RAMAIAH
Institute of Technology

UEFI BIOS Compatibility for Multiple SOC Based on Device Capability Detection

Submitted in Partial fulfillment of Requirement for the award of the Degree of

MASTER OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

Submitted By

ASHRAF ALI S

1MS18SCS03

Under Guidance of

Dr. Mohana Kumar S

Associate Professor

Dept. of CSE RIT, Bangalore

Department of Computer Science and Engineering

RAMAIAH INSTITUTE OF TECHNOLOGY

(Autonomous Institute, Affiliated to VTU)
Accredited by National Board of Accreditation & NAAC with 'A' Grade
MSR Nagar, MSRIT Post, Bangalore-560054
www.msrit.edu

2020

RAMAIAH
Institute of Technology

CERTIFICATE

This is to certify that the dessertation work entitled **"UEFI BIOS Compatibility for Multiple SOC Based on Device Capability Detection"** is carried out by Ashraf Ali S (1MS18SCS03), a bonafide student of Ramaiah Institute of Technology, Bangalore, in partial fulfillment for the award of Master of Technology in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum, during the year 2019-20. It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the thesis deposited in the departmental library. The thesis has been approved as it satisfies the academic requirements in respect to dessertation work prescribed for the said degree.

Guide	HOD	Principal
Dr. Mohana Kumar S	**Dr. Anita Kanavalli**	**Dr. N V R Naidu**
Associate Professor	**HOD and Professor**	**Principal**
Dept. of CSE, RIT	**Dept. of CSE, RIT**	**Ramaiah Institute of Technology**

Name & Signature of Examiners with Date: -

1)

2)

RAMAIAH
Institute of Technology

DECLARATION

I, Ashraf Ali S, a student of Master of Technology in Computer Science and Engineering, Ramaiah Institute of Technology, Bangalore hereby declare that the project entitled **"UEFI BIOS Compatibility for Multiple SOC Based on Device Capability Detection"** has been carried out independently at the Institute under the Guidance of Dr. Mohana Kumar S, Associate Professor, Department of CSE, Bangalore. I hereby declare that work submitted in this thesis is my own, except where acknowledged in the text and has not been previously submitted for the award of the degree of Visvesvaraya Technological University, Belgaum or any other institute or University.

Date: 20-Sep-20 **ASHRAF ALI S**

Place: Bangalore **1MS18SCS03**

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, Direct Media Interface, Peripheral Component Interconnect (PCI), and Advanced Configuration and Power Interface (ACPI) for every Intel System on a Chip (SoC). For each IP functional flow of each generation will vary. Currently Intel BIOS supports only one generation of each IP and its functional flow in its Firmware Image. If we change a processor N with N+1 generation the system won't boot. Each Processor has its own Boot Flow. In our Simics environment we have designed a firmware in such a way that it should able to boot with N and N+1 generation processor with a same BIOS and support features of driver which are supported by that processor. This project involves working on development of new features for Intel's upcoming processor, focusing on BIOS development for IPs residing on North or Uncore part of SOC. Creating a single silicon package that supports all platforms as well as across generations.

ACKNOWLEDGEMENT

The project work could not have been successful without the guidance and inspiration that I received from all the people around me. I feel it obligatory to acknowledge the contribution of these people.

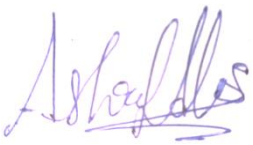
I express my deep gratitude and respectful regards to Dr. N.V.R Naidu, Principal, RIT, who provided me with all the facilities, both technical and non-technical.

It is with a deep sense of gratitude that I express my heartfelt and sincere thanks to Dr. Anita Kanavalli, Professor and Head, Department of Computer Science and Engineering.

I am also highly indebted to the Proctor in charge Dr. Jayalakshmi D. S, Associate Professor, Department of Computer Science and Engineering, for his guidance and constant supervision as well as providing necessary information regarding the Project.

The project work mapped its way through and was able to achieve my desired success due to inspirational guidance provided by my guide Dr. Mohana Kumar S, Associate Professor, Department of Computer Science and Engineering, who was along with me during every inch of development in the project work.

Heartfelt regards to all types of help, co-operation and encouragement received from all the staff (teaching and non-teaching) of Department of Computer Science and Engineering, and my friends, who helped me to pass through the practical difficulties I encountered during the project work.



Ashraf Ali S

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The interface between the OS and platform in is defined in the Unified Extensible Firmware Interface (UEFI) is a specification. UEFI describes a programmatic interface to a platform [1]. The platform includes the chipset, CPU, motherboard and other components. Pre-operating system (PreOS) agents are allowed in the UEFI Spec, Pre-OS agents includes OS loader and other applications that the system needs for applications to execute and interoperate, including UEFI Drivers and applications [1]. The cornerstones for understanding UEFI application and drivers are several UEFI concepts they're defining within the UEFI specifications.

The helpful frameworks in the UEFI concepts to keep in mind as you study the specifications:

- UEFI based firmware objects – used to manage system state, including I/O devices, memory, and events.
- System Table – It's the interface with system function calls and data information table in the form of data structures.
- Handle database and Protocols – Interface Register which are callable.
- UEFI Images – The code can deploy in the executable content format.
- Device Paths – The hardware locations of an entity are described in the data structures, such as the spindle, partition, bus, and file name of a UEFI image etc.

The UEFI provides a driver model for support of devices that attach to today's industry standard buses such as Peripheral Component Interconnect (PCI) and Universal Serial Bus (USB) and architectures of tomorrow. design and implementation of device drivers are simplified in the UEFI Model and produce small executable image sizes. A device driver must produce a Driver Binding Protocol on an equivalent image handle on which the driver was loaded. It then waits for system firmware to connect the driver to a controller. These device controllers are producing a protocol on the controller's device handle that abstracts of I/O operations.

1.1. Problem Statement

Every Intel Processor has its only Silicon initialization. So that for each processor has its own Firmware to initialize. In the Firmware other than silicon initialization and IPs like PCIe, Graphics for a specific version remains same. So, whenever user tried to change the processor the system won't boot because that firmware doesn't have that silicon

initialization process. For that user needs to update their firmware based on the processor they are using. And when customer want switch back to older processor which can't be possible. The only option for customer is that they should upgrade their processor.

1.2. Objectives

Following are the objectives of this project implementation:

- Implementing the Firmware in such a way that firmware remains same for multiple processor.
- Implementing the common IP code structure for implementing the BIOS compatibility.

1.3. Scope

The scope of the project includes the implementation firmware for multiple processor supports so that whenever customer wants to change the processor, they just need to replace the existing SOC with the newer SOC so that they can get the required speed based on the processor capability.

1.4. Report Organization

This thesis provides a detailed description of the project "Intel Next Gen Client BIOS Development for Uncore IPs".

Chapter 2 talks about the Background of BIOS, UEFI, PCIe, GOP, and ACPI specifications and the description of each one of them.

Chapter 3 discusses the design of UEFI, platform boot flow and its working on each stage and build process for firmware.

Chapter 4 discusses the Architecture of BIOS firmware images and its sections.

Chapter 5 discusses the software requirement specification of the project which explains the functional and non-functional requirements. It also gives a brief explanation of software and hardware requirements for the project.

Chapter 6 gives the detailed design of the project with the architecture diagram.

Chapter 7 gives details about the implementation of the project. It talks about system setup and process of implemented in the project.

Chapter 8 concludes the project which indicates the project outcome. It also discusses future work that can be done to improve the project.

2 BACKGROUND

2.1. Uncore Intellectual Properties

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

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

2.2. LEGACY BIOS and UEFI

First code to execute at boot Initialize the hardware Establish root-of-trust then Hand-off to the operating system. BIOS is the dominant standard which deftness a firmware interface. "Legacy" (as in Legacy BIOS), in the context of firmware specifications, refer to an older, widely used specification. Major responsibility of BIOS is to set up the hardware, load and start an Operating System (OS). When the system boots, the BIOS initializes and identifies system devices including video display card, mouse, hard disk drive, keyboard, solid state drive and other hardware followed by locating software held on a boot device. a hard disk or removable storage such as CD/DVD or USB and loads and executes that software, giving it control of the computer. This process is also referred to as "booting" or "boot strapping". Table 1 overviews of comparisons of UEFI with legacy BIOS

Table 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	x*86 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

2.2.1. 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 can run different Operating System (OS), such as MS-DOS, equally well on systems other than IBM. Additionally, Legacy BIOS has a defined OS-independent interface for hard- ware that enables interrupts to communicate with video, disk and keyboard services along with the BIOS ROM loader and bootstrap loader, to name a few. Use of legacy BIOS is diminishing and is expected to be phased out in new systems by the year 2020.

2.2.2. Limitations of legacy BIOS

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

market demands opened the opportunity for a fresh BIOS architecture to be developed and introduced to the market. The UEFI specifications and resulting implementations have begun to effectively address these persisting market needs.

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

2.3. Unified Extensible Firmware (UEFI)

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

The UEFI Forum board of directors consists of representatives from 11 industry leaders as described in Figure1. These involved organizations work to ensure that the UEFI specifications meet industry needs.

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

Figure 2.1 Board of Directors of UEFI Forum

2.3.1. UEFI Driver Model Extension

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

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

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

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

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

2.3.2. UEFI Goals

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

The main properties of this specification can be summarized by these attributes:

- Coherent, scalable platform environment. The specification defines a complete solution for the firmware to describe all platform features and surface platform capabilities to the OS during the boot process. The definitions are rich enough to cover a range of contemporary processor designs.
- Abstraction of the OS from the firmware. The specification defines interfaces to platform capabilities. Using abstract interfaces, the specification allows the OS loader to be constructed with far less knowledge of the platform and firmware that underlie those interfaces. The interfaces represent a well-defined and stable boundary between the underlying platform and firmware implementation and the OS loader. Such a boundary allows the underlying firmware and the OS loader to change provided both limit their interactions to the defined interfaces.
- Reasonable device abstraction free of legacy interfaces. "PC-AT" BIOS interfaces require the OS loader to have specific knowledge of the workings of certain hardware devices. This specification provides OS loader developers with something different: abstract interfaces that make it possible to build code that works on a range of underlying hardware devices without having explicit knowledge of the specifics for each device in the range.
- Abstraction of Option ROMs from the firmware. This specification defines interfaces to platform capabilities including standard bus types such as PCI, USB, and SCSI. The list of supported bus types may grow over time, so a mechanism to extend to future bus types is included. These defined interfaces, and the ability to extend to future bus types, are components of the UEFI Driver Model. One purpose of the UEFI Driver Model is to solve a wide range of issues that are present in

existing "PC-AT" option ROMs. Like OS loaders, drivers use the abstract interfaces so device drivers and bus drivers can be constructed with far less knowledge of the platform and firmware that underlie those interfaces.

- Architecturally shareable system partition. Initiatives to expand platform capabilities and add new devices often require software support. In many cases, when these platform innovations are activated before the OS takes control of the platform, they must be supported by code that is specific to the platform rather than to the customer's choice of OS. The traditional approach to this problem has been to embed code in the platform during manufacturing (for example, in flash memory devices). Demand for such persistent storage is increasing at a rapid rate. This specification defines persistent store on large mass storage media types for use by platform support code extensions to supplement the traditional approach. The definition of how this works is made clear in the specification to ensure that firmware developers, OEMs, operating system vendors, and perhaps even third parties can share the space safely while adding to platform capability.

2.3.3. UEFI Design Overview

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

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

- **Runtime services**. A minimal set of runtime services is presented to ensure appropriate abstraction of base platform hardware resources that may be needed by the OS during its normal operations.

Figure 2.1 shows the principal components of UEFI and their relationship to platform hardware and OS software.

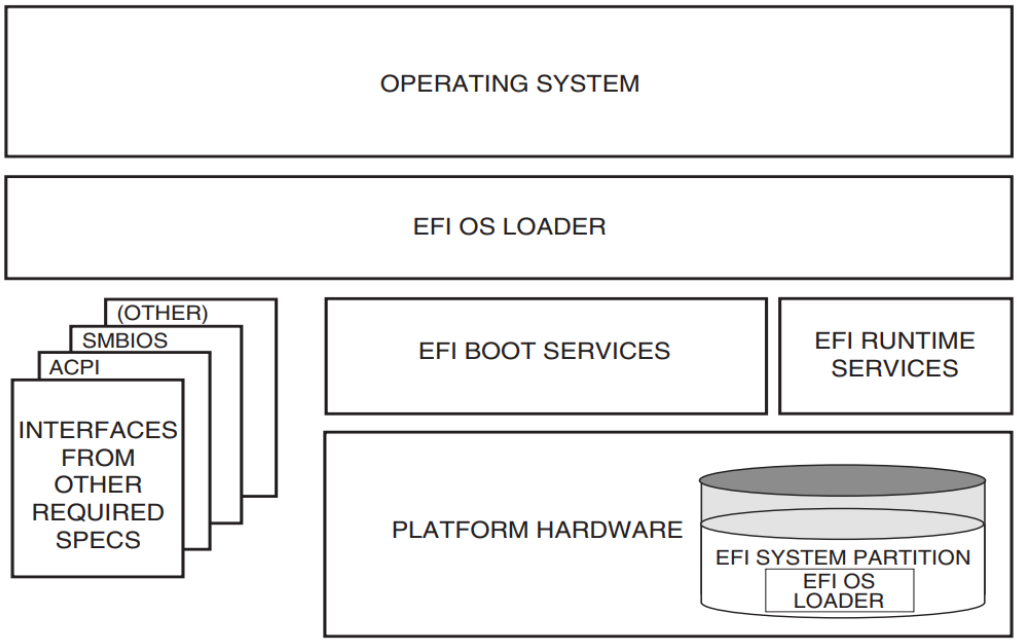


Figure 2.2 UEFI Conceptual Overview

Figure 2.2 illustrates the interactions of the various components of an UEFI specification-compliant system that are used to accomplish platform and OS boot [3].

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

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

2.3.4. UEFI Driver Goals

The UEFI Driver Model has the following goals:

- **Compatible** – Drivers conforming to this specification must maintain compatibility with the EFI and the UEFI Specification. This means that the UEFI Driver Model takes advantage of the extensibility mechanisms in the UEFI Specification to add the required functionality.
- **Simple** – Drivers that conform to this specification must be simple to implement and simple to maintain. The UEFI Driver Model must allow a driver writer to concentrate on the specific device for which the driver is being developed. A driver should not be concerned with platform policy or platform management issues. These considerations should be left to the system firmware.
- **Scalable** – The UEFI Driver Model must be able to adapt to all types of platforms. These platforms include embedded systems, mobile, and desktop systems, as well as workstations and servers.
- **Flexible** – The UEFI Driver Model must support the ability to enumerate all the devices, or to enumerate only those devices required to boot the required OS. The minimum device enumeration provides support for more rapid boot capability, and the full device enumeration provides the ability to perform OS installations, system maintenance, or system diagnostics on any boot device present in the system.
- **Extensible** – The UEFI Driver Model must be able to extend to future bus types as they are defined.
- **Portable** – Drivers written to the UEFI Driver Model must be portable between platforms and between supported processor architectures.
- **Interoperable** – Drivers must coexist with other drivers and system firmware and must do so without generating resource conflicts.
- **Describe complex bus hierarchies** – The UEFI Driver Model must be able to describe a variety of bus topologies from very simple single bus platforms to very complex platforms containing many buses of various types.
- **Small driver footprint** – The size of executables produced by the UEFI Driver Model must be minimized to reduce the overall platform cost. While flexibility and extensibility are goals, the additional overhead required to support these must be kept to a minimum to prevent the size of firmware components from becoming unmanageable.

- **Address legacy option rom issues** – The UEFI Driver Model must directly address and solve the constraints and limitations of legacy option ROMs. Specifically, it must be possible to build add-in cards that support both UEFI drivers and legacy option ROMs, where such cards can execute in both legacy BIOS systems and UEFI-conforming platforms, without modifications to the code carried on the card. The solution must provide an evolutionary path to migrate from legacy option ROMs driver to UEFI drivers.

2.4. Advanced Configuration and Power Interface

The ACPI Component Architecture (ACPICA) defines and implements a group of software components that together create an implementation of the ACPI specification. A major goal of the architecture is to isolate all operating system dependencies to a relatively small translation or conversion layer (the OS Services Layer) so that the bulk of the ACPICA code is independent of any individual operating system. Therefore, hosting the ACPICA code on new operating systems requires no source changes within the ACPICA code itself.

The components of the architecture include:

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

In the Figure 2.3, the ACPICA subsystem is shown in relation to the host operating system, device driver, OSPM software, and the ACPI hardware.

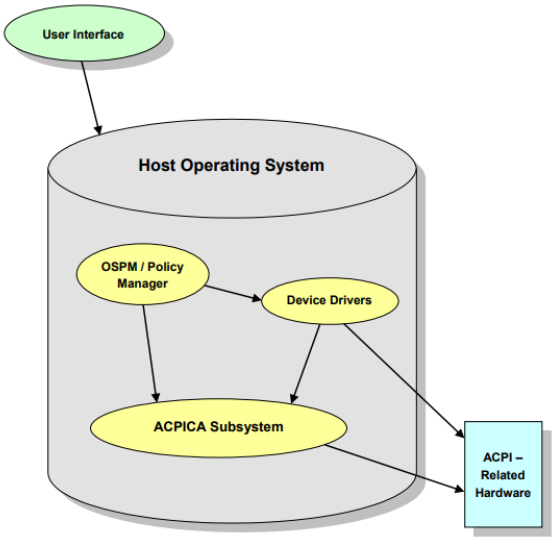


Figure 2.3The ACPI Component Architecture

2.4.1. Overview of ACPICA Subsystem

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

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

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

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

When combined into a single static or loadable software module such as a device driver or kernel subsystem, these two major components form the ACPICA Subsystem. Throughout this document, the term “ACPICA Subsystem” refers to the combination of the OS-independent ACPICA Subsystem with an OS Services Layer components combined into a single module, driver, or load unit.

2.4.2. OS-independent ACPICA Subsystem

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

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

The ACPICA Subsystem implementation makes no assumptions about the host operating system or environment. The only way it can request operating system services is via interfaces provided by the OS Services Layer. The primary user of the services provided by the ACPICA Subsystem are the host OS device drivers and power/thermal management software.

2.4.3. Operating System Services Layer

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

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

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

The overall ACPI Component Architecture in relation to the host operating system is Figure 2.4

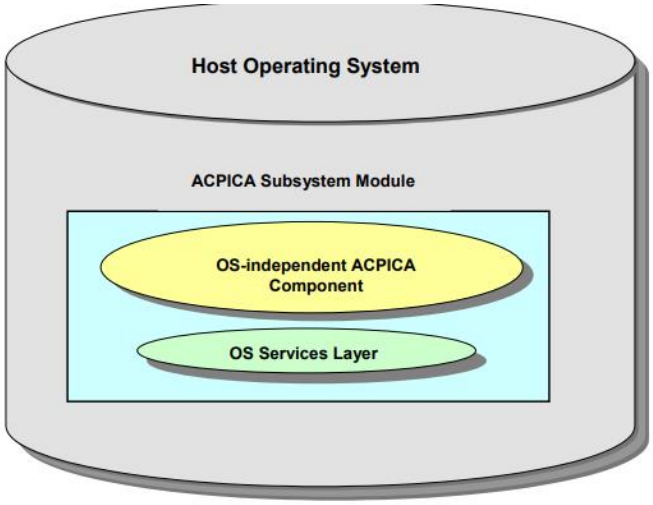


Figure 2.4 ACPICA Subsystem Architecture

2.4.4. ACPICA Subsystem Interaction

The ACPICA Subsystem implements a set of external interfaces that can be directly called from the host OS. These ACPI interfaces provide the actual ACPI services for the host. When operating system services are required during the servicing of an ACPI request, the Subsystem makes requests to the host OS indirectly via the fixed AcpiOs interfaces. The diagram below illustrates the relationships and interaction between the various architectural elements by showing the flow of control between them. Note that the OS-independent ACPICA Subsystem never calls the host directly – instead it makes calls to the AcpiOs interfaces in the OSL. This provides the ACPICA code with OS-independence. The Interaction between the Architectural Components Is shown in Figure 2.5

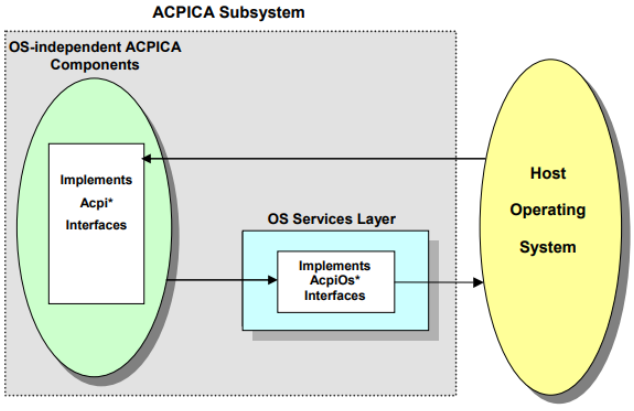


Figure 2.5 Interaction between the Architectural Components

2.5. Peripheral Component Interconnect Express

The PCI architecture has proven to be successful beyond even the most optimistic expectations. Today nearly every new computer platform comes outfitted with multiple

PCI slots. In addition to the unprecedented number of PCI slots being shipped, there are also hundreds of PCI adapter cards that are available to satisfy virtually every conceivable application. This enormous momentum is difficult to ignore.

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

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

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

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

applications. It offers the performance needed for today's and tomorrow's applications in an easy-to-adopt, backward-compatible standard.

2.5.1. Functional Description

PCI BIOS functions provide a software interface to the hardware used to implement a PCI based system. Its primary usage is for generating operations in PCI specific address spaces (configuration space and Special Cycles).

PCI BIOS functions are specified to operate in the following modes of the X86 architecture. The modes are: real-mode, 16:16 protected mode (also known as 286 protected mode), 16:32 protected mode (introduced with the 386), and 0:32 protected mode (also known as "flat" mode, wherein all segments start at linear address 0 and span the entire 4-GB address space).

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

2.5.2. UEFI PCI Services

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

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

2.5.3. UEFI Driver Model

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

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

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

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

2.5.4. Graphics Output Protocol

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

2.5.5. BUS Performances and Number of Slots Compared

The various architectures defined by the PCISIG. The table shows the evolution of bus frequencies and bandwidths., as it obvious, increasing bus frequency compromises the number of electrical loads or number of connectors allowable on a bus at that frequencies. At some point for a given bus architecture there is an upper limit beyond which one cannot further increase the bus frequency, hence requiring the definition of a new bus architecture.

A PCI express (PCIe) Interconnect that connects two devices is referred to as a Link. A link consists if either x1, x2, x4, x8, x12, x16 or x32 signal pairs in each direction. These signals are referred to as Lanes. A designer determines how many Lanes to implement based on the targeted performance benchmark required on a given Link.

Table 2 Comparison of Bus Frequency, Bandwidth and Number of Slots

Bus Type	Clock Frequency	Peak Bandwidth *	Number of Card Slots per Bus
PCI 32-bit	33 MHz	133 MBytes/sec	4-5
PCI 32-bit	66 MHz	266 MBytes/sec	1-2
PCI-X 32-bit	66 MHz	266 MBytes/sec	4
PCI-X 32-bit	133 MHz	533 MBytes/sec	1-2
PCI-X 32-bit	266 MHz effective	1066 MBytes/sec	1
PCI-X 32-bit	533 MHz effective	2131 MByte/sec	1
*** Double all these bandwidth numbers for 64-bit bus implementations**			

The Table 2 shows aggregate bandwidth numbers for various Link width implementations, as is apparent from this table, the peak bandwidth achievable with PCIe is significantly higher than any existing bus today. Figure 2.6 shows the comparison of Bus Frequency, Bandwidth and the number of slots.

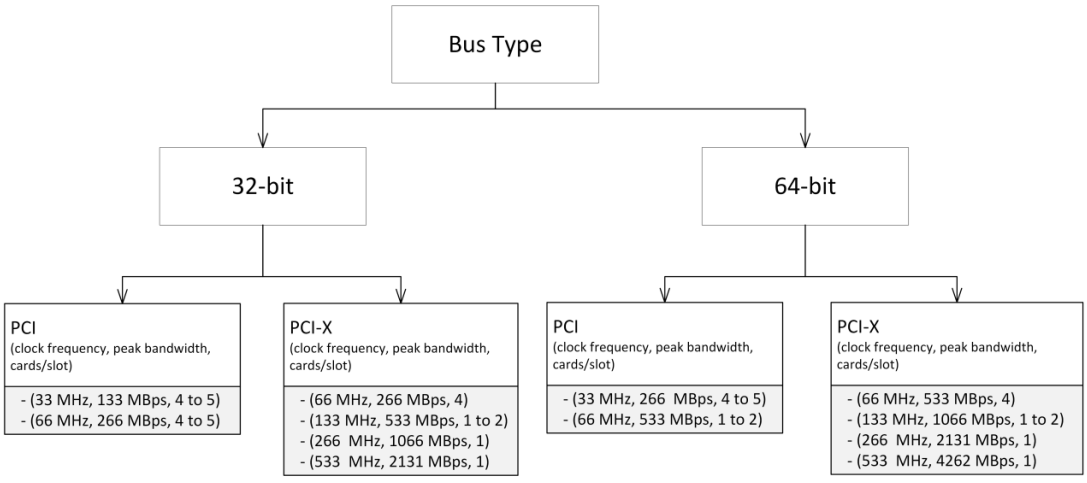


Figure 2.6: Comparison of Bus Frequency, Bandwidth and Number of Slots in Tree Structure

2.5.6. 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

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

2.6. Graphics Controller

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

- UEFI graphics drivers follow the UEFI driver model.
- Depending on the adapter that the driver manages, a graphics driver can be categorized as either a single or a multiple output adapter.
- The graphics driver must create child handles for each output.
- Graphics drivers must create child handles for some of the graphics output ports and attach the Graphics Output Protocol (GOP Protocol), EDID Discovered Protocol, and EDID Active Protocol to each active handle that the driver produced.
- Graphics drivers are chip-specific because of the requirement to initialize and manage the graphics device.

A UEFI driver is required for any PC hardware device needed for the boot process to complete. Hardware devices can be categorized into the following:

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

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

2.6.1. Graphics Output Protocol

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

2.6.2. GOP Overview

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

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

2.6.3. GOP DRIVER

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

Table 3 Difference between Video BIOS and GOP gives a quick comparison of GOP and video BIOS:

Table 3 Difference between Video BIOS and GOP

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

The GOP Driver is part of the UEFI boot time drivers responsible for bringing up the display during bios boot. This driver enables logo display during bios boot time. The GOP driver interacts with the PCI Driver which is responsible for enumerating the PCI devices such as graphics. For each controller detected on the PCI driver, it installs a PCI IO Protocol that get used by the underlying child driver for implementing its services.

The UEFI graphics driver (GOP Driver) is responsible for the Display unit functionality. It acts as a bus driver for the display subsystem and creates child devices for each of the output interfaces discovered by it. It uses the PCI IO protocol for implementing the GOP protocol that is installed on each of the output child devices created by it.

2.6.4. GOP INTEGRATION

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

- Platform firmware must be compliant to UEFI 2.1 or later.
- Platform must enumerate and initialize the graphics device.

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

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

Table 4 GOP driver files

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

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.

3 DESIGN

3.1. UEFI/PI Firmware Images

UEFI and PI specifications define the standardized format for EFI firmware storage devices (FLASH or other non-volatile storage) which are abstracted into "Firmware Volumes". Build systems must be capable of processing files to create the file formats described by the UEFI and PI specifications. The tools provided as part of the EDK II Base Tools package process files compiled by third party tools, as well as text and Unicode files in order to create UEFI or PI compliant binary image files. In some instances, where UEFI or PI specifications do not have an applicable input file format, such as the Visual Forms Representation (VFR) files used to create PI compliant IFR content, tools and documentation have been provided that allows the user to write text files that are processed into formats specified by UEFI or PI specifications. Figure 3.1 shows the Image creation of Firmware image of UEFI.

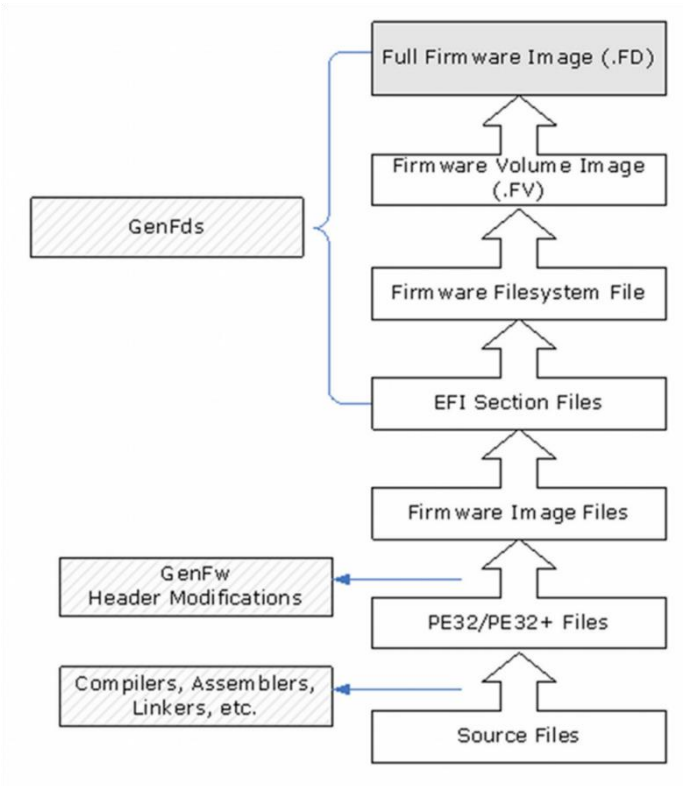


Figure 3.1 UEFI/PI Firmware Image Creation

A Firmware Volume (FV) is a file level interface to firmware storage. Multiple FVs may be present in a single FLASH device, or a single FV may span multiple FLASH devices. An FV may be produced to support some other type of storage entirely, such as a disk partition or network device. For more information consult the Platform Initialization

Specification, Volume 3. In all cases, an FV is formatted with a binary file system. The file system used is typically the Firmware File System (FFS), but other file systems may be possible in some cases. Hence, all modules are stored as "files" in the FV. Some modules may be "execute in place" (linked at a fixed address and executed from the ROM), while others are relocated when they are loaded into memory and some modules may be able to run from ROM if memory is not present (at the time of the module load) or run from memory if it is available. Files themselves have an internally defined binary format. This format allows for implementation of security, compression, signing, etc. Within this format, there are one or more "leaf" images. A leaf image could be, for example, a PE32 image for a DXE driver.

Therefore, there are several layers of organization to a full UEFI/PI firmware image. These layers are illustrated below in Figure 3.1 Each transition between layers implies a processing step that transforms or combines previously processed files into the next higher level. Also shown in Figure 3.1 are the reference implementation tools that process the files to move them between the different layers.

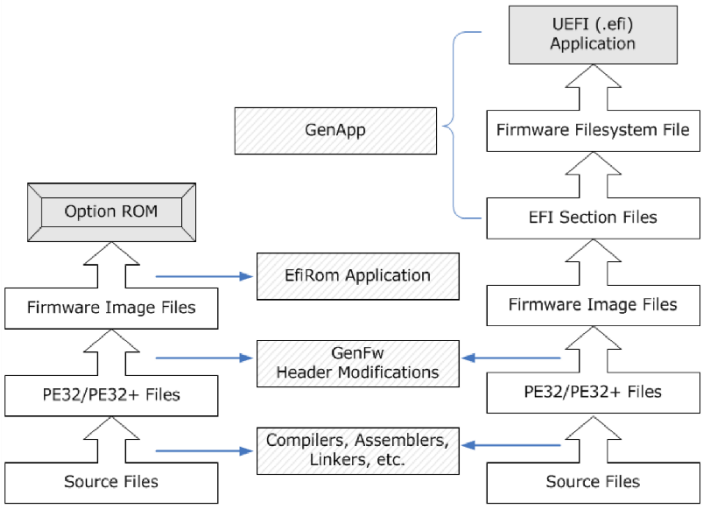


Figure 3.2 UEFI/PI Firmware Image Creation Flow

In addition to creating images that initialize a complete platform, the build process also supports creation of stand-alone UEFI applications (including OS Loaders) and Option ROM images containing driver code. Figure 3.2, shows the reference implementation tools and creation processes for both image types.

The final feature that is supported by the EDK II build process is the creation of Binary Modules that can be packaged and distributed for use by other organizations. Binary

modules do not require distribution of the source code. This will permit vendors to distribute UEFI images without having to release proprietary source code.

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

3.2. Platform Initialization Boot Sequence

PI compliant system firmware must support the six phases: security (SEC), pre-eft initialization (PEI), driver execution environment (DXE), boot device selection (BDS), run time (RT) services and After Life (transition from the OS back to the firmware) of system. Refer to Figure 3.3 below

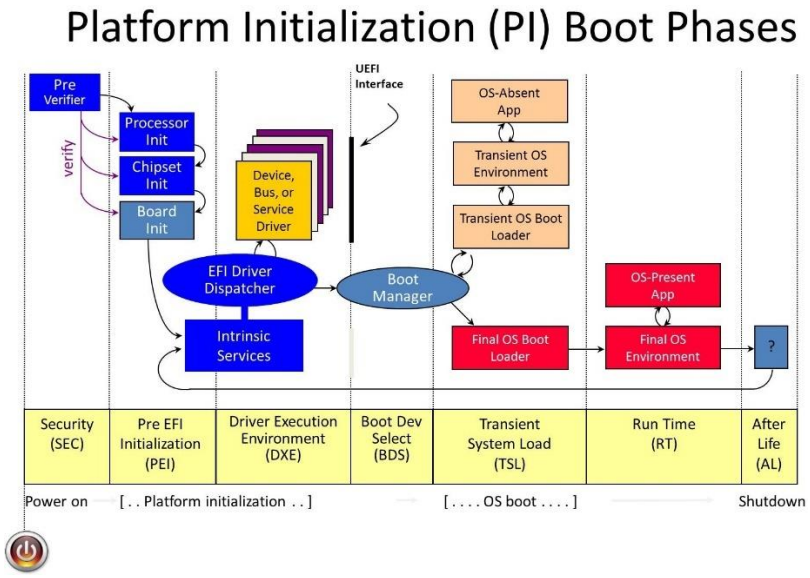


Figure 3.3 PI Boot Phases

3.2.1. Security Phase

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

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

The security section may contain modules with code written in assembly. Therefore, some EDK II module development environment (MDE) modules may contain assembly code.

Where this occurs, both Windows and GCC versions of assembly code are provided in different files

3.2.2. Pre-EFI Initialization (PEI)

The Pre-EFI Initialization (PEI) phase described in the PI Architecture specifications is invoked quite betimes in the boot period. Specifically, after about preliminary processing in the Security (SEC) phase, any machine restart event will invoke the PEI phase [4]. The PEI phase is designed to be developed in many parts and consists of:

- PEI Foundation (core code)
- Pre-EFI Initialization Modules (specialized plug-ins)

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

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

- Locating and validating PEIMs
- Dispatching PEIMs
- Facilitating communication between PEIMs

Providing handoff data to later phases These PEIMs are responsible for the following:

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

(DXE) phase Figure 3.3 shows a diagram describes the action carried out during the PEI phase

3.2.3. Drive Execution Environment (DXE)

Prior to the DXE phase, the Pre-EFI Initialization (PEI) phase is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position independent data structures called Hand-Off Blocks (HOBs). HOBs are described in detail in the Platform Initialization Specification.

There are several components in the DXE phase:

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

3.2.4. BOOT Device Selection (BDS)

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

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

3.2.5. Transient System Load (TSL) and Runtime (RT)

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

3.2.6. After Life (AL)

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

3.3. 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 most 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 enough, 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 a Section header followed by the section data (driver binary).

3.3.1. Extensible Firmware Interface Section Files

The general section format for sections less than 16MB in size is shown in Figure 3.4 shows the section format for sections 16MB or larger in size using the extended length field. Figure 3.5 shows section format of EFI.

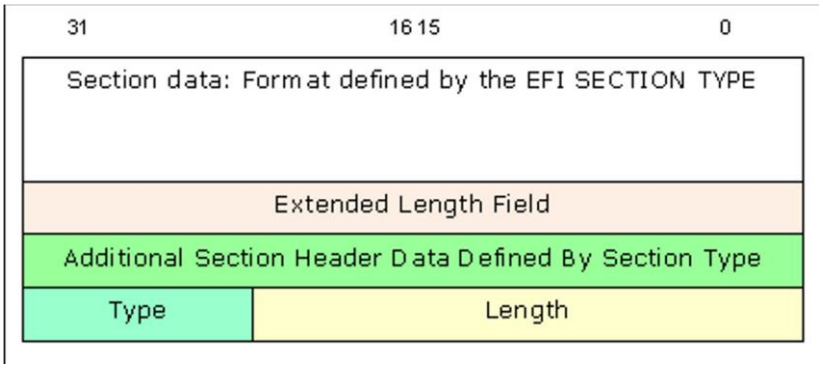


Figure 3.4 General EFI Section Format for large size Sections (greater than 16 MB)

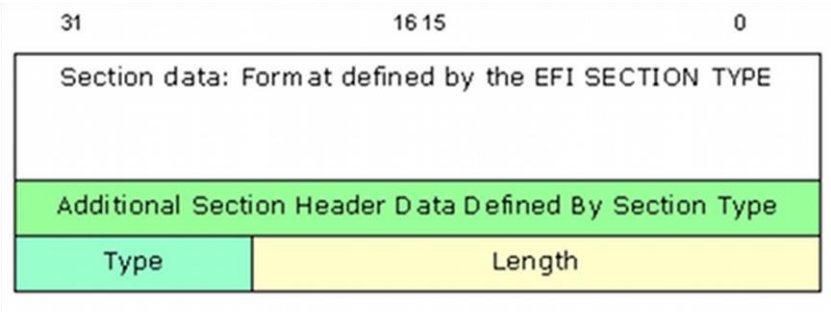


Figure 3.5 General EFI Section Format (less than 16 MB)

3.4. Firmware Support Package

The Firmware Support Package (FSP) provides chipset and processor initialization in a format that can easily be incorporated into many existing bootloaders. The FSP performs the necessary initialization steps as documented in the BIOS Writers Guide (BWG) / BIOS

Specification including initialization of the processor, memory controller, chipset and certain bus interfaces, if necessary.

FSP is not a stand-alone bootloader; therefore, it needs to be integrated into a bootloader to carry out other functions such as:

- Initializing non-Intel components
- Bus enumeration and device discovery
- Industry standards

3.4.1. FSP Binary Format

The FSP binary follows the UEFI Platform Initialization Firmware Volume Specification format. The Firmware Volume (FV) format is described in the Platform Initialization (PI) Specification. FV is a way to organize/structure binary components and enables a standardized way to parse the binary and handle the individual binary components that make up the Firmware Volume (FV). The Figure 3.6 shows the Component Logical View of FSP and Figure 3.7 shows Component Layout View of FSP. Each FSP component has an FSP_INFO_HEADER table and may optionally have additional tables. All FSP tables must have a 4-byte aligned base address and a size that is a multiple of 4 bytes. All FSP tables must be placed back-to-back. All FSP tables must begin with a DWORD signature followed by a DWORD length field. A generic table search algorithm for additional tables can be implemented with a signature search algorithm until a terminator signature ‘FSPP’ is found [2].

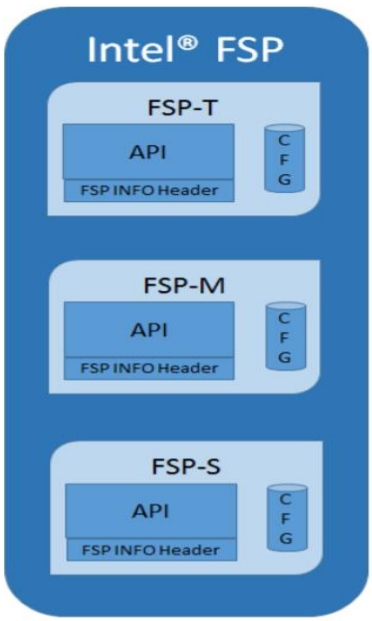


Figure 3.6: FSP Component Logical View

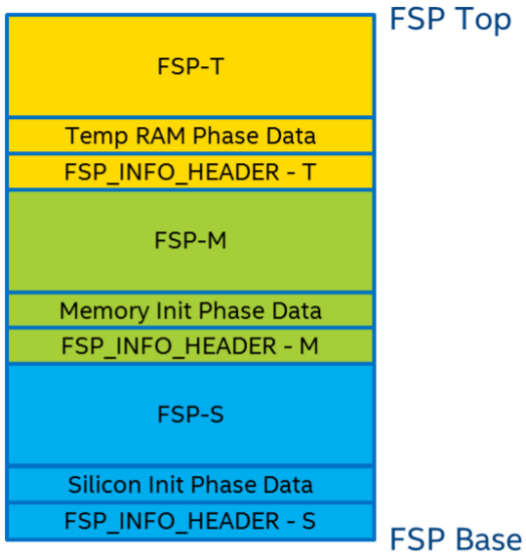


Figure 3.7: FSP Component Layout View

The FSP will have several components each containing one or more Firmware Volumes (FV). Each component provides a phase of initialization as below.

- FSP-T: Temp RAM initialization phase

  Primary purpose of this phase is to initialize the Temporary RAM along with any other early initialization. This phase consists of below FSP APIs

  - TempRamInit()
- FSP-M: Memory initialization phase Primary purpose of this phase is to initialize the permanent memory along with any other early silicon initialization. This phase consists of below FSP APIs
  - FspMemoryInit()
  - TempRamExit()
- FSP-S: Silicon initialization phase Primary purpose of this phase is to complete the silicon initialization including CPU and IO controller initialization. This phase consists of below FSP APIs
  - FspSiliconInit()
  - NotifyPhase() -Post PCI bus enumeration, Ready To Boot and End of Firmware.

3.4.2. Locating FSP_INFO_HEADER

The FSP_INFO_HEADER structure is stored in a firmware file, called the FSP_INFO_HEADER file and is placed as the first firmware file within each of the FSP

component's first FV. All firmware files will have a GUID that can be used to identify the files, including the FSP_INFO_HEADER file.

The FSP_INFO_HEADER file GUID is FSP_FFS_INFORMATION_FILE_GUID

#define FSP_FFS_INFORMATION_FILE_GUID \

{0x912740be, 0x2284, 0x4734, {0xb9, 0x71, 0x84, 0xb0, 0x27, 0x35, 0x3f, 0x0c}};

The bootloader can find the offset of the FSP_INFO_HEADER within the FSP component's first Firmware Volume (FV) by the following steps described below:

- Use EFI_FIRMWARE_VOLUME_HEADER to parse the FSP FV header and skip the standard and extended FV header.
- The EFI_FFS_FILE_HEADER with the FSP_FFS_INFORMATION_FILE_GUID is located at the 8-byte aligned offset following the FV header.
- The EFI_RAW_SECTION header follows the FFS File Header.
- Immediately following the EFI_RAW_SECTION header is the raw data. The format of this data is defined in the FSP_INFO_HEADER and additional header structures.

A pictorial representation of the data structures that is parsed in the above flow is provided below Figure 3.8.

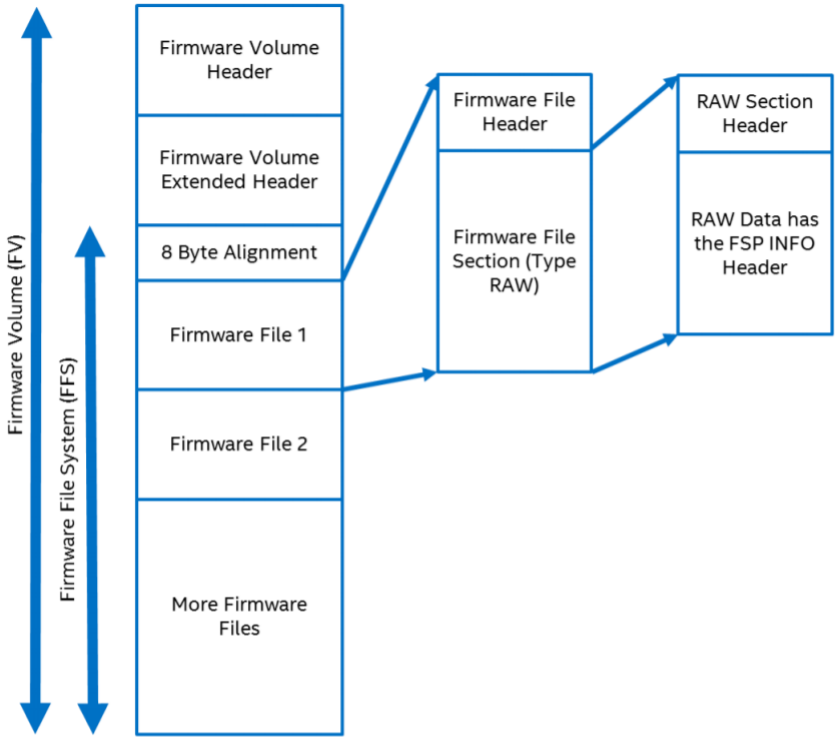


Figure 3.8: FSP Component Headers

3.4.3. FSP Description File

An FSP component may optionally include an FSP description file. This file will provide information about the FSP including information about different silicon revisions the FSP supports. The contents of the FSP description file must be an ASCII encoded text string [7].

The file, if present, must have the following file GUID and be included in the FDF file as shown below.

```
#define FSP_FFS_INFORMATION_FILE_GUID \
{0xd9093578, 0x08eb, 0x44df, {0xb9, 0xd8, 0xd0, 0xc1, 0xd3,0xd5, 0x5d, 0x96}};
#
# Description file
#
FILE RAW = D9093578-08EB-44DF-B9D8-D0C1D3D55D96 {
        SECTION RAW = FspDescription/FspDescription.txt
}
```

3.4.4. FSP Patch Table (FSPP)

FSP Patch Table contains offsets inside the FSP binary which store absolute addresses based on the FSP base. When the FSP is rebased, the offsets listed in this table needs to be patched accordingly [7].

A PatchEntryNum of 0 is valid and indicates that there are no entries in the patch table and should be handled as a valid patch table by the rebasing software. Table 5 shows FSPP – PatchData Encoding of each bit in FSP_PATCH_TABLE [7].

```
typedef struct {
        UINT32 Signature; ///< FSP Patch Table Signature “FSPP”
        UINT16 Length; ///< Size including the PatchData
        UINT8 Revision; ///< Revision is set to 0x01
        UINT8 Reserved;
        UINT32 PatchEntryNum; ///< Number of entries to Patch
        UINT32 PatchData[]; ///< Patch Data
} FSP_PATCH_TABLE;
```

Table 5: FSPP – PatchData Encoding

BIT [23:00]	Image OFFSET to patch
BIT [27:24]	Patch type 0000: Patch DWORD at OFFSET with the delta of the new and old base. NewValue = OldValue + (NewBase - OldBase) 1111: Same as 0000 Others: Reserved
BIT [28:30]	Reserved
BIT [31]	0: The FSP image offset to patch is determined by Bits [23:0] 1: The FSP image offset to patch is calculated by (ImageSize – (0x1000000 – Bits [23:0])) If the FSP image offset to patch is greater than the ImageSize in the FSP_INFO_HEADER, then this patch entry should be ignored.

3.4.5. FSP Boot Flow

1. Bootloader starts executing from Reset Vector
    a. Switches the mode to 32-bit mode
    b. Initializes the early platform as needed
    c. Finds FSP-T and calls TempRamInit () API. If Bootloader initializes the temporary memory this step and step2 can be skipped.
2. FSP initializes temporary memory and returns from TempRamInit () API.
3. Bootloader initializes the stack in temporary memory
    a. Initializes the platform as needed
    b. Finds FSP-M and calls the FspMemoryInit () API
4. FSP initializes memory and returns from FspMemoryInit () API.
5. Bootloader relocates itself to Memory.
6. Bootloader calls TempRamExit () API. If Bootloader initialized the temporary memory, this step and next step can be skipped.
7. FSP returns from TempRamExit () API.
8. Bootloader finds FSP-S and calls FspSiliconInit () API.
9. FSP returns from FspSiliconInit () API.
10. Bootloader continues and device enumeration.
11. Bootloader calls NotifyPhase () API with After PciEnumeration parameter.

12. Bootloader calls NotifyPhase () API with Ready to Boot parameter before transferring control to OS loader.
13. When booting to non-UEFI OS, Bootloader calls NotifyPhase () API with EndOfFirmware parameter immediately after Ready to Boot.
14. When booting to UEFI OS, Bootloader calls NotifyPhase () with EndOfFirmware parameter during ExitBootServices

Figure 3.9 shows the Boot Flow of FSP

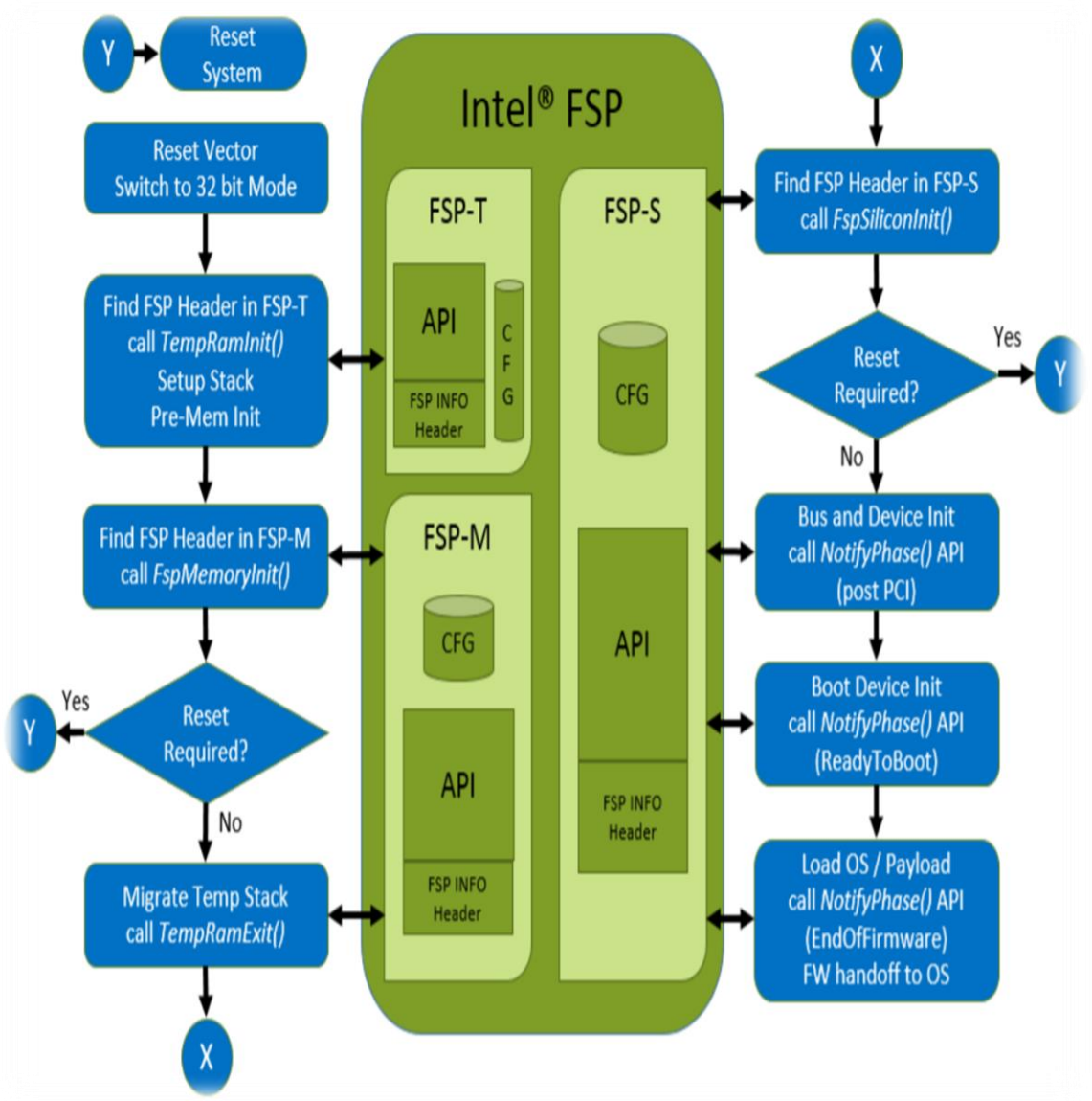


Figure 3.9: FSP Boot Flow

3.4.6. Peripheral Component Interconnect Configuration Space

The set of registers for each device which are referred as a device config space. In each device config space will have their own capability registers which will identify the capacity of those devices. The config space of Peripheral Component Interconnect is show in Figure 3.10.

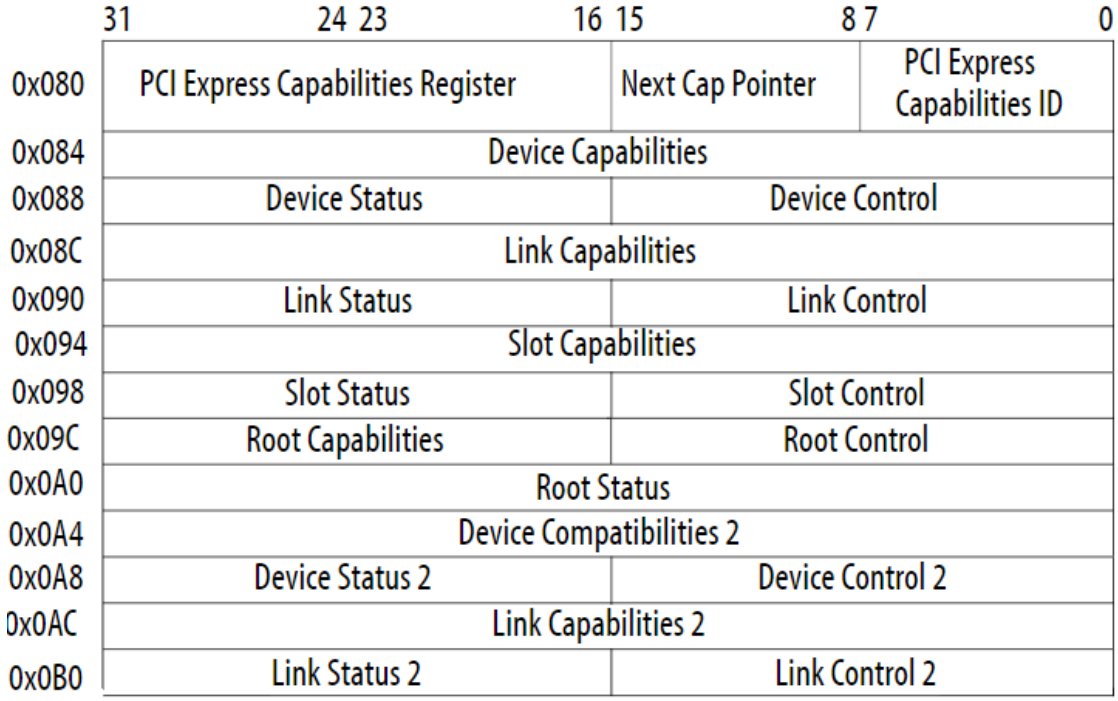


Figure 3.10 PCI Capability Register

As “Figure 3.10” Capability register which has a 32bit length. The first 8bits which are [7:0] which assign by PCI-SIG which are unique for each capability of devices. Each capability registers have its own significance like Link, Slot, Device capabilities. Device capabilities and link capabilities bits indicates the max speed which can be handled by devices.

4 ARCHITECTURE OF BIOS FIRMWARE

4.1. 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

4.2. 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 must 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 into multiple other smaller pieces to form many other logical firmware devices from it and vice-versa.

Flash - 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 based on sector-by-sector. After ensuring, every bits of sector return their erase value1.
- 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, FFS places restraints on the trading operations that can be executed while a write or erase is in progress.

4.3. 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 6 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.

4.4. 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 6**.** 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 4.1 enables to support non-standard file types.

Table 6: Firmware Volume Attributes

Attribute	**Description**
Name	each volume has a unique identifier name having UEFI Glob ally 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 require to be memory-mapped which de terminus whether the entire content of the volume can appear at once in the processor's memory address space.
Sticky Write?	Specifies whether special erase cycles require 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 cycles every bit 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 7: 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 4.1.

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 hands 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.

4.5. 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 about 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.

4.6. Firmware File Section

Firmware File Section is individual distinct unit of certain file types which has following attributes shown in Figure 4.1.

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.

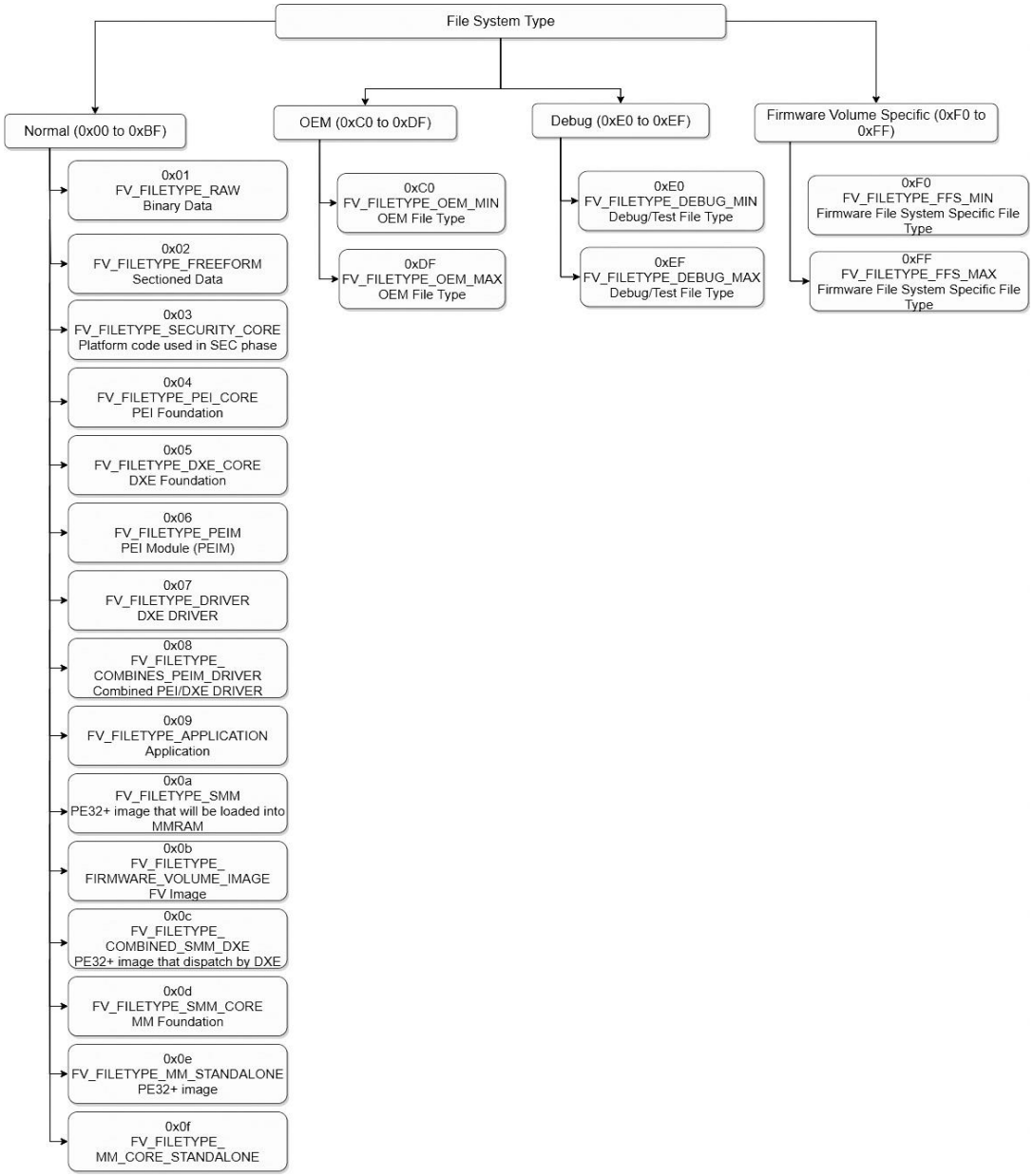


Figure 4.1: Firmware File Type

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

As illustrated in Figure 4.2, 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 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.

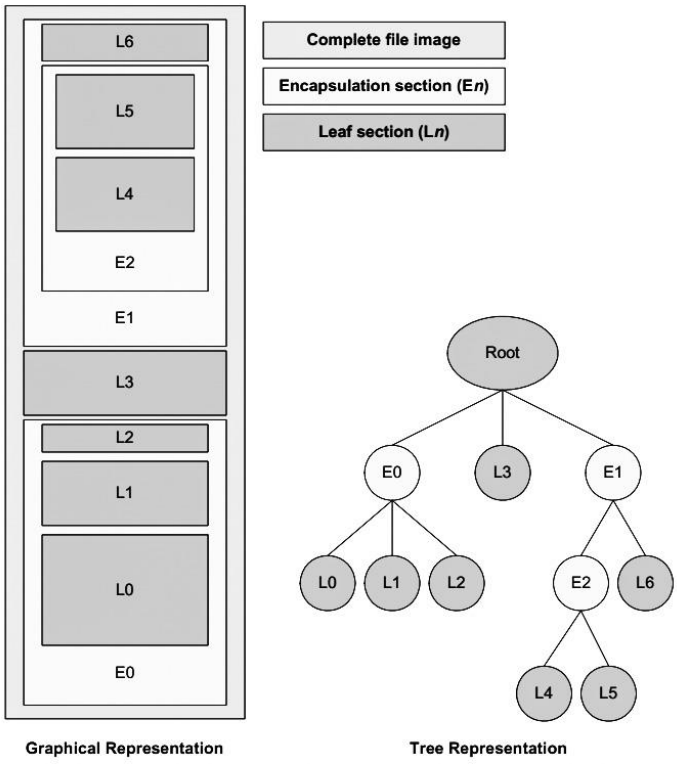


Figure 4.2: Example File System Image

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 [5].

The guided formatting of firmware volume Figure 4.3 made of two parts: The FV header and FV data. Header of FV describes every attribute mentioned in “Firmware Volumes" in Table 6. 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 File format" describes how files are organized. The firmware file format made of two parts: the firmware file header and the firmware file data.

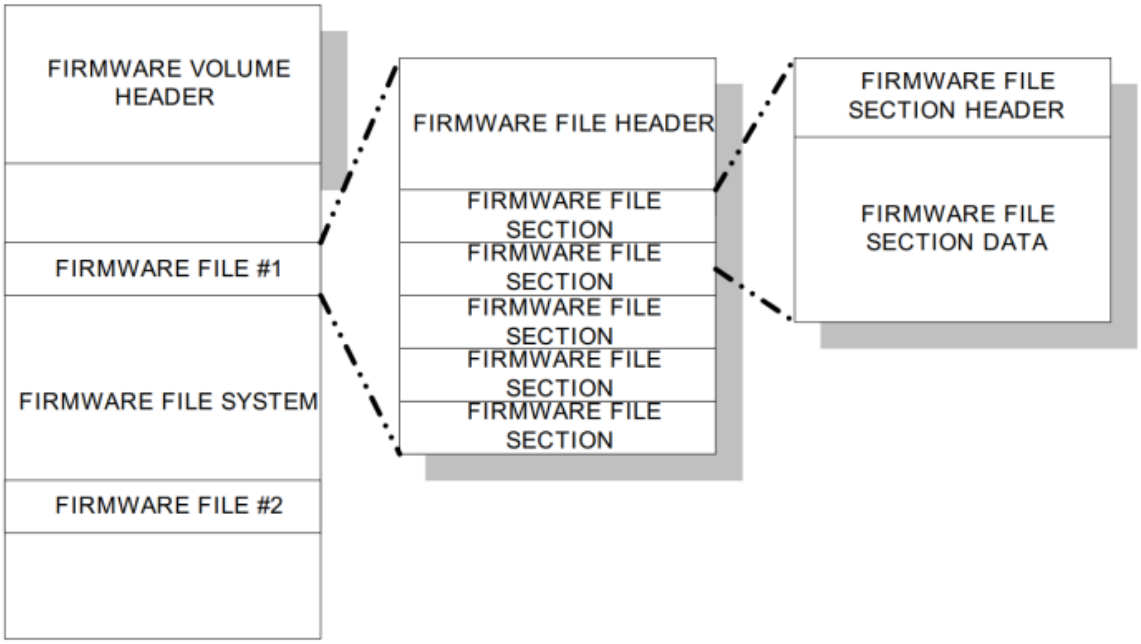


Figure 4.3 The Firmware Volume Format

4.7. 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 described by a GUID.

Valid files system GUID values are EFI_FIRMWARE_FILE_SYSTEM2_GUID and EFI_FIRMWARE_FILE_SYSTEM3_GUID.

4.8. 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 file 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 an 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 must 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 must have GUID for the file name which is declared as variable EFI_FFS_VOLUME_TOP_FILE_GUID. Driver code if Firmware file system must 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.

4.9. Firmware File Format (FFS)

Every FFS files begin 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 consist of the below parts:

- Header
- Data

When a zero-length file is created without any data it still must 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 4.4 exemplifies the typical layout of a (i.e. EFI_FFS_FILE_HEADER) \PI Architecture Firmware File" (≤ 16Mb).

Figure 4.5 exemplifies the typical layout of “PI Architecture Firmware File" (> 16Mb).

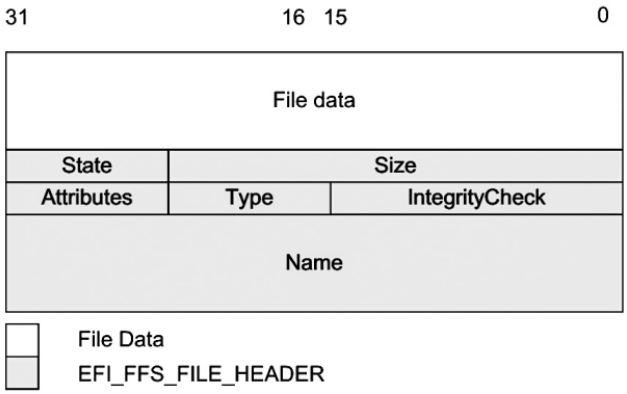


Figure 4.4: Layout representation of FFS File Header (≤ 16M b)

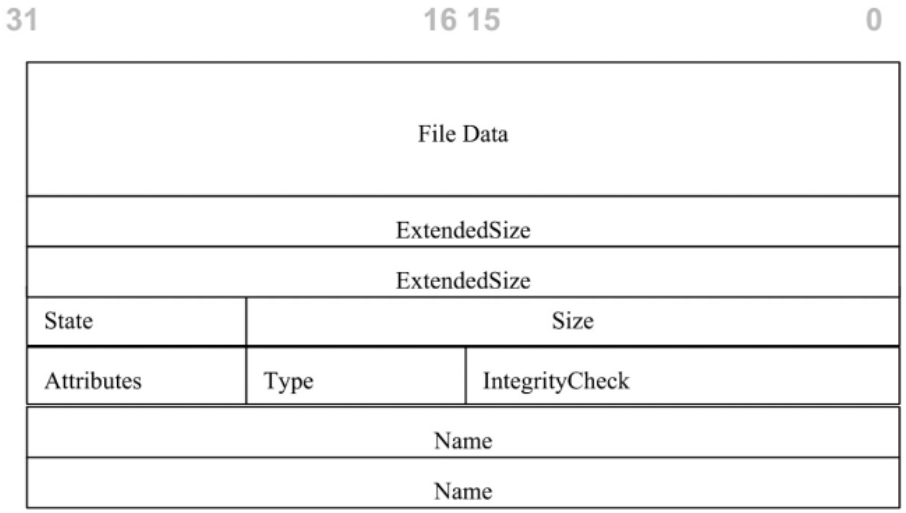


Figure 4.5: Layout representation of FFS File Header 2 layout for files (>16M b)

4.10. 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 start with a 24 − bit integer telling the section size, and 8 − bit section type. The format of the rest of the 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.

4.11. 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 operation on any FFS. It is desired that an FFS driver produces an instance of Firmware Volume Protocol so that every normal file operation carried out in that context. Every file operation must follow all the rules of creation, update, and deletion mentioned in this specification to avoid corruption of the file system. Figure 4.6 and Figure 4.7 shows the typical layout of a section data format.

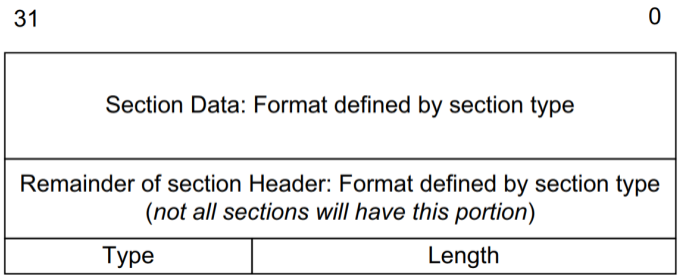


Figure 4.6: Section Header Format when size < 16M b

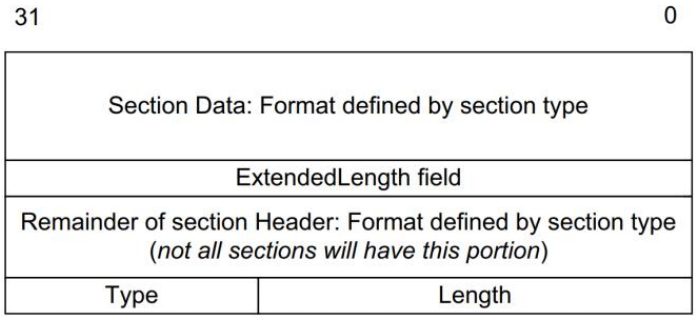


Figure 4.7: Section Header Format of when size ≥ 16M b using Extended Length field

4.12. 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 must 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 cannot 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. 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.

4.13. 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.

5 SOFTWARE REQUIREMENT SPECIFICATION

This section describes the necessary resources required for the project. It consists of four requirement types. They are functional, non-functional, software and hardware requirements. The functional requirements describe the core behavior and functionalities of the systems. It also explains the inputs and expected outputs. Non-functional requirements majorly focus on the quality of the project and enhancements. All the necessary software required by the system and their scope is described in the software requirements section. Also, all the hardware required is listed in the hardware requirements section.

5.1.1. General Constraints

Some of the general constraints in the implementation of graph partitioning are as follows:

- Mentioned hardware and software requirements should be met.
- The developer should have Visual Studio with C++ development environment setup.
- EDK-II package should be cloned to workspace. All the required environmental variables should be setup properly.
- Python 3.7 or above should be present in the development machine.

5.2. Functional Requirements

- Source code in modules, packages, and resulting binary components
- Execution in control, data, error, and debug flows
- Functionality as solutions evolve from initial to complete
- Scaling from silicon development to products

5.3. Software Requirements

Following are the software required for this project:

- Operating System: Windows 7,8,10 or Linux Operating System
- Development Platform: Visual Studio 2015
- Distribution Platform: TianoCore
- Editor: Visual Studio Code
- Simulator: Simics 6

5.4. Hardware Requirements

Following are the hardware required for this project:

- RAM size: 8 GB
- Disk: 500GB
- Octa-core processor

6 IMPLEMENTATION

This chapter discusses about the implementation requirements, development platforms, experimental setup and algorithms of this project. Further, the details about the implementation APIs required are discussed. Issues related to development and execution are also highlighted.

6.1. Implementation Requirements

This section briefly explains the implementation requirements of this project. It majorly highlights the details about the selection of the platform/environment. It also discusses the experimental setup needed for this project.

6.1.1. Selection of the Development Platform

For the implementation of this project TianoCore is used as a development platform. TianoCore community supporting an open source implementation of the UEFI. EDK II is a modern, feature-rich, cross-platform firmware development environment for the UEFI and UEFI Platform Initialization (PI) specifications.

In this project it is important to setup the EDK II environment properly and also development software's should be installed as per specifications provided.

In this project, it is very important to store and process the data in parallel. Also, this data requires further parallelism and fault tolerance while algorithms process them. So Apache Spark is selected for the development process [6].

Following steps shows how to install EDK II in local machine.

- Download EDK II Project
    - Open https://github.com/tianocore/edk2 in web browser
    - Click on the Clone or Download button (Right Green)
    - Click on Download ZIP
    - Unzip to C:/
    - Rename directory "edk2-master" to "edk2"
- Compile Tools for EDK II project developers on Windows with source BaseTools:
    - Create a workspace directory
    - Change to the workspace directory
    - Clone the EDK II project repository

        - Example: git clone https://github.com/tianocore/edk2
    - Install Python37 or late version (https://www.python.org/) to run python tool from source
    - Compile BaseTools C source tools
        - Example: Open Command prompt and *CD C:\edk2:*
        - *C:\edk2> set PYTHON_HOME=C:\Python37*
        - *C:\edk2> edksetup.bat Rebuild*
- Build
    - Set up the Nasm open source assembly compiler
    - Set up the ASL Compiler
    - Compile Tools above
    - Open a Windows CMD prompt:
    - Change to the edk2 directory
    - Run the edksetup.bat script
        - C:\Users\MySid> CD \edk2
        - C:\edk2> edksetup
        - C:\edk2> build

6.2. Implementation of BIOS

We have environment setup in our machine. During the development of BIOS for the compatibility we should take care of Firmware Volumes for each Silicon since the driver which are supported for one processor may not support for another processor. So, the section 4 we have discussed about the FV, FFS and its implantations. For selecting the FSS for the processor we are having a separate register in the silicon for identification. Depending upon the that register we are going to select the which FFS should be loaded into memory.

The objective of our model is to detect the capability of the system on chip and connected end-point devices to dispatch the UEFI driver during the Firmware boot flow. And maintain same firmware BIOS for the different system on chip.

UEFI drivers are mainly of two type which are discussed in the Platform Initialization Boot Phases. During the PEI phase it will detect the platform and Processor then loads the PEI drivers based on the capability of the processor which can handle the devices.

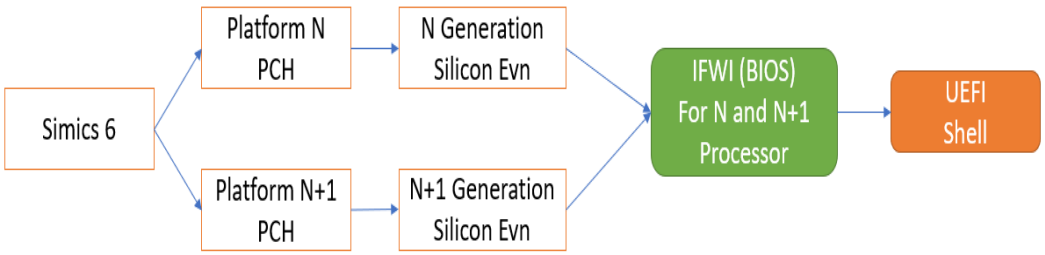


Figure 6.1 Proposed flow of OS in Simics

As in the "Figure 6.1" shows same firmware BIOS can be used for two different platforms as a result firmware size will be increased gradually, it can choose any of the paths so during the BIOS execution it will loads the drivers based on the configuration which are executing. The compatibility of the bios is shown "Figure 6.2" in which same bios supporting multiple system on chip on a specific platform. This can be implemented in the real time hardware.

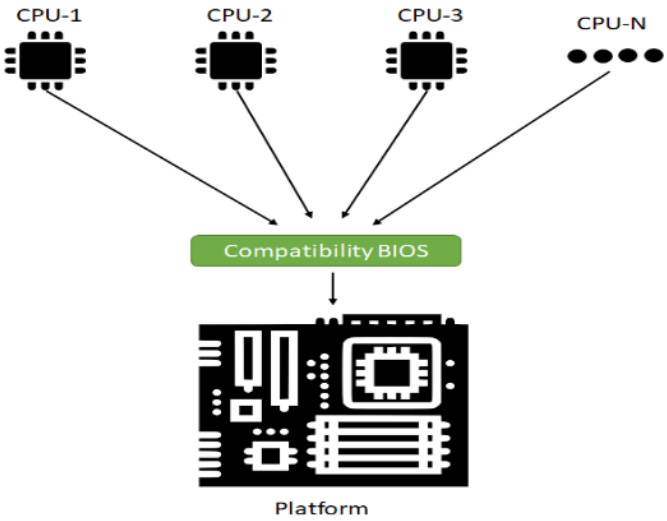


Figure 6.2 BIOS Support for Cross Compatibility

The Figure 6.3 shows the proposed model for dispatching the drivers during the firmware boot flow. Our model will read the System on chip capability during the Pre EFI-Initialization phase and loads the PEI drivers for handling the system on chip functionality. So that even when system on chip changes will it up and running. During the driver execution phase peripheral devices will be initialized. During the initialization of each device before dispatching the drivers will detect the capability of connected device and system on chip and loads the supported drivers for those environments. Since model must detect capability, it will increase the boot time based on the number of connected devices. This extra boot time can be calculated by Equation 1.

$$\text{Boot time} = T_{i+}(N * T_i)$$

Equation 1

Where T_i is the time taken to run the if statement and N is the number of connected devices during run time.

6.3. System Architecture

System Architecture identifies underlying modules of the system. It is very important to identify the modules at the early stages of development and recognize the communication between the already established modules. The functionalities of each module are defined clearly.

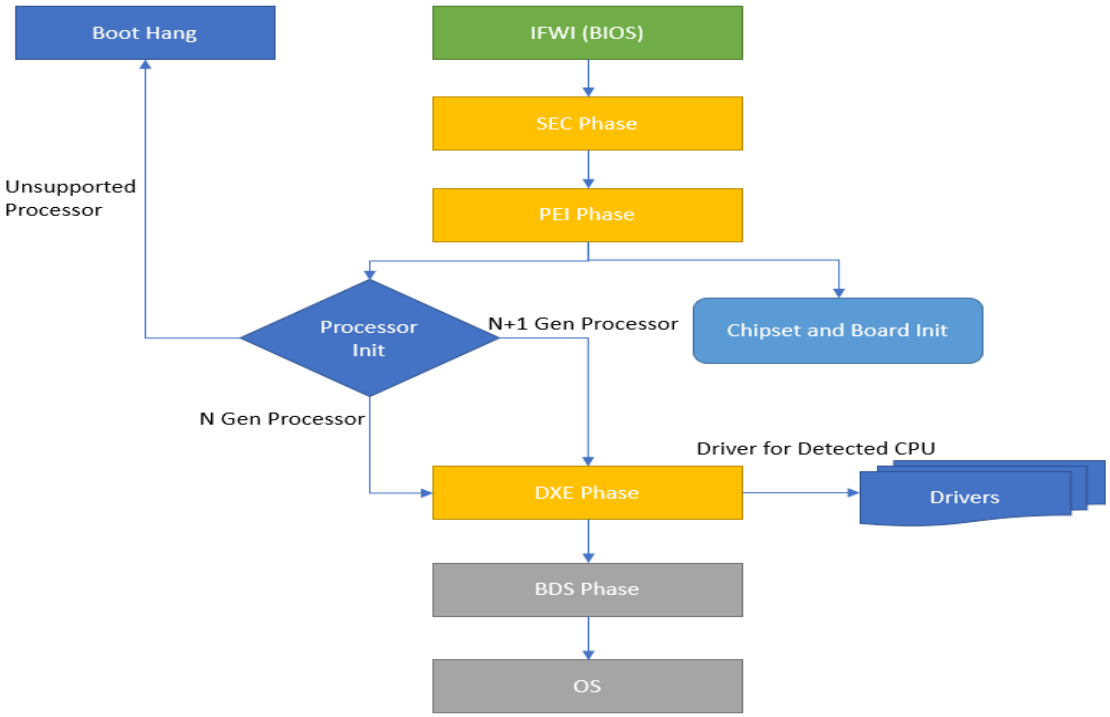


Figure 6.3 Proposed model for Driver Dispatcher

Algorithm 1 shows the Pseudo code for SOC/PCI capability detection

Alogithm 1 : Algorithm for SOC/PCI Capability Detection

Input	:	read capability register bits from devide
Output	:	loads EFI driver based on capability
Initialisation	:	BIOS boot flow
Step 1	:	X ← read SOC capability
Step 2	:	Y ← read PCI capability of connected device
Step 3	:	**if** (Y supported by X) **then**

Step 4	:	loads driver for Y from BIOS
Step 5	:	**else if (** check Y is capable of runnung lower speed) **then**
Step 6	:	**if** (check if Y drivers present in BIOS) **then**
Step 7	:	load appropriate drivers from BIOS
Step 8	:	**else:**
Step 9	:	disconnect logical link of devide
Step 10	:	**Endif**
Step 11	:	**else:**
Step 12	:	disconnect logical link of devide
Step 13	:	**endif**

6.4. Simulation results and analysis

To evaluate the working of model, there are 3 system on chip which are Intel I7 server CPU in simulation model, 2 version of PCI, Graphics devices and Intel I7 CPU supported platform. Model is configured in such a way that it will check the capability of system on chip and the connected PCI device. test conducted results using 3 virtual CPUs namely Silicon X, Silicon Y, and Silicon Z, 2 Virtual PCIe devices having the speed of Gen2, Gen3 and Graphics Gen X5 and X6. And the results are shown in “Table 8”.

Table 8: Simulation Results

Test No.	Test Environment	PCIe Version	Graphics Version	Result
1	Silicon X + Platform X	Gen 3	X6	Success
2	Silicon X + Platform X	Gen 4	X6	Success
3	Silicon Y + Platform X	Gen 2	X5	Success
4	Silicon Y + Platform X	Gen 3	X5	Fail
5	Silicon Z + Platform X	Gen2	N/A	Success
6	Silicon Z + Platform X	Gen 3	N/A	Success

In the Table 8 shows the test results which is conducted on specific platform with different Silicon on Chip and PCI devices in a virtual Simics platforms. Success results indicated it's booted to UEFI shell and capable of running the max speed of that connected device. Fail results indicated it is not capable of running those devices and it won's affected the boot flow, it will not detect the connected device. In test 1 can conclude that Silicon X supports both Gen3 and Gen 4 speed. Test 3 and 4 we can see Silicon Y can support max PCIe gen 2 speed only, so it is not detecting Gen 3 device in test 4. As the test 5 and 6 we can conclude that Gen 2 and Gen 3 is handled by Silicon X.

Experimental tests are conducted on the Simics 6 environment. This can be carried forward to real hardware platform in the future.

6.5. Implementation Results

Run the Wind River Simics 6, open the Simics file under the target PCH [8]. And provide the path for the BIOS and workaround for the platform, as shown the in below Figure 6.4. and run the simulation provided the option in the menu.

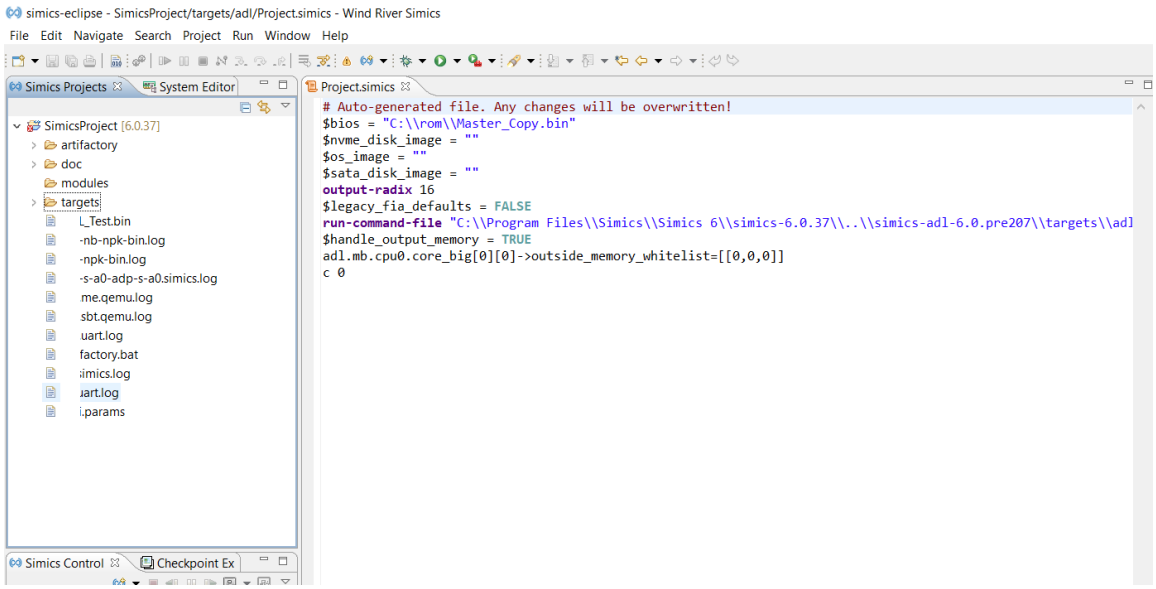


Figure 6.4 Simics Environment Setup

Once the Simulation Model start executing when the user sees "Start Boot Option" as shown in Figure 6.5. Press "F2" Button in the keyboard to enter "BIOS Setup Menu". If user unable to press "F2" button it will be booted to Simics EDK II Shell which is shown in below Figure 6.6. type Exit to inter into the BIOS Setup Menu.

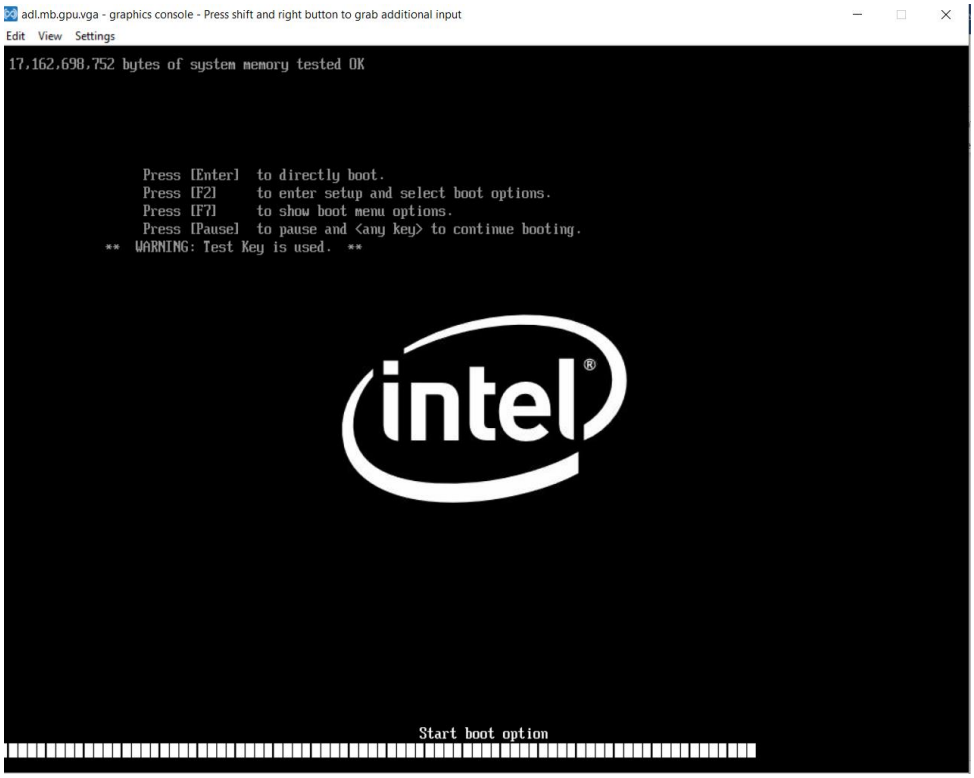


Figure 6.5 Start Boot Option

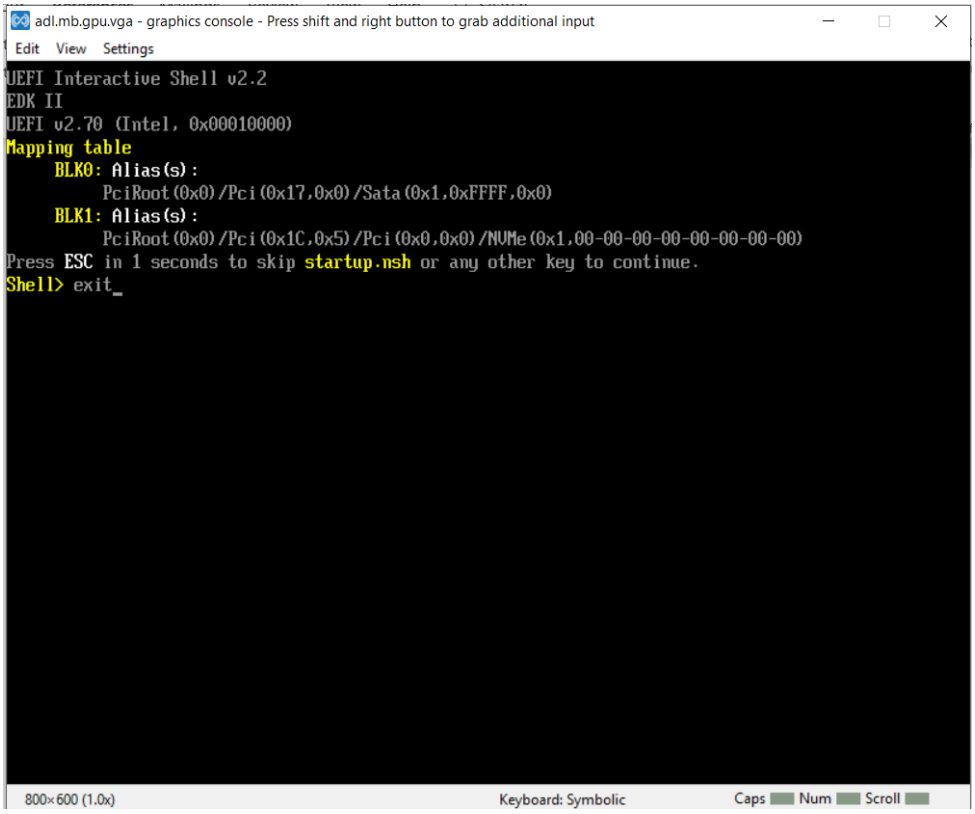


Figure 6.6 EDK II Shell

In the “BIOS Setup Menu” basic information of Processor Type and its memory will be displayed as shown in Figure 6.7. in the Setup Menu user can change the Setting based on

the user requirements, for our project it is impossible to change the Silicon physically in a Simics Environment, so we have created an option in which user can change the Silicon.

Figure 6.7 BIOS Setup Menu

The Platform Information Menu is shown in Figure 6.8 in which CPU name, Platform name and its supported PCIe, Graphics, and Graphics Output Protocol Version are displayed. So with the combination of “Platform X” with “Silicon X” BIOS supports PCIe Gen3 Speed and Graphics X6 Speed as shown in Figure 6.8.

Figure 6.8 Platform Information Menu

When user change the processor under “Intel Advanced Menu” Under Silicon Selection Option. Default Silicon is showed in the Figure 6.9. when the user selects the “Silicon Selection” option it will show supported Silicon which is supported for the platform as shown in Figure 6.10. when user select “Silicon Y” and click “F4” in keyboard to save the changes as shown in Figure 6.11. and click “Y”. click “ESC” button to go back to main menu. And select “Reset” as shown in Figure 6.12.

Figure 6.9 Intel Advanced Menu Default

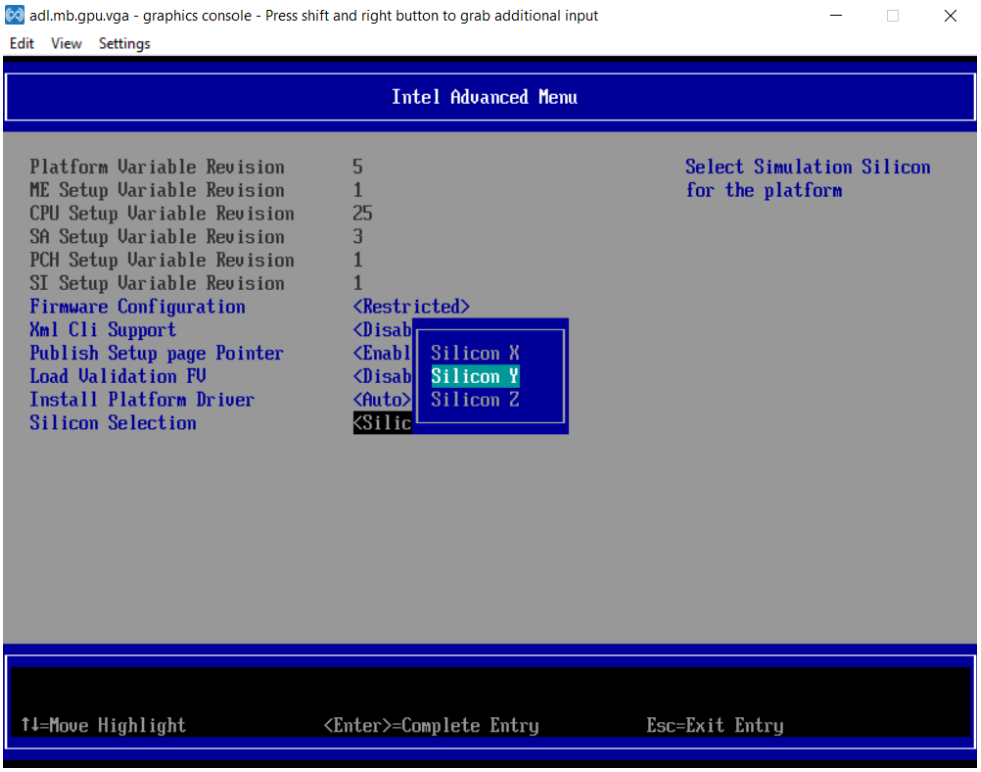


Figure 6.10 Intel Advanced Menu Silicon Option

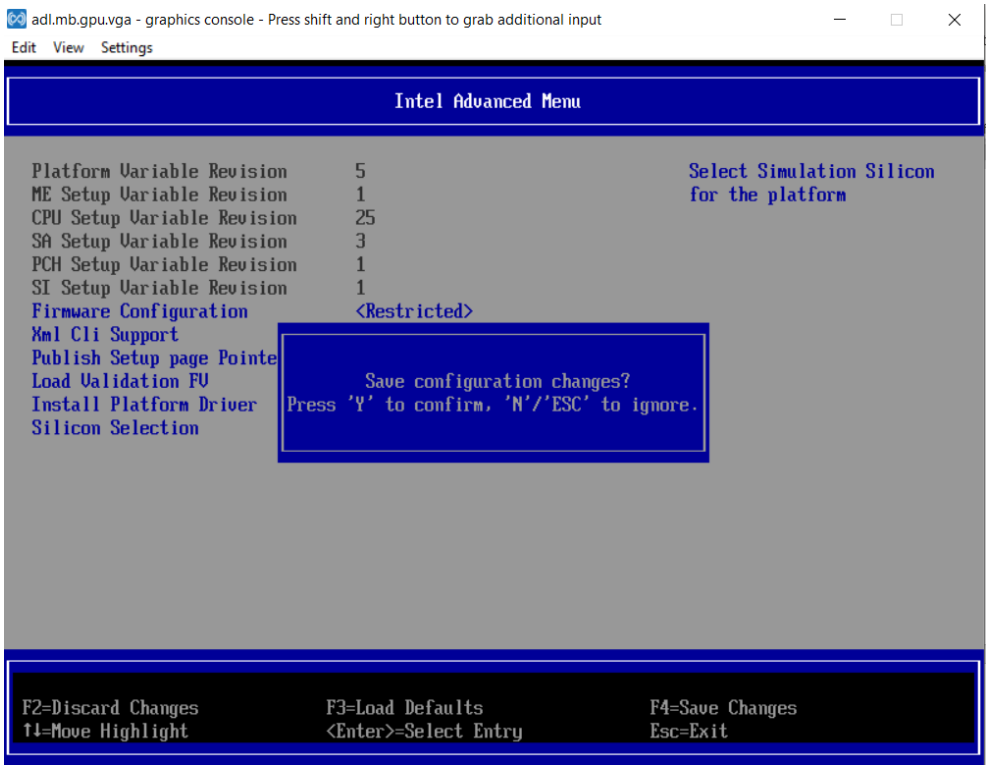


Figure 6.11 Intel Advanced Menu Silicon Save Option

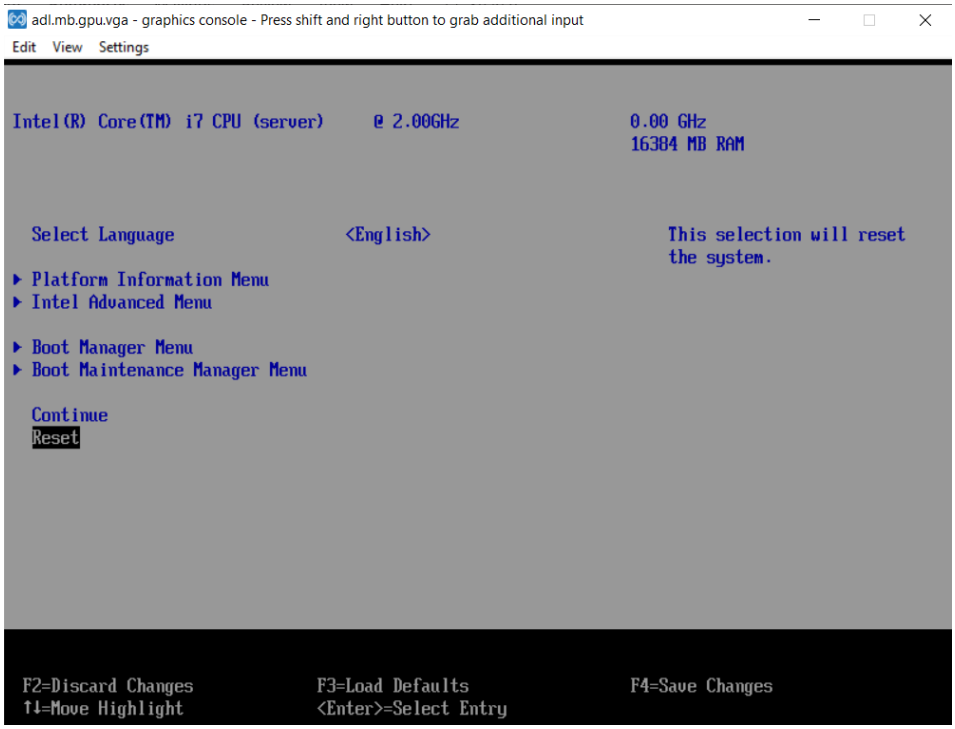


Figure 6.12 Intel Advanced Menu Silicon Reset Option

Upon clicking the “Reset” button it will restart the Simics Platform. Once it booted go to “Platform Information Menu” we can see the information as shown in below Figure 6.13. from the Figure 6.13 we can see that combination of “Silicon Y” and “Platform X” the Supported PCIe version is PCIe V2 and for Graphics its X5.

Figure 6.13 Platform Information Menu Silicon Y

The Figure 6.14 shows the combination of “Silicon Z” and “Platform X” which doesn’t supports Graphics and PCIe version is PCIe Gen3.

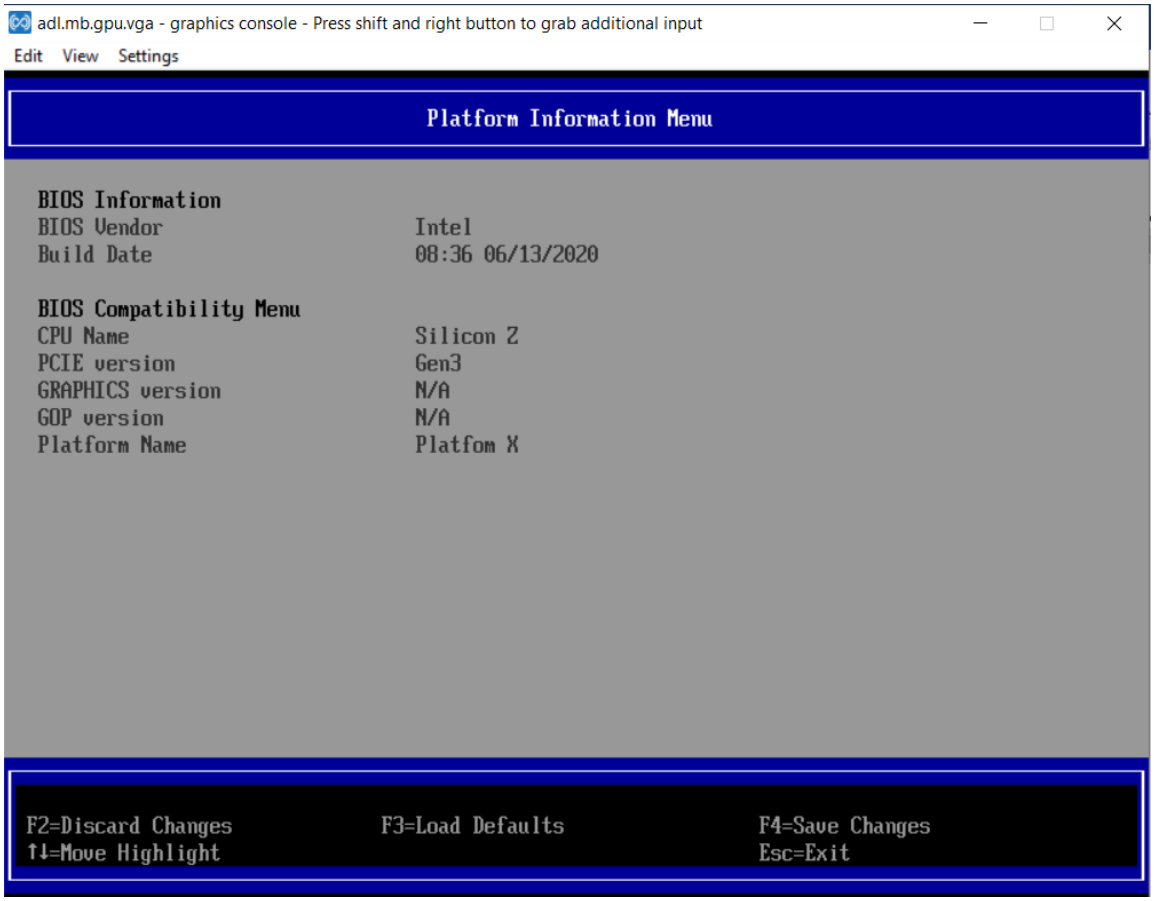


Figure 6.14 Platform Information Menu Silicon Z

7 CONCLUSION

Simics is a full-system simulator used to run unchanged production binaries of the target hardware at high-performance speeds. Simics contains both Instruction set simulators and hardware models, and can simulate systems such as x86-64, IA-64, and x86 CPUs. Many operating systems have been run on various varieties of the simulated hardware, including Windows and UEFI. The current version of Simics is 6 which was released publicly in 2019[8]. Simics runs on 64-bit Intel Architecture machines running Microsoft Windows and Linux.

In this proposed work within a Simics 6 environment if we change the N gen SOC in the “Intel Advanced Setup Menu” with N+1 Gen SOC with a same BIOS it will boot and support the drivers featured which are SOC capable. If SOC doesn’t support, it will hang. Since we cannot change the Processor Physically in Simulation. We have created a option to change to processor in the BIOS Setup Menu. This work has been carried out in a Simics 6 environment and in future this can be implemented with the real time hardware also.

8 REFERENCES

[1] Muhammad Irfan Afzal Butt, “BIOS integrity, an advanced persistent threat”, IEEE Information Assurance and Cyber Security (CIACS), 2014 Conference on,2014.

[2] Rahul Khanna, Fadi Zuhayri, Murugasamy Nachimuthu, “Unified extensible firmware interface: An innovative approach to DRAM power control”, IEEE Energy Aware Computing (ICEAC), 2014 International Conference on 2014.

[3] Rahul Khanna, Fadi Zuhayri, Christian Le, “Unified extensible firmware interface: An innovative infrastructure for power/thermal autonomics”, IEEE EnergyAware Computing (ICEAC), 2014 International Conference on 2014

[4] Vincent Zimmer, Michael Krau, “Establishing the root of trust”, Unified Extensible Firmware Interface Forum, August 2016

[5] Shirley Radack, “guidelines for protecting basic input/output system (bios) firmware” NIST Special Publication (SP) SP 800-61 June 2011

[6] Sven Amann, Sebastian Proksch, Sarah Nadi, “A Study of Visual Studio Usage in Practice”, Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on 2016,2016, DOI: 10.1109/SANER.2016.39

[7] https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ fsp-architecture-spec-v2.pdf

[8] Engblom, Jakob (September 10, 2019). "Simics 6 at the Mountain Top". Intel Developer Zone Blog.

Project_Report_Phase_-_II_-_Copy.pdf

ORIGINALITY REPORT

7%	6%	2%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	acpica.org Internet Source	2%
2	www.pcisig.com Internet Source	2%
3	www.microbe.cz Internet Source	2%
4	Submitted to Visvesvaraya Technological University Student Paper	1%
5	syzygyx.com Internet Source	<1%
6	www.tianocore.org Internet Source	<1%
7	Submitted to B.V. B College of Engineering and Technology, Hubli Student Paper	<1%
8	www.uefi.org Internet Source	<1%

ICOSEC 2020

International Conference on
Smart Electronics and Communication
(ICOSEC 2020)

10-12, September 2020 | http://icsec.in/ | icsec.conf@gmail.com

Letter of Acceptance

To
Ashraf Ali S, Mohana Kumar S,
Department of Computer Science and Engineering,
Ramaiah Institute of Technology,
Bangalore, India.

Greetings!!

On behalf of the Kongunadu College of Engineering and Technology, we take great pleasure in inviting you to attend the "International Conference on Smart Electronics and Communication (ICOSEC 2020)" which is going to be held during 10-12, September 2020 at Trichy, India.

We are glad to inform you that your submitted article entitled "***BIOS Compatibility for Multiple System On Chip" [Paper id: ICOSEC138]*** has been accepted after the double-blinded peer-review process, for publication at ICOSEC 2020. We welcome you to join us and share your research and views. This conference will offer you an unforgettable experience in exploring new opportunities.

We look forward to seeing you at Trichy, India.

For more details about ICOSEC 2020, visit our website: http://icsec.in/index.html

With Thanks,

Yours' Sincerely

Dr. J. Yogapriya,
Dean (R&D),
Kongunadu College of Engineering and
Technology, Trichy, Tamil Nadu, India.

Proceedings by

IEEE