# Generic IP independent BIOS Signing and Parsing

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Technology

in

Computer Science & Engineering

with specialization in Computer Science & Engineering

by

Gahan Saraiya

18MCEC10

NIRMA UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED 'A' GRADE

Department of Computer Science & Engineering,

Institute of Technology,

Nirma University, Ahmedabad,

Gujarat - 382481, India.

December, 2019

NIRMA
UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED ‘A’ GRADE

# Declaration

I hereby declare that the dissertation ***Generic IP independent BIOS Signing and Parsing*** submitted by me to the Institute of Technology, Nirma University, Ahmedabad, 382481 in partial fulfillment of the requirements for the award of **Master of Technology** in **Computer Science & Engineeringwith specialization in Computer Science & Engineering** is a bona-fide record of the work carried out by me under the supervision of ***Prof. Dvijesh Bhatt***.
I further declare that the work reported in this dissertation, has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or University.

Sign:

Name & Roll. No.:

Date:

NIRMA
UNIVERSITY
INSTITUTE OF TECHNOLOGY
NAAC ACCREDITED 'A' GRADE

## Computer Science & Engineering

# Certificate

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

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

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

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

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

# *Abstract*

Here goes the BOOM... aka abstract..

# *Acknowledgements*

is this page needed ???????

# Contents

**Declaration** i

**Certificate** ii

**Abstract** iii

**Acknowledgements** iv

**List of Figures** vi

1. Introduction . . . . . . . . . . 1
   - 1.1 Legacy BIOS and UEFI . . . . . . . . . . 1
     - BIOS . . . . . . . . . . 1
     - 1.1.1 Background of Legacy BIOS . . . . . . . . . . 1
   - 1.2 Unified Extensible Firmware Interface (UEFI) . . . . . . . . . . 1
     - 1.2.1 UEFI's Role in boot process . . . . . . . . . . 2
   - 1.3 Comparing of Legacy BIOS and UEFI . . . . . . . . . . 2
   - 1.4 ACPI . . . . . . . . . . 2

**List of Figures** 1

2. Design . . . . . . . . . . 3
   - 2.1 UEFI/PI Firmware Images . . . . . . . . . . 3
   - 2.2 Platform Initialization PI Boot Sequence . . . . . . . . . . 5
   - 2.3 Security (SEC) . . . . . . . . . . 5
   - 2.4 Pre-EFI Initialization (PEI) . . . . . . . . . . 6
     - 2.4.1 PEI Services . . . . . . . . . . 7
     - 2.4.2 PEI Foundation . . . . . . . . . . 7
   - 2.5 PEI Dispatcher . . . . . . . . . . 8
   - 2.6 Drive Execution Environment (DXE) . . . . . . . . . . 9
   - 2.7 Boot Device Selection (BDS) . . . . . . . . . . 9
   - 2.8 Transient System Load (TSL) and Runtime (RT) . . . . . . . . . . 10
   - 2.9 After Life (AL) . . . . . . . . . . 10
   - 2.10 Generic Build Process . . . . . . . . . . 10
     - 2.10.1 EFI Section Files . . . . . . . . . . 10

# List of Figures

1 Board of Directors of UEFI Forum . . . . . . . . . . . . . . . . . . . . . 2
2 UEFI/PI Firmware Image Creation . . . . . . . . . . . . . . . . . . . . . 3
3 UEFI/PI Firmware Image Creation . . . . . . . . . . . . . . . . . . . . . 4
4 PI Boot Phases . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
5 Diagram of PI Operations . . . . . . . . . . . . . . . . . . . . . . . . . 7
6 General EFI Section Format for large size Sections(greater then 16 MB) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
7 General EFI Section Format (less then 16 MB) . . . . . . . . . . . . . 11

# 1. Introduction

## 1.1 Legacy BIOS and UEFI

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

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

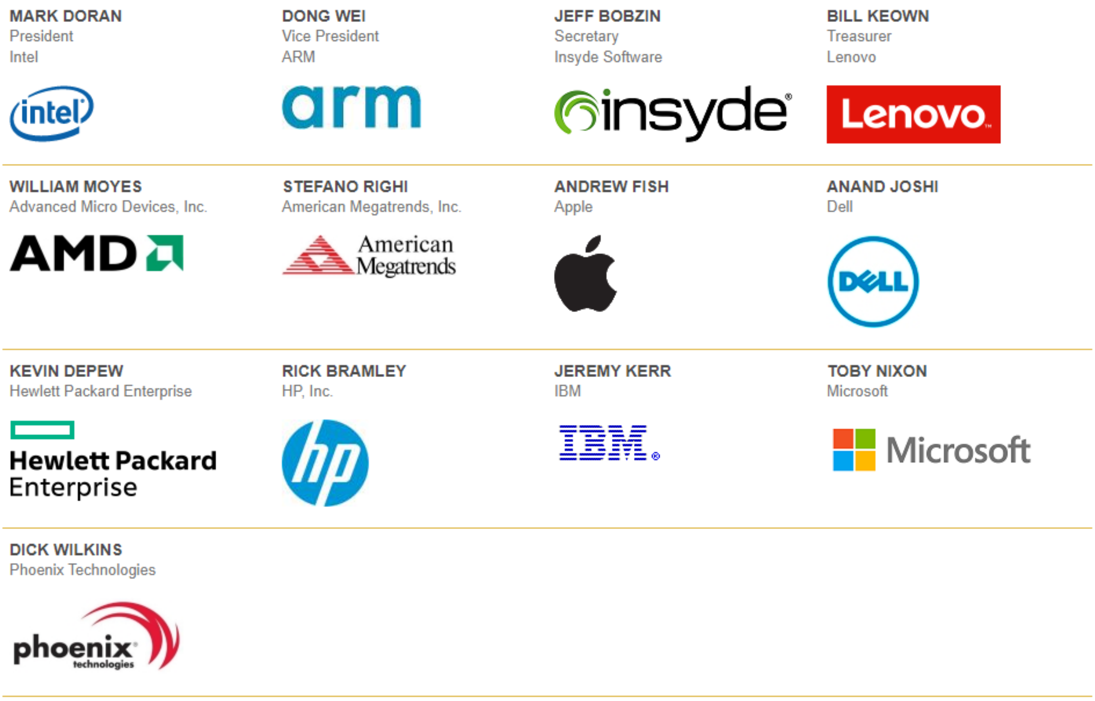
### 1.1.1 Background of Legacy BIOS

In 1980s, IBM developed the personal computer with a 16-bit BIOS with the aim of ending the BIOS after the first 250,000 products. Legacy BIOS is based upon Intel's original 16-bit architecture, ordinarily referred to as "8086" architecture. And as technology advanced, Intel extended that 8086 architecture from 16 to 32-bit. Legacy BIOS is able to run different Operating System (OS), such as MS-DOS, equally well on systems other than IBM. Additionally, Legacy BIOS has a defined OS-independent interface for hardware that enables interrupts to communicate with video, disk and keyboard services along with the BIOS ROM loader and bootstrap loader, to name a few.

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

## 1.2 Unified Extensible Firmware Interface (UEFI)

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



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

FIGURE 1: Board of Directors of UEFI Forum

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

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

### 1.2.1 UEFI's Role in boot process

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

## 1.3 Comparing of Legacy BIOS and UEFI

## 1.4 ACPI

# 2. Design

## 2.1 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 BaseTools package process files compiled by third party tools, as well as text and Unicode files in order to create UEFI or PI compliant binary image files. In some instances, where UEFI or PI specifications do not have an applicable input file format, such as the Visual Forms Representation (VFR) files used to create PI compliant IFR content, tools and documentation have been provided that allows the user to write text files that are processed into formats specified by UEFI or PI specifications.

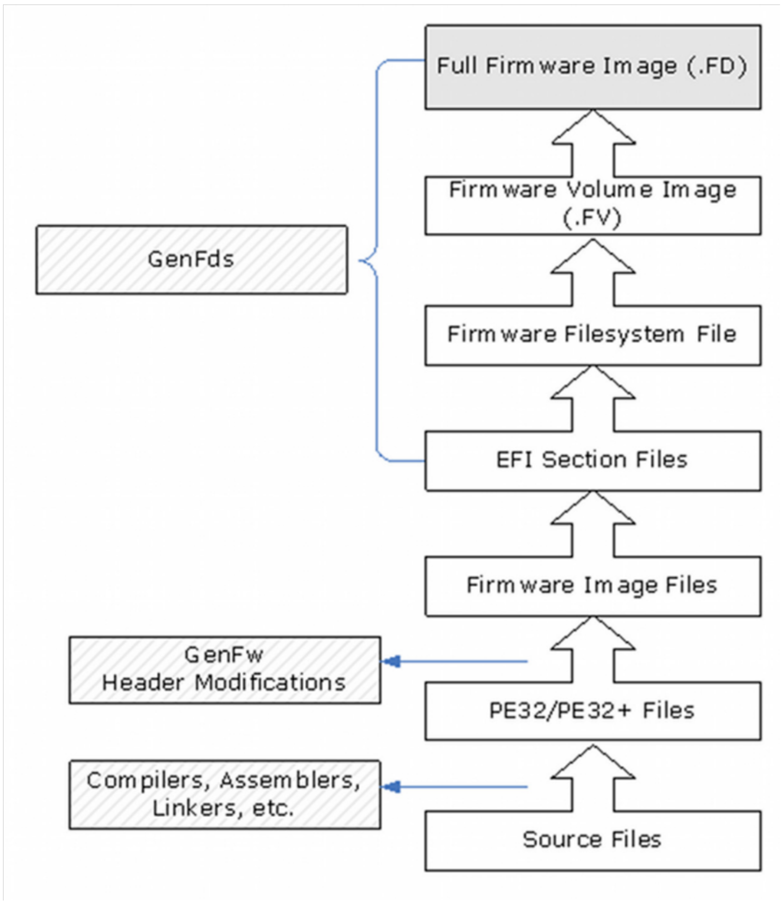

Full Firmware Image (.FD)
Firmware Volume Image (.FV)
GenFds
Firmware Filesystem File
EFI Section Files
Firmware Image Files
GenFw Header Modifications
PE32/PE32+ Files
Compilers, Assemblers, Linkers, etc.
Source Files

FIGURE 2: 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 2. Each transition between layers implies a processing step that transforms or combines previously processed files into the next higher level. Also shown in Figure 2 are the reference implementation tools that process the files to move them between the different layers.

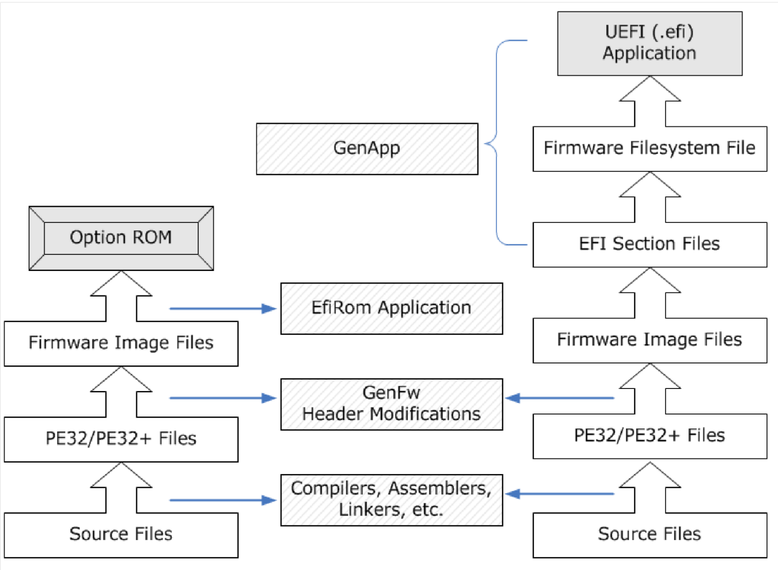


FIGURE 3: UEFI/PI Firmware Image Creation

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, below, shows the reference implementation tools and creation processes for both of these image types

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

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

## 2.2 Platform Initialization PI Boot Sequence

PI compliant system firmware must support the six phases: security (SEC), pre-efi initialization (PEI), driver execution environment (DXE), boot device selection (BDS), run time (RT) services and After Life (transition from the OS back to the firmware) of system. Refer to Figure 4 below.

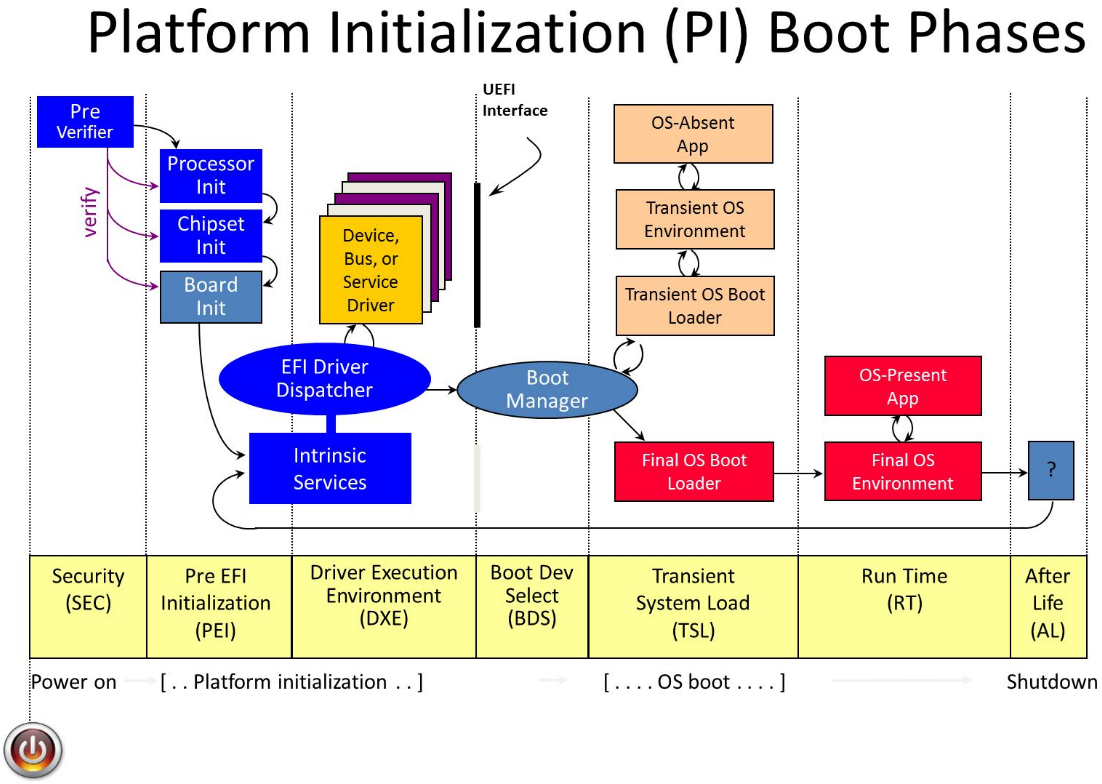


FIGURE 4: PI Boot Phases

## 2.3 Security (SEC)

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

## 2.4 Pre-EFI Initialization (PEI)

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

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

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

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

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

These PEIMs are responsible for the following:

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

Figure 5 shows a diagram describes the action carried out during the PEI phase

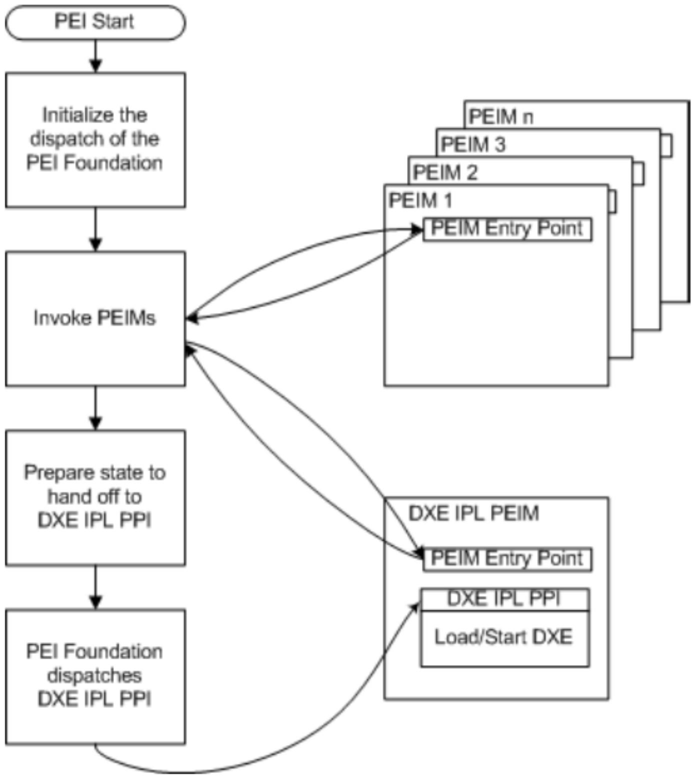

PEI Start
Initialize the dispatch of the PEI Foundation
Invoke PEIMs
Prepare state to hand off to DXE IPL PPI
PEI Foundation dispatches DXE IPL PPI
PEIM n
PEIM 3
PEIM 2
PEIM 1
PEIM Entry Point
DXE IPL PEIM
PEIM Entry Point
DXE IPL PPI
Load/Start DXE

FIGURE 5: Diagram of PI Operations

### 2.4.1 PEI Services

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

The PEI Foundation provides the classes of services listed in Table 1

### 2.4.2 PEI Foundation

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

- Dispatching of Pre-EFI initialization modules (PEIMs)
- Maintaining the boot mode

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

TABLE 1: Services provided by PEI Foundation Classes

- Initialization of permanent memory
- Invoking the DXE loader

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

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

## 2.5 PEI Dispatcher

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

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

The PEI Dispatcher will exit Once the PEI Dispatcher has examined and evaluated all of the PEIMs in all of the uncovered firmware volumes and no more PEIMs can be dispatched (i.e. the dependency expressions do not evaluate from $FALSE$ to $TRUE$). At this stage, the PEI Dispatcher cannot invoke any additional PEIMs. The PEI Foundation then takes back control from the PEI Dispatcher and invokes the $DXEIPLPPI$ to pass control to the DXE phase of execution.

## 2.6 Drive Execution Environment (DXE)

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

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

## 2.7 Boot Device Selection (BDS)

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

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

## 2.8 Transient System Load (TSL) and Runtime (RT)

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

## 2.9 After Life (AL)

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

## 2.10 Generic Build Process

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

### 2.10.1 EFI Section Files

he general section format for sections less than 16MB in size is shown in Figure 7. Figure 6 shows the section format for sections 16MB or larger in size using the extended length field.

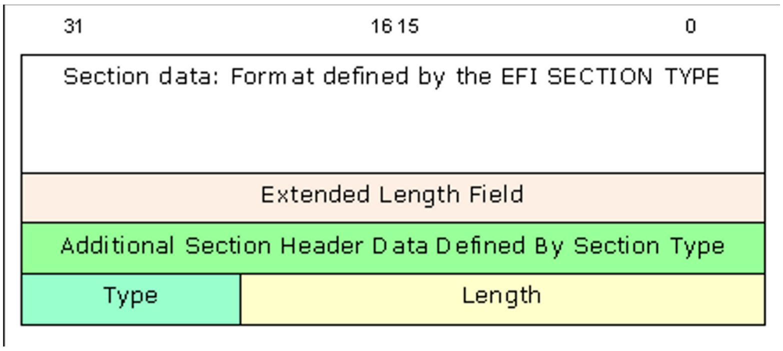

31
16 15
0
Section data: Format defined by the EFI SECTION TYPE
Extended Length Field
Additional Section Header Data Defined By Section Type
Type
Length

FIGURE 6: General EFI Section Format for large size Sections(greater then 16 MB)

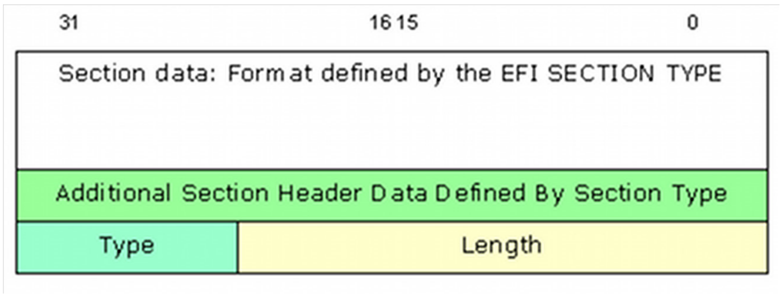

31
16 15
0
Section data: Format defined by the EFI SECTION TYPE
Additional Section Header Data Defined By Section Type
Type
Length

FIGURE 7: General EFI Section Format (less then 16 MB)