IP Independent Generic Framework to accelerate Development Process Iteration

Submitted by

Gahan Saraiya

18MCEC10

Department of Computer Science & Engineering,

Institute of Technology,

Nirma University, Ahmedabad,

Gujarat - 382481, India.

May, 2020

IP Independent Generic Framework to accelerate Development Process Iteration

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 ***IP Independent Generic Framework to accelerate Development Process Iteration*** 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 ***IP Independent Generic Framework to accelerate Development Process Iteration*** 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 bona-fide work carried out under my supervision. The dissertation fulfills the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission. The contents of this dissertation have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certified.

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

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

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

Dr. R. N. Patel
I/C 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). Section 1.1 describes all the basic information required on the Intel SoC.

Section 2.1 involves the design of the Basic Boot Flow of the BIOS followed by Section 2.2 and Section 2.3 explains the architecture and protocols which are the concept used to build the proposed framework which is described under Section 3.1 to aid the development and debugging iteration for various stakeholders including but not limited to BIOS Developers, Validation Engineers, Automation team.

The framework is designed and implemented to aid the development process by eliminating longer duration of common debugging steps and providing a sophisticated way to build and test the various scenarios includes but not limited to Setup Options, Firmware Flashing, UEFI Variable Creation.

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

Chapter 1

Introduction

1.1 Introduction

Intel System on a Chip (SoC) [**intel-soc**] features a new set of Intel Uncore Intellectual Property (IP) for every generation. Section 1.1 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 2.1 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 2.2. Proposed work to reducing the process of build iteration described in Section 3.1.

1.1.1 Uncore IP

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 communicating system is highly-developed via PCIe for use the transfer data between the host and the peripheral devices. Intel developed the hardware interface which allows the connection among the external peripherals to a computer called Thunderbolt. This interface not only has "PCI Express (PCIe)" and DisplayPort (DP) combined into two serial signals but additionally provides DC power also, bundled in just one cable. "Graphics Processing Engine (GPE)", "Integrated graphics", "shared graphics solutions", "integrated graphics processors (IGP)" or

Table 1.1: Comparison of Legacy BIOS and UEFI

	Legacy BIOS	EFI
Programming Language used	Assembly Language	C Language (99%)
Resources	Interrupt Hardcode Memory Access hardcore Input/Output Access	Divers, Handlers and Protocols
Processor Type	$x86\ 16-bit$	CPU Protects Mode
Expand	Interrupt through hook	Driver to be loaded
OS Communication Bridge	via ACPI	through runtime driver
3^{rd} Party ISV & IHV	Support Bas	Ease of Support and for Cross-Platforms

"unified memory architecture(UMA)" utilize a portion of a memory of computer system instead of having dedicated graphics memory. GPEs can be integrated onto the motherboard as part of the chipset. Guest virtual machines use "Virtual Technology for Directed-IO (Vt-d)", an "input/output memory management unit(IOMMU)", to directly use peripheral devices, such as hard-drive controllers, accelerated graphics cards and Ethernet, through interrupt remapping.

1.1.2 Legacy BIOS and UEFI

BIOS is the governing reference which specifies a firmware interface.

"Legacy" (as in Legacy BIOS) - in terms of firmware specifications it refers to an older, broadly used specification. Major responsibility of BIOS is to initiate the support for hardware devices, loading and commencing an OS. When the system boots, the BIOS initializes and identifies every connected system devices including keyboard, mouse, hard disk drive, solid state drive, video display card and other hardware followed by locating software stored on a boot device i.e. a hard disk or removable storage such as USB or CD/DVD and loads and executes that software, transferring control of the system to it. This flow of actions is also known as "booting" or "boot strapping".

Table 1.1 overviews of comparisons of UEFI with legacy BIOS.

1.1.2.1 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 very well irrespective if the system is IBM or not. Additionally, Legacy BIOS also has a defined OS-independent interface for hardware that enables interrupts to communicate with keyboard, disk and video 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.

1.1.2.2 Limitations of legacy BIOS

With progress in technologies, the BIOS implementations were also updated with many new configuration and power management technologies and added support for many generations of Intel® architecture hardware. Although a few of limitations namely, upper memory block (UMB) dependencies, PC AT hardware dependencies, 1 MB addressable space and 16-bit addressing mode persisted throughout the years. The need to integrate libraries of third-party firmware modules into a single platform solution across multiple product lines and ensuring quality of individual firmware modules arises in the industries. The existing market demands to overcome inherent limitations lead towards development of a fresh BIOS architecture which is introduced in market as The UEFI specifications.

One major problem with existing BIOS implementations is that since they are highly customized for a specific motherboard, there maintenance is difficult. A lot of effort is required in significant porting, integration, testing and debug work of changes in component modules. The UEFI architecture is designed to considering these limitations and to resolve them.

1.1.3 Unified Extensible Firmware Interface (UEFI)

UEFI is a replacement for legacy BIOS to act as the interface between a operating system and its platform firmware streamlining the booting process. It offers a rich extensible pre-OS environment with advanced boot and runtime services, replacing most BIOS functions. Unified Extensible Firmware Interface (UEFI) is grounded in Intel's initial Extensible Firmware Interface (EFI) specification 1.10, which defines a software interface providing linking to an operating system and platform firmware. It has intrinsic networking capabilities, is designed to work with multi-processors (MP) system and also allows users to execute applications on a command line interface.

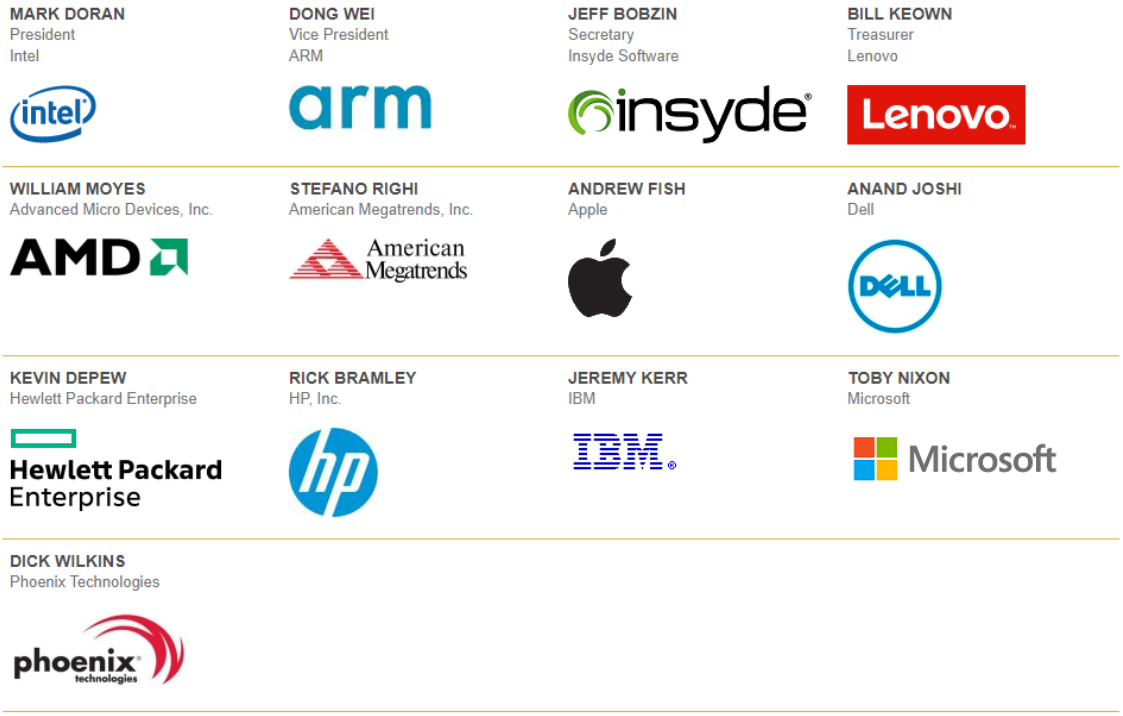


FIGURE 1.1: Board of Directors of UEFI Forum [**uefi-board-of-directors**]

The UEFI Forum board of directors consists of representatives from 11 industry leaders as described in Figure 1.1. These organizations work to ensure that the UEFI specifications meet industry needs.

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

1.1.3.1 UEFI Driver Model Extension

Boot devices are accessible via a set of protocol interfaces. The UEFI Driver Model provides a replacement for PC-AT-style option ROMs.

The UEFI Driver Model was not designed to replace the high-performance OS specific drivers but to access boot devices in the pre-boot environment, to support the execution of modular pieces of code, also known as drivers.These drivers control hardware buses and devices on the platform, and also they may provide some software-derived, platform specific service. The information required by the driver developers for implementing combination of bus drivers which boot an UEFI-complaint OS are included in the UEFI Driver Model.

Thus the UEFI Driver Model is designed to be generic.The UEFI Specification describes how to write USB bus drivers, USB device drivers, PCI device drivers, PCI bus drivers and SCSI drivers. Additional details are provided that allow UEFI

drivers to be stashed within PCI option ROMs along with maintaining the compatibility with legacy option ROM images.

The UEFI Specification is designed keeping in mind the goal of having compact driver images. However to facilitate support for multiple processor architectures, a driver object file for each architecture is required to be included leading to a space issue. To resolve this issue, UEFI defines EFI Byte Code Virtual Machine. Every driver file is compiled into just a single EFI Byte Code object which is run by an UEFI Byte Code Interpreter included in the UEFI Specification complaint firmware. Another very common method to resolve this issue is compression. The UEFI specification defined compression and decompression algorithms which may be used to reduce the size of UEFI Drivers.

This information can used by OEMs, IHVs, OSVs, and firmware vendors for developing drivers that produce standard protocol interfaces, and operating system loaders which could be utilized to boot UEFI compliant OS.

1.1.3.2 UEFI's Role in boot process

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

The `PC-AT` boot environment is challenging to innovate as each new firmware capability requires firmware developers to craft more complex solutions, often requiring OS developers to perform manipulation for their boot code. Since this is a time-consuming process and also required investment of resources, the UEFI specification undertakes it as a primary goal to overcome this issue.

1.1.4 Advanced Configuration and Power Interface (ACPI)

The "ACPI Component Architecture (ACPICA)" is an implementation of a group of software components according to the ACPI specification. It is created with the goal of isolating operating system dependencies to a relatively small translation or conversion layer (the OS Services Layer). This makes the bulk of the ACPICA code independent of any individual operating system.so it can used for new operating systems with no source changes within the ACPICA code itself.

Tthe architecture include below component:

- ACPICA Subsystem - independent of OS and kernel which serves the primal ACPI services like the AML interpreter and management of namespace.
- ACPICA Subsystem - independent of OS and OS Services Layer for every host OS to serve OS support.

- The ASL compiler/disassembler for translating the source code from ASL to AML and also disassembling the ASL source code from the binary ACPI tables if exists.
- Many ACPI utilities for running the interpreter in level 3 user space taking out the binary ACPI tables residing in the output result of ACPI Dump utility along with translating ACPICA source code to output format of Linux/Unix.

Figure 1.2 portrays the ACPICA subsystem in relation with the device driver(s), host OS, and the ACPI hardware.

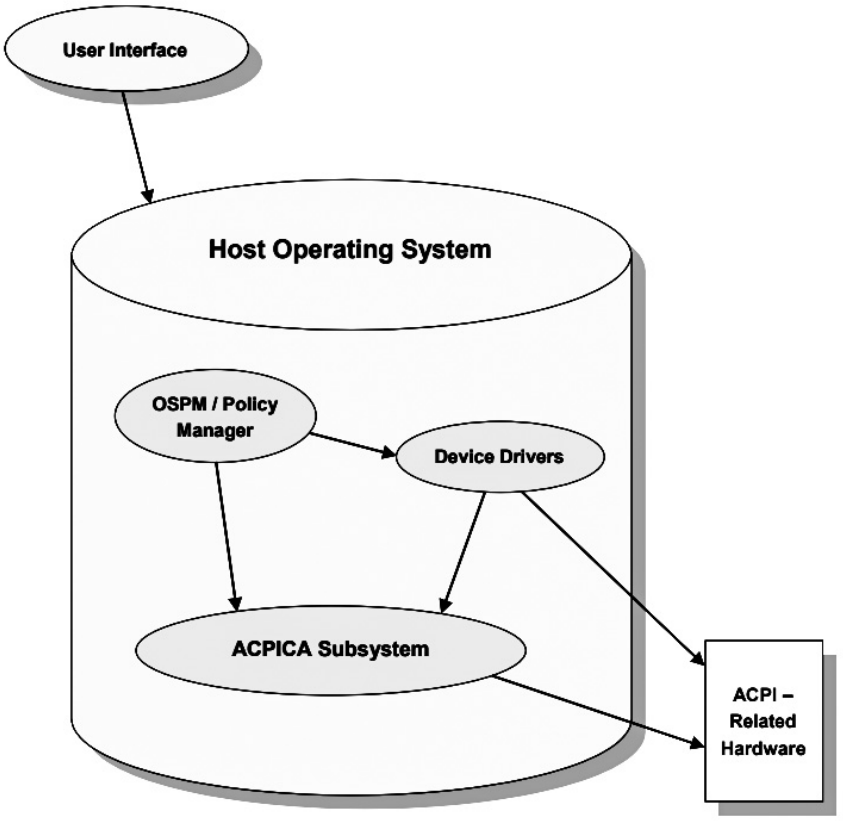


FIGURE 1.2: The ACPI Component Architecture

1.1.4.1 Overview of ACPICA Subsystem

The "ACPICA Subsystem" develops the basic primal aspects of the ACPI specification. Includes an AML parser/interpreter, ACPI table and device support, ACPI namespace management, and event handling. As the ACPICA subsystem serves the lower level services for system, it also involves low-level services of OS like memory management, scheduling, synchronization and I/O.

To allow the ACPICA Subsystem to easily link between any operating system that engage such services, an Operating System Services Layer transforms ACPICA-to-OS requests inside the system calls publicized by the host OS. This OS Services

Layer is the one and only element of the ACPICA which pertains source code which is limited to a particular host OS.

1.1.4.2 Operating System Services Layer OSL

"OS Services Layer (OSL)" act as a request translation service for host os from OS-independent ACPICA subsystem. The OSL develops a common subset for interfaces of OS service by utilizing the primitives usable from host OS.

The OSL has to be developed afresh for each and every supported host OS. There exists only one ACPICA Subsystem which OS-independent but there has to be a different OSL for each OS backed by the ACPICA.

The whole ACPICA in relation to the host OS is portrayed in Figure 1.3

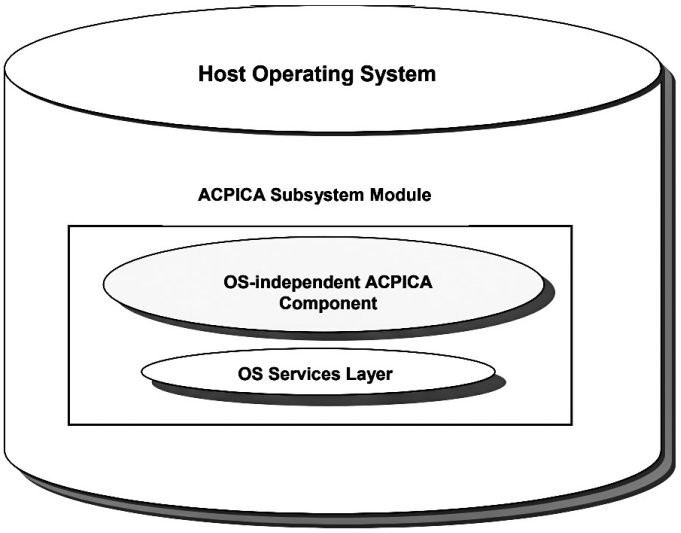


FIGURE 1.3: ACPICA Subsystem Architecture

1.1.4.3 ACPICA Subsystem Interaction

ACPICA Subsystem develops a subset of external interface links that could directly summoned via host OS. These Acpi interfaces serve the literal ACPI services for host. When OS services are needed while servicing of request of an ACPI the Subsystem makes oblique request to host OS through the fixed AcpiOs interfaces.

Figure 1.4 portrays the kinship and fundamental interaction linking the diverse architectural modules by screening the control flow among them. Note that OS independent ACPICA Subsystem could never call the host OS directly and instead it has to make call(s) to the AcpiOs interfaces inside the OSL. This serves the ACPICA code as OS-independence.

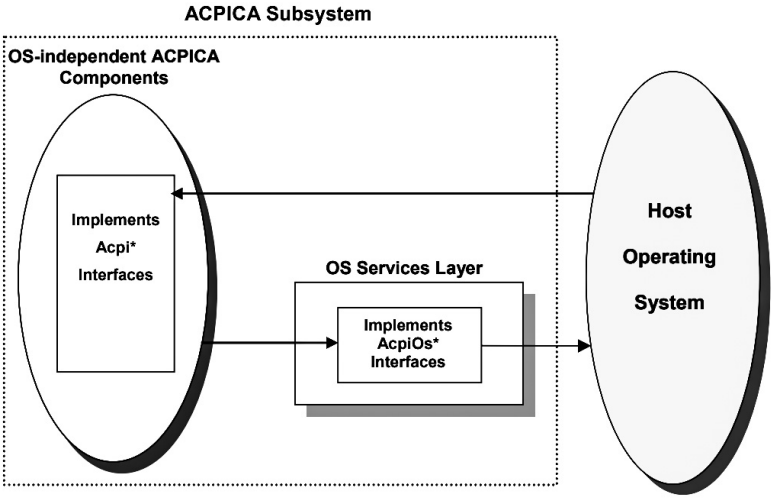


FIGURE 1.4: Interaction between the Architectural Components

1.1.5 Peripheral Component Interconnect Express (PCIe)

The "Peripheral Component Interconnect (PCI)" architecture has emerged out to be a very thriving beyond even the many more optimistic prospects. Today about every new computer system arrives equipped with at least one PCI slots. Even though there are about to countless PCI slots are shipped globally, there does exists tons of PCI adapter cards which are available to fulfill the all the virtual possible application needs.

This fast growing generation has also raised the demand for many new higher performance interface for input-output communication to support rising technology such as ultra high bandwidth technologies like $10 - Gb$ Ethernet, $10 - Gb$ "FibreChannel", $12X$ InfiniBand and many more. A regulation which could adapt to carrying out these high performance objectives, while withholding compatibility for previous generation of PCI would beyond any doubt can offer the idealistic solution.

"PCI-X 2.0" standard has been developed to meet these objectives. It is capable to serve the uninterrupted performance to feed the nearly all high-bandwidth programs while at the same instance keeping up the full hardware and software backward compatibility for previous "PCI" and "PCI-X" generations. The PCI-X 2.0 regulation establishes two new grades of speed and performance which are PCI-X 266 and PCI-X 533. These speed grades endeavors bandwidths that are twice and fourfold that of previous generation PCI-X 133 which results to finally supplying bandwidths which are $32x$ faster compared to the older version of PCI. It successfully succeeds to bring of additional required performance via time-proven "Double Data Rate (DDR)" and "Quad Data Rate (QDR)" mechanisms that transmit data at either twice or fourfold the base frequency of clock. As the PCI-X 2.0 conserves too many modules of previous generation PCI it's beneficiary for terrific amount of preceding development job. The OS, device drivers, protocols, connectors, form factor, BIOS, electrical signals, Bus functional modal among many more original PCI modules are all greatly

rendered in the new PCI-X 2.0 specification. Also, many of these modules actually remains untouched in PCI-X 2.0. Due to not having the much of the dissimilarity, it enables development easier because these modules have been already designed and developed with required engineering and familiar to the developers. As a end result, risk is dramatically decreased because the time-to-market became short.

1.1.5.1 Functional Description

The hardware which is used to implement a PCI based system consumes a software interface served by PCI BIOS feature. It has elementary use to generate operations in address spaces specific to PCI (PCI configuration space [**pci-config-space**]).

The X86 architecture allows following mode to operate as per PCI BIOS features:

- Real mode
- Protected mode
  - 286 protected mode (16 : 16)
  - 386 protected mode (16 : 32)
- Flat mode (0:32 protected mode)

In the Flat mode, all the segments begun at linear address 0 and span till the end of the whole $4 - GB$ address space.

1.1.5.2 BUS Performances and Number of Slots Compared

The several architectures characterized by the PCI-SIG[**pcisig**]. Figure 1.5 portrays the development of PCI bus clock frequencies and bandwidths. Its pretty obvious that the increasing the bus frequency does comprise the load in terms of electrical nodes as also to the maximum permissible connectors on a bus at that clock frequencies.

A "PCI express (PCIe) Interconnect" is responsible in connecting two PCI devices via PCI link. A PCI link consists if any signal pairs in each direction. These signals $(x1, x2, x4, x8, x12, x16, x32)$ known as the Lanes. A BIOS designer decides that how many Lanes implementation should be permissible based on the benchmark performance of targeted platform on a given PCI link.

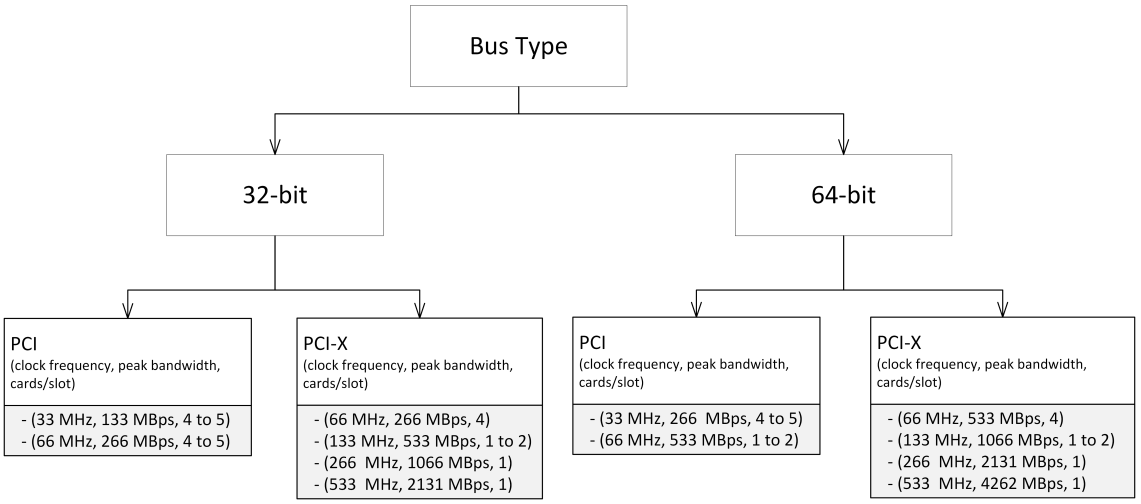


FIGURE 1.5: Comparison of Bus Frequency, Bandwidth and Number of Slots

1.1.6 Graphics Controller

Almost every graphics controllers are merely PCI controllers only. And it is also obvious that the graphics drivers who are responsible to control and manage these graphics controllers are also PCI drivers. Note that even if the most graphics controllers are PCI controllers but even then the graphics controllers can also utilize many of the other buses i.e. USB buses.

Characterizes of Graphics drivers are listed below:

- Follows UEFI Driver Modal
- Depending on the driver managed adapter, a graphics driver could be classified as into a single output adapter and a multiple output adapter.
- For each output expected, the graphics driver has to construct child handles.
- For some of the output ports and protocols (such as GOP Protocol) the graphics drivers must create child handles.
- Graphics drivers are hardware-dependent (i.e. specific to the corresponding chip) due to the need of initializing and managing the graphics device.

Note that (IHV) has the privilege of choosing whether to support and implement all the required modules of the UEFI specification. i.e., all modules might not be implemented to support on a specified system configuration which doesn't support all of the services and features understood by the needed modules.

1.1.6.1 Graphics Output Protocol (GOP)

The "Graphics Output Protocol GOP" Driver is member of the driver of UEFI boot time which are responsible for running up the display while the bios is booting. This driver triggers displaying of logo while the bios is booting.

1.1.6.2 GOP Overview

The GOP driver is the successor for video controller of legacy BIOS and sheers the utilization of UEFI pre-boot firmware without the use of CSM. The GOP driver can be $32 - bit$, $64 - bit$, or $IA - 64$ with no binary support. Pre-boot firmware architecture of UEFI which could be either $32 - bit$ or $64 - bit$ has to adapt the corresponding GOP driver architecture ($32 - bit$ or $64 - bit$). The GOP driver could be one of the boot mode: "fastboot" (for specific platform optimized mode to speedup the boot time) or "generic" (the normal boot process).

1.1.6.3 GOP DRIVER

The EFI specification characterizes the "Universal Graphic Adapter (UGA)" protocol to provide graphics that could be device-independent. However, Specification of UEFI eliminated the inclusion of UGA and replaced it with its successor GOP so that VGA hardware dependencies can be removed.

1.1.6.4 GOP Integration

The platform firmware has to align with the below listed requirements for integration of GOP Driver:

- Platform firmware has to be obedient with 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 has to bring forth the standard protocol `EFI_PCI_IO_PROTOCOL` and also the `EFI_DEVICE_PATH_PROTOCOL` on the graphics device handle. Additionally, the platform must produce `PLATFORM_GOP_POLICY_PROTOCOL`.
- The platform firmware should not establish the legacy BIOS Video.

Table 1.2: 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

The GOP Driver solution comprises the following files shown in Table 1.2 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.

Chapter 2

Design and Architecture

2.1 Design

2.1.1 Design Overview of UEFI

The UEFI construction is depending on the below listed primal elements:

- **Re-utilizing of already existing interfaces** - In order to keep up assets in existing infrastructure codebase, both at the OS and firmware level, many different existing specifications which are usually implemented on platforms harmonious with supported processor specifications has to be developed on platforms which is able to comply with specification of the UEFI.
- **System partition** characterizes a partition and file system which are to developed to grant safe sharing between various different vendors and various purposes. The power to include a disjoint, shareable system partition exists a chance to gain platform value-add without importantly thriving the need for nonvolatile memory of platform.
- **Boot services** are responsible to offer interfaces for devices and system functionality which could be utilized during the time of ongoing boot process. Device access is abstracted by "handles" and "protocols". This features reuse the investment in already existing BIOS codebase by persisting underlying implementation necessity out of the specification without giving execution load to the consumer accessing device.
- **Runtime services** - A stripped set of runtime services is given to guarantee seize abstraction for resources of platform hardware which could be required by the OS while its conventional operations.

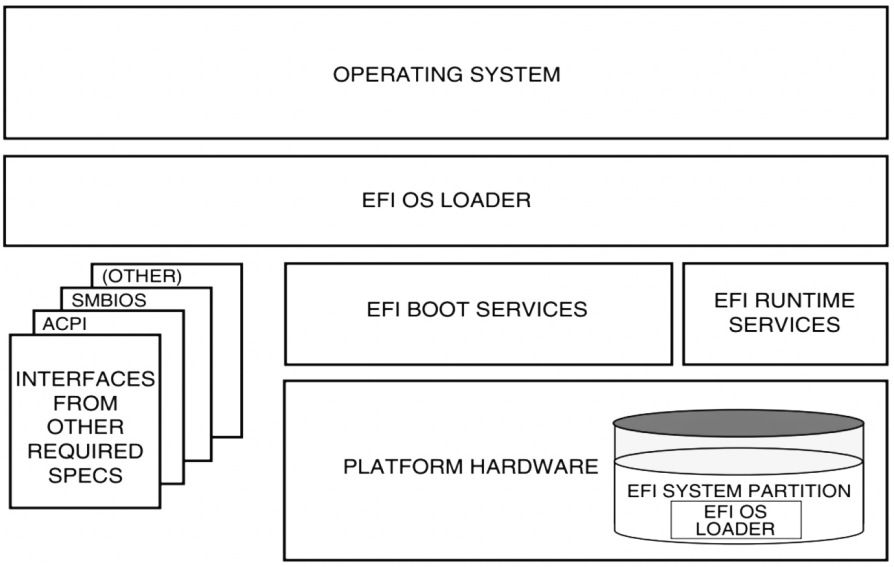


FIGURE 2.1: UEFI Conceptual Overview

Error! Reference source not found determines the fundamental interaction of the different component parts of an UEFI specification-amenable system which are utilized to carry out platform and OS boot.

From the system partition, the os loader image is retrieves by the platform firmware. The specification supplies for a diversity of media and mass storage device types as "disk", "CD-ROM", and "DVD" as well as "remote boot" via a network (also known as LAN boot or network boot). By the use of extensible protocol interfaces, there is possibility to include many other boot media types but also these would need OS loader alteration if they need to use the protocols.

Once begun, the OS loader proceeds to boot the whole operating system. To achieve this it could utilize the EFI boot services and interfaces characterized by respected specifications to analyze, embrace, and initialize the several platform components and the OS software which controls them. Also, for the OS loader will be capable to access EFI runtime services while in boot phase.

2.1.1.1 Goal of UEFI Driver

Majorly below are the listed motives to be expected to achieve by the UEFI Driver:

- **Compatible** - Any driver who conformist to the specification has to hold up compatibility along with the UEFI and EFI Specification. Hence, the UEFI Driver Modal brings up benefits of the extensibility mechanisms in the UEFI Specification to include the desired and necessary features.
- **Simple** - Any driver who conformist to the specification has to be simpler to develop and maintain. The UEFI Driver Modal has to permit a driver writer

to focus on the ad hoc device for which the driver is to be developed. A driver should not be related to issues correspondence to platform management or policy of platform. These circumstance should be left over for the system firmware.

- **Scalable** - The UEFI Driver Modal requires the adaptability for all kind of platforms including mobile, embedded systems, workstations, servers as well as desktop systems.
- **Flexibility** - The UEFI Driver Modal have to have capability to enumerate over every the devices (or only the relevant devices needed to boot the OS). With the minimum device enumeration support for fast boot ability can be achieved and the full device enumeration results to provide capability for performing system maintenance, or system diagnostics, OS installations on any boot device exists on the system.
- **Extensible** - The UEFI Driver Modal must be able to unfold to succeeding bus types as and when they are characterized.
- **Portability** - Every drivers transcribed in the UEFI Driver Model has to be portable among platforms and within every founded processor architectures.
- **Interoperability** - Every drivers has to coexist along with every other firmware and drivers and also do so without incurring conflicts for any resource.
- **Describing hierarchies of complex bus** - The UEFI Driver Modal has to be capable to key out a all kind of bus topologies from the platforms as simple as single bus to platforms with extremely complex bus which may consists of multiple buses of different kind.
- **Address the issues for legacy ROM option** - The UEFI Driver Modal needs to instantly come up and resolve the restrictions and regulations of legacy ROM options. Especially, it has to be capable to build add-in device cards which supports both UEFI drivers and legacy ROM options. However, maintaining this backward compatibility, the solution with proposed methodology should also provide a way to migrate from legacy ROM option driver to UEFI driver.

2.1.2 UEFI/PI Firmware Images

UEFI and PI specifications characterize the standard format for EFI firmware storage devices which includes FLASH or any other nonvolatile storage which are separated in “Firmware Volumes”.

Build systems should have the capability of processing files to construct the file formats represented by the UEFI and PI specifications. The tools which are supplied as part of the “EDK II BaseTools package” processes files compiled via third party

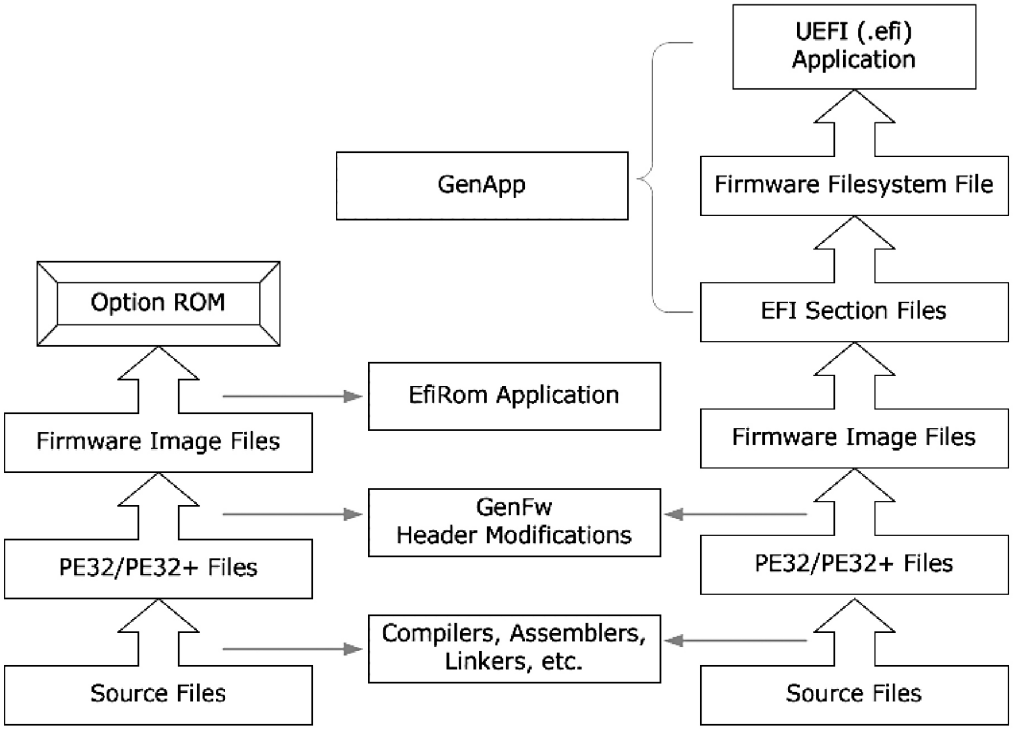


FIGURE 2.2: UEFI/PI Firmware Image Creation

scripts and tools, as well as unicode files and text files in order to construct UEFI or PI amenable binary image file. In few cases, where UEFI or PI specifications don't have an corresponding file format for input such as the "Visual Forms Representation (VFR)" files utilized to make PI compliant IFR contents, scripts, tools and documentation have been supplied which permits the user to create text files that are treated into formats specified by UEFI or PI specifications.

There are different layers of structure to a complete UEFI/PI firmware image. These layers are exemplified in Figure 2.3. Every Shifts between layers means that a processing block that transforms or unites previously treated files into the another higher level. Also in Figure 2.3 portrayed the reference tools utilized that processes the files to transit them among the different layers. The layers are described in more emphasized manner in Section 2.2

Apart from constructing images that initialize the whole platform, the build process also sustains creation of standalone UEFI applications (such as OS Loaders) and ROM images having driver code. Figure 2.2 portrayed the reference implementation tools and creation processes for both the image type.

The closing feature that is backed by the EDK II build process is to packaged and distributed the founding of Binary Modules to be utilized by other governing body. Binary modules doesn't need to distribute the source code. This shall only allow vendors to publicize UEFI images **without releasing copyrighted source code**.

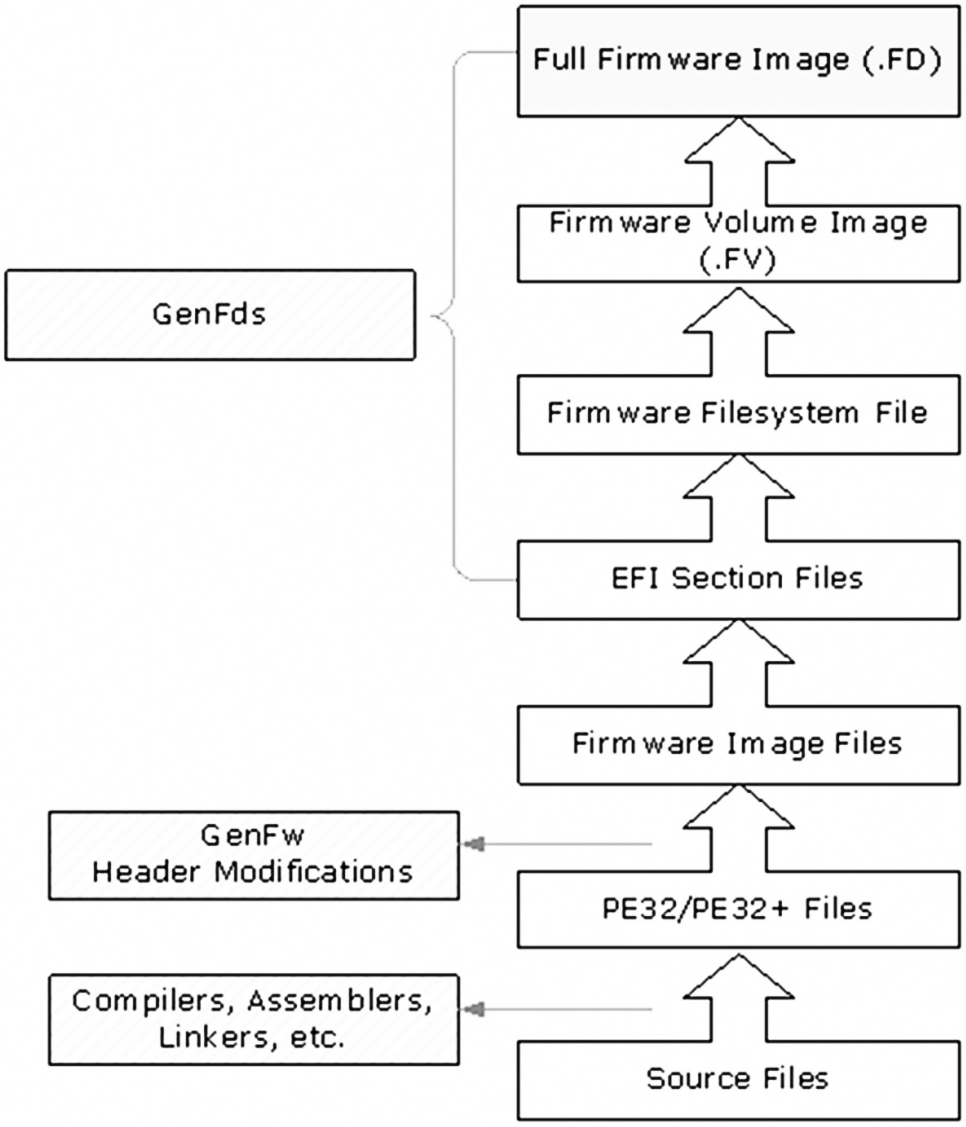


FIGURE 2.3: UEFI/PI Firmware Image Creation

The process of packaging allows construction of an archive file having multiple binary files which can be either Firmware Image files or higher (FFS, EFI Section files, etc.). The build process would only allow insertion of such binary files in to the corresponding level of the build stages.

2.1.3 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
6. "After Life (AL)" of system.

Figure 2.4 describes the phases and transition in detail.

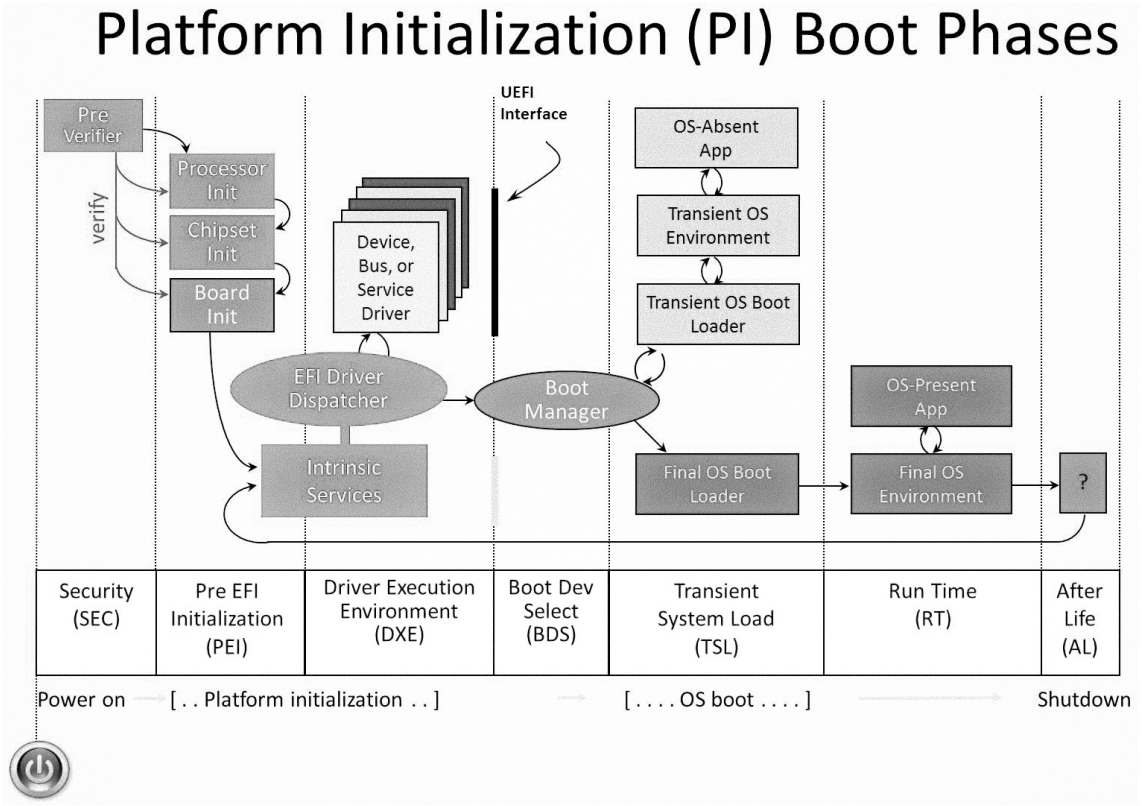


FIGURE 2.4: PI Boot Phases [**beyond-bios**]

2.1.4 Security (SEC)

"Security (SEC)" phase is the initial phase through which the boot flow begins. This phase is responsible for the following:

- Handling restart events of every platform
- Creation of a temporary memory stash
- Bringing the trust root in the system
- Transit handoff content to next phase - the "PEI Foundation"

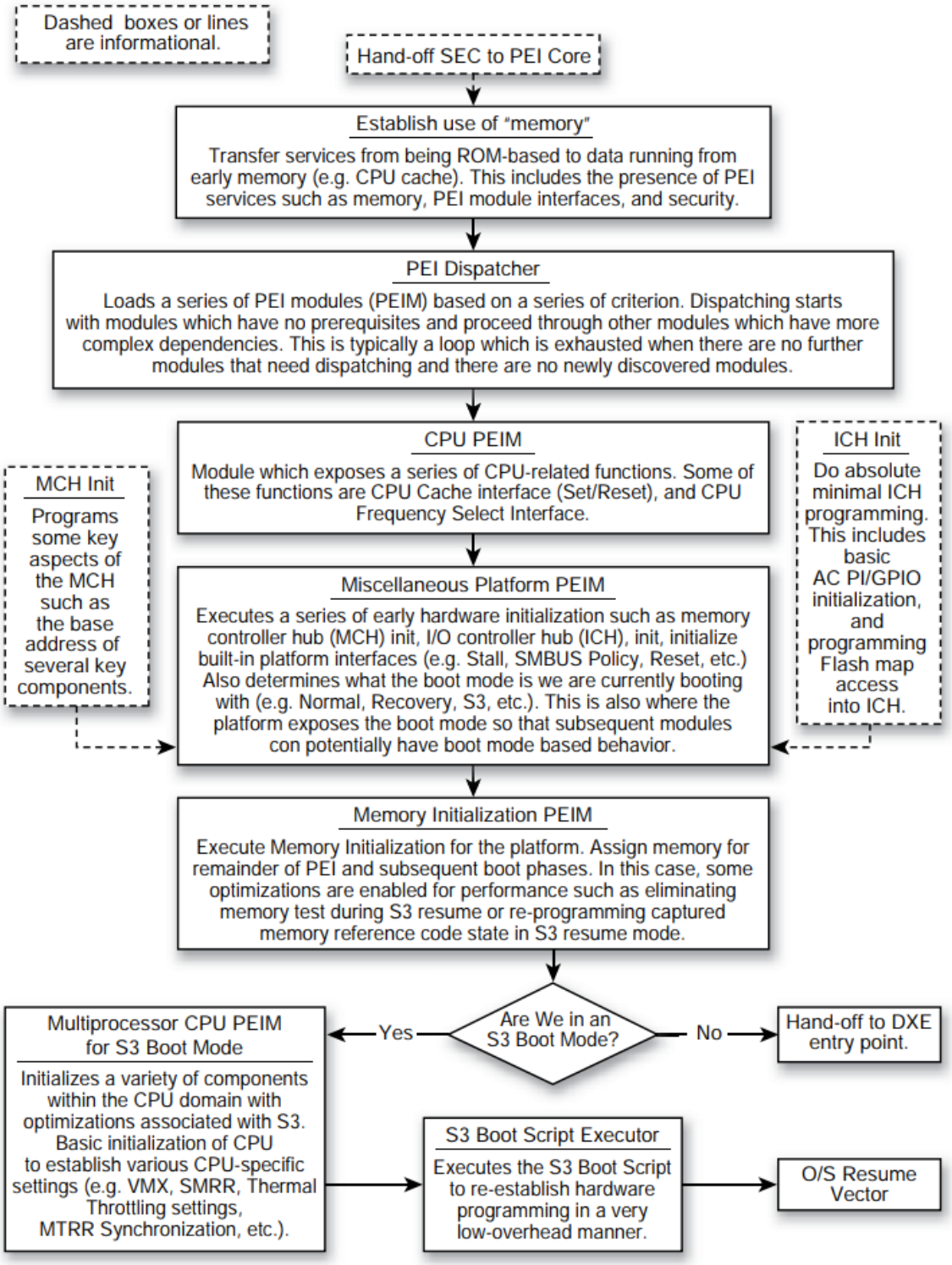


FIGURE 2.5: SEC Phase

Figure 2.5 portrays the Flow of the DXE phase.

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.

2.1.5 Pre-EFI Initialization (PEI)

"Pre-EFI Initialization (PEI)" phase represented within the PI Architecture specifications brought up quite betimes in the period of boot. Especially, after about preliminary processing of the Security (SEC) phase, any system restart event will bring up 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 along the platform in a developing stage, holding only resources on processor, such as the cache for maintaining call stack and dispatching the "Pre-EFI Initialization Modules (PEIMs)".

Figure 2.6 portrays the Flow of the DXE phase.

The PEI phase can not presume the convenience of amount of main memory (RAM) as DXE and hence PEI phase support is limited to:

- Locates and validates PEIMs
- Dispatches of PEIMs
- Facilitates commuting of PEIMs
- Provides handoff information content to later phases

These PEIMs can be considered for accountable for:

- Initializing few permanent memory complement
- Characterizing the main memory in "Hand-Off Blocks (HOBs)"
- Characterizing locations of the firmware volume in HOBs
- Transit the control flow into next phase - the "Driver Execution Environment (DXE)" phase

Figure 2.7 shows a diagram describes the action carried out during the PEI phase

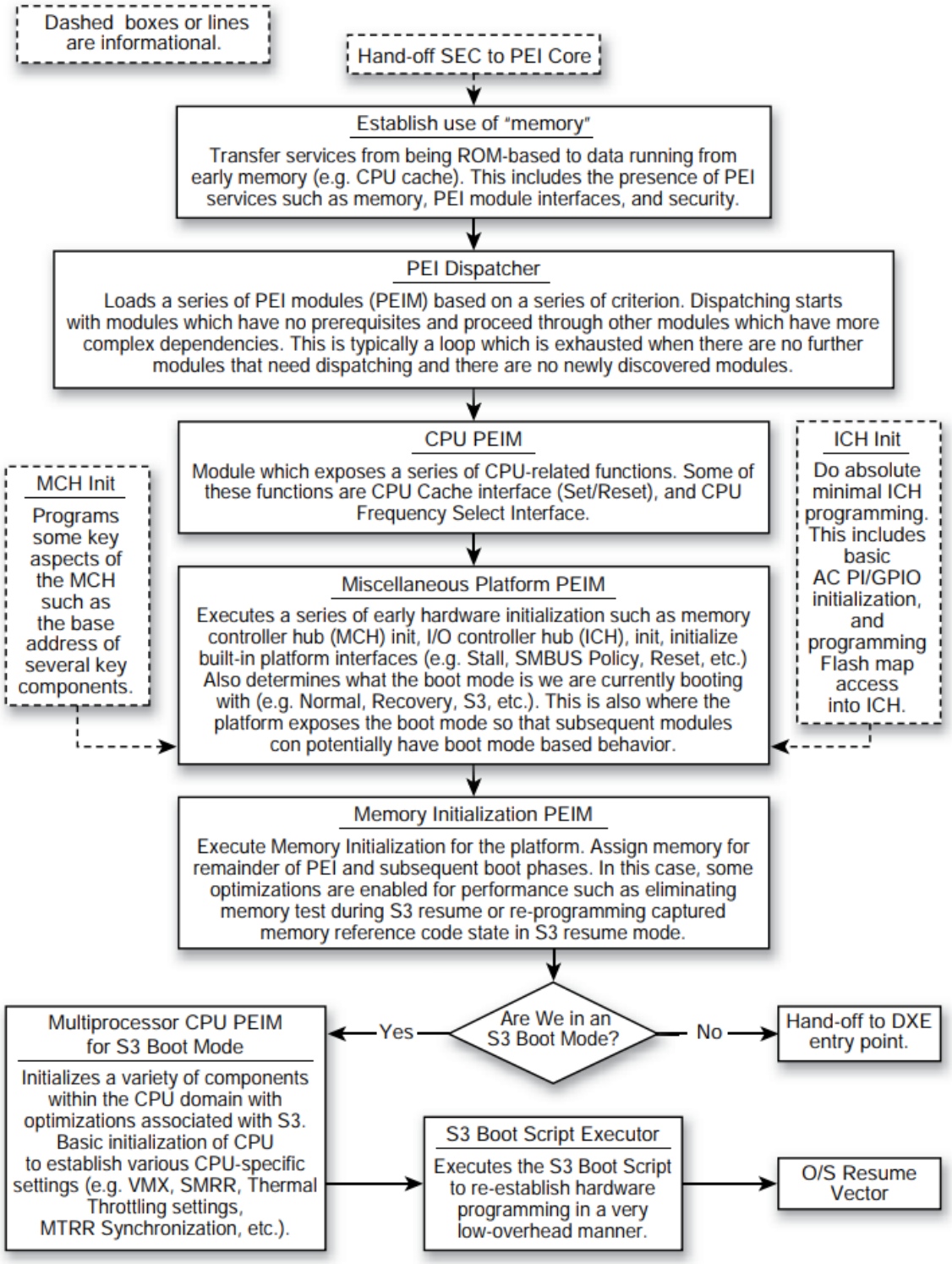


FIGURE 2.6: PEI Phase

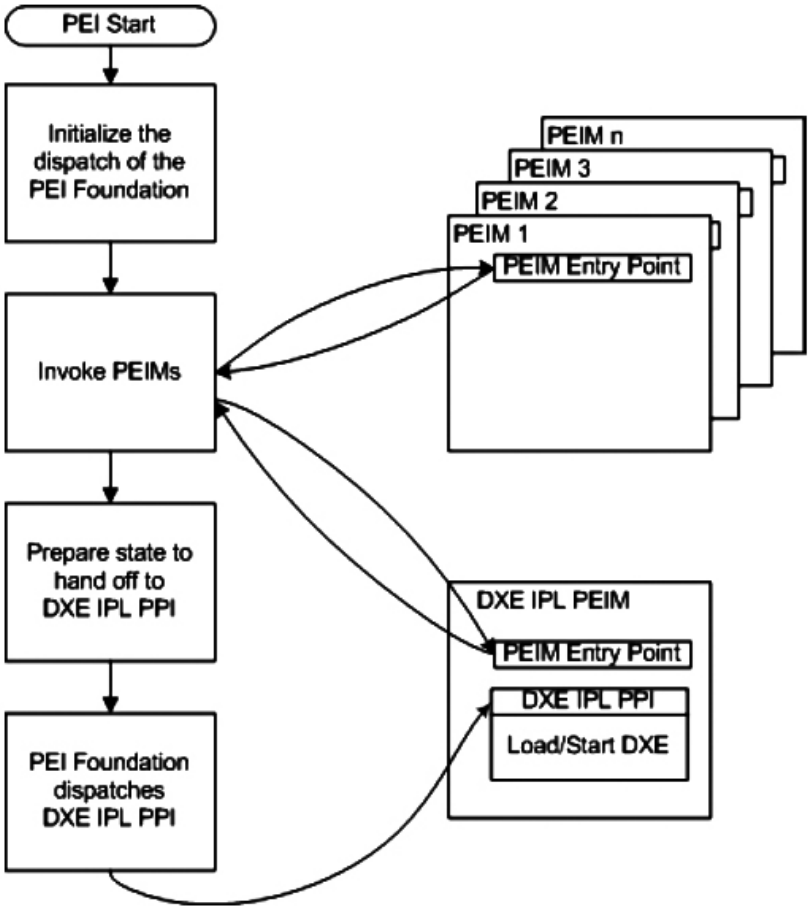


FIGURE 2.7: Diagram of PI Operations

2.1.5.1 PEI Services

"PEI Foundation" institutes a system table for the PEI Services named as "PEI Services Table" which is viewable to every PEI Modules (PEIMs) exists on the system. A PEI Service could be defined as a method, command or some other potentiality manifested by the "PEI Foundation" when the requirements of that service initialization are fulfilled. As the PEI phase having no permanent memory accessible until almost the end of life of the phase, all the various types of services constructed during this phase ("PEI phase") cannot be as enrich as those constructed during later phases. A pointer referenced to PEI Services Table is sent to the entry point of each and every PEIM and also within part of each "PEIM-to-PEIM Interface (PPI)" because the location of PEI Foundation and its temporary storage memory is unknown at the time of build.

2.1.5.2 PEI Foundation

The Phase PEI Foundation carries out following activity:

- Dispatching of "Pre-EFI initialization modules (PEIMs)"
- Maintaining and managing the boot mode
- Initialization of permanent main memory
- Conjure the DXE loader

The PEI Foundation developed to be portable among all the various platforms architecture of a specified instruction-set. i.e. A binary for "IA-32" (32-bit Intel architecture) works across all Pentium processors and similarly "Itanium® processor family" works on all the Itanium processors.

Irrespective of the processor's micro architecture, the set of service routines uncovered by the PEI Foundation has to be the same. This consistent layered area over the PEI Foundation allows PEIMs to be developed by the "C programming language" and also be compiled across any micro architecture.

The PEI Foundation responsible in providing the service classes listed in Figure 2.8

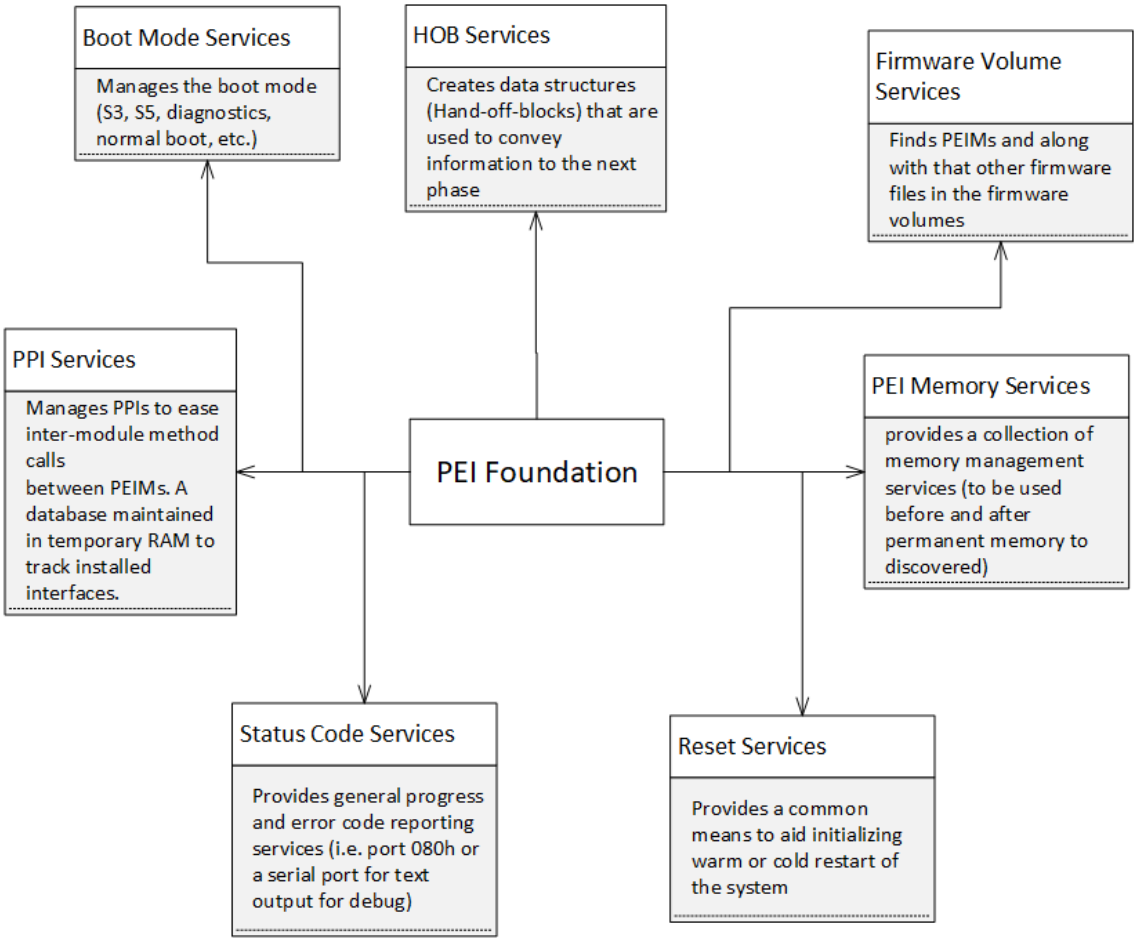


FIGURE 2.8: Services provided by PEI Foundation classes

2.1.5.3 PEI Dispatcher

The implementation of a state machine in PEI Foundation is known as the PEI Dispatcher. It evaluates the dependency expressions (DEPEX) in "Pre-EFI initialization modules (PEIMs)" that are lying in the FVs being analyzed.

Dependency expressions (DEPEX) are coherent alliance of "PEIM-to-PEIM Interfaces (PPIs)". These expressions distinguish the PPIs that has to be available for use before invoked by a given PEIM. 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 (DEPEX) will evaluate the result to `TRUE` which informs PEI Dispatcher that it can execute PEIM. At this very stage, the PEI Foundation handovers control flow to the PEIM with DEPEX result evaluated to `TRUE`.

The PEI Dispatcher will exit when it has examined and evaluated the results of all the PEIMs in all of the uncovered firmware volumes and none of the PEIMs can be dispatched (such as the DEPEX do not evaluate from `FALSE` to `TRUE` and vice-versa). At this stage, the PEI Dispatcher can not make call to any extra PEIMs. Control flow is than taken back by the PEI Foundation from the PEI Dispatcher and call to the `DXE IPL PPI` is made to navigate control flow to the "DXE phase" of execution.

2.1.6 Driver eXecution Environment (DXE)

Before the "DXE phase" the "Pre-EFI Initialization (PEI)" phase is held liable for initializing the permanent memory on the system platform. Hence, DXE phase could be loaded and executed in the permanent memory. At the very end of the PEI phase, state of the system is handed over to the DXE phase via utilizing the Hand-Off Blocks (HOBs).

Figure 2.9 portrays the Flow of the DXE phase.

DXE phase includes three major components as shown in Figure 2.10 which work among each other with the aim to initialize the platform and serve the services needed for performing OS boot.

2.1.7 Boot Device Selection (BDS)

The "BDS Architectural Protocol" has part of implementation of the Boot Device Selection (BDS) phase. After evaluation of all the dependencies of the DXE drivers along with their satisfied dependencies are loaded and execution is completed by the DXE Dispatcher, the DXE Foundation transfer the control flow to the BDS Architectural Protocol.

The "BDS Phase" held liable for:

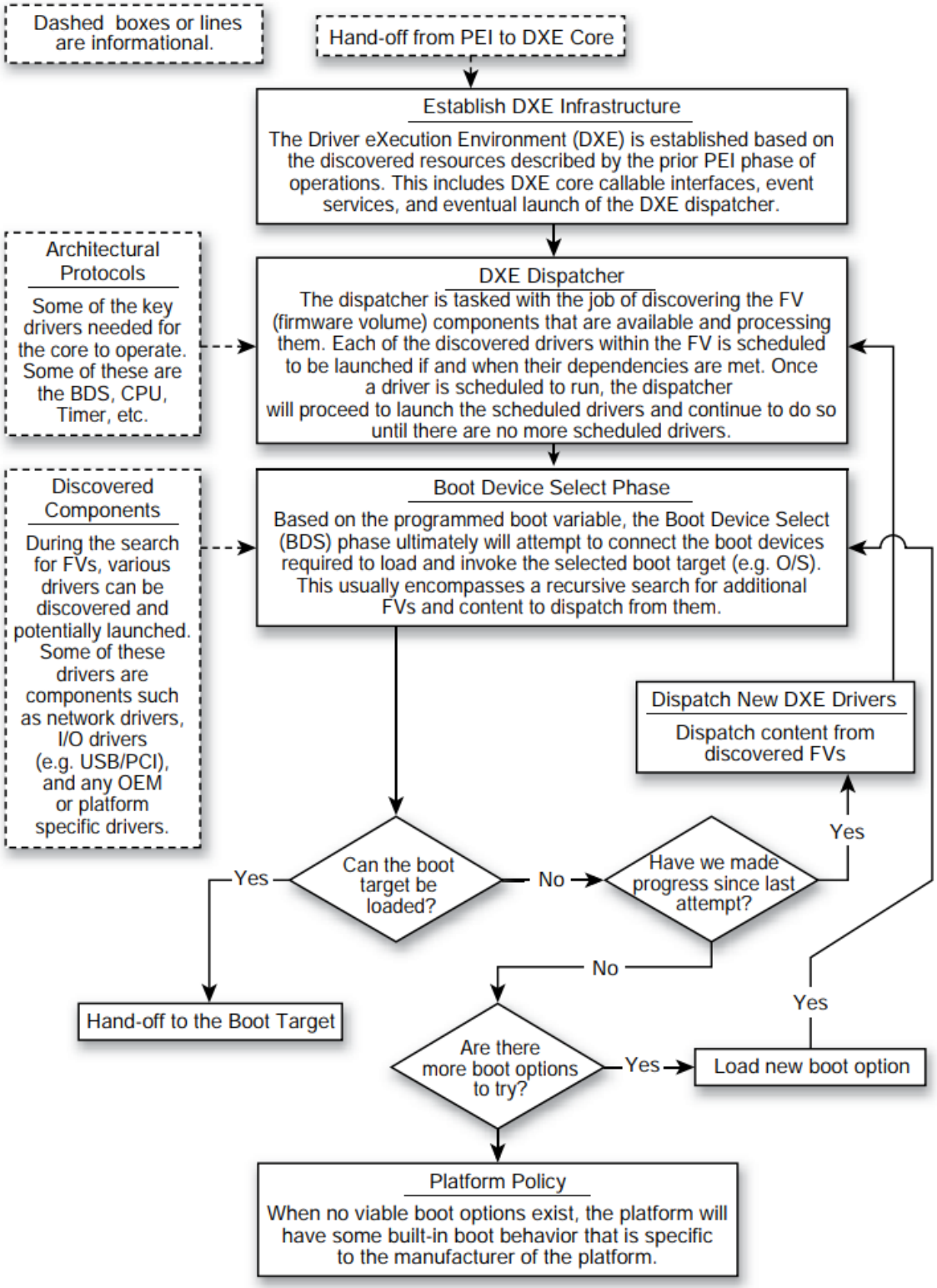


FIGURE 2.9: DXE Phase

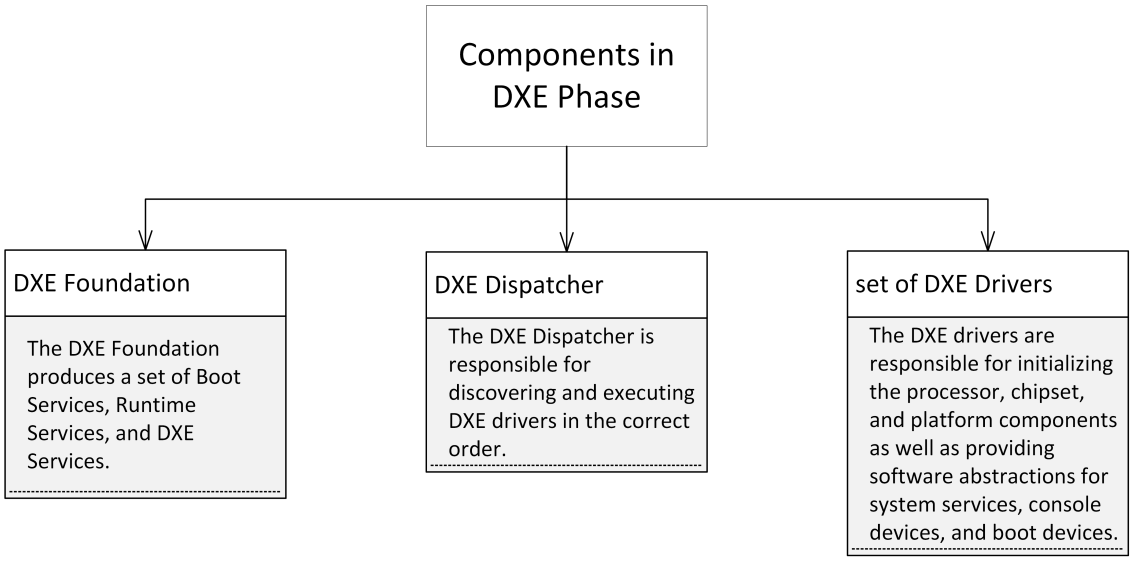


FIGURE 2.10: Components of DXE Phase

- Initialize the console devices
- Load the device drivers
- Attempt to load and execute the boot selection

2.1.8 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.

2.1.9 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.

2.1.10 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).

2.1.10.1 EFI Section Files

he general section format for sections less than 16MB in size is shown in Figure 2.12. Figure 2.11 shows the section format for sections 16MB or larger in size using the extended length field.

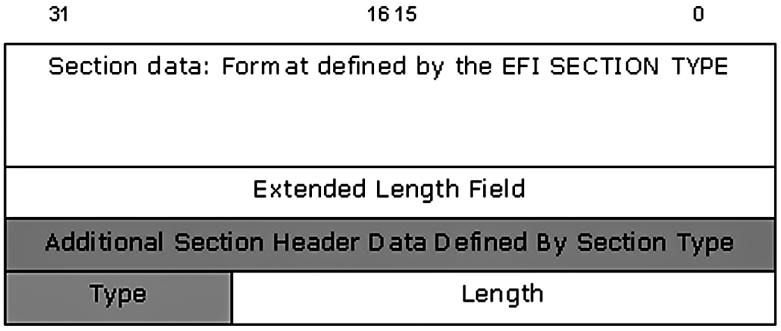


FIGURE 2.11: General EFI Section Format for large size Sections(greater then 16 MB)

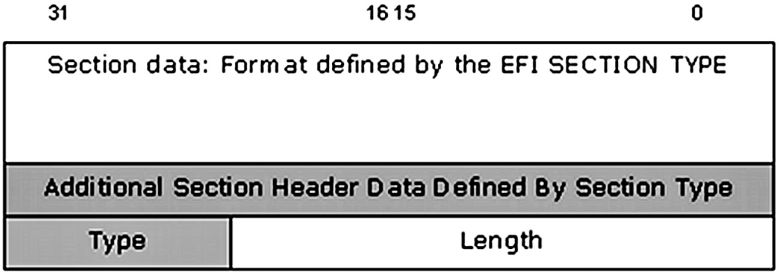


FIGURE 2.12: General EFI Section Format (less then 16 MB)

2.1.11 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.

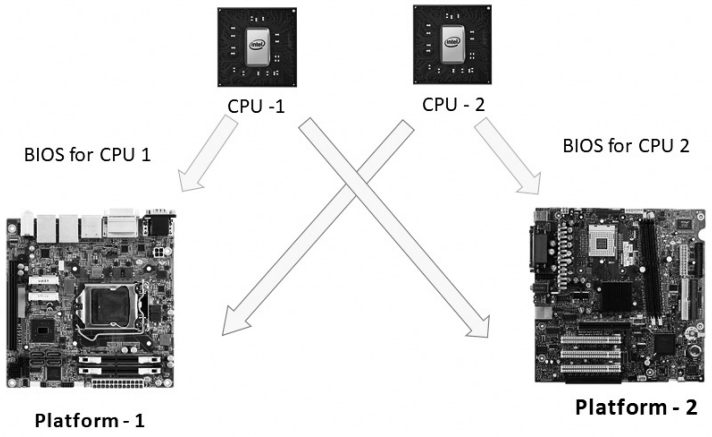


FIGURE 2.13: Cross Compatibility Design

Figure 2.13 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 2.14 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 2.15

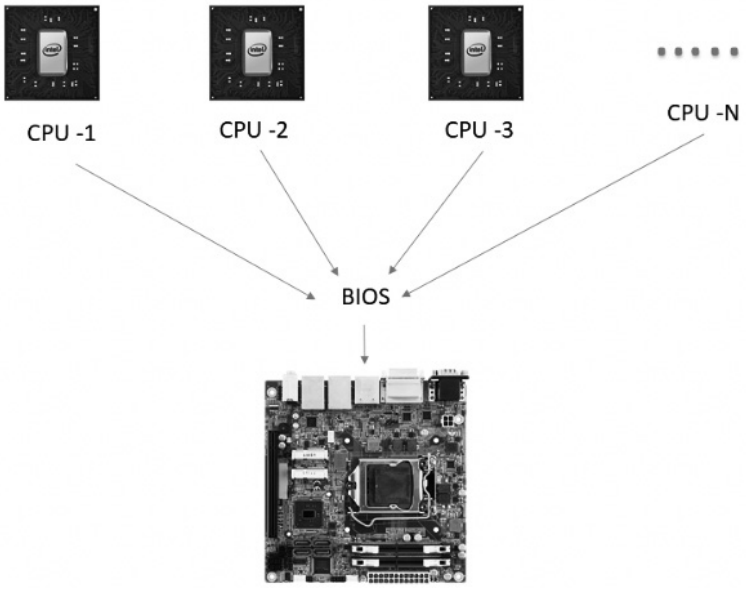


FIGURE 2.14: BIOS Support for Cross Compatibility

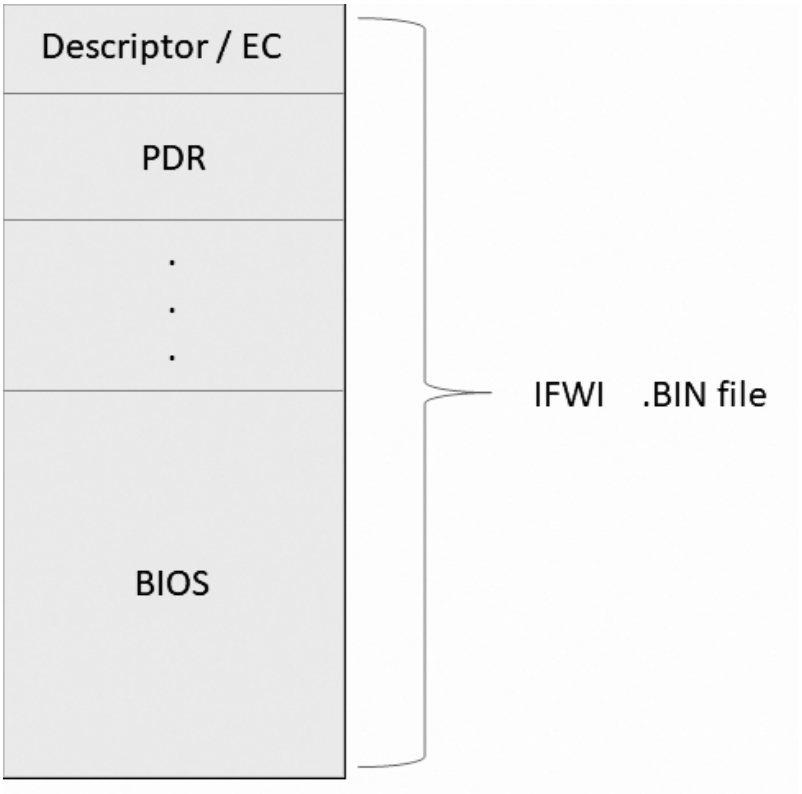


FIGURE 2.15: Integrated Firmware Image

2.2 Architecture of BIOS Firmware

2.2.1 Overview

If you interpret BIOS image as close look then it is nothing but the file system which is made in a byte format to be read by low level programming language which is most efficient method to store the data or content.

The concept of initialization of Platform includes the execution of this BIOS image which is stored on the every SoC

The components which plays role in Platform Initialization are listed below:

- Firmware Volume (FV) - consists of one or more firmware file systems
- Firmware File System (FFS) which consists of one or more firmware files
- Firmware File - Encapsulated section or leaf section
- Reference Layout of Binary
- Pre-EFI Initialization (PEI) PEIM to PEIM Interfaces (PPIs)
- Driver Execution Environment (DXE) Protocols

2.2.2 Design of Firmware Storage

Design of firmware storage elaborates the way that how files needs to be stored and accessed in nonvolatile storage environment. Implementation of any firmware has to support and follow the standard structure for PI Firmware Volume and the structure of FFS.

Firmware Device - a persistent physical repository consisting data and/or firmware code. Typically it is a component of flash but may also be any other type of persistent storage. Singular physical firmware device can be partitioned in to multiple other smaller pieces to form many other logical firmware devices from it and vice-versa.

Flash devices are most usual nonvolatile storage mechanism for firmware volumes. Often, flash devices are partitioned into many sectors or blocks of potentially differing sizes, each along with various runtime characteristics.

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

- Erase operation processed on the basis of sector-by-sector. After ensuring, every bits of sector return their `erase value`[1].
- Write operation can be performed on a bit-by-bit basis. i.e. In case erase value is 0 then bit value 0 can be changed to 1.
- Only by performing erase operation on the whole 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.
- Operations like writes and erases are much longer than reads operation.
- Many times places restraints on the trading operations that can be executed while a write or erase is in progress.

2.2.3 Firmware Volume (FV)

The BIOS image is consisting of one or more logical firmware devices known as a Firmware Volume (FV). Firmware Volume is the very basic and efficient logical storage mechanism for data and/or code. If you consider file system as a basic unit then firmware volume is unionized into these one or more file system units. Table 2.1 describes attributes in each firmware volume.

Apart from this Firmware volumes also consisting of few more information about the correspondence between OEM file types and a `GUID`.

2.2.4 Firmware File System (FFS)

The logical data payload within firmware volume is known as a Firmware File System (FFS) which illustrates the structure of files and free space (if any). To affiliate a driver to firmware volume every firmware file systems contains a globally unique ID (GUID).

Firmware files consists of code or raw data or both. Attributes of files are described in Table 2.2.

Integrity check and staged file creation are some extra attribute formats which might spotted in some firmware volume. Firmware File Sections are unit which unionized in a standard fashion to form certain file types for the file data.

OEM file types (described in detail in Figure 2.16) enables to support non-standard file types.

[1] either all 0 or all 1

Table 2.1: 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 data (includes all information like headers, files and free/reserved space)
Format	describes type of Firmware File System (FFS) which is unionized in construction of the volume.
Memory is Mapped or not?	some volumes may requires to be memory-mapped which determines whether the entire content of the volume can appear at once in the processor's memory address space.
Sticky Write?	Specifies whether or not special erase cycles requires in order to change value of bits into an erase value from non-erase value
Erase Polarity	In case a volume supports *Sticky Write*, then after processing an erase cycle every bits in the volume will return to this value (0 or 1)
Alignment	A volume is required to be aligned on some power-of-two (2^x) boundary such that *minimum* >= highest file alignment value.
Enable/Disable Read capable status	Decides whether to keep volumes as hidden from readable or not
Enable/Disable Write capable Status	Decides whether to keep volumes as hidden from writable or not
Lock Capable/Status	Volumes could also have their locking mechanism
Read-Lock Capable/Status	Volumes could also have the power to lock their read status
Write-Lock Capable/Status	Volumes could also have the power to lock their write status

Table 2.2: Firmware Files Attributes

Attribute	**Description**
Name	each volume has a unique identifier name having UEFI Globally Unique Identifier (GUID). Name of the File(s) has to be unique within a same firmware volume.
Type	Type of the individual file which can be Normal, OEM, Debug, FV Specific. More file types information are described in Figure 2.16.
Alignment	Every data of file to be aligned on some power-of-two (2^x) boundary such that these boundaries are founded depending on the alignment of firmware volume.
Size	Describes size of each file which consists of data of size zero or more bytes

PEI phase is responsible to serve the file related services which are carried out using PEI Service Table. On the Other hand the `EFI_FIRMWARE_VOLUME2_PROTOCOL` services which are attached to a volume's handle (`ReadFile`, `ReadSection`, `WriteFile` and `GetNextFile`) are responsible to carried out file related services in DXE phase.

2.2.4.1 Firmware File Types

If you consider an application with file name such as `XYZ.exe`, in which content format of `XYZ.exe` is implied by the ".exe" in the file name. Based on the situation of operation, this extension normally signals the contents of `XYZ.exe`. The PI Firmware File System characterizes the contents of a file that is returned by the firmware volume interface.

Firmware File System of the Platform Initialization dictates an enumeration of many file types. For example, the type `EFI_FV_FILETYPE_RAW` implies that the file is a RAW Binary Data. In the same way, files with the type `EFI_FV_FILETYPE_SMM_CORE` supports MM traditional mode .

2.2.5 Firmware File Section

Firmware File Section is individual distinct unit of certain file types which has following attributes:

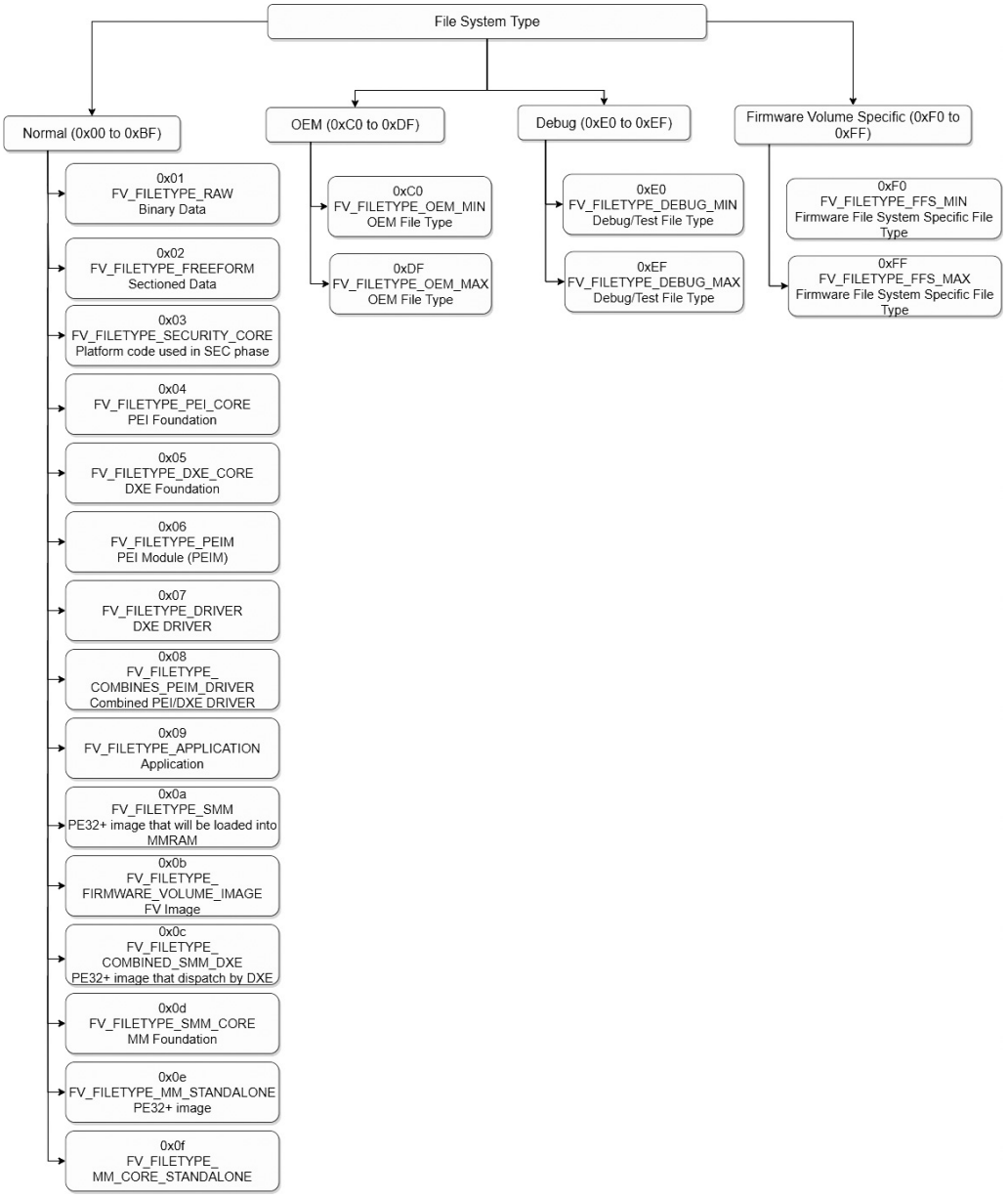


FIGURE 2.16: Firmware File Type

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

However as many as types of sections are present, they eventually fall in one of the below broadly described categories:

- **Encapsulation section** - logical storage consisting of the one or more section. The child section(s) which are lying within the encapsulation section (parent section) can be another encapsulation section or a leaf section which are also called relative peers to each other. An encapsulation section never consists of data in itself; however it is just a container that ultimately ends in leaf section(s). Files which are stacked with section can be imagined as tree consisting of nodes (encapsulation section) and leaves (leaf section). The root which can be interpreted as the file image itself may have a discrete number of sections. Sections that exist in the root have no parent section but are still considered peers.
- **Leaf Sections** - Contrary to the encapsulation section, leaf section does contain data and only data within it. Type of section defines which kind of data is stored within the leaf section.

As illustrated in Figure 2.17, the root which we interpret as the file image has two encapsulation sections which are $E0$, $E1$ and one leaf section which is $L3$. $E0$ which is the first encapsulation section possessing three child node which are all leaves ($L0$, $L1$, and $L2$). $E1$ is the another encapsulation section which possesses only two children, where one of them is encapsulation ($E2$) and the another is the leaf ($L6$). $E2$ which is the very last encapsulation section consists two children which are both leaves only ($L4$ and $L5$).

With the help of `FfsFindSectionData`, services related to section are populated with the help of PEI Service Table in the PEI phase. On the other hand `ReadSection` which is attached to service protocol `EFI_FIRMWARE_VOLUME2_PROTOCOL` responsible to populate services related to section during the DXE phase.

2.2.6 Firmware File Section Types

Subjective types of section are described in Table 2.3.

2.2.7 PI Architecture Firmware File System Format

Basic encoding of binary used for PI firmware file, firmware volume and file system is illustrated in this section. Development which carries out the non-vendor firmware files or firmware volumes to be enclosed into the system must have the standard formats. These sections also describes the way features of the standard format mapped into the standard interfaces of DXE and PEI.

TABLE 2.3: Types of Section

Name of Section	Value	Details
EFI_SECTION_COMPRESSION	$0x1$	non-leaf section containing compressed section(s) within
EFI_SECTION_GUID_DEFINED	$0x2$	non-leaf section which only to be used while in process of build and not for execution
EFI_SECTION_DISPOSABLE	$0x3$	non-leaf section which only to be used while in process of build and not for execution
EFI_SECTION_PE32	$0x10$	Image executable of PE32+
EFI_SECTION_PIC	$0x11$	Code independent of position
EFI_SECTION_TE	$0x12$	Image of Terse Executable
EFI_SECTION_DXE_DEPEX	$0x13$	Expression for DXE driver dependency
EFI_SECTION_VERSION	$0x14$	version of the section - text/numeric
EFI_SECTION_USER_INTERFACE	$0x15$	Human readable and easily interpretable name for driver
EFI_SECTION_COMPATIBILITY16	$0x16$	16-bit exe of DOS fashion
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	$0x1b$	Expression for PEI driver dependency
EFI_SECTION_SMM_DEPEX	$0x1c$	Leaf section which determine the order of dispatch for the MM Traditional driver in SMM.

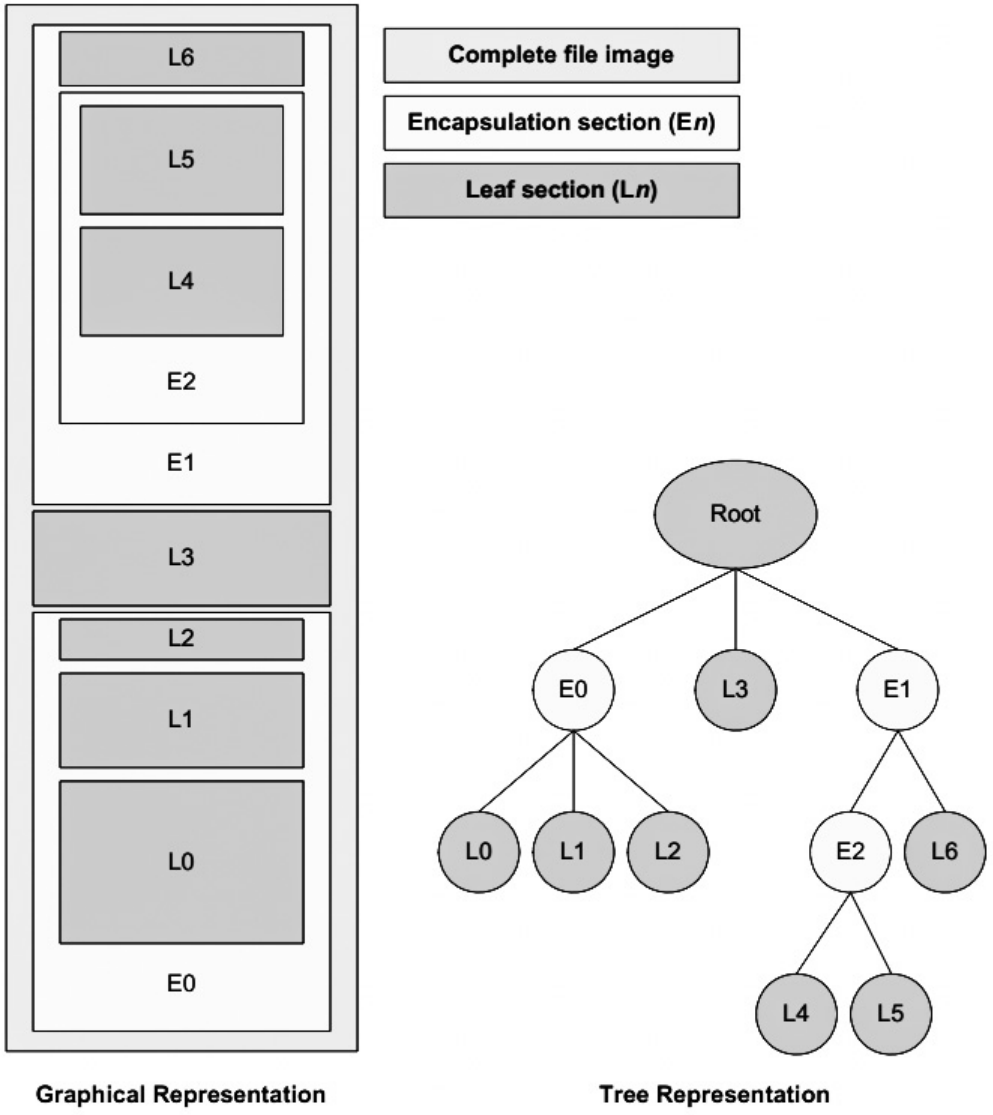


FIGURE 2.17: Example File System Image

The standard format of firmware file and volume also brings in extra dimensions and potential that are used to assure the unity of firmware volume. The standard format is unionized by three different levels: firmware volume, firmware file system, and firmware file.

The guided formatting of firmware volume (Figure 2.18) made of two parts: The FV header and FV data. Header of FV describes every attributes mentioned in "Firmware Volumes" in Table 2.1. This header also has GUID which identifies format of the firmware file system utilized to orchestrate data in the firmware volume. The "firmware volume header" is compatible with every other firmware file systems except the PI Firmware File System.

"Firmware File System format" explains the way the firmware files and free space are conceived inside the firmware volume. However on the other hand "Firmware

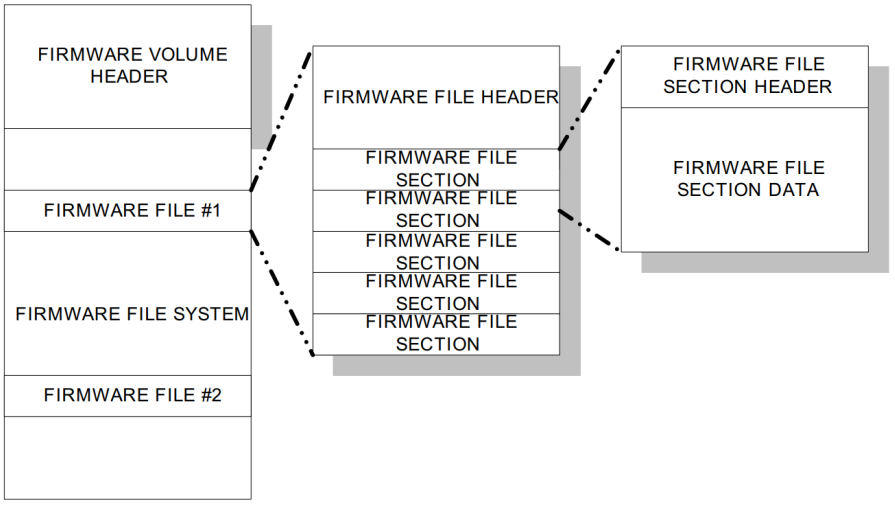


FIGURE 2.18: The Firmware Volume Format

File format" describes how files are organized. The firmware file format made of two parts: the firmware file header and the firmware file data.

2.2.7.1 Firmware Volume Format

The PI Architecture Firmware Volume format key outs the binary structure of a firmware volume. The firmware volume format possesses a FV header followed by the FV data. The FV header is represented by variable `EFI_FIRMWARE_VOLUME_HEADER`. The format layout of the FV data is described by a GUID. Valid files system GUID values are `EFI_FIRMWARE_FILE_SYSTEM2_GUID` and `EFI_FIRMWARE_FILE_SYSTEM3_GUID`.

2.2.7.2 Firmware File System Format

The PI Architecture Firmware File System is a binary design of logical file storage within firmware volumes. It is a flat file system in which there is no rendering of any directory hierarchy structure. Each and every files lies directly in the root of the storage. Files are stored end to end without any directory entry to explain which files are present. Parsing the information stored in a firmware volume to find a itemization of files exists needs the complete walk through over the firmware volume in and out.

Firmware File System GUID The firmware volume header has a unique data field for the file system GUID. The two valid FFS file systems are defined by the GUID values in variable `EFI_FIRMWARE_FILE_SYSTEM2_GUID` and `EFI_FIRMWARE_FILE_SYSTEM3_G` In case of the FFS file system, if it does allows files larger than 16 MB along with backward compatibility `EFI_FIRMWARE_FILE_SYSTEM2_GUID` then `EFI_FIRMWARE_FILE_SYSTEM3_G` is used.

Volume Top File known as VTF is a file that has to be presented such that the very last byte of file is also the very last byte of the firmware volume. Irrespective of type of the file, a VTF have to have GUID for the file name which is declared as variable `EFI_FFS_VOLUME_TOP_FILE_GUID`. Driver cide if Firmware file system has to be exposed of this GUID and infix an alignment pad file as and when needed to assure that the VTF is situated correctly at the top of the firmware volume. Length and alignment of File requirements needs to be coherent with the top of volume so that a write error does occurs and the unwanted firmware volume modification can be prevented.

2.2.8 Firmware File Format (FFS)

Every FFS files begins with its header data that is aligned on an $8-byte$ (which is power-of-two 2^3) boundary with respect to the origin of the firmware volume. FFS files consists of the below parts:

- Header
- Data

When a zero-length file is created without any data it still has to have header and will consume minimum $24bytes$ of space.

The data (if any) exists in file then it immediately conjugated after the header. How the data within a file is formed can be identified by the "Type" field in the header which can be either of `EFI_FFS_FILE_HEADER` and `EFI_FFS_FILE_HEADER2`.

Figure 2.19 exemplifies the typical layout of a (i.e. `EFI_FFS_FILE_HEADER`) "PI Architecture Firmware File" ($\leq 16Mb$).

Figure 2.20 exemplifies the typical layout of "PI Architecture Firmware File" ($> 16Mb$).

2.2.9 Firmware File Section Format

Storage Data format mechanism of section is described in this section. Each individual section starts with a section header which is followed by the data defined using the section type. Section headers are always aligned at $4-byte$ boundaries with respect to the start of the file image. In case of padding required between the section then to achieve the $4-byte$ alignment as defined, every bit value of padding is set to zero. There some section types which are variable in terms of data length and are more precisely represented as data streams instead of data structures.

Irrespective of type of the section, all section headers starts with a $24-bit$ integer telling the section size, and $8-bit$ section type. The format of the rest of the

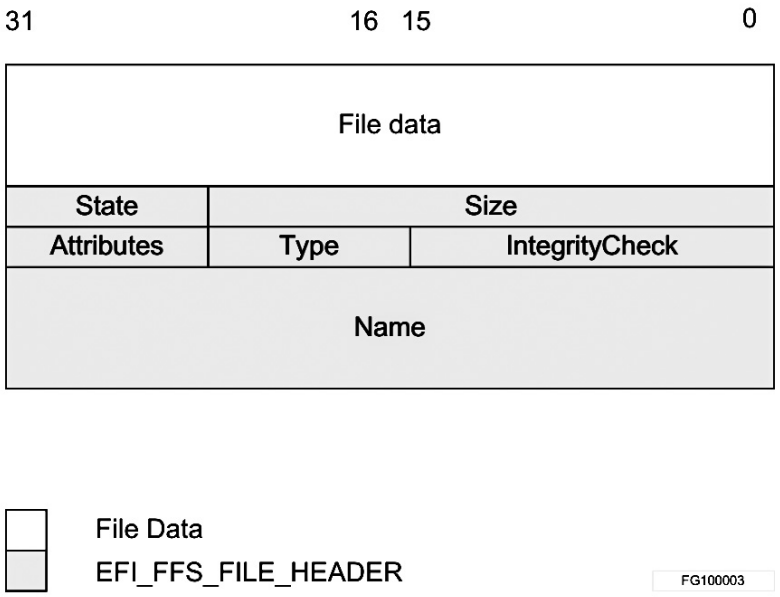


FIGURE 2.19: Layout representation of FFS File Header ($\leq 16Mb$)

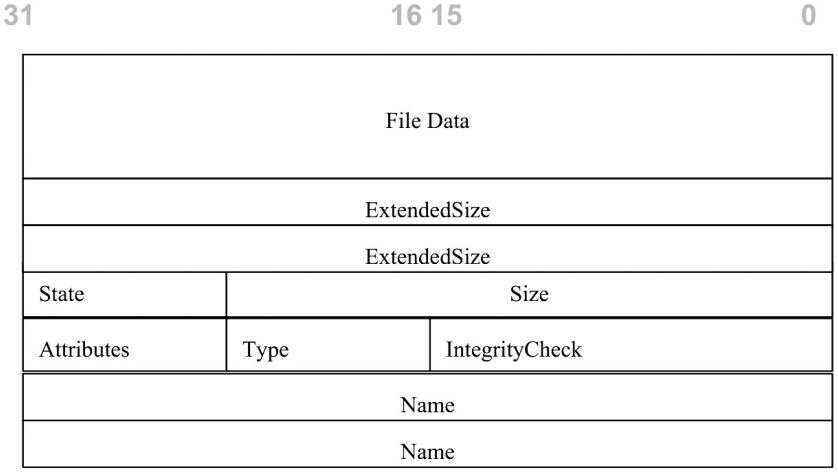


FIGURE 2.20: Layout representation of FFS File Header 2 layout for files ($> 16Mb$)

section header and data is defined by the section type. If size of the section size is $0xFFFFFF$ then the size is defined by a $32 - bit$ integer that follows the $32 - bit$ section header. Figure 2.21 and Figure 2.22 shows the typical layout of a section data format.

2.2.10 File System Initialization

To ensure unity of the file system it is mandatory to maintain state byte of each file correctly such that it won't compromised even in case of power failure during

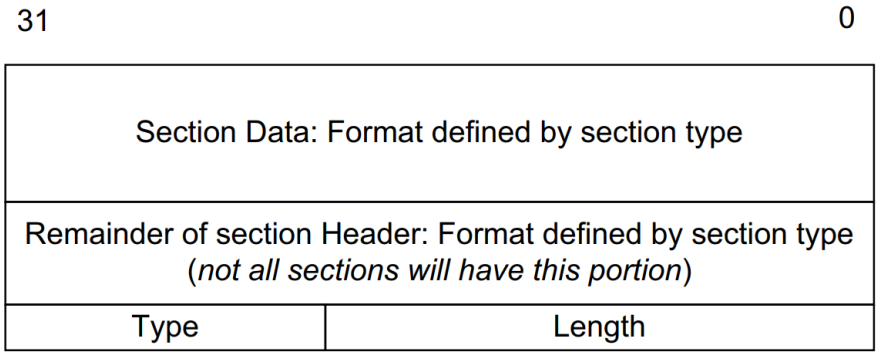


FIGURE 2.21: Section Header Format when $size < 16Mb$

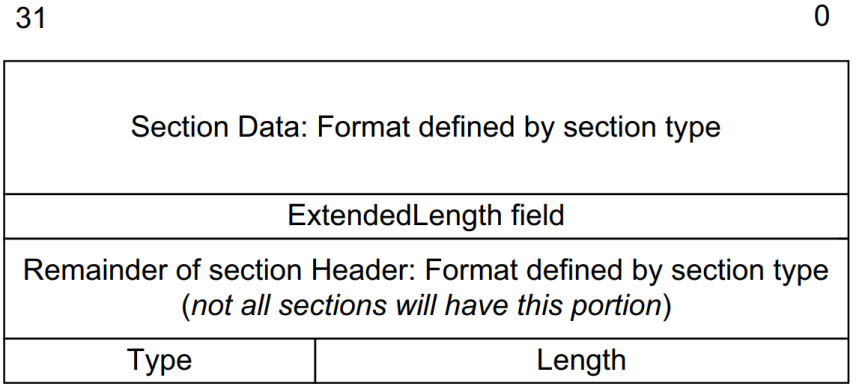


FIGURE 2.22: Section Header Format of when $size \geq 16Mb$ using Extended Length field

operation on any FFS. It is desired that an FFS driver produces an instance of Firmware Volume Protocol so that every normal file operations carried out in that context. Every file operations has to follow all the rules of creation, update, and deletion mentioned in this specification to avoid corruption of the file system.

2.2.11 Traversal and Access to Files

The Security (SEC), PEI, and early DXE code needs to be capable to traverse the FFS such that it's read and execute operation on files carried out before a write-enabled DXE FFS driver is started it's execution so that the FFS may not have any inconsistencies because of any kind of previous system failure. Hence, it has to follow a set of rules to assert the credibility of files before using them. It is not incumbent on SEC, PEI, or the early read-only DXE FFS services to make any effort to perform recovery or modification the file system. If any case exists where execution can not continue because of inconsistencies in file system, a recovery boot must be initiated.

As there is one mutual exclusiveness that the SEC, PEI, and early DXE code can affect without instantiating a recovery boot. This condition can be summoned by any previous system failure such as power failure that come along while a file update on a previous boot. In such case, a failure can cause two files with an identical file name GUID to coexist within the same firmware volume where one of them will have the `EFI_FILE_MARKED_FOR_UPDATE` bit set to its state field but are going to be otherwise totally valid file. The another file may be in unknown state of building up to and including `EFI_FILE_DATA_VALID`. All files used preceding to the initialization of the write-enabled DXE FFS driver must be filtered with this test prior to their use. If this condition is observed, it's tolerable to trigger a recovery boot and allow the recovery DXE to perform the completion of update.

Note There's no ascertain for redundant files once a file found in the `EFI_FILE_DATA_VALID` state. The condition where two files in same firmware volume coexist having the same file name GUID and both are within the `EFI_FILE_DATA_VALID` state cannot occur if the set of rules for creation and update are strictly followed.

2.2.12 File Integrity and State

File corruption, no matter the cause, must be detectable in order to carried out appropriate steps for file system repair. File corruption can come from various sources but broadly falls into three categories listed below:

- Any general failure
- Failure on erase
- Failure on write

A general failure is characterized to be evidently random corruption of the storage media. This corruption can occur because of the design problem or obsolete storage media i.e. This type of failure can be as perceptive as replacing any single bit inside the file content. Using a good design of system along with reliable storage media, general failures can be avoided. However, the FFS enables catching of this kind of failure.

An erase failure happens when a block erase of firmware volume media isn't completed because of any system failure i.e. power failure. As the erase operation is not outlined, it is likely that most of the implementation of FFS that allow file write and delete operations will also develop a mechanism to rectify deleted files and unite free space. In case the operation is not carried out successfully, the file system can be left out in a state which is not consistent.

Likewise, a write failure takes place when a file system write is in motion and is left incomplete because of any system failure i.e. power failure. This type of failure can lead the file system to be in an inconsistent state.

All of these failures can be traced while FFS initialization is in progress hence depending on the cause of the failure, many recovery schemes can be carried out.

2.3 System Management Mode (SMM)

2.3.1 Overview

On "IA-32 processors", System Management Mode (SMM) is a manner of operation which is different from flat modal so as from protected mode of the DXE and PEI phases. It is outlined as a real mode environs with $32 - bit$ data bus access and its carried out in effect to either with a specified interrupt type or with the System Management Interrupt (SMI) pin. Note that Operation mode of SMM is OS independent mode and is discrete operational mode, however it may also lies in both within and OS runtime.

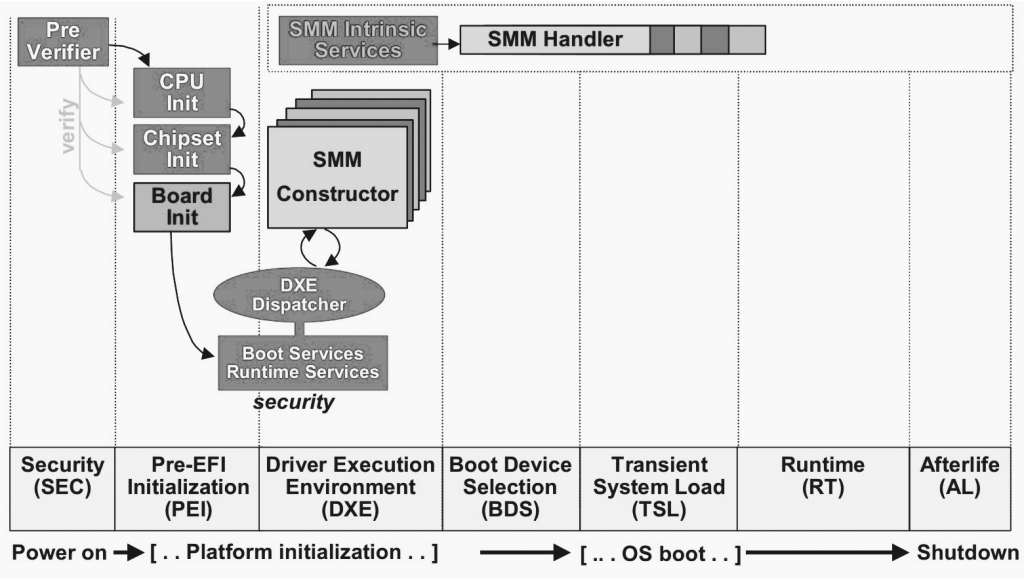


FIGURE 2.23: SMM Framework Architecture [**beyond-bios**]

2.3.2 System Management System Table (SMST)

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

SMST table allows access to service based on the SMST which also known as SMM Services. Driver can only use SMM services during execution inside context of the SMM. `EFI_SMM_BASE_PROTOCOL.GetSmstLocation()` service used to discover the address of SMST.

SMST is a set of potentiality exported for utilization by any driver that is loaded into SMRAM. It's similar to the EFI System Table except that by design it's fixed set of services and data and also doesn't acknowledge to the resiliency of an EFI protocol interface.

SMM infrastructure component of Framework provides SMST, which manages:

- Dispatching drivers inside SMM
- Allocation of memory (SMRAM)
- Switching the framework in and out of the applicable SMM of the processor

2.3.3 SMM and Available Services

2.3.3.1 SMM Services

As EFI runtime drivers have their constraints, similarly the model of SMM framework will have them too. Especially, dispatch of drivers in SMM won't be capable of using any core protocol service. However SMST-based services, called SMM Services allows the drivers to be access using an SMM identical of the EFI System Table, but services of the core protocol won't grantees the availability in runtime. As an alternative, the complete mass of EFI Boot Services and EFI Runtime Services can be available while the driver loading or "constructor" phase.

With utilizing the visibility of constructor, SMM driver is capable to leveraging rich set of EFI service to perform:

- Marshall interface for EFI services.
- Observing EFI protocols which are populated by other SMM drivers while in constructor phase.

For drivers while not in SMM and during the initial load inside SMM, EFI protocol database becomes quite useful by utilizing design.

Available services which are SMST-based includes:

- Negligible, blocking kind of the device Input/Output protocol
- Memory allocator from SMRAM

These services are exposed through the entries present in the System Management System Table (SMST).

2.3.3.2 SMM Library

During constructor phase of SMM driver inside SMM, all additional service within SMM Library (SMLib) are uncovered like EFI protocols. i.e., An identical status code in SMM is merely an EFI protocol with interface referencing an SMM-based driver service. To avoid error or information of progress during runtime, other SMM drivers also locates this SMM based status code.

2.3.4 SMM Drivers

2.3.4.1 Process to Load Drivers in SMM

Process to load driver modal in SMM is merely a DXE SMM runtime driver having a DEPEX (dependency expression) having at least `EFI_SMM_BASE_PROTOCOL`. This kind of dependency is essential as the DXE runtime driver which is planned for SMM will utilize the `EFI_SMM_BASE_PROTOCOL` to load up itself again in SMM and re-execute its entry point. Also, other SMM-loaded protocols permitted to be stayed in the DEPEX of specified SMM DXE runtime driver. Principle of the DXE Dispatcher is to verifying if the GUIDs to be consumed by the protocols does exists in the database of protocol and capable to identifying if the driver can be loaded or not.

After formerly loaded in SMM the DXE SMM runtime driver becomes capable utilize only minor set of services. While in its constructor entry point, the driver can use EFI Boot Services as it executes within space of boot service and SMM.

In secondary entry point in SMM driver is capable to perform:

- Registration of an interface - In the formal protocol database, naming the SMM occupant interfaces with future-loaded SMM drivers
- Registration with the SMM codebase - For a callback hook in effect to an SMI pin stimulation or an SMI based interrupt message from outside of SMM Code (i.e. a boot service, runtime agent)

After this **constructor** phase in SMM, the SMM driver needs not be rely on any other boot services as the mode of operation to carrying out execution can move away from these services. Many EFI Runtime Services could possess the majority of their execution shifted into SMM and viewable runtime portion simply becomes a proxy which merely utilizes the `EFI_SMM_BASE_PROTOCOL` to perform callback in SMM to carry out services. By Possessing a proxy which allows for a modal of sharing code blocks of error handling, like services for flash access and also the EFI Runtime Services `GetVariable()` or `SetVariable()`.

2.3.4.2 SMM Drivers for IA-32

In SMM the “IA-32 runtime drivers” can not called as on the image the action by `SetVirtualAddress()` is performed. Hence, code segment that requires to be accessible among SMM and EFI runtime needs to be migrated in SMM.

2.3.4.3 "Itanium® Processor Family" SMM Drivers

From "Platform Management Interrupt (PMI)" the runtime drivers for the "Itanium® processor family" are called as if each of them is kind of "Position Independent Code (PIC) runtime driver".

2.3.5 SMM Protocols

System Architecture of SMM broke in to below two parts:

- "SMM Base Protocol" - exposed by the processor. This protocol is liable to perform:
  - To initialize the state of processor
  - Registration of the handlers
- "SMM Access Protocol" - interprets the specific enabling and locking mechanism that an IA-32 memory controller may allows during execution in SMM. (Not needed for "Itanium® processor family")

2.3.5.1 SMM Protocols for IA-32

Figure 2.24 shows the SMM protocols which are published for an IA-32 system.

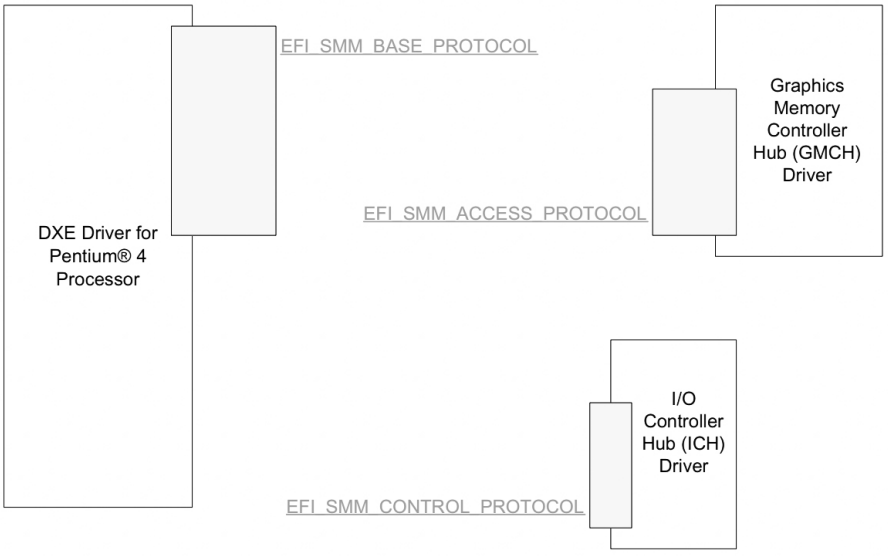


FIGURE 2.24: Protocols Published for IA-32 Systems

2.3.5.2 SMM Protocols for "Itanium®-Based Systems"

Figure 2.25 shows the way SMM protocols are published for an "Itanium®-Based system".

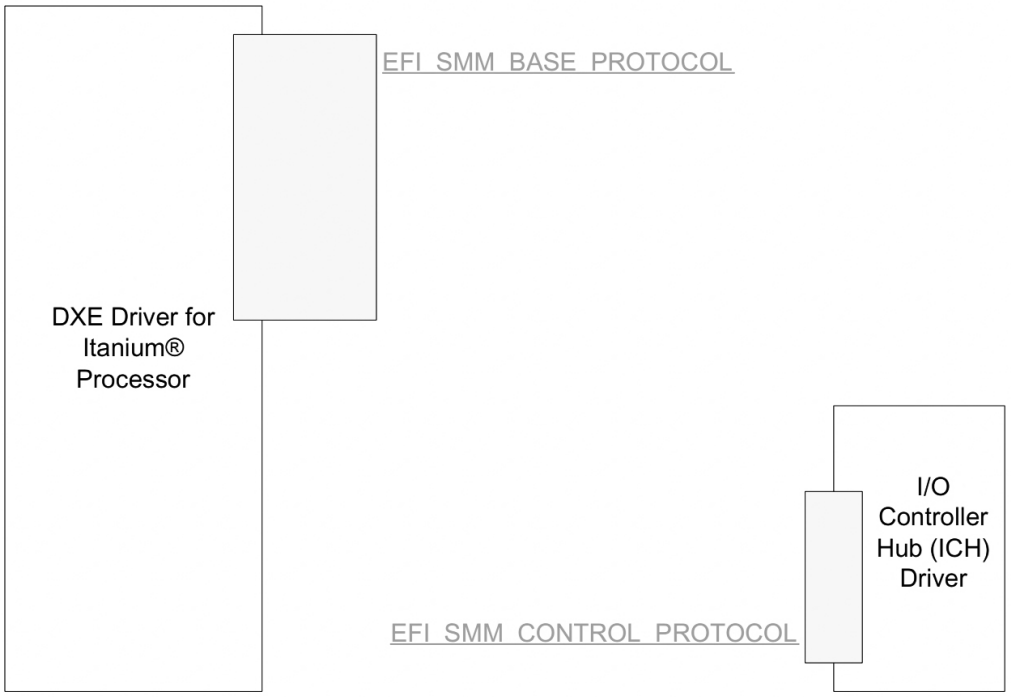


FIGURE 2.25: Protocols Published for "Itanium®-Based Systems"

2.3.6 SMM Dispatcher and infrastructure

SMM Code segment lies within the SMM Dispatcher. Major Role of SMM Dispatcher is to give the control mode to the SMM handlers in a systematic methodology. SMM Infrastructure Code aids to drive communication for SMM to SMM. SMM handlers are PE32+ images.

2.3.7 Initializing SMM Phase

The SMM driver for the Framework is fundamentally a enrollment transport mode to dispatch the drivers in outcome to the:

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

2.3.8 Relation of "System Management RAM (SMRAM)" to conventional memory

Figure 2.26 shows relationship between SMRAM and main memory in IA-32. Where SMRAM is isolated secure part inside the conventional main memory.

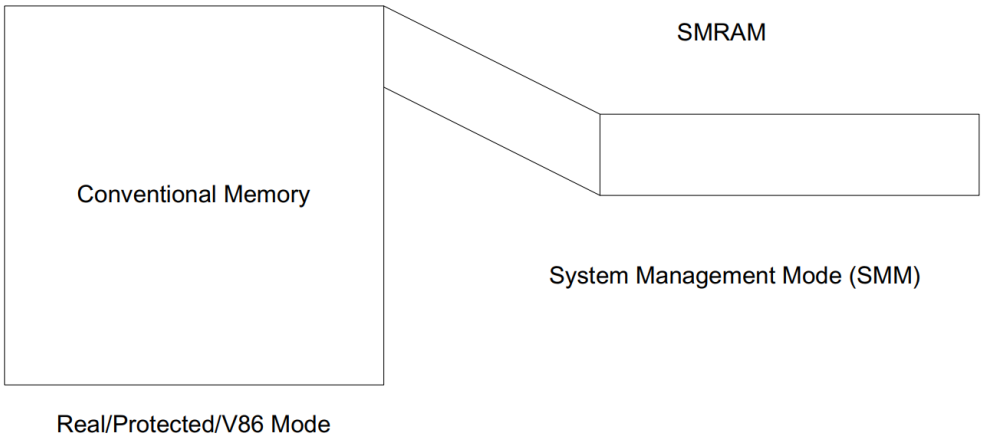


FIGURE 2.26: SMRAM kinship with conventional memory

2.3.9 Execution Mode of SMM on Processor

SMM is acceded asynchronously with the ongoing main flow of program. SMM was primitively developed to be clear to the OS and provide a power management facility more transparent.

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

- Applicable Workaround for SoC exaggeration
- Logging of error(s)
- Security for the platform

A SMI can be launched by energizing either the SMI logic pin via dedicated on the board or by utilizing the local APIC.

"Itanium® architecture" possess no independent separate mode for processor for the tractability of interruption however it does supports "Platform Management Interrupt (PMI)" which indeed is a maskable interruption. However, there is this another way to enter PMI using a interrupt message on local "Streamlined Advanced Programmable Interrupt Controller (SAPIC)".

This architecture informs a techniques to load modules of needful code segment that substantiate the functionality specified above. The internal representation of protocol which enables the loading of images of various handler and runs in normal memory of boot-services. Only the handlers does to run in SMRAM.

2.3.10 Accessing Platform Resources

As par policy outcome process of the execution of SMM handlers is reasonably prevents from accessing conventional memory resources. Hence, there does not exist any ease binding technique such as a call or trap interface to render the services in preemptive bid of non-SMM state.

Besides, SMM Services - the library of service which abides a sub set of the core EFI services, i.e. device input-output protocol, memory allocation and others. Also, execution mode of SMM driver has the equivalent structure as per the EFI criterion - namely a components which executes under boot services and it could perhaps run in runtime mode. When `ExitBootServices()` invoked, the mechanism of an unregister event occurs.

Chapter 3

Implementation

3.1 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.

3.1.1 Stake holders

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

- **BIOS development team** : main development group in contributing BIOS firmware, this is the only stake holder 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

3.1.2 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 stake holders which are not meant to be the BIOS developer
- Runtime BIOS Support for temporary UEFI variable creation

3.1.3 Requirements

3.1.3.1 Software Requirements

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

3.1.4 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.

3.1.5 Module: Setup Knob modification

3.1.5.1 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.

3.1.5.2 Additional Tech Stack Used

Below are the listed technologies consumed in development of this module in addition to the already specified requirements in Section 3.1.3

- Tkinter
- XML
- JSON

3.1.5.3 Flow of the module

Figure 3.1 describes the flow of setup knobs modification on the System Under Test (SUT).

3.1.5.4 Screenshots of Module

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 3.2 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 3.3)

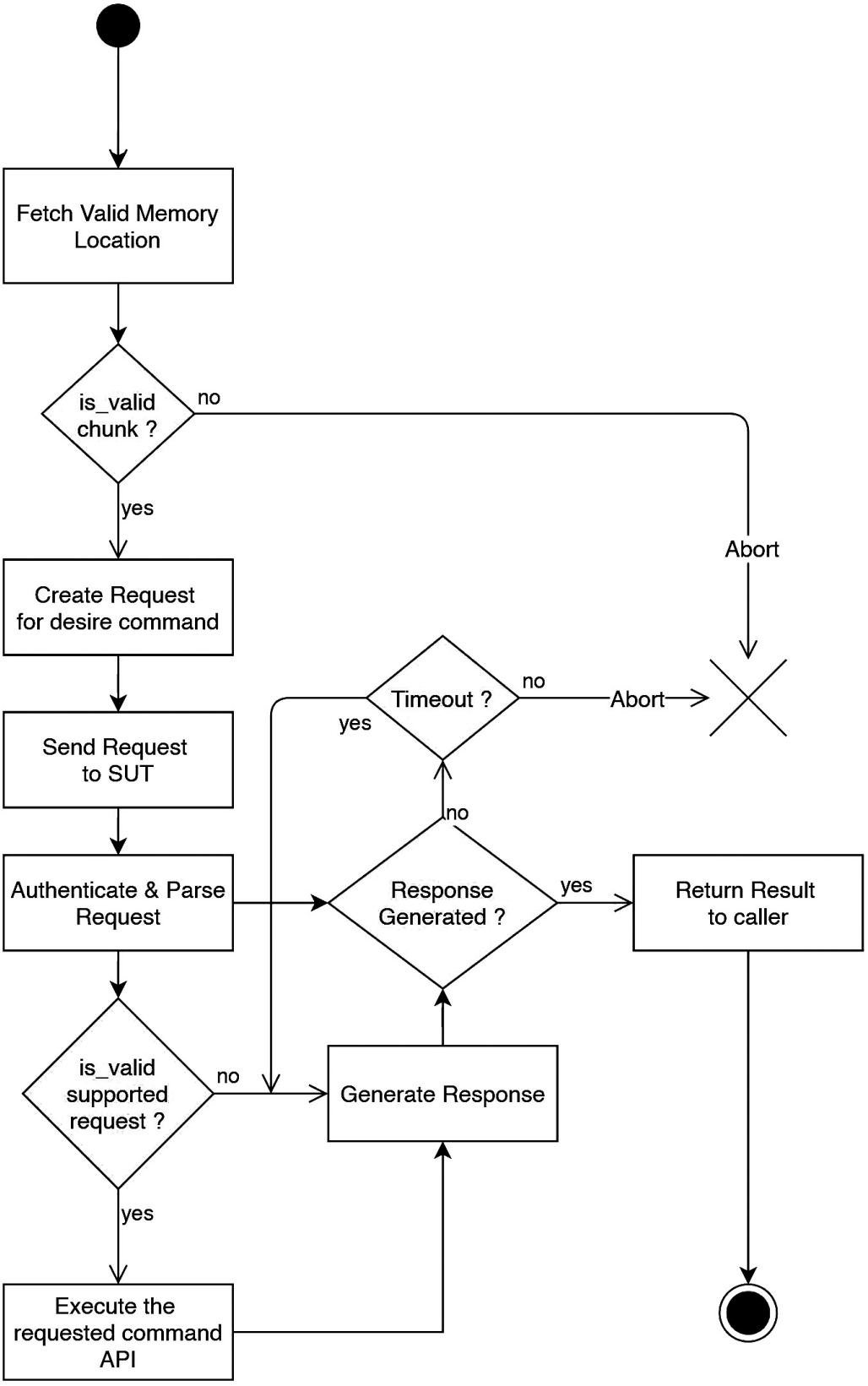


FIGURE 3.1: Flow of Setup Knobs Modification

  - `online` - to work on SUT and require to select valid access method for online mode from menu
  - `offline` - to work on BIOS binary

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

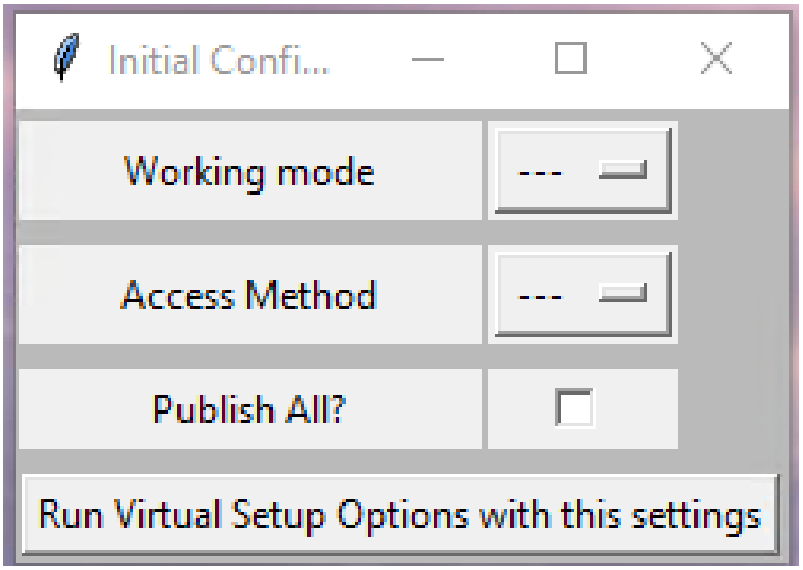


FIGURE 3.2: Menu to Select initial configuration for work

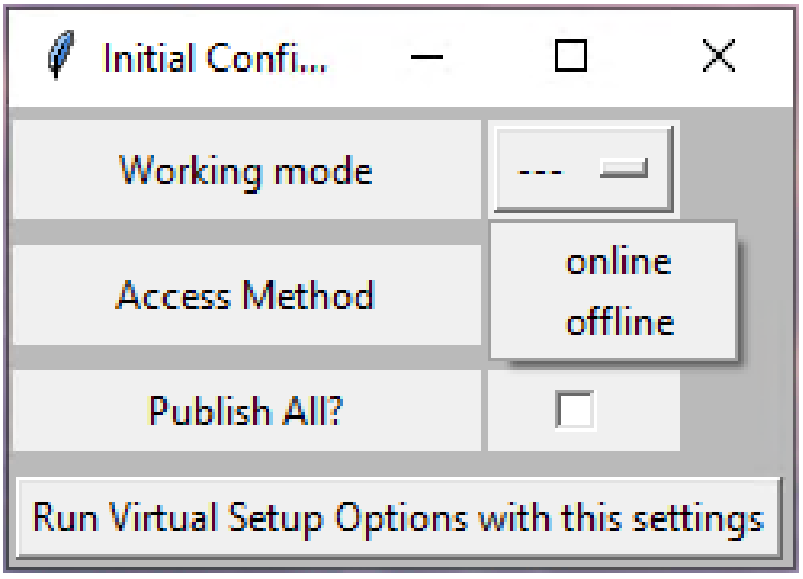


FIGURE 3.3: Available work mode for the system: Online and Offline

Table 3.1 describes the interpretation of each button action on specific condition as remarks if applicable

Table 3.1: 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

3.1.5.5 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 3.1.5.1.
- 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.

3.1.6 Module: Parsing

Figure 3.4 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 2.2

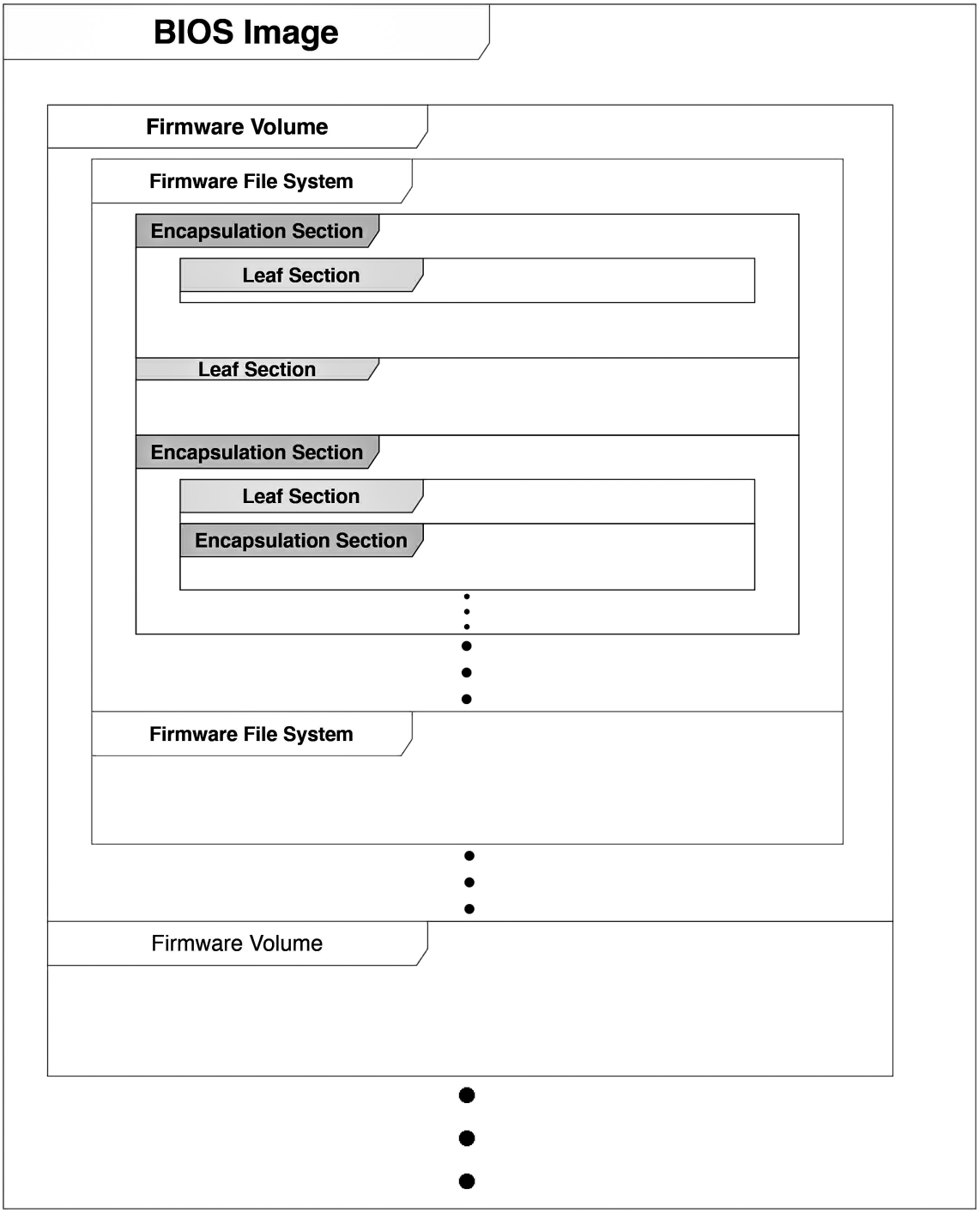


FIGURE 3.4: Overview of BIOS image as a File System

3.1.6.1 Additional Tech Stack Used

Below are the listed technologies consumed in development of this module in addition to the already specified requirements in Section 3.1.3

- Decompression binaries
- XML
- JSON

3.1.6.2 Flow of the module

Figure 3.5 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.

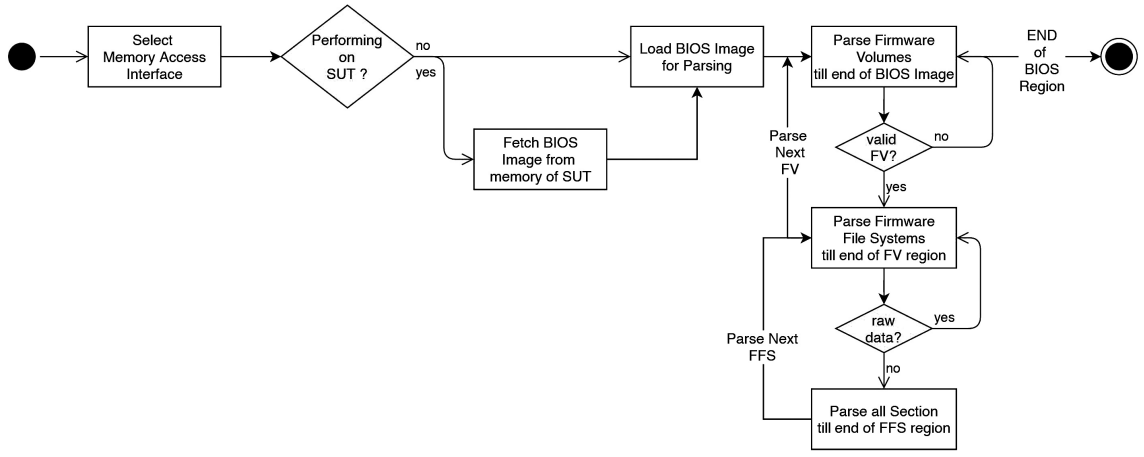


FIGURE 3.5: Flow of Parser

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 3.4. 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.

3.1.6.3 Outcome of Module

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

3.1.7 Module: 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 2.3.

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.

3.1.7.1 Additional Tech Stack Used

Below are the listed technologies consumed in development of this module in addition to the already specified requirements in Section 3.1.3

- Flask
- Ajax
- jQuery
- Javascript

- HTML/CSS
- XML
- JSON

3.1.7.2 Flow of the module

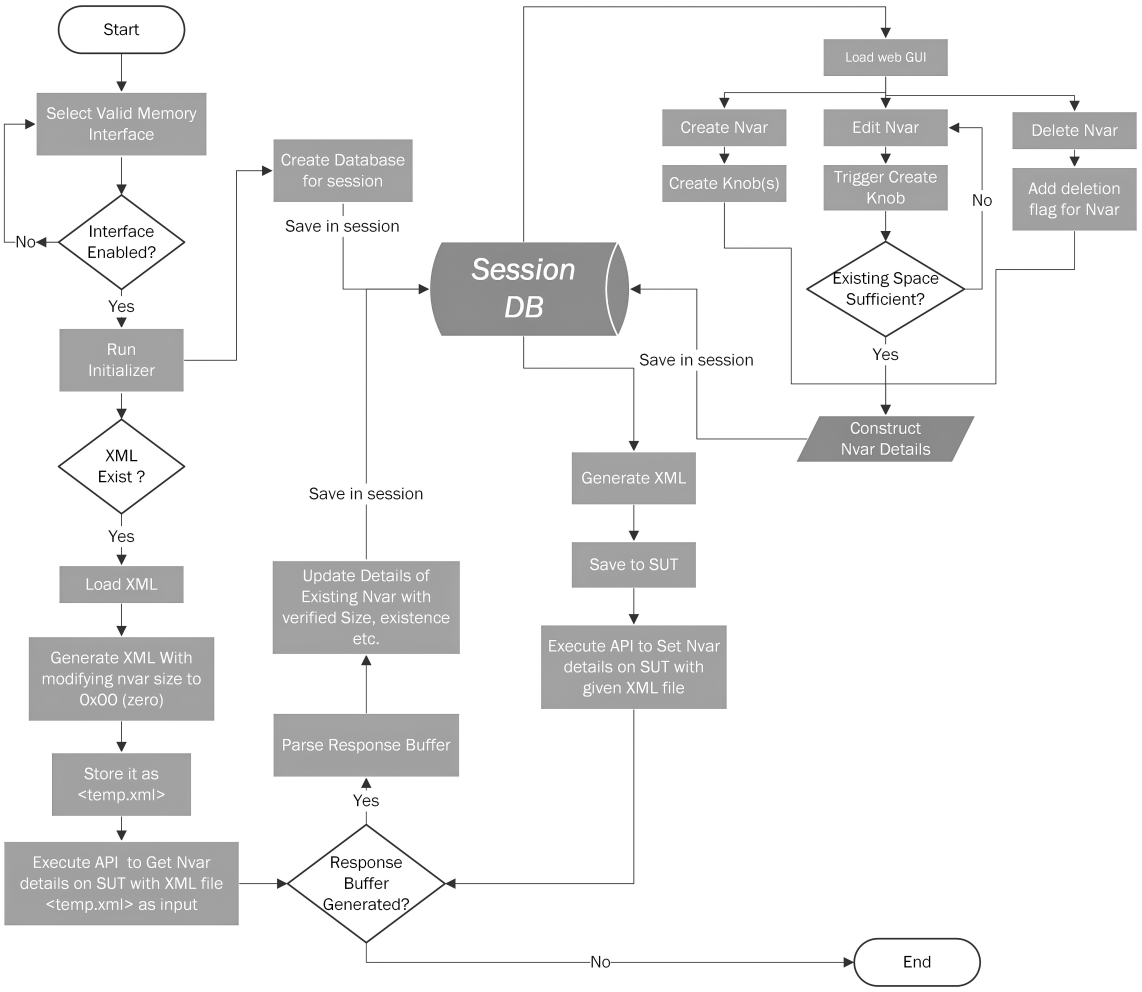


FIGURE 3.6: Flow of Nvar Web GUI

The Flow of the module is described in section 3.1.7.3 along with screenshots which is easier to interpret the flow chart in Figure 3.6.

3.1.7.3 Screenshots of Module

Whenever the User launches the module the home page screen to select valid communication interface will appear as displayed in Figure 3.7. This is the crucial stage as if valid interface for communication is not selected one may not be able to use the functionality of the service.

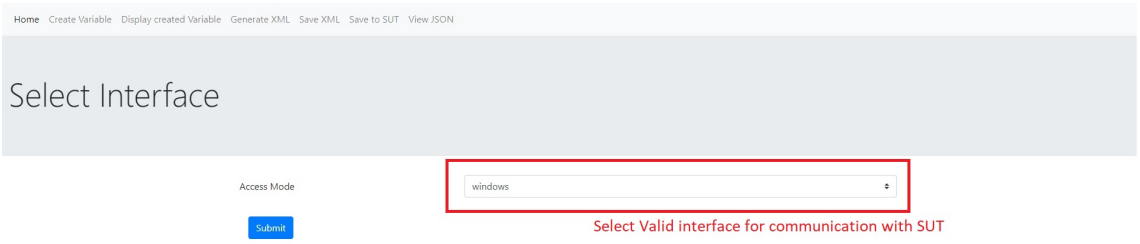


FIGURE 3.7: Home Page to Create UEFI Variable

TABLE 3.2: Navigation Bar Action

Button	Interpretation
Create Variable	Opens a form to create new Variable as in Figure 3.9
Display Created Variable	lists out created variable as in Figure 3.8
Generate XML	Generate XML from the stored session database as in Figure 3.15
Save XML	Saves the generated XML on the storage device
Save to SUT	Applies the Pending changes action (Create/Delete/Modify) to the SUT
View JSON	View the stored session database in the json format as in Figure 3.16

After selection of valid Interface one may operate the desired options listed in navigation bar which are:

Figure 3.8 lists the variable created under the current session which is to be applied

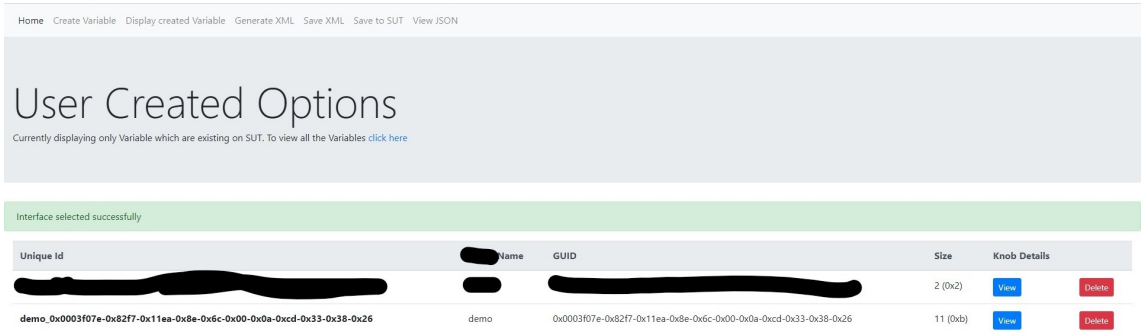


FIGURE 3.8: Variables created or exists on SUT

Figure 3.9 displays form which allows user to create Variable, where user needs to specify the name of the variable with certain restriction of input field. To identify and lookup the Variable the GUID is required which is automatically generated by the module with required format, however if user wishes then they can modify the GUID.

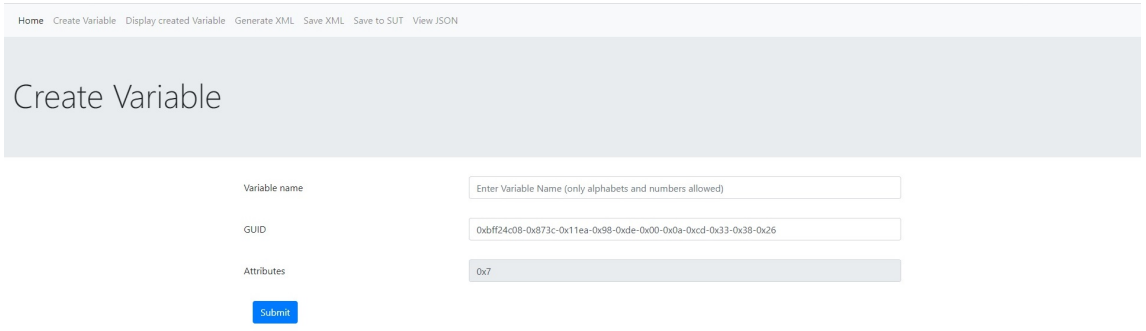


FIGURE 3.9: Create new UEFI Variable on SUT

Figure **??** opens the list of the options if created and allows to edit their current values too. However one can also add the new option to the Variable. It allows user to create various types of options under the variable which are oneof type as in Figure 3.11, string type as in Figure 3.12, numeric type as in Figure 3.13 and the

checkbox type which allows user to toggle the option value in as Boolean interpretation. Common fields for creating options including its name, type, description and size.

If user wants to change the value set for the variable created while creation of option as described in Figure 3.10 forms one can actually modify the value.

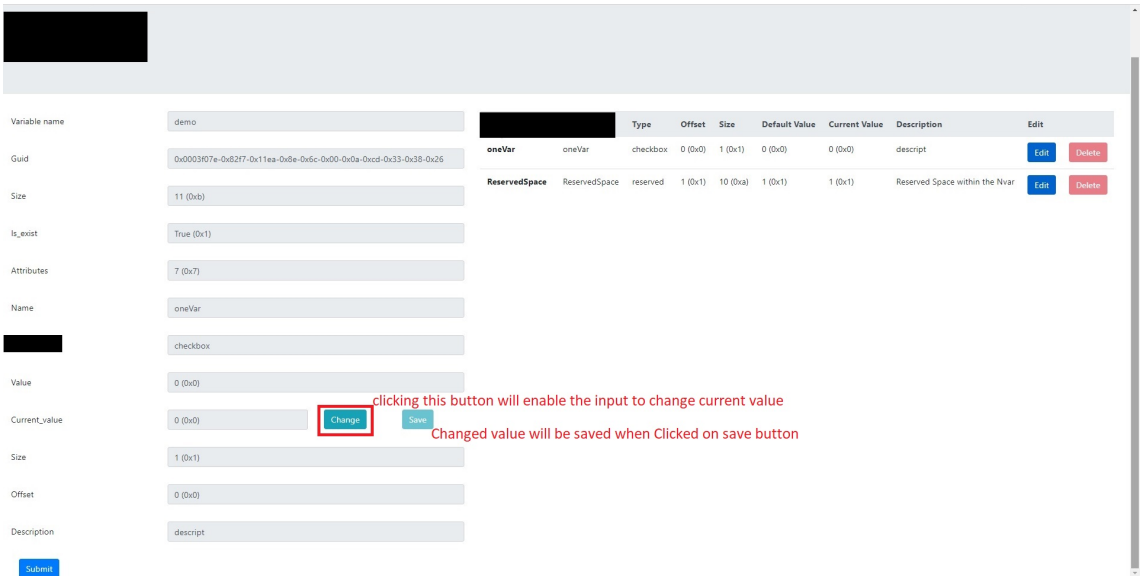


FIGURE 3.10: Edit the Existing Option Created under Variable SUT

The highlighted prompt in Figure 3.11 allows user to create the choices for the option where one of the multiple values to be selected as a result, By clicking `Add Option` button user can create choices and under drop down menu besides Value, user can select default value to be selected for the option.

Option type string as in Figure 3.12 allows user to create a option which accepts minimum and maximum characters to be supported in the string as well as the default string value to be selected.

To set the numeric input for the option, minimum and maximum value along with the default value to be set as in Figure 3.13

For the future use one can create a reserved space under the UEFI variable as in Figure 3.14

Figure 3.15 shows the XML which is generated from the existing and newly created variables and options under it.

Figure 3.16 represents session data of existing and newly created data (if any) as json

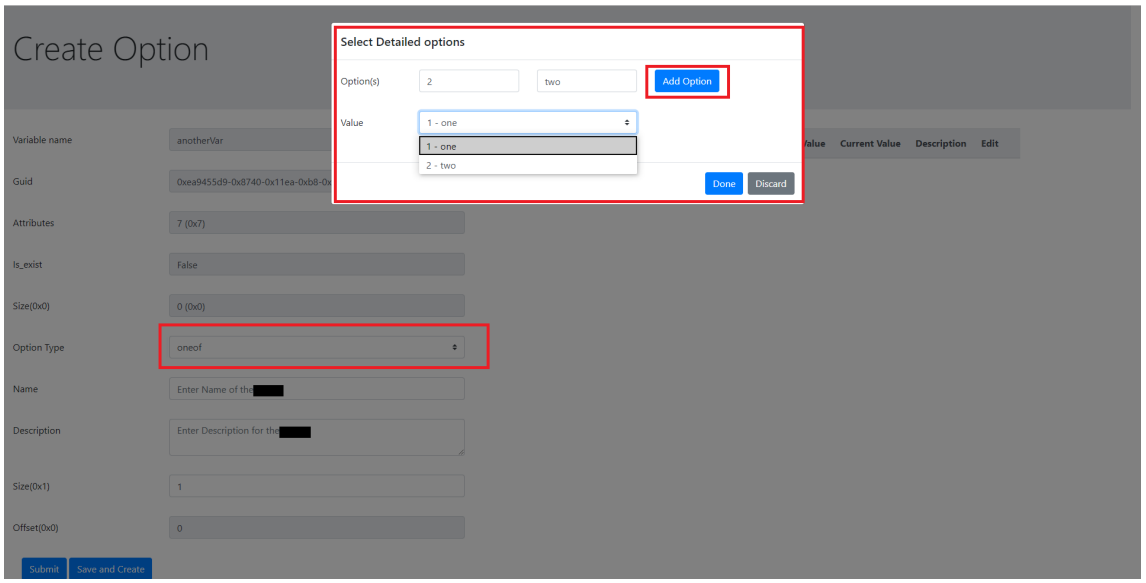


FIGURE 3.11: Create New Option(s) under Variable - Oneof Type

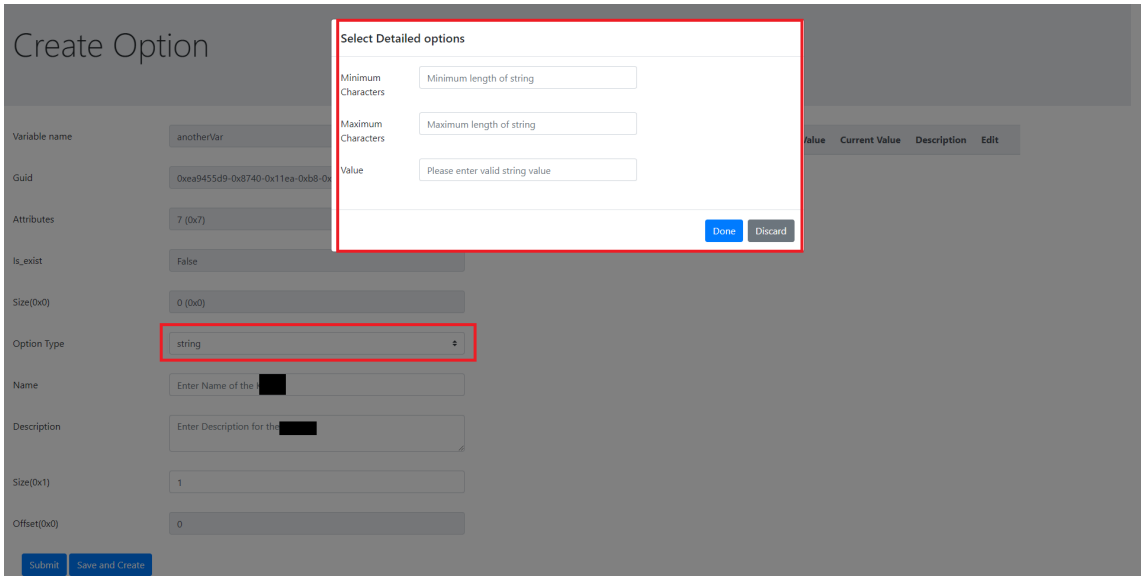


FIGURE 3.12: Create New Option(s) under Variable - String Type

3.1.7.4 Outcome of the module

- Enables creation of UEFI variable from OS layer.
- Lifts headache of maintaining variable creation from BIOS development for individuals.
- Interactive Web GUI for better User Experience
- Quick Modification of already created Variable

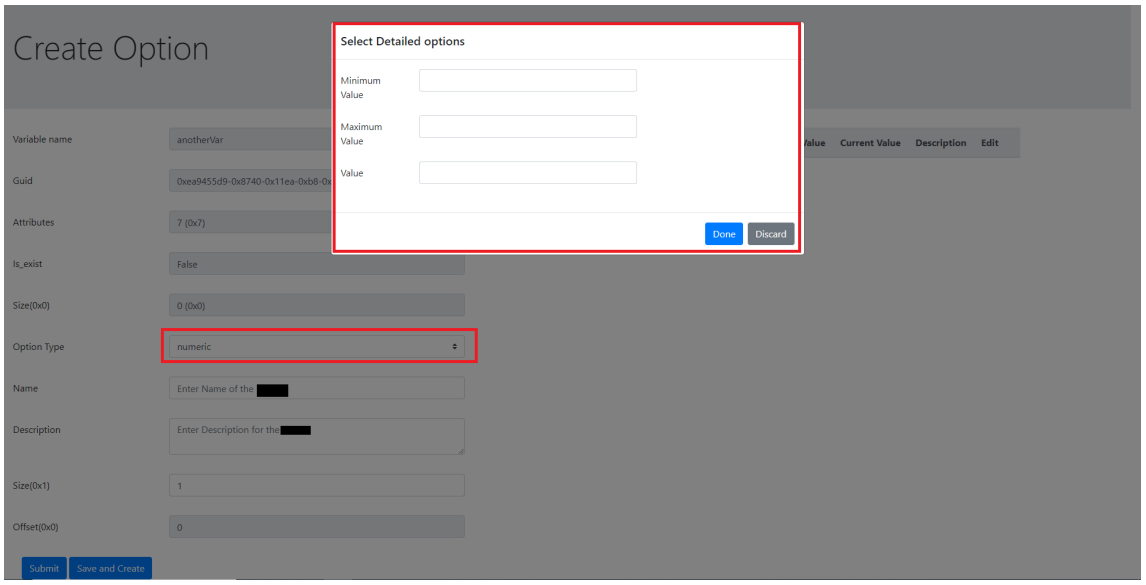


FIGURE 3.13: Create New Option(s) under Variable - Numeric Type

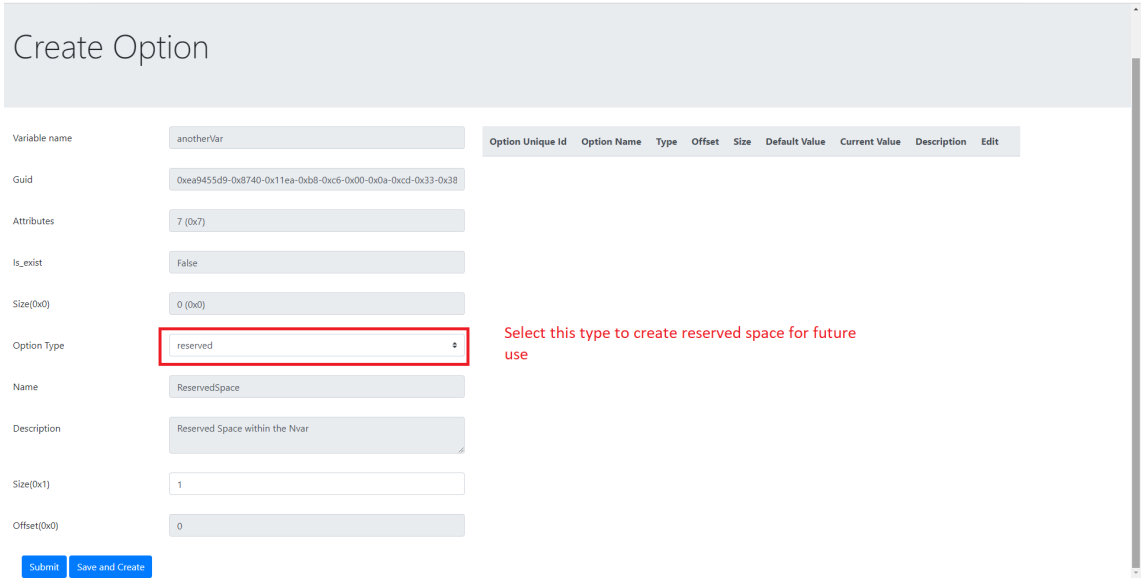
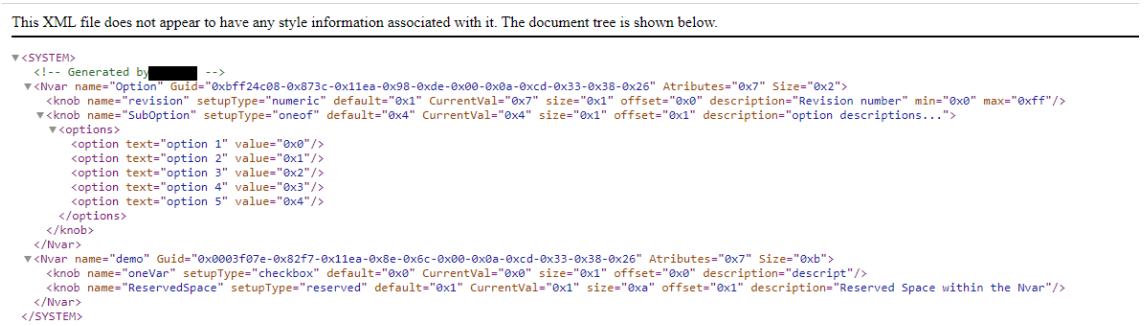


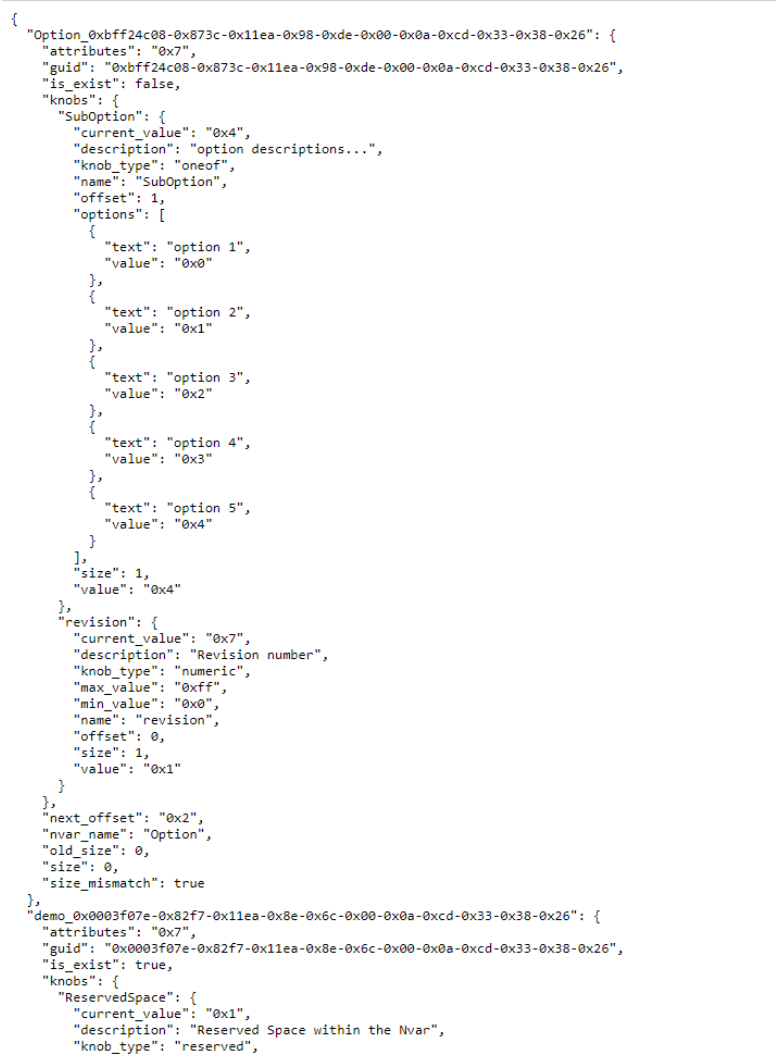
FIGURE 3.14: Create Reserved Space for future use under Variable



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<SYSTEM>
  <!-- Generated by [redacted] -->
  <Nvar name="Option" Guid="0xbff24c08-0x873c-0x11ea-0x98-0xde-0x00-0x0a-0xcd-0x33-0x38-0x26" Atributes="0x7" Size="0x2">
    <knob name="revision" setupType="numeric" default="0x1" CurrentVal="0x7" size="0x1" offset="0x0" description="Revision number" min="0x0" max="0xff"/>
    <knob name="SubOption" setupType="oneof" default="0x4" CurrentVal="0x4" size="0x1" offset="0x1" description="option descriptions...">
      <options>
        <option text="option 1" value="0x0"/>
        <option text="option 2" value="0x1"/>
        <option text="option 3" value="0x2"/>
        <option text="option 4" value="0x3"/>
        <option text="option 5" value="0x4"/>
      </options>
    </knob>
  </Nvar>
  <Nvar name="demo" Guid="0x0003f07e-0x82f7-0x11ea-0x8e-0x6c-0x00-0x0a-0xcd-0x33-0x38-0x26" Atributes="0x7" Size="0xb">
    <knob name="oneVar" setupType="checkbox" default="0x0" CurrentVal="0x0" size="0x1" offset="0x0" description="descript"/>
    <knob name="ReservedSpace" setupType="reserved" default="0x1" CurrentVal="0x1" size="0xa" offset="0x1" description="Reserved Space within the Nvar"/>
  </Nvar>
</SYSTEM>
```

FIGURE 3.15: Generate XML SUT




```
{
  "Option_0xbff24c08-0x873c-0x11ea-0x98-0xde-0x00-0x0a-0xcd-0x33-0x38-0x26": {
    "attributes": "0x7",
    "guid": "0xbff24c08-0x873c-0x11ea-0x98-0xde-0x00-0x0a-0xcd-0x33-0x38-0x26",
    "is_exist": false,
    "knobs": {
      "SubOption": {
        "current_value": "0x4",
        "description": "option descriptions...",
        "knob_type": "oneof",
        "name": "SubOption",
        "offset": 1,
        "options": [
          {
            "text": "option 1",
            "value": "0x0"
          },
          {
            "text": "option 2",
            "value": "0x1"
          },
          {
            "text": "option 3",
            "value": "0x2"
          },
          {
            "text": "option 4",
            "value": "0x3"
          },
          {
            "text": "option 5",
            "value": "0x4"
          }
        ],
        "size": 1,
        "value": "0x4"
      },
      "revision": {
        "current_value": "0x7",
        "description": "Revision number",
        "knob_type": "numeric",
        "max_value": "0xff",
        "min_value": "0x0",
        "name": "revision",
        "offset": 0,
        "size": 1,
        "value": "0x1"
      }
    },
    "next_offset": "0x2",
    "nvar_name": "Option",
    "old_size": 0,
    "size": 0,
    "size_mismatch": true
  },
  "demo_0x0003f07e-0x82f7-0x11ea-0x8e-0x6c-0x00-0x0a-0xcd-0x33-0x38-0x26": {
    "attributes": "0x7",
    "guid": "0x0003f07e-0x82f7-0x11ea-0x8e-0x6c-0x00-0x0a-0xcd-0x33-0x38-0x26",
    "is_exist": true,
    "knobs": {
      "ReservedSpace": {
        "current_value": "0x1",
        "description": "Reserved Space within the Nvar",
        "knob_type": "reserved",
```


FIGURE 3.16: View JSON representation SUT

Chapter 4

Future Scope

4.1 Future Scope of Work

Few implementation modules of Section 3.1 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