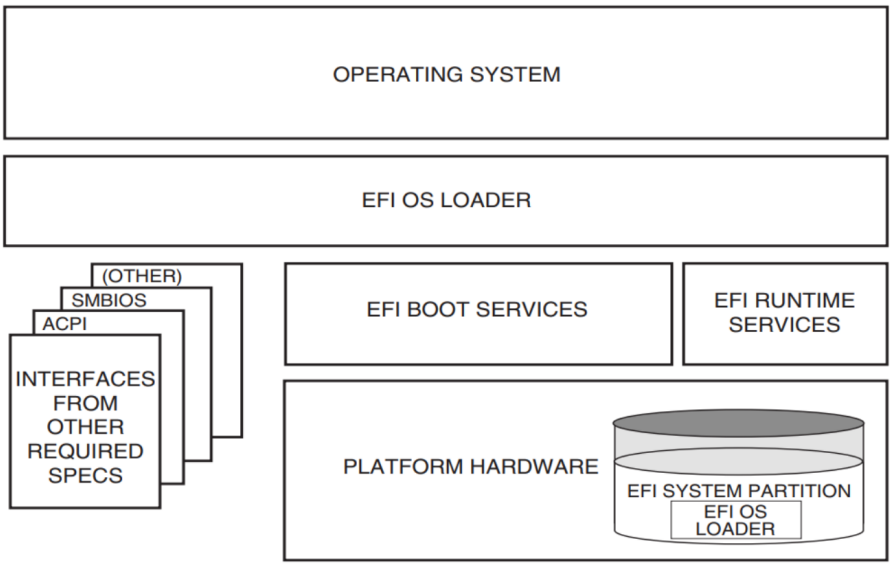

FIGURE 5: UEFI Conceptual Overview

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 latforms, 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

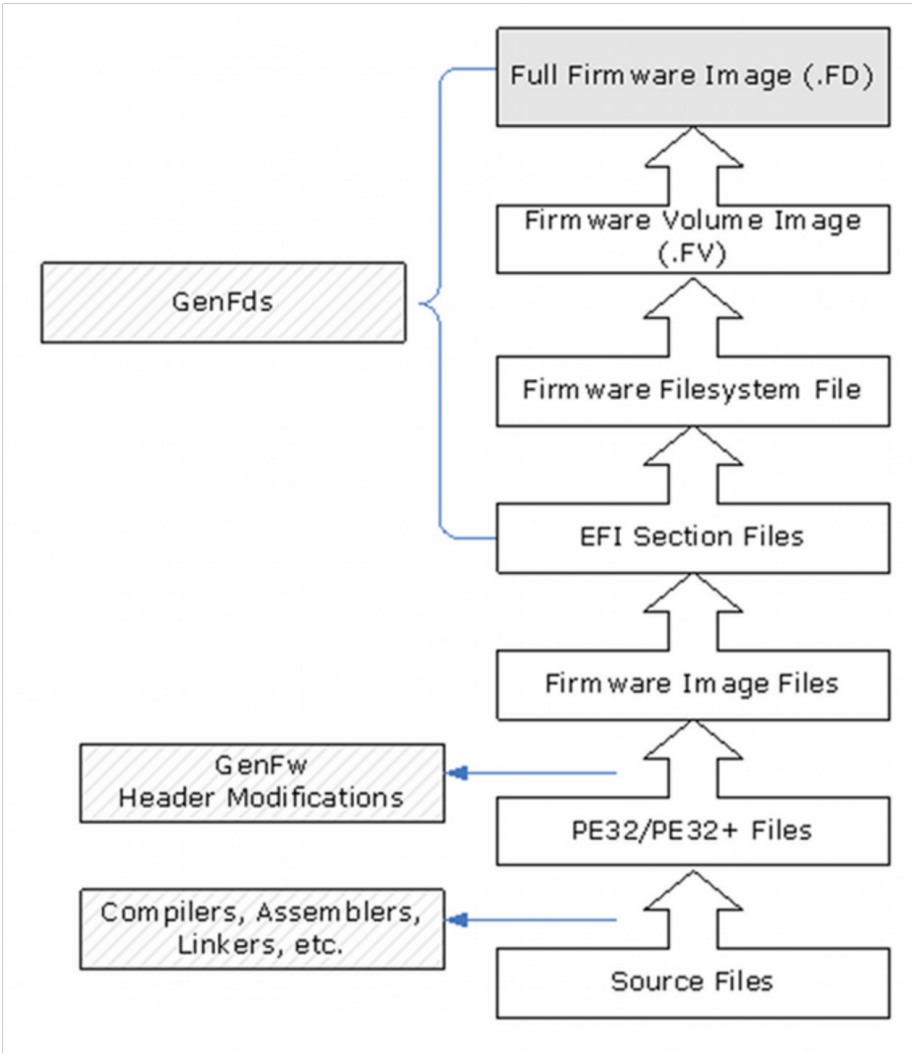

FIGURE 6: UEFI/PI Firmware Image Creation

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 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

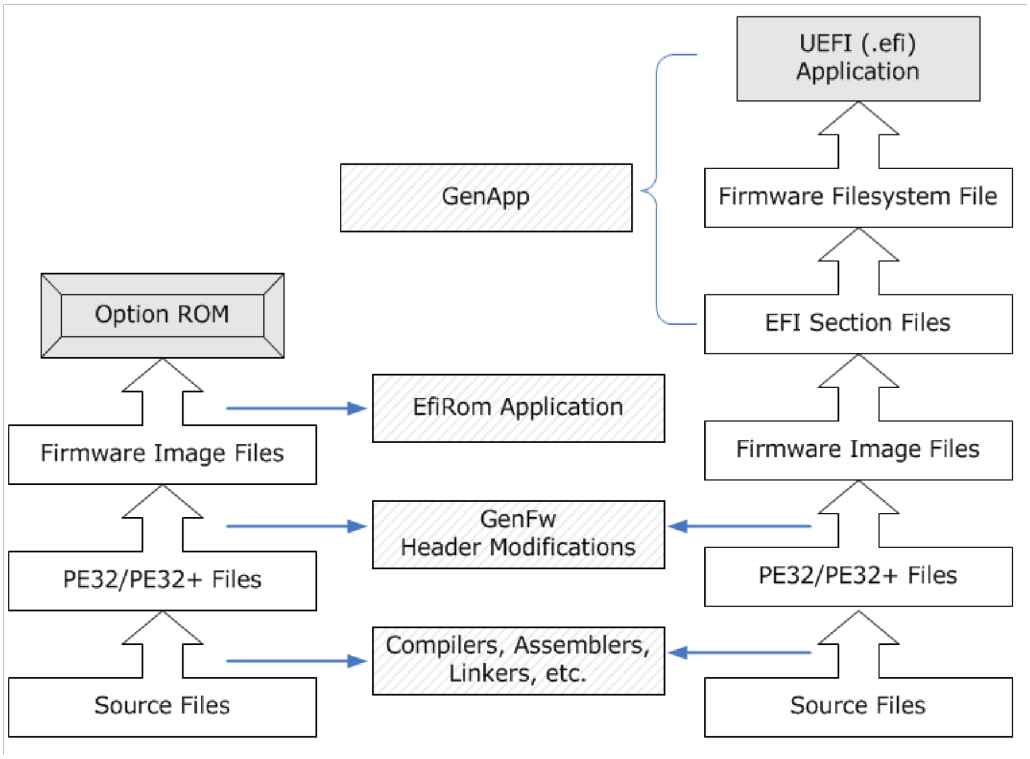

FIGURE 7: UEFI/PI Firmware Image Creation

Option ROM images containing driver code. Figure 7, below, shows the reference implementation tools and creation processes for both of these image types

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

Platform Initialization PI compliant system firmware has to support the six phases:

1. Security (SEC) Phase
2. Pre-efi Initialization (PEI) Phase
3. Driver Execution Environment (DXE) Phase
4. Boot device selection (BDS) Phase

5. Run time (RT) services and After Life (AL) (transition from the OS back to the firmware) of system.

Figure 8 describes the phases and transition in detail.

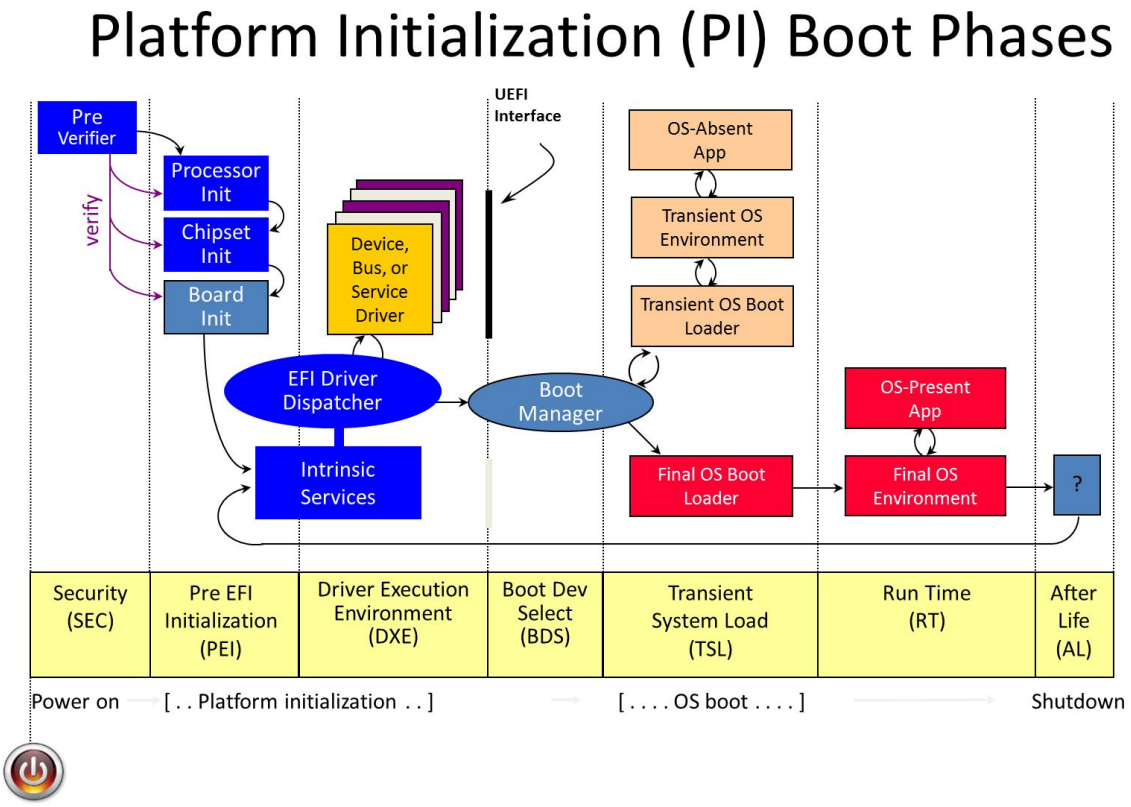

FIGURE 8: PI Boot Phases

2.4 Security (SEC)

The Security (SEC) phase is the initial phase in the PI Architecture and is liable for the following:

- Handling restart events of all platform
- Creation of a temporary memory store
- Bringing the root of trust in the system
- Transit handoff information to next phase - the PEI Foundation

The security section may have the modules with source code scripted in assembly language. Hence, some EDK II module development environment (MDE) modules can consist of assembly code. During Occurrence of this, both Windows and GCC versions of assembly language code are served 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 at first operates with the platform in a developing state, holding only on-processor resources, such as the cache of processor for call stack, to dispatch the 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
- Characterizing the memory in Hand-Off Blocks (HOBs)
- Characterizing the firmware volume locations in HOBs
- Transit the control into next phase - the Driver Execution Environment (DXE) phase

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

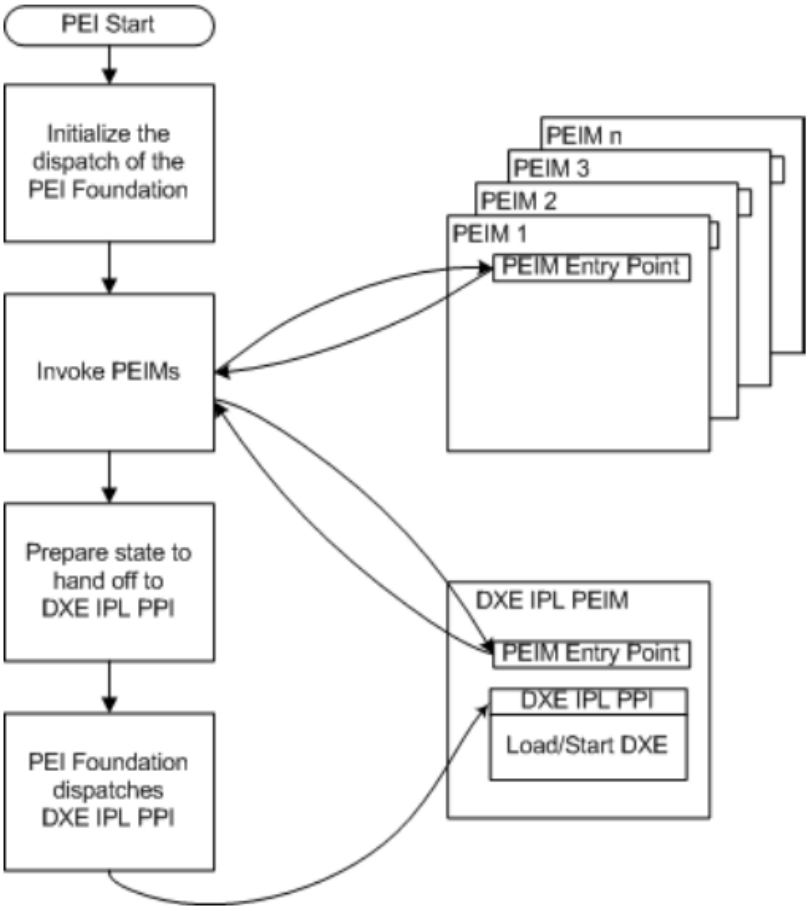

FIGURE 9: Diagram of PI Operations

2.5.1 PEI Services

The PEI Foundation institutes a system table for the PEI Services named as PEI Services Table that is viewable to all PEI Modules (PEIMs) in the system. A PEI Service is defined as a method, command or other potentiality manifested by the PEI Foundation when that service's initialization needs are met. As the PEI phase having no permanent memory available until almost the end of the phase, all the various types of services created during this phase (PEI phase) cannot be as enrich as those created during later phases. A pointer to PEI Services Table is sent into entry point of each PEIM's and also to part of each PEIM-to-PEIM Interface (PPI) because the location of PEI Foundation and its temporary memory is unknown at build time.

The PEI Foundation provides the classes of services listed in Table 5

2.5.2 PEI Foundation

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

TABLE 5: 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

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

The PEI Foundation written to be portable across all the various platforms architecture 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.

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 (DEPEX) 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 calls the `DXE IPL PPI` to navigate control to the DXE phase of execution.

2.7 Drive Execution Environment (DXE)

Before the DXE phase, the Pre-EFI Initialization (PEI) phase is held responsible for initializing permanent memory in the platform. Hence, DXE phase can be loaded and executed. At the very end of the PEI phase, state of the system is handed over to the DXE phase through Hand-Off Blocks (list of position independent data structures).

There are three components in the DXE phase:

1. DXE Foundation
2. DXE Dispatcher
3. A set of DXE Drivers

2.8 Boot Device Selection (BDS)

The BDS Architectural Protocol has part of implementation of the Boot Device Selection (BDS) phase. After evaluation of all of the DXE drivers dependencies, DXE drivers with satisfied dependencies are loaded and executed by the DXE Dispatcher, the DXE Foundation will pass the control to the BDS Architectural Protocol. The BDS phase held responsible for the following:

- Initializing console devices
- Loading device drivers
- Attempt of loading and executing boot selections

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

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

2.10 After Life (AL)

The After Life (AL) phase contains the persistent UEFI drivers used to store the state of the system during the OS systematically shutdown, sleep, hibernate or restart processes.

2.11 Generic Build Process

All code initialized as either C sources and header files, assembly language sources and header files, Unicode files (UCS-2 HII strings), Virtual Forms Representation files or binary data (native instructions, such as microcode) files. Per the UEFI and PI specifications, the C files and Assembly files must be compiled and coupled into PE32 or PE32+ images. While some code is configured to execute only from ROM, most UEFI and PI modules code are written to be relocatable. These are written and built different i.e. XIP (Execute In Place) 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 needs the relocatable code.

Some modules may also allow dual mode, where it will execute from memory only if memory is sufficient, 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

he 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.

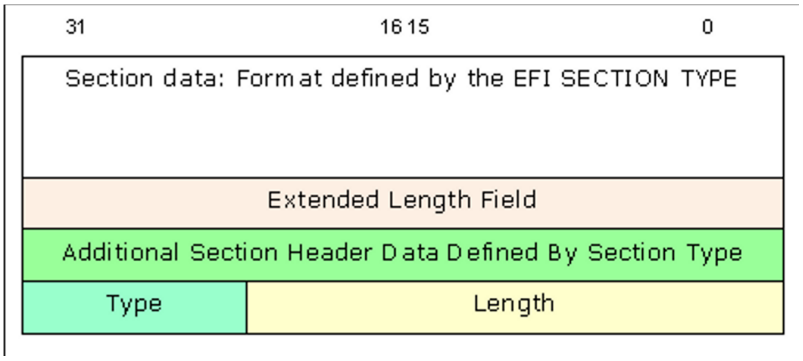

FIGURE 10: General EFI Section Format for large size Sections(greater then 16 MB)

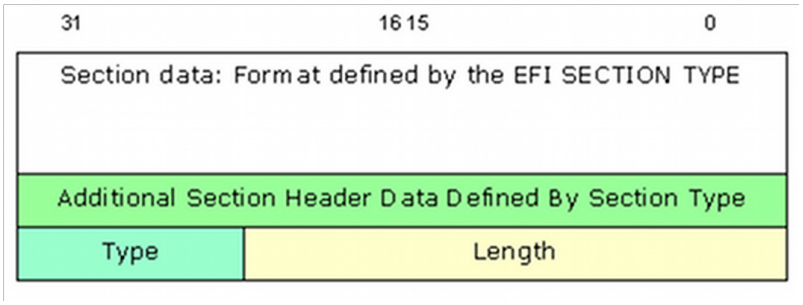

FIGURE 11: General EFI Section Format (less then 16 MB)

2.12 Cross Compatibility of CPUs

Whenever customer try to change the default Intel motherboard CPU with different Intel silicon chip which won't works. The specific CPU Chip initialization varies for each generation. So, the board designs should be designed is such a way that specific generation CPU should support., if we change the CPU with a different Intel Board it will not even boot, because the BIOS doesn't support for other Silicon Initialization for other CPUs.

So, we are integrating the runtime detection of the silicon during the Pre-Extensible Firmware Initialization (PEI) phase. So, within single Integrated Firmware Image (IFWI) should support the Multi Generation CPUs which is never tried before.

Each silicon has a fixed register from which the CPU generation can be identified., so the BIOS should read that register and program in such a way the is CPU1 is in Platform it should support the CPU1 Features like PCIe, Graphics & DMI., if the

CPU1 is replaced with CPU2 then it should support the CPU2 speed. That should be taken care by the BIOS.

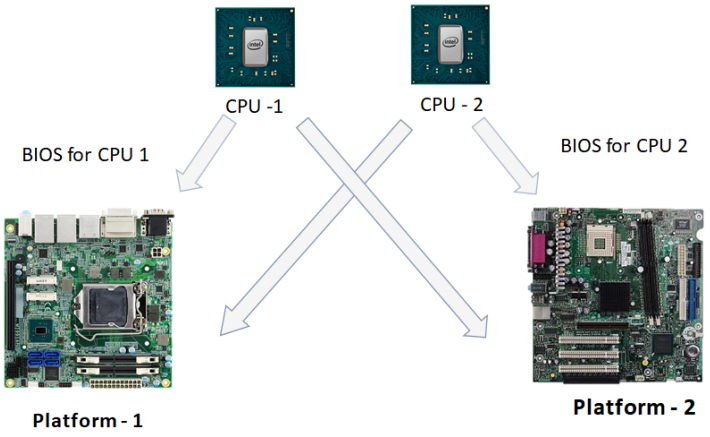

FIGURE 12: Cross Compatibility Design

Figure 12 shows the general view of the Cross Compatibility of CPUs.

BIOS is the part of Integrated Firmware Image which resides at the End of the Binary table. The CPU swap can only occur in Specially designed Intel Designed Board only. Mainly because for each and every feature it required some hardware(H/W) requirements. If that H/W requirement not present. Then It will boot but doesn't support the Maximum speed.

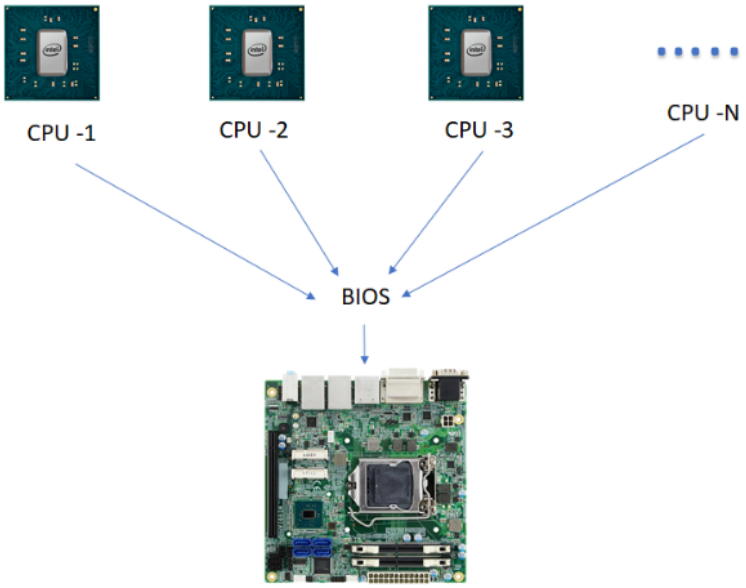

FIGURE 13: BIOS Support for Cross Compatibility

Figure 13 shows the BIOS role for identifying the CPUs during PEI phase.

As the number of Feature increases in the Silicon BIOS size also increases, usually the BIOS size varies from Platform to Platform and CPU to CPU., as we are integrating the Compatibility the BIOS size obviously increases.

The structure of IFWI is Shown in Figure 14

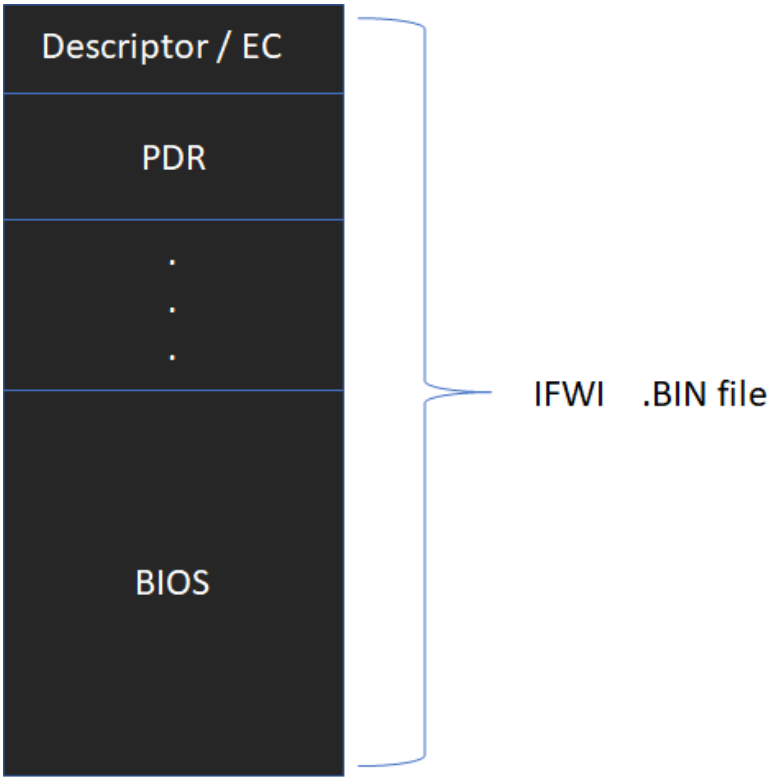

FIGURE 14: Integrated Firmware Image

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. Implementation of any firmware must support and follow the standard PI Firmware Volume structure and the structure of 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`[1].
- 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.

[1] 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 6 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 7.

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 8).

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`).

TABLE 6: 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 *Sticky Write*, 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 `EFI_FVB2_WEAK_ALIGNMENT` 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 7: 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 8.
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 `EFI_FVB2_WEAK_ALIGNMENT` 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

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:

TABLE 8: 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

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 15, 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).

In the PEI phase, section-related services are provided through the PEI Service Table, using `FfsFindSectionData`. In the DXE phase, section-related services are provided through the `EFI_FIRMWARE_VOLUME2_PROTOCOL` services attached to a volume's handle (`ReadSection`).

3.6 Firmware File Section Types

Table 9 list outs the defined architectural section types.

3.7 PI Architecture Firmware File System Format

This section describes the standard binary encoding for PI Firmware Files, PI Firmware Volumes, and the PI Firmware File System. Implementations that allow the non-vendor firmware files or firmware volumes to be introduced into the system must

TABLE 9: Architectural Section Types

Name	Value	Description
EFI_SECTION_COMPRESSION	$0x1$	Encapsulation section where other sections are compressed
EFI_SECTION_GUID_DEFINED	$0x2$	Encapsulation section used during the build process but not required for execution
EFI_SECTION_DISPOSABLE	$0x3$	Encapsulation section used during the build process but not required for execution
EFI_SECTION_PE32	$0x10$	PE32+ Executable image
EFI_SECTION_PIC	$0x11$	Position-Independent Code
EFI_SECTION_TE	$0x12$	Terse Executable Image
EFI_SECTION_DXE_DEPEX	$0x13$	DXE Dependency Expression
EFI_SECTION_VERSION	$0x14$	Version, Text and numeric
EFI_SECTION_USER_INTERFACE	$0x15$	User-Friendly name of the driver
EFI_SECTION_COMPATIBILITY16	$0x16$	DOS-style 16-bit EXE
EFI_SECTION_FIRMWARE_VOLUME_IMAGE	$0x17$	PI Firmware Volume Image
EFI_SECTION_FREEFORM_SUBTYPE_GUID	$0x18$	Raw data with GUID in header to define format
EFI_SECTION_RAW	$0x19$	Raw data
EFI_SECTION_PEI_DEPEX	$0x1b$	PEI Dependency Expression
EFI_SECTION_SMM_DEPEX	$0x1c$	Leaf section type for determining the dispatch order for an MM Traditional driver in MM Traditional Mode or MM Standalone driver in MM Standalone Mode.

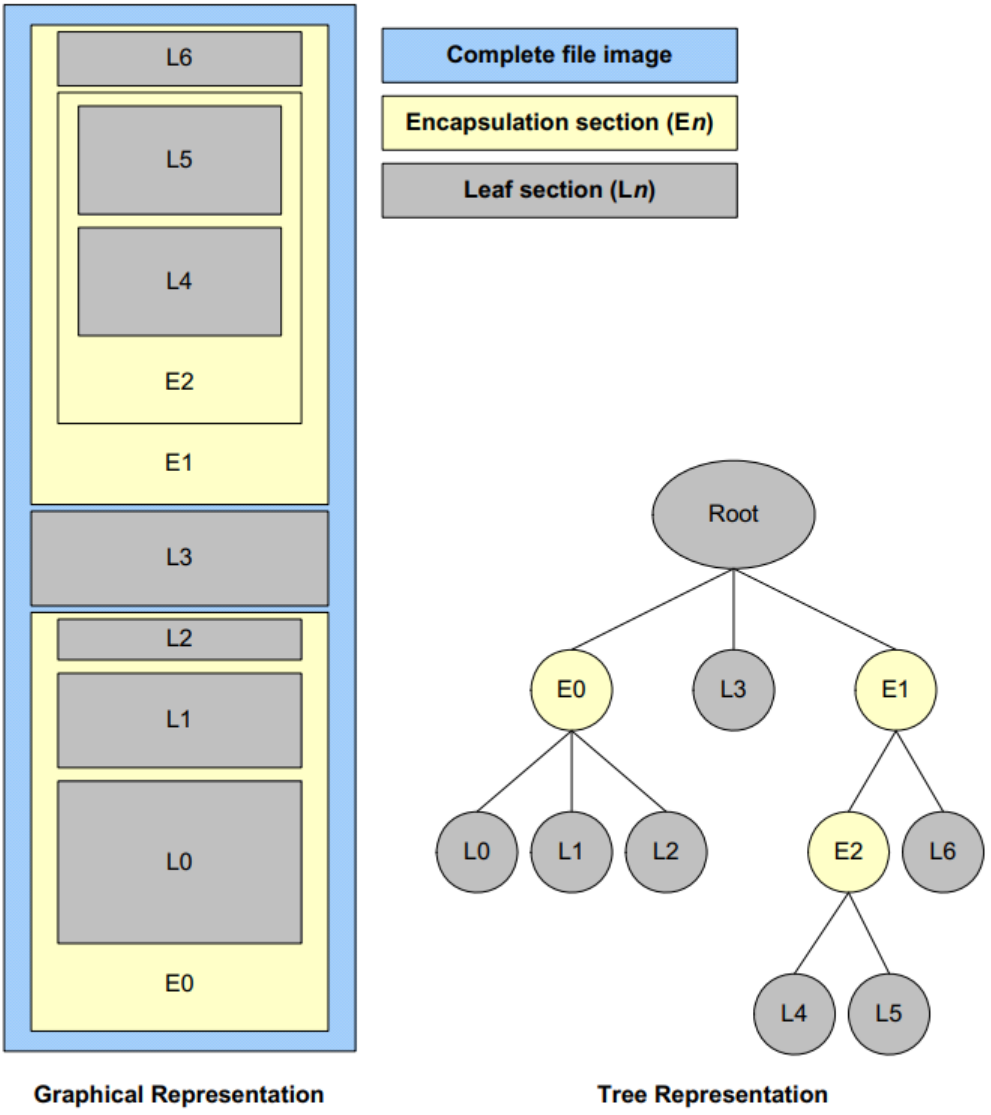

FIGURE 15: Example File System Image

support the standard formats. This section also describes how features of the standard format map into the standard PEI and DXE interfaces.

The standard firmware file and volume format also introduces additional attributes and capabilities that are used to guarantee the integrity of the firmware volume. The standard format is broken into three levels: the firmware volume format, the firmware file system format, and the firmware file format.

The standard firmware volume format (Figure 16) consists of two parts: the firmware volume header and the firmware volume data. The firmware volume header describes all of the attributes specified in “Firmware Volumes” on Table 6. The header also contains a GUID which describes the format of the firmware file system used to organize the firmware volume data. The firmware volume header can support other firmware file systems other than the PI Firmware File System.

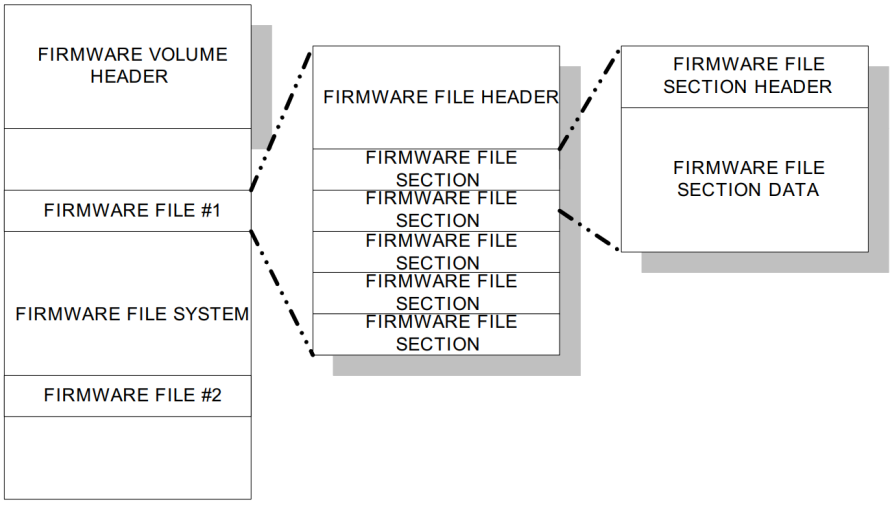

FIGURE 16: The Firmware Volume Format

The PI Firmware File System format describes how firmware files and free space are organized within the firmware volume.

The PI Firmware File format describes how files are organized. The firmware file format consists of two parts: the firmware file header and the firmware file data.

3.7.1 Firmware Volume Format

he PI Architecture Firmware Volume format describes the binary layout of a firmware volume. The firmware volume format consists of a header followed by the firmware volume data. The firmware volume header is described by `EFI_FIRMWARE_VOLUME_HEADER`. The format of the firmware volume data is described by a GUID. Valid files system GUID values are `EFI_FIRMWARE_FILE_SYSTEM2_GUID` and `EFI_FIRMWARE_FILE_SYSTEM3_GUID`.

3.7.2 Firmware File System Format

The PI Architecture Firmware File System is a binary layout of file storage within firmware volumes. It is a flat file system in that there is no provision for any directory hierarchy; all files reside in the root directly. Files are stored end to end without any directory entry to describe which files are present. Parsing the contents of a firmware volume to obtain a listing of files present requires walking the firmware volume from beginning to end.

Firmware File System GUID The PI Architecture firmware volume header contains a data field for the file system GUID. There are two valid FFS file system, the GUID is defined as `EFI_FIRMWARE_FILE_SYSTEM2_GUID` and `EFI_FIRMWARE_FILE_SYSTEM3_GUID`. If the FFS file system is backward compatible with `EFI_FIRMWARE_FILE_SYSTEM2_GUID`

and supports files larger than 16 MB then EFI_FIRMWARE_FILE_SYSTEM3_GUID is used.

Volume Top File A Volume Top File (VTF) is a file that must be located such that the last byte of the file is also the last byte of the firmware volume. Regardless of the file type, a VTF must have the file name GUID of EFI_FFS_VOLUME_TOP_FILE_GUID. Firmware file system driver code must be aware of this GUID and insert a pad file as necessary to guarantee the VTF is located correctly at the top of the firmware volume on write and update operations. File length and alignment requirements must be consistent with the top of volume. Otherwise, a write error occurs and the firmware volume is not modified.

3.8 Firmware File Format (FFS)

All FFS files begin with a header that is aligned on an $8-byte$ boundary with respect to the beginning of the firmware volume. FFS files can contain the following parts:

- Header
- Data

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

If the file contains data, the data immediately follows the header. The format of the data within a file is defined by the Type field in the header, either EFI_FFS_FILE_HEADER or EFI_FFS_FILE_HEADER2.

Figure 17 illustrates the layout of a (typical i.e. EFI_FFS_FILE_HEADER) PI Architecture Firmware File smaller than $16Mb$.

Figure 18 illustrates the layout of a PI Architecture Firmware File larger than $16Mb$.

3.9 Firmware File Section Format

This section describes the standard firmware file section layout. Each section begins with a section header, followed by data defined by the section type. The section headers aligned on 4 byte boundaries relative to the start of the file's image. If padding is required between the end of one section and the beginning of the next to achieve the 4-byte alignment requirement, all padding bytes must be initialized to

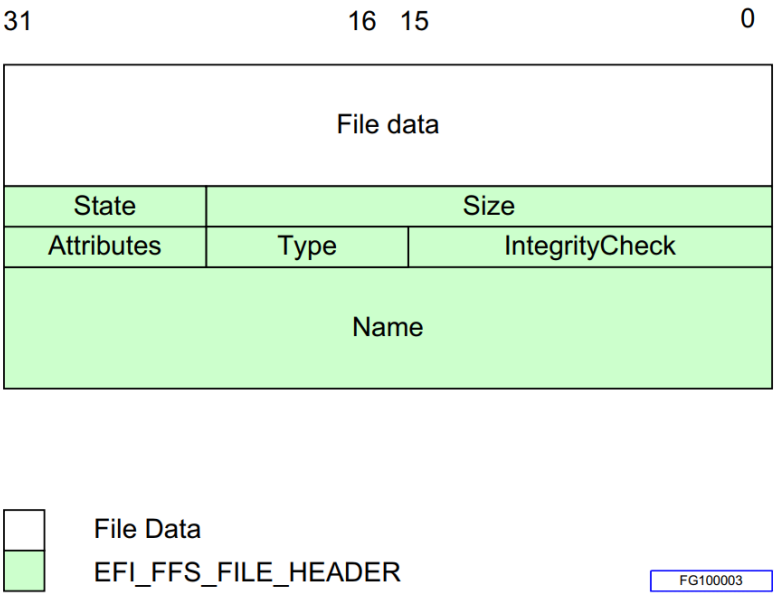

FIGURE 17: Typical FFS File Layout

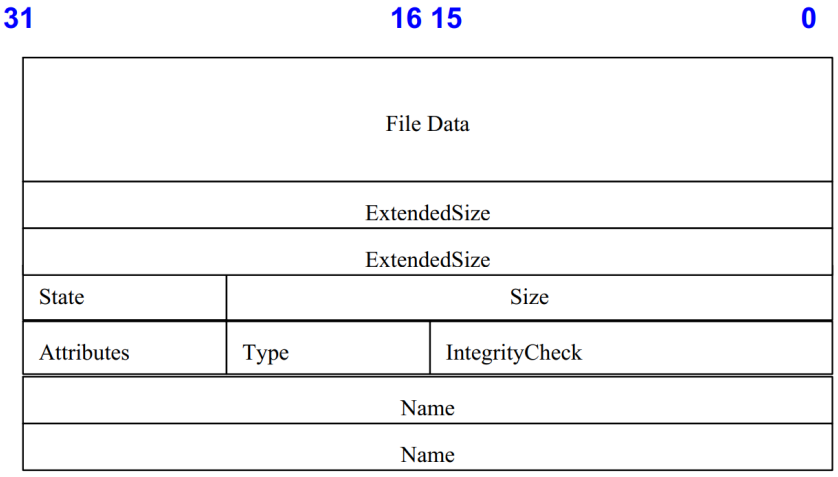

FIGURE 18: File Header 2 layout for files larger than $16Mb$

zero. Many section types are variable in length and are more accurately described as data streams rather than data structures.

Regardless of section type, all section headers begin with a $24-bit$ integer indicating the section size, followed by an 8-bit section type. The format of the remainder of the section header and the section data is defined by the section type. If the section size is $0xFFFFFF$ then the size is defined by a $32-bit$ integer that follows the $32-bit$ section header. Figures 19 and 20 shows the general format of a section.

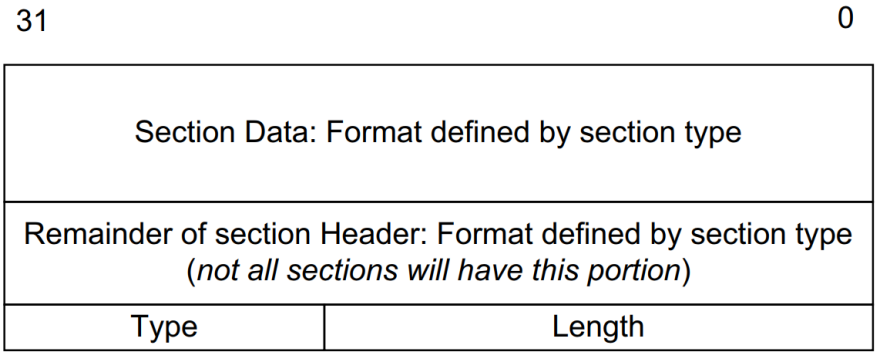

FIGURE 19: Format of a Section below $16Mb$

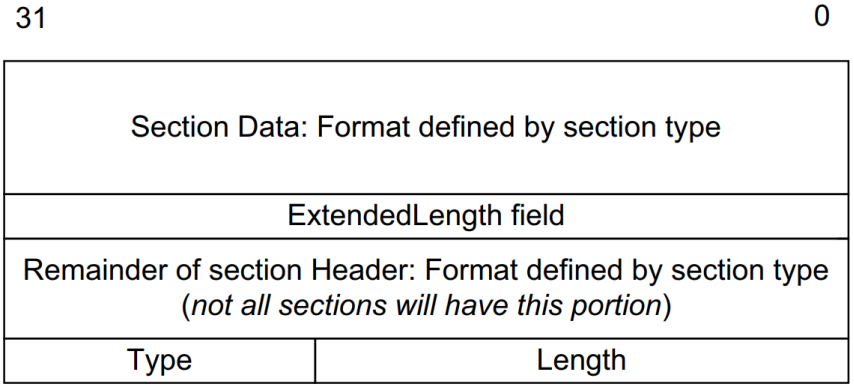

FIGURE 20: Format of a Section using Extended Length field $16Mb$

3.10 File System Initialization

The algorithm 3.10 describes a method of FFS initialization that ensures FFS file corruption can be detected regardless of the cause.

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

The following `FvCheck()` pseudo code must be executed during FFS initialization to avoid file system corruption. If at any point a failure condition is reached, then the firmware volume is corrupted and a crisis recovery is initiated.All FFS files, including files of type `EFI_FV_FILETYPE_FFS_PAD` must be evaluated during file system initialization. It is legal for multiple pad files with this file type to have the same Name field in the file header. No checks for duplicate files should be performed on pad files.

```
// Firmware volume initialization entry point - returns TRUE if FFS driver can use this firmware volume.
BOOLEAN FvCheck(Fv) {
    // first check out firmware volume header
    if (FvHeaderCheck(Fv) == FALSE) {
        FAILURE();// corrupted firmware volume header
    }
    if (!((Fv->FvFileSystemId == EFI_FIRMWARE_FILE_SYSTEM2_GUID) || (Fv->FvFileSystemId == EFI_FIRMWARE_FILE_SYSTEM3_GUID))){
        return (FALSE); // This firmware volume is not formatted with FFS
    }
    // next walk files and verify the FFS is in good shape
    for (FilePtr = FirstFile; Exists(Fv, FilePtr); FilePtr = NextFile(Fv, FilePtr)) {
        if (FileCheck (Fv, FilePtr) != 0) {
            FAILURE(); // inconsistent file system
        }
    }
    if (CheckFreeSpace (Fv, FilePtr) != 0) {
        FAILURE();
    }
    return (TRUE); // this firmware volume can be used by the FFS
    // driver and the file system is OK
}

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

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

// BufferErased - returns TRUE if no bits are set in buffer
```

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

// GetFileState - returns high bit set of state field.
UINT8 GetFileState (Fv, FilePtr) {
    UINT8 FileState;
    UINT8 HighBit;
    FileState = FilePtr->State;
    if (Fv.ErasePolarity != 0) {
        FileState = ~FileState;
    }
    HighBit = 0x80;
    while (HighBit != 0 && (HighBit & FileState) == 0) {
        HighBit = HighBit >> 1;
    }
    return HighBit;
}

// FileCheck - returns TRUE if the file is OK
BOOLEAN FileCheck (Fv, FilePtr) {
    switch (GetFileState (Fv, FilePtr)) {
        case EFI_FILE_HEADER_CONSTRUCTION:
            SetHeaderBit (Fv, FilePtr, EFI_FILE_HEADER_INVALID);
            break;
        case EFI_FILE_HEADER_VALID:
            if (VerifyHeaderChecksum (FilePtr) != TRUE) {
```

```
73                  return (FALSE);
74              }
75              SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
76              Break;
77          case EFI_FILE_DATA_VALID:
78              if (VerifyHeaderChecksum (FilePtr) != TRUE) {
79                  return (FALSE);
80              }
81              if (VerifyFileChecksum (FilePtr) != TRUE) {
82                  return (FALSE);
83              }
84              if (DuplicateFileExists (Fv, FilePtr,
85                  EFI_FILE_DATA_VALID) != NULL) {
86                  return (FALSE);
87              }
88              break;
89          case EFI_FILE_MARKED_FOR_UPDATE:
90              if (VerifyHeaderChecksum (FilePtr) != TRUE) {
91                  return (FALSE);
92              }
93              if (VerifyFileChecksum (FilePtr) != TRUE) {
94                  return (FALSE);
95              }
96              if (FilePtr->State & EFI_FILE_DATA_VALID) == 0) {
97                  return (FALSE);
98              }
99              if (FilePtr->Type == EFI_FV_FILETYPE_FFS_PAD) {
100                 SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
101             }
102             else {
103                 if (DuplicateFileExists (Fv, FilePtr,
                     ↪  EFI_FILE_DATA_VALID)) {
104                     SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
105                 }
106                 else {
107                     if (Fv->Attributes & EFI_FVB_STICKY_WRITE) {
108                         CopyFile (Fv, FilePtr);
109                         SetHeaderBit (Fv, FilePtr, EFI_FILE_DELETED);
```

```
110                     }
111                     else {
112                         ClearHeaderBit (Fv, FilePtr,
                            ↪ EFI_FILE_MARKED_FOR_UPDATE);
113                     }
114                 }
115             }
116             break;
117         case EFI_FILE_DELETED:
118             if (VerifyHeaderChecksum (FilePtr) != TRUE) {
119                 return (FALSE);
120             }
121             if (VerifyFileChecksum (FilePtr) != TRUE) {
122                 return (FALSE);
123             }
124             break;
125         case EFI_FILE_HEADER_INVALID:
126             break;
127 }
128     return (TRUE);
129 }
130
131 // FFS_FILE_PTR * DuplicateFileExists (Fv, FilePtr, StateBit)
132 // This function searches the firmware volume for another occurrence
133 // of the file described by FilePtr, in which the duplicate files
134 // high state bit that is set is defined by the parameter StateBit.
135 // It returns a pointer to a duplicate file if it exists and NULL
136 // if it does not. If the file type is EFI_FV_FILETYPE_FFS_PAD
137 // then NULL must be returned.
138
139 // CopyFile (Fv, FilePtr)
140 // The purpose of this function is to clear the
141 // EFI_FILE_MARKED_FOR_UPDATE bit from FilePtr->State
142 // in firmware volumes that have EFI_FVB_STICKY_WRITE == TRUE.
143 // The file is copied exactly header and all, except that the
144 // EFI_FILE_MARKED_FOR_UPDATE bit in the file header of the
145 // new file is clear.
146 // VerifyHeaderChecksum (FilePtr)
```

```
147 // The purpose of this function is to verify the file header
148 // sums to zero. See IntegrityCheck.Checksum.Header definition
149 // for details.
150 // VerifyFileChecksum (FilePtr)
151 // The purpose of this function is to verify the file integrity
152 // check. See IntegrityCheck.Checksum.File definition for details.
```

3.11 Traversal and Access to Files

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

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

```
1 // Screen files to ensure we get the right one in case
2 // of an inconsistency.
3 FFS_FILE_PTR EarlyFfsUpdateCheck(FFS_FILE_PTR * FilePtr) {
4   FFS_FILE_PTR * FilePtr2;
5     if (VerifyHeaderChecksum (FilePtr) != TRUE) {
6     return (FALSE);
7   }
8   if (VerifyFileChecksum (FilePtr) != TRUE) {
```

```
9       return (FALSE);
10    }
11    switch (GetFileState (Fv, FilePtr)) {
12      case EFI_FILE_DATA_VALID:
13        return (FilePtr);
14        break;
15      case EFI_FILE_MARKED_FOR_UPDATE:
16        FilePtr2 = DuplicateFileExists (Fv, FilePtr,
17        EFI_FILE_DATA_VALID);
18        if (FilePtr2 != NULL) {
19          if (VerifyHeaderChecksum (FilePtr) != TRUE) {
20            return (FALSE);
21          }
22          if (VerifyFileChecksum (FilePtr) != TRUE) {
23            return (FALSE);
24          }
25          return (FilePtr2);
26        } else {
27          return (FilePtr);
28        }
29        break;
30    }
31  }
```

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

3.12 File Integrity and State

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

- General failure
- Erase failure

- Write failure

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

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

All of these failures are detectable during FFS initialization, and, depending on the nature of the failure, many recovery strategies are possible. Careful sequencing of the State bits during normal file transitions is sufficient to enable subsequent detection of write failures. However, the State bits alone are not sufficient to detect all occurrences of general and/or erase failures. These types of failures require additional support, which is enabled with the file header `IntegrityCheck` field. For sample code that provides a method of FFS initialization that can detect FFS file corruption, regardless of the cause, see “File System Initialization” on Section 3.10.

4. System Management Mode (SMM)

4.1 Overview

On IA-32 processors, System Management Mode (SMM) is a mode of operation which is distinct from the flat model, protected mode operation of the DXE and PEI phases. It is defined as a real-mode environment with 32-bit data access and its activated in effect to an interrupt type or using the System Management Interrupt (SMI) pin. Note that SMM is OS-transparent mode of operation and is distinct operational mode and also it coexist within and OS runtime.

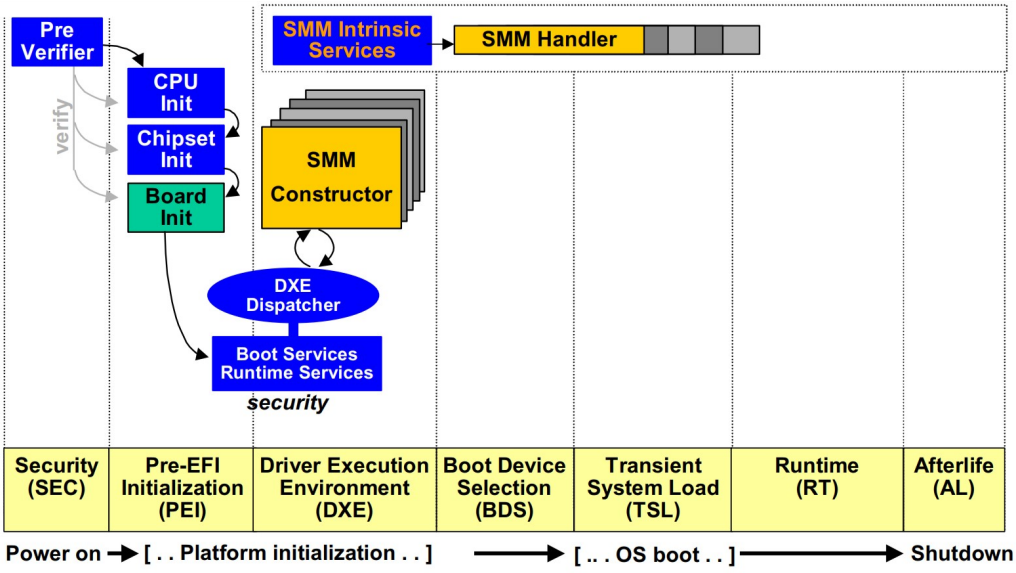

FIGURE 21: SMM Framework Architecture

4.2 System Management System Table (SMST)

System Management System Table (SMST) is core mechanism of SMM handler to pass information and enabling activity.

SMST table provides access to the SMST-based services, known as SMM Services. Driver can only use SMM services while executing within the SMM context. `EFI_SMM_BASE_PROTOCOL.G` service used to discover the address of SMST.

SMST is a set of capabilities exported for use by any driver that is loaded into SMRAM. It's akin to the EFI System Table, where it is a fixed set of services and data, by design, and does not acknowledge to the extensibility of an EFI protocol interface.

SMM infrastructure component of Framework provides SMST, which manages:

- Dispatching drivers in SMM
- Allocations of SMRAM
- Transitioning the framework into and out of the respective SMM of the processor

4.3 SMM and Available Services

4.3.1 SMM Services

The model of SMM in the Framework will have constraints similar to those of EFI runtime drivers. Specifically, the dispatch of drivers in SMM will not be able to use core protocol services. There will be SMST-based services, called SMM Services, that the drivers can access using an SMM equivalent of the EFI System Table, but the core protocol services will not necessarily be available during runtime. Instead, the full collection of EFI Boot Services and EFI Runtime Services are available only during the driver load or "constructor" phase. This constructor visibility is useful in that the SMM driver can leverage the rich set of EFI services to do the following:

- Marshall interfaces to other EFI services.
- Discover EFI protocols that are published by peer SMM drivers during their constructor phases.

This design makes the EFI protocol database useful to these drivers while outside of SMM and during their initial load within SMM. The SMST-based services that are available include the following:

- A minimal, blocking variant of the device I/O protocol
- A memory allocator from SMM memory

These services are exposed by entries in the System Management System Table (SMST)

4.3.2 SMM Library (SMLib Services)

Additional services in the SMM Library (SMLib) are exposed as conventional EFI protocols that are located during the constructor phase of the SMM driver in SMM. For example, the status code equivalent in SMM is simply an EFI protocol whose interface references an SMM-based driver's service. Other SMM drivers locate this SMM-based status code and can use it during runtime to emit error or progress information.

4.4 SMM Drivers

4.4.1 Loading Drivers into SMM

Driver loading modal into SMM is that the DXE SMM runtime driver contains a dependency expression that at least have the `EFI_SMM_BASE_PROTOCOL`. This dependency is essential because the DXE runtime driver that is planned for SMM will use the `EFI_SMM_BASE_PROTOCOL` to reload itself into SMM and re-execute its entry point in SMM. Also, other SMM-loaded protocols allowed to be situated in the dependency expression of a given SMM DXE runtime driver. The principle of the DXE Dispatcher, verifying if the GUIDs for the protocols that are exist in the protocol database can then be used to identify if the driver can be loaded.

Once loaded into SMM, the DXE SMM runtime driver can utilize a very minor set of services. While in its constructor entry point, the driver can use EFI Boot Services as it runs in the 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 effect to an SMI pin activation or an SMI based message from outside of SMM Code (i.e. a boot service, runtime agent)

After this **constructor** phase in SMM, the SMM driver should not depend upon any other boot services because the operational mode of execution can migrate away from these services (the `ExitBootServices()` call is asynchronous to calling the SMM infrastructure code). Several EFI Runtime Services can have the bulk of their processing shifted into SMM, and the runtime visible portion would simply be a proxy that uses the `EFI_SMM_BASE_PROTOCOL` to callback into SMM to carry out the services. Having a proxy allows for a model of sharing error handling code, such as flash access services, along with runtime code, such as the EFI Runtime Services `GetVariable()` or `SetVariable()`.

4.4.2 IA-32 SMM Drivers

In SMM the IA-32 runtime drivers are not callable because of the `SetVirtualAddress()` action that is performed upon the image. As such, code that needs to be accessible between SMM and EFI runtime needs to migrate into SMM.

4.4.3 Itanium® Processor Family SMM Drivers

From Platform Management Interrupt (PMI) the runtime drivers for the Itanium® processor family are callable as each is a variant of position-independent code (PIC) runtime driver.

4.5 SMM Protocols

System Architecture of SMM broke in to below two parts:

- SMM Base Protocol - published by a processor and its responsible for:
  - To initialize the state of processor
  - Registration of the handlers
- SMM Access Protocol - interprets the specific enable and locking techniques that an IA-32 memory controller might support during execution in SMM. (Not needed for Itanium® processor family)

4.5.1 SMM Protocols for IA-32

Figure 22 shows the SMM protocols which are published for an IA-32 system.

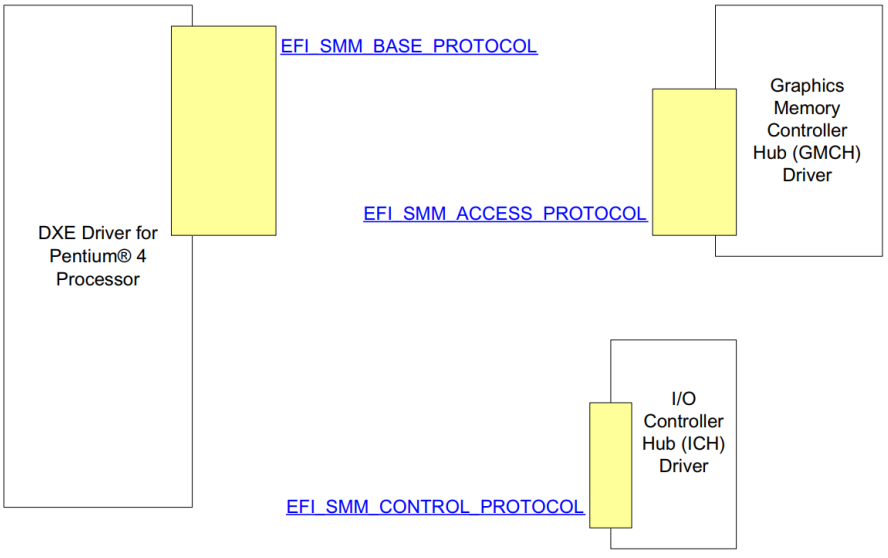

FIGURE 22: Protocols Published for IA-32 Systems

4.5.2 SMM Protocols for Itanium®-Based Systems

Figure 23 shows the SMM protocols which are published for an Itanium®-Based system.

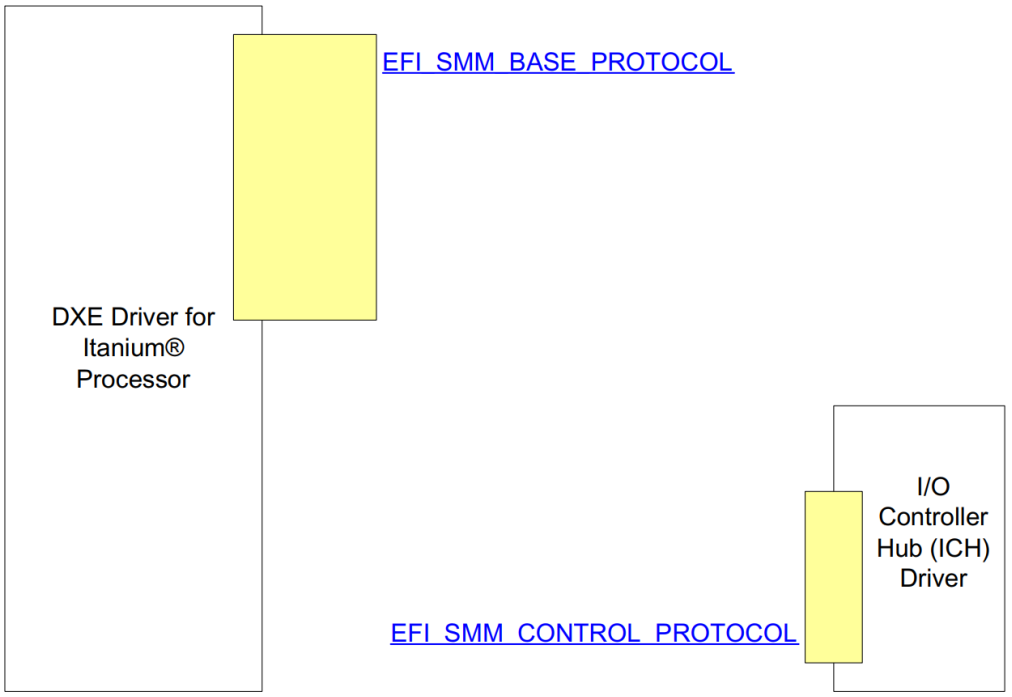

FIGURE 23: Protocols Published for Itanium®-Based Systems

4.6 SMM Infrastructure Code and Dispatcher

SMM Infrastructure Code centers within the SMM Dispatcher. Role of SMM Dispatcher is to hand over the control to the SMM handlers in an orderly manner. SMM Infrastructure Code assists to drive SMM to SMM communication. SMM handlers are PE32+ images that have and image type of `EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER`.

4.7 Initializing SMM Phase

The SMM driver for the Framework is essentially a enrollment vehicle for dispatching drivers in response to the:

- System Management Interrupts for IA-32
- Platform Management Interrupts (PMIs) for Itanium® processor family

4.8 Relation of System Management RAM (SMRAM) to main memory

Figure 24 shows relationship between SMRAM and main memory in IA-32.

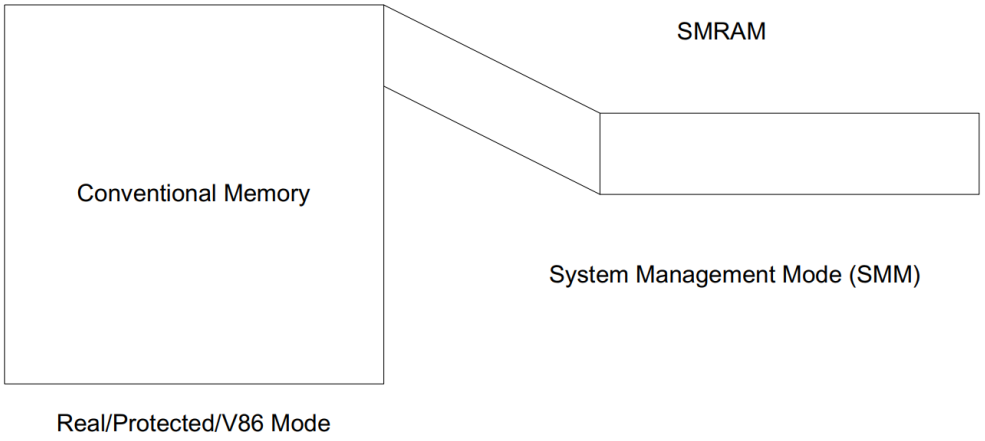

FIGURE 24: SMRAM relationship to main memory

4.9 Processor Execution Mode

SMM is entered asynchronously to the main flow of program. SMM was originally designed to be clear to the OS and provides a transparent power management facility.

Preboot agents are responsible to initiate alternate uses of SMM which are:

- Workarounds for chipset errata
- Error logging
- Platform security

A SMI can be entered by energizing either the SMI logic pin on the baseboard dedicated or using the local APIC.

Itanium® architecture has no separated processor mode for the manageability interruption, but it supports Platform Management Interrupt (PMI), which is a maskable interruption. Also, another way to enter PMI is using a message on local Streamlined Advanced Programmable Interrupt Controller (SAPIC).

This architecture describes a mechanism for loading modules of needful code that substantiate the functionality mentioned above. The instantiation of protocol that enables the loading of handler images runs in normal boot-services memory. Only handler need to run in the SMRAM.

4.10 Access to Platform Resources

As a policy outcome, execution of SMM handlers is logically precluded from accessing traditional memory resources. Hence, there is no ease binding technique through a call or trap interface to leverage services in the preempted, non-SMM state.

Besides, SMM Services - the library of service, supports a subset of the core EFI services, i.e. memory allocation, device I/O protocol, and others. Also, SMM driver execution mode has the same structure as the EFI baseline - namely a components that runs in boot services mode and that can perhaps execute in runtime. Another mechanism occurs using an unregister event when `ExitBootServices()` is invoked.

5. Proposed Work

In general to generate BIOS image (*.rom file), compilation of XYZ.c (source code) has to be done, this compilation not only involves compilation of DXE driver, PEI driver, EFI Application but also includes pre-processing checks, compression of raw files which takes huge amount of time depending on the system configuration. Implementation of this project aids in reduction of this compilation time.

5.1 Stack holders

The proposed work is applicable but not limited to below stack holders:

- **BIOS development team** : main development group in contributing BIOS firmware, this is the only stockholder who are having access to the BIOS development environment and access to the source code of the complete BIOS firmware
- **Validation team** : performs various validation on developed BIOS image
- **Automation team** : brings various integration and validation automation to module(s)
- Other Development team who wishes to ease the debugging process

5.2 Issues

The Proposed work is capable of mitigating below issues:

- Generation of BIOS image - includes compilation of whole source code
- Time complexity - took enormous amount of time to generate the BIOS image
- Accessing and modifying BIOS Setup Option(s) remotely
- Firmware Flashing of BIOS remotely
- Updating CPU microcode
- Summarizing changes among BIOS image
- Avoiding exposing the source code support for OEM to fill their OEM information
- Avoid setting of BIOS development platform for stack holders which are not meant to be the BIOS developer
- Runtime BIOS Support for temporary UEFI variable creation

5.3 Requirements

5.3.1 Software Requirements

- Visual C/C++ binaries
- Python 3
- Visual Studio Code (IDE)
- Memory Access Interface - supported mechanism to communicate over target memory

5.4 Development Process of Modules

Framework development process is driven by implementation of independent modules which can serve functionality and having flexibility to integration to the framework.

5.5 Module 1: Processing Firmware individually

To process apply the whole firmware changes individually for BIOS, signing is require to be performed for individual Firmware before stitching it to the BIOS binary in the valid structure.

5.5.1 Primary Goals of the module

- Remove other Intellectual Property's dependency (IP dependency) during firmware loading
- IP Subsystem :
  - Loader and Verifier
  - IP is always consumer
- Signature verification using SHA hash algorithm and should be ease support for adding new algorithmic support as needed.
- Should support hardware based and software based verification support modifying memory requirements for given IP without impacting eco-system
- Prevent common security threats
- Allow easier OEM adoption and modification based on the respective design
- Reusability/Portability of design across many IPs

- Generic design which supports any new IP integration

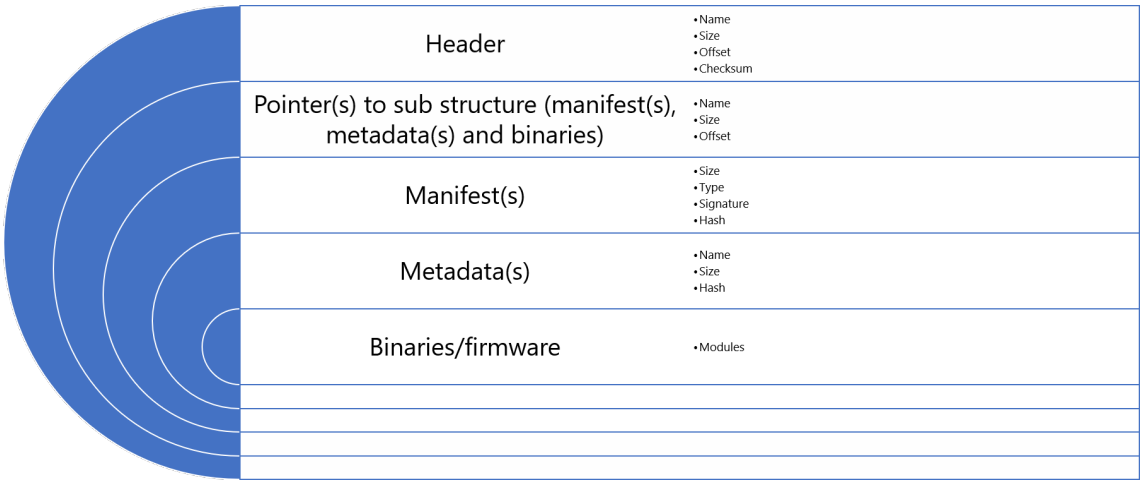

FIGURE 25: Proposed Structure for firmware signing

Note: Due to Confidentiality other details of this module is not to be disclosed such as flow diagram, structures or pseudo code.

5.6 Module 2: Setup Knob modification

5.6.1 Processing Unsigned debug BIOS

Before Releasing the BIOS firmware for public use, those are signed for security and integrity purpose, however the debug BIOS which are used Pre-release to test and verify all the functional features until all the requirements are met.

Every SoC system which are under test known is SUT are configured in such a way that it supports debug BIOS. The proposed framework is designed to simulate the process of SUT in terms of processing BIOS binary similar to SUT performs it after flashing BIOS firmware on SoC.

Processing the debug BIOS can be classified in to two ways:

1. Applying changes directly to the SUT
2. Applying changes on to the BIOS image

At the high level the flow for both the above classification remains the same but will be differentiated at the backend support. An additional driver is attached with BIOS firmware to aid the framework to be able to apply changes directly to the SUT.

5.6.2 Flow of the module

Figure 26 describes the flow of setup knobs modification on the System Under Test (SUT).

The iteration of the development could be reduce in two ways:

1. Processing Debug/Unsigned BIOS in section 5.6.1
2. Processing Firmware individually in section 5.5

5.6.3 Outcome of Module

- The module is capable of cross platform usage.
- The module can work with all the platform binary and SUT.
- A communication bridge as a driver in BIOS firmware to aid the framework run directly on SUT is implemented.
- Generic solution is provided for end-user while running any of the classification listed in 5.6.1.
- Simulating the information from system or binary image is provided as native GUI application.
- Real time sync with simulation framework is supported.
- Seamless Integration of any new features or modules in framework is made possible.

5.6.4 Framework Proof of Concept and working demo

As a PoC for the framework, this section shows snapshots of the working module to mimic the setup options of BIOS, however as a simulation framework, it also provides quite more features which are not available in the actual BIOS due to memory limitation.

Figure 27 shows the prompt asked to user to select basic configurations before launching the module of framework. Configurations available to select are:

- Working Mode (options to be selected as in figure 28)
  - `online` - to work on SUT and require to select valid access method for online mode from menu
  - `offline` - to work on BIOS binary

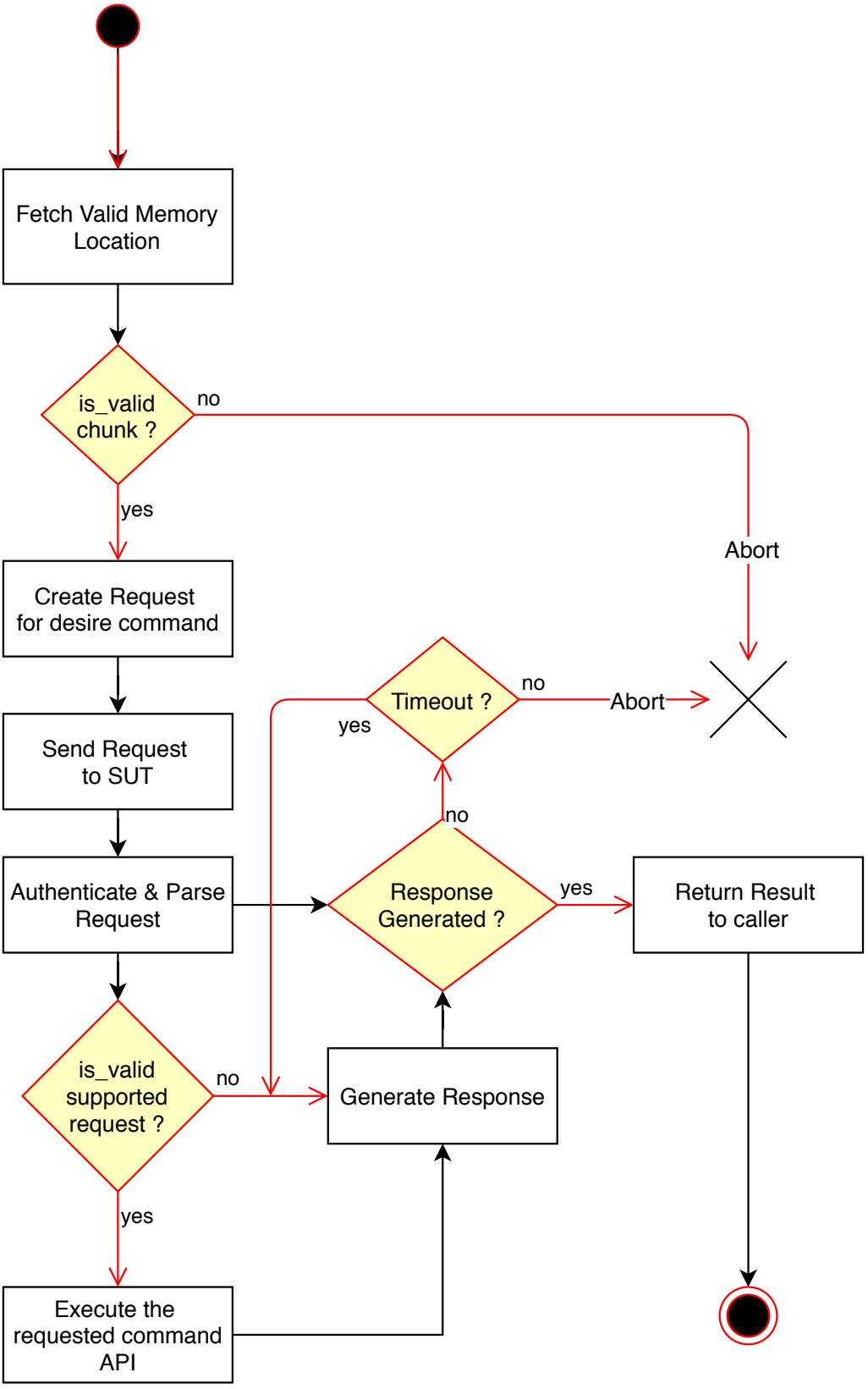

FIGURE 26: Flow of Setup Knobs Modification

- Access Method - selecting valid access method for working on SUT
- Publish all? - Boolean options to decide whether to evaluate DEPEX or not.

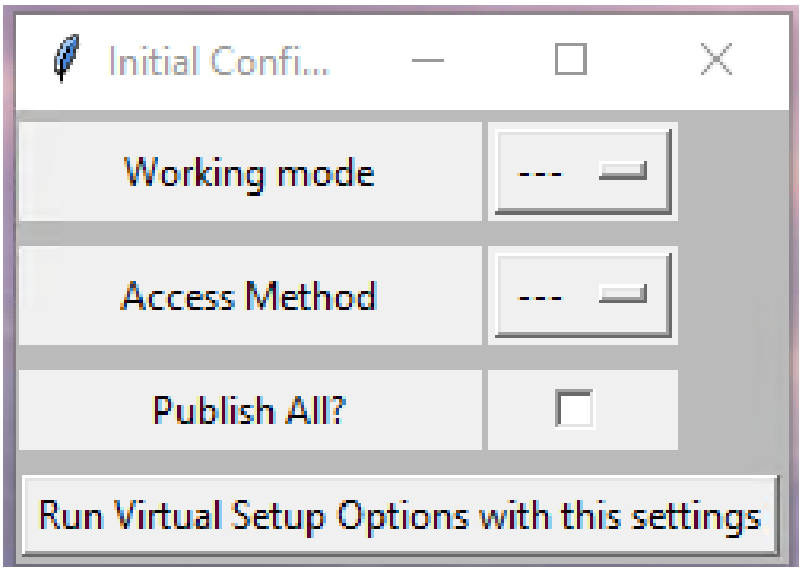

FIGURE 27: Menu to Select initial configuration for work

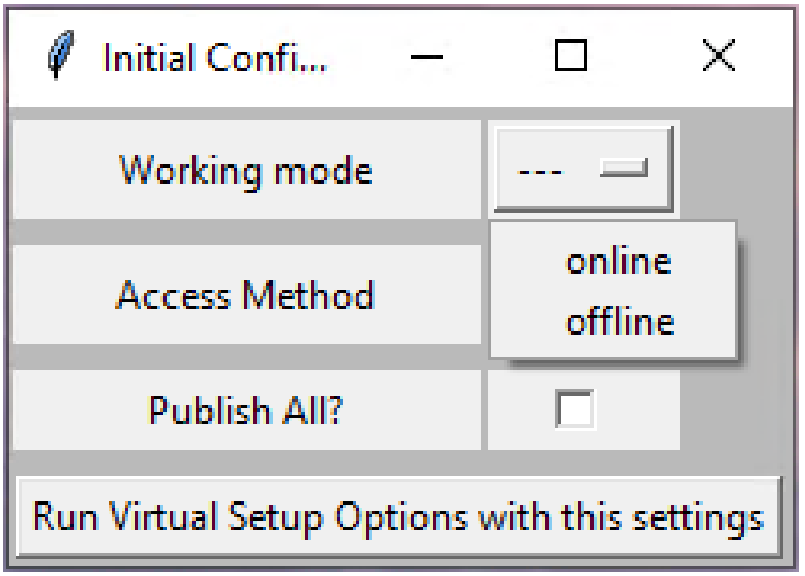

FIGURE 28: Available work mode for the system: Online and Offline

Table 10 describes the interpretation of each button action on specific condition as remarks if applicable

TABLE 10: Interpretation of buttons on Virtual Setup Page GUI

Button	Interpretation
Push Changes	Apply changes to system if online mode else apply changes to 'bin' file
View Changes	View saved changes in new window
Exit	Exit the GUI
Reload	Reload the GUI
Discard Changes	Discard any change made, any value if modified are restored to current value
Load Defaults	Restore to default values and revert any changes made

5.7 Module 3: Parsing

Figure 29 represents the overview of the BIOS as a File system which is interpreted and parsed from the BIOS image. Detail architecture of the same is explained in Section 3.

5.7.1 Flow of the module

Figure 30 describes the flow of the Parsing module. The Initial part is performed by user who is responsible to select valid memory interface to work. Note that some memory interface are supported by the module which requires additional hardware and software setup which are considered to be the part of dependency of interface itself which is not in the scope of the module.

When User select valid Interface the module will determine whether user is on Target SUT or on the local BIOS image. If user is working on SUT with valid memory interface and privileges then BIOS image will be parsed from the memory.

As on both the cases BIOS Image is available to act on, the module will start the parsing of the BIOS image as interpretation described in Figure 29. It parses All the valid firmware volumes only till the end of BIOS image (skips the free space or firmware volumes with invalid signature and GUID). Decompression of file system under the firmware volume if any is handled by the module too, for the decompression of file system it uses the binary for decompression technique available to public i.e. lzma, tianocore, brotli etc.

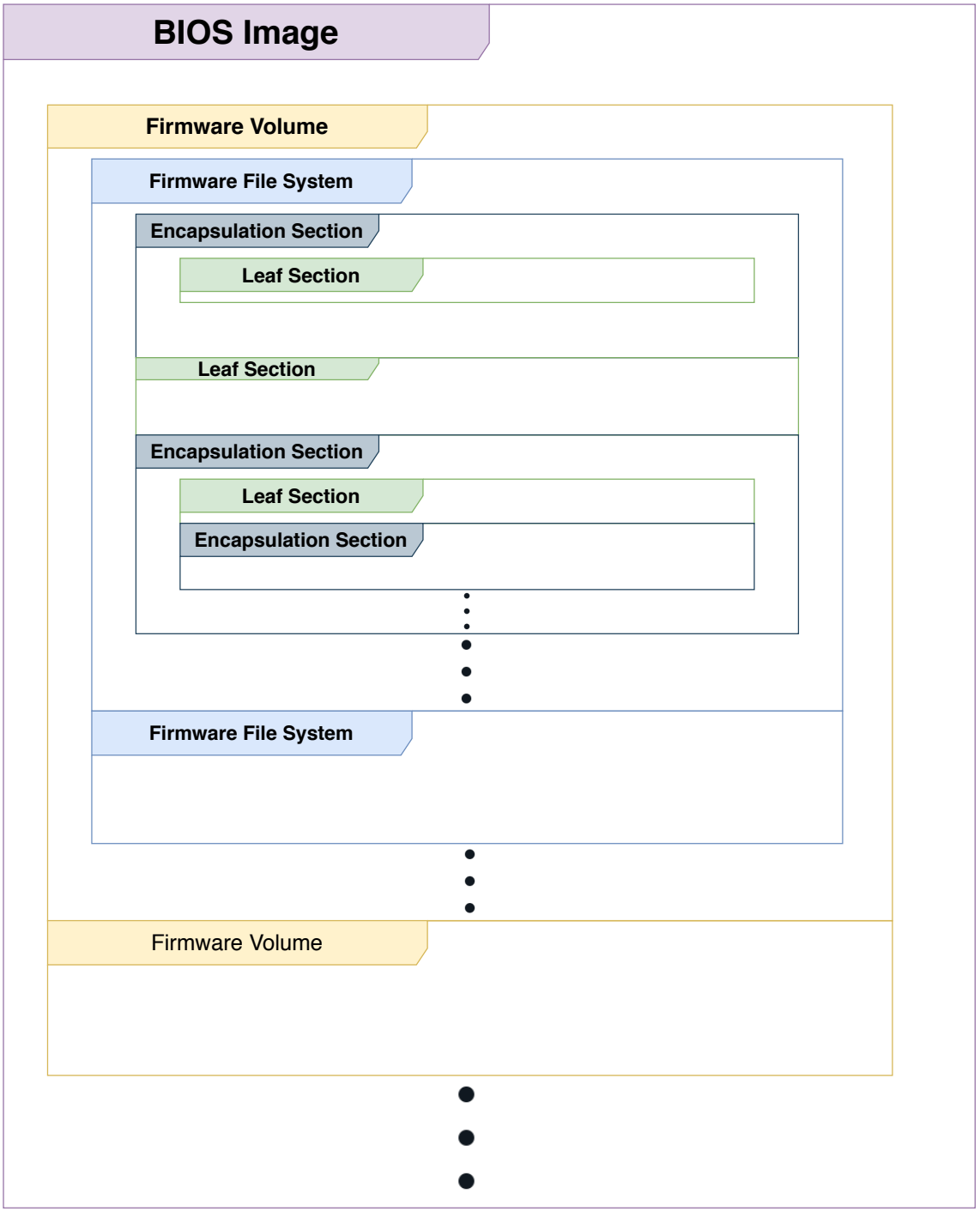

FIGURE 29: Overview of BIOS image as a File System

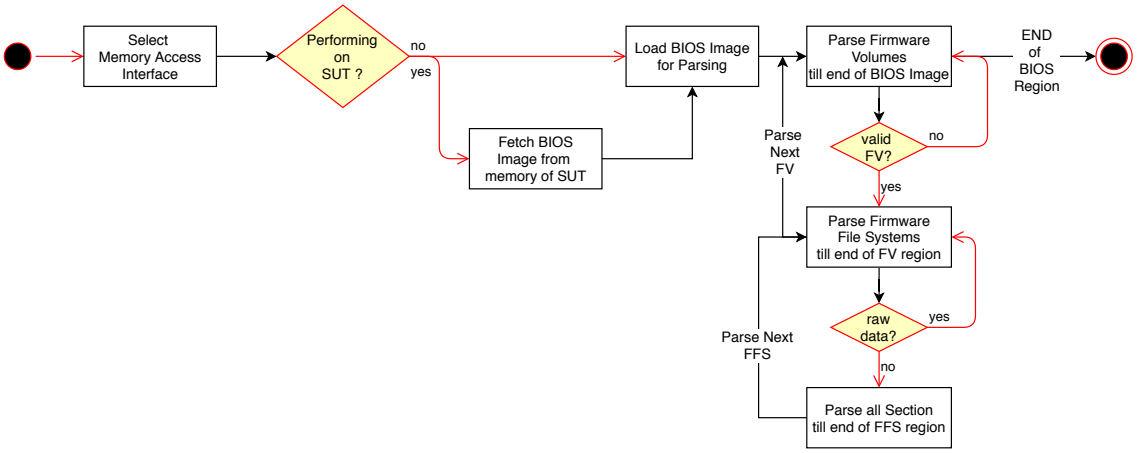

FIGURE 30: Flow of Parser

5.7.2 Outcome of Module

- Human Readable interpretation of BIOS image
- Possible to debug the BIOS via setup knobs comparison
- Lookup of order of the module in BIOS image as readable file system
- Verification of integration of module via GUID
- Extracting and storing file system or module of BIOS image by GUID
- Summarizing changes of two BIOS image

5.8 Module 4: Runtime UEFI variable Creation

Each variable in BIOS has a scope for each variable where Runtime support is one of the attribute, to simply state the run time variable one can interpret it as the variable which will be available during and after the completion boot flow (while OS is running). Such a variable require special access mechanism, which is carried out by the System Management mode SMM described in Section 4..

Earlier Challenges are described as below:

- Providing and maintaining native driver support from BIOS for creation of UEFI variable
- Setting of Build environment for non-BIOS development team

Note: As all the variable created at runtime the scope of such variable are limited to the flashing of the BIOS. i.e. when BIOS is flashed/re-flashed or updated, those variable won't be available on the SUT.

5.8.1 Flow of the module

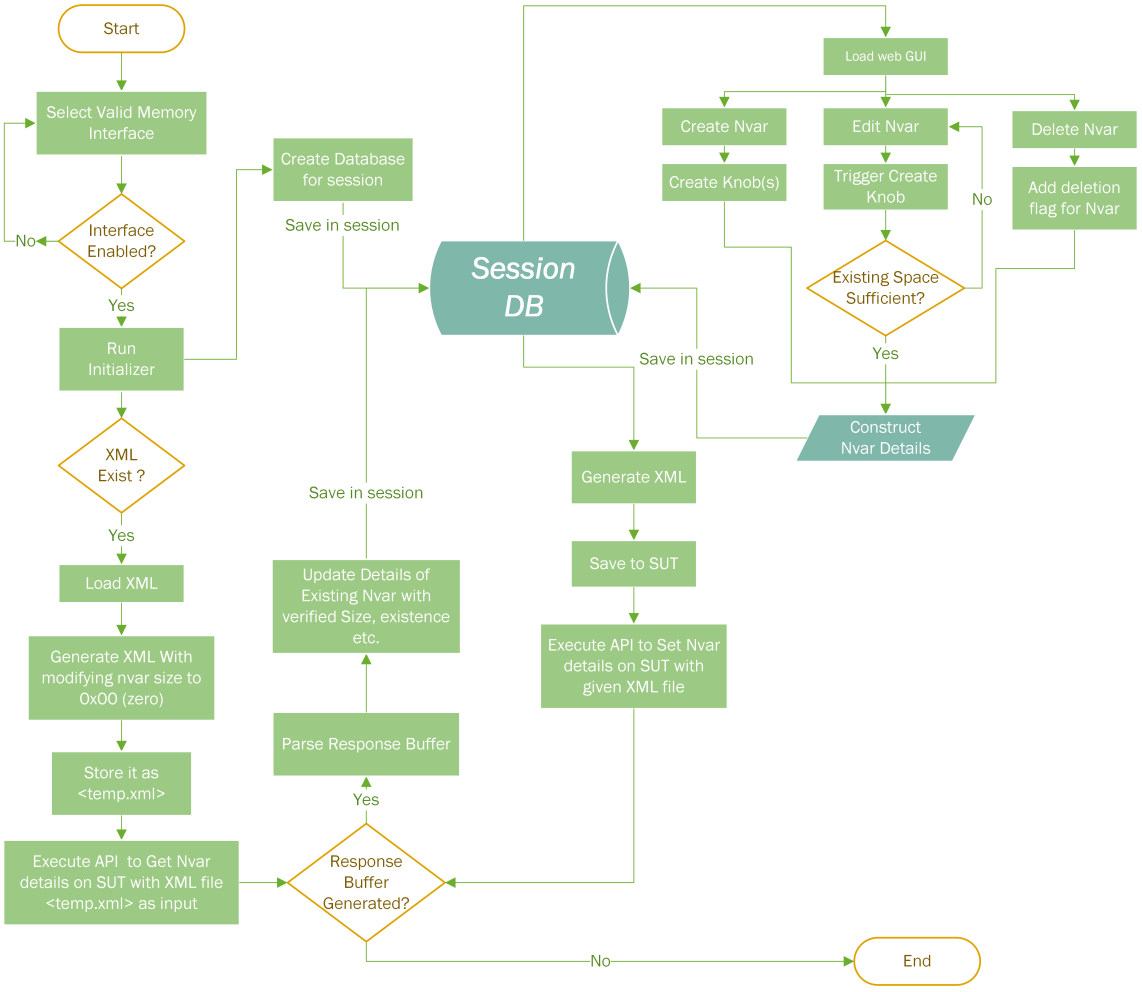

FIGURE 31: Flow of Nvar Web GUI

6. Future Scope of Work

Few implementation modules of Section 5. are not well developed at production launch which a slight modification and standard checks have to be performed to make the modules qualify for production level. Also as the release of production other stuff to be maintained is user guide, FAQs and other "how-to" articles to help out others to ease in using the framework.

Along with the enhancing of existing modules there will still be exercise to analyze existing system to explore more use cases which are taking a longer time for every build iteration for the system.

Few of the possible use cases to study and decide the feasibility of implementation would be:

- Development and testing of individual driver component rather than building the whole BIOS image
- AI powered Search Engine to enhance the findings of FAQs for relevant existing queries and articles
- Automating the initial BIOS Environment Setup
- Platform independent easy installation setup for the framework

Glossary

ACPI Advanced Configuration and Power Interface . iii, v, viii, 1, 5, 6

BDS Boot Device Selection . vi, 20, 25

BIOS Basic Input Output System . iii, v, 1, 2, 5

CSME Converged Security and Mobility Engine . 61

DEPEX Dependency Expression . 25, 54

DXE Driver Execution Environment . vi, 20, 25, 30

EDK II Extensible Firmware Interface Developer Kit II . 18, 21

FFS Firmware File System . vi, 30, 31, 39

FV Firmware Volume . vi, 25, 30, 31

GOP Graphics Output Protocol . v, 11, 14, 15

IFWI Integrated Firmware Image . 27, 28

IP Intellectual Property . iii, 52

OS Operating System . 1, 2

PCI Peripheral Component Interconnect . iii, 1

PCIe Peripheral Component Interconnect Express . v, 1, 8, 12

PEI Pre-EFI Initialization . vi, 20, 22, 23, 27, 30

PI Platform Initialization . vi, viii, 18, 20, 21, 30

PMI Platform Management Interrupt . 47, 50

PoC Proof of Concept . 54, 61

SAPIC Streamlined Advanced Programmable Interrupt Controller . 50

SEC Security . vi, 20, 21

SHA Secure Hash Algorithm . 52

SMI System Management Interrupt . 45, 50

SMM System Management Mode . vi, 45, 59

SMRAM System Management Random Access Memory . 50

SMST System Management System Table . vi, 45

SoC System on a Chip . iii, 1, 53

SUT System Under Test . 53, 54, 58

UEFI Unified Extensible Firmware Interface . v, vi, 1–5, 16, 18