Development of Firmware Support Package BIOS for Next Generation Processors

Major Project Report

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology

in

Electronics & Communication Engineering

(Communication Engineering)

By

Niravkumar T. Rajpara

(14MECC17)

Electronics & Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad - 382 481

May 2016

Development of Firmware Support Package BIOS for Next Generation Processors

Major Project Report

Submitted in partial fulfillment of the requirements

for the degree of

Master of Technology

in

Electronics & Communication Engineering

(Communication Engineering)

By

Niravkumar T. Rajpara

(14MECC17)

Under the Guidance of

Mr. Satya P. Yarlagadda

Sr. BIOS Engineer,

Intel Technology Pvt. Ltd.,

Bangalore

Dr. D. A. Pujara

Elec. & Comm. Department,

Nirma University,

Ahmedabad

Electronics & Communication Engineering Branch

Electrical Engineering Department

Institute of Technology

Nirma University

Ahmedabad - 382 481

May 2016

Declaration

This is to certify that,

1. The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at Institute of Technology,Nirma University and has not been submitted elsewhere for a degree.

2. Due acknowledgement has been made in the text to all other material used.

Rajpara Niravkumar T.

Certificate

This is to certify that the major project entitled **"Development of Firmware Suppport Package BIOS for Next Generation Processors"** submitted by **Rajpara Niravkumar T. (Roll No: 14MECC17)**, towards the partial fulfillment of the requirements for the award of degree of Master of Technology in Electronics and Communication Engineering of Nirma University, Ahmedabad, is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahemdabad

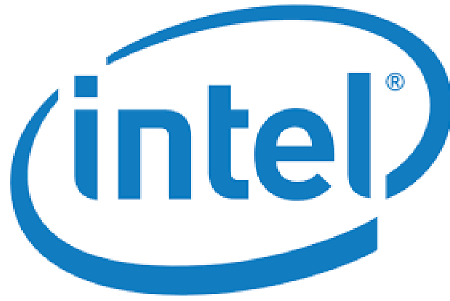
Guide **Program Coordinator**

Dr. D. A. Pujara
(Professor, EC)

Dr. D. K. Kothari
(Professor, EC)

HOD & Director

Dr. P. N. Tekwani
(Head of EE Department, Director, IT-NU)



Certificate

This is to certify that the major project entitled **“Development of Firmware Suppport Package BIOS for Next Generation Processors”** submitted by **Rajpara Niravkumar T. (Roll No: 14MECC17)**, towards the partial fulfillment of the requirements for the award of degree of Master of Technology in Electronics and Communication Engineering of Nirma University, Ahmedabad, is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven’t been submitted to any other university or institution for award of any degree or diploma.

Date:

Place: Bangalore

Guide

Mr. Satya P. Yarlagadda
(Guide, Intel Technology Pvt. Ltd.)

Manager

Mr. Bimod Narayanan
(Manager, Intel Technology Pvt. Ltd

Acknowledgements

I would like to express my heartedly thanks to following peoples who were able to give their willingness to help, for their unwavering and undying support, encouragement and precious time for accomplishment of this report.

First of all I would like to thank **Prof. Dr. D. A. Pujara (Internal Guide)**, M.Tech. Communication and **Mr. Satya P. Yarlagadda (External Guide), Sr. BIOS Engineer, Intel Technology Pvt. Ltd.** for support, advice, guidance, valuable comments, suggestions that benefited in the success of this project. I would also like to thank **Prof. Dr. P. N. Tekwani, Head of Electrical Engineering Department** and **Prof. D. K. Kothari (Program Coordinator)**, M.Tech. Communication for allowing me to this thesis work and for giving me encouragement. I would like to thank **Prof. Dr. P. N. Tekwani, Director, Institute of Technology, Nirma University** for his encouragement during project work.

I am extremely thankful to **Mr. Bimod Narayanan (Manager), Component BIOS Engineering, Intel Technology Pvt. Ltd.** who allowed me to enrich my skills in Intel Technology Pvt. Ltd.

Last but not the least, I would also like to give credit to my all BIOS development team members, my family and my friends who helped me a lot by giving guidance and support throughout this project training.

- Niravkumar T. Rajpara
14MECC17

Abstract

This thesis mainly presents a discussion about the Firmware Support Package BIOS. In broader view, Firmware Support Package Basic Input Output System is the binary distribution of necessary Intel Silicon initialization code which provide access to programming information which is not publicly available. The research work described in the thesis is mainly focused on improving the performance of BIOS and flexibility to the users to use any open source boot loaders.

Basically the BIOS in modern PCs is to initialize and test the system hardware components, and to load a boot loader or an operating system from a mass memory device which is used for silicon initialization. It act as a layer between OS and hardware to initialize and interface and finally loads boot loader to give control to OS and BIOS played supporting hardware role for the devices. The FSP provides chipset and processor initialization in a format that can easily be incorporated into many existing boot loaders. The first design goal of FSP is to provide access to the key programming information that is open source. The second design goal is to abstract the complexities of Intel Silicon initialization and expose a limited number of well-defined interfaces. A fundamental design philosophy is to provide the ubiquitously required silicon initialization code. As such, FSP will often provide only a subset of the products features.

The next generation processors for FSP used to give better performance to the customers are Kabylake, Broxton and Apollolake which will release at the end of 2016. Mainly Kabylake and Apollolake are for PCs and Broxton is for tablet. So, by using FSP for next generation processors, Intel provides best solution to that customers who want to use open source bootloaders for particular BIOS.

KEYWORDS: BIOS, Boot loader, FSP, Initialization, Next generation processors

Contents

Declaration **iii**

Certificate **iv**

Certificate **v**

Acknowledgements **vi**

Abstract **vii**

Contents **ix**

List of Tables **x**

List of Figures **xii**

Abbreviations **1**

1 Introduction **2**
1.1 Background 2
1.2 Motivation 4
1.3 Problem Statement 5
1.4 Thesis Organization 6

2 Literature Survey **8**
2.1 Intel Platform Architecture 8
2.2 Platform Software Architecture 10
2.3 BIOS Overview 11
2.4 Legacy and EFI BIOS 13
2.5 UEFI Specifications 15
2.6 UEFI BIOS Boot Phases 16
2.7 EDK and EDK II Platform 24

3 Firmware Support Package **26**
3.1 Overview 26
3.2 FSP Usage Model 27
3.3 FSP Code Delivery Model 28
3.4 FSP Integration 29
3.5 FSP Boot Flow 30
3.6 FSP Versions 31

4 FSP Implementation Strategy **39**
4.1 POST and Phase Code 39
4.1.1 Why POST Code Required 39
4.1.2 How POST Status Code Works 40
4.1.3 Types of POST Code 41
4.1.4 Implementation 42
4.1.5 Phase Code 43
4.1.6 Implementation 43
4.2 FSP Separation from Source Code 44
4.2.1 Flow Chart of Script 46
4.3 FSP 2.0 49
4.4 Firmware Configuration for FSP 52
4.5 Responsiveness Infrastructure of FSP 55
4.5.1 What is Responsiveness? 55
4.5.2 Why Responsiveness? 55
4.5.3 Implementation 56
4.6 eMMC Boot Flow for FSP 58
4.6.1 Overview 58
4.6.2 Implementation 58

5 Automated BIOS generation using System RDL **60**
5.1 Overview 60
5.2 Firmware Developement Model 62
5.3 System RDL Usage Model 63
5.4 Implementation 66

6 Conclusions and Future Scope **69**
6.1 Conclusions 69
6.2 Future Scope 70

Bibliography **72**

List of Tables

2.1 Legacy BIOS and EFI BIOS [4] 14
2.2 EFI Boot Phases and Services [4] 17
2.3 PEI Services and its Functions [4] 20

4.1 POST Code Classes [6] 42
4.2 EDK II Reference Code Packages [4] 49

List of Figures

2.1 Intel Platform Architecture [4] . . . 9
2.2 High Level Diagram of BIOS Space [4] . . . 12
2.3 UEFI Interface between BIOS and OS Loader [4] . . . 14
2.4 UEFI Interface [4] . . . 16
2.5 Phases of BIOS Execution [4] . . . 17
2.6 PEI Phase Flow [4] . . . 20
2.7 PEI to DXE Foundation [4] . . . 22
2.8 DXE Phase Flow [4] . . . 23
2.9 BDS Phase Flow [4] . . . 24

3.1 FSP Usage Model [5] . . . 27
3.2 Code Delivery Model for FSP A) 2012 B) 2013 C) 2014 D) 2015 [5] . . . 28
3.3 FSP Boot Flow [5] . . . 30
3.4 FSP Boot flow Versions (a) 1.0 (b) 1.1[5] . . . 31
3.5 TempRamInit API prototype [5] . . . 33
3.6 FspInit API Prototype [5] . . . 33
3.7 NotifyPhase API Prototype [5] . . . 34
3.8 FspMemoryInit API Prototype [5] . . . 35
3.9 TempRamExit API Prototype [5] . . . 35
3.10 FspSiliconInit API Prototype [5] . . . 36
3.11 FSP Boot Flow Version 2.0 [6] . . . 37

4.1 BIOS POST Code in Board [6] . . . 41
4.2 Proposed Method for Phase Code . . . 43
4.3 Finalized Implementation of Phase Code . . . 44
4.4 (a) Regular BIOS Execution (b) FSP Execution . . . 45
4.5 Flow Chart of Script . . . 46
4.6 Silicon Folder before Python Script . . . 47
4.7 Silicon Folder after Python Script . . . 48
4.8 Firmware Volume for (a)FSP 1.1 (b) FSP 2.0 [5] . . . 50
4.9 FSP 2.0 Implementation [5] . . . 51
4.10 High Level Diagram of BCT [6] . . . 52
4.11 GUI of BCT [6] . . . 53
4.12 API called to UPD [6] . . . 54
4.13 Definition of UPD . . . 54
4.14 FV's of FSP for responsiveness [7] . . . 56
4.15 Intel's System Scope Tool . . . 57
4.16 Result of Responsiveness . . . 57

5.1 High Level Diagram of Proposed Solution[11] . . . 61

5.2 Firmware Development Model [11] 62
5.3 System RDL Usage Model [8] 63
5.4 Back End Overviewn [8] 65
5.5 Snippet of XML file 66
5.6 Snippet of Source file 67
5.7 Integration of BIOS code 68

6.1 Re usability Structure [12] 71

Abbreviations

ACPI	Advanced Configuration and Power Interface
API	Application Programming Interface
BCT	Binary Configuration Tool
BDS	Boot Device Selection
BIOS	Basic Input Output System
CRB	Customer Reference Board
CSM	Compatibility Support Module
DXE	Driver Execution Environment
ECP	EFI Compatibility Package
EFI	Extensible Firmware Interface
FPDT	Firmware Performance Data Table
FSP	Firmware Support Package
FV	Firmware Volume
GUID	Globally Unique Identifier
HOB	Hand Off Block
IPL	Initialization Program Load
ME	Management Engine
OEM	Original Equipment Vendor
OS	Operating System
OSPM	Operating System-directed configuration and Power Management
PC	Personal Computer
PCD	Platform Configuration Database
PCH	Peripheral Cotroller Hub
PEIM	PEI Module
PEI	Pre EFI Initialization
PI	Platform Initialization
POST	Power On Self-Test
PPI	PEIM to PEIM Interface
ROM	Read Only Memory
SMBIOS	System Management Basic Input Output System
UEFI	Unified Extensible Firmware Interface
UPD	Updatable Product Data
VPD	Vital Product Data

Chapter 1

Introduction

1.1 Background

Initially, computers had no boot firmware. Instead, the user had to enter a boot program by hand using switches on the front panel of the computer. This was happen for so long time but was so tough, error prone, slow and very laborious. At the later stage, when the change became mandatory, the initial switch positions were encoded as diodes on cards and were treated as peripheral and all important programs are started to enter into RAM. For small computers, programs were also not more than 256 words long at that time peoples started stored in ROM which gave access of boot from very few devices if you want to access booting from more devices from ROM then you have to change the ROM. After some time a major change came into process in early 1970 by Gary Kildall who was owner of digital research. He proposed that boot firmware provides an abstraction layer between the system hardware and operating system. This layer was to be used both to boot the system and to provide low level communication with the basic peripherals. When PCs are came into market developers implemented same concept called Basic Input Output System [1].

The BIOS consisted of two main pieces mainly Power On Self Test and the run time, which is worked as an abstraction layer for early PC based operating systems. POST is activated by the BIOS. It runs series of checks and diagnostics on motherboard. When BIOS executes, it is checking for registers of CPU, check the unification of the BIOS with itself, check the different peripheral components, check for main memory and select devices which are able to booting. Actually BIOS starts its POST when the reset

is given to CPU. When reset occurs then first memory location which CPU tries to execute is called as reset vector. During re boot, the CPU saves this code fetch to the BIOS stored on the system flash memory [1].

Before run time phase one more phase is occuring called Boot phase. The main functionality of the boot phase is to load the boot loader from memory into portion of operating system. It should execute immediately after POST phase but it can execute by operating system or it can be invoked multiple times in a attempt to find a valid boot media. When boot phase completes, run time phase will execute. Objective of this phase is to support these legacy calling interfaces and compatibility with BIOS standards. These legacy setting are used to set up CPU register state, called BIOS and returned CPU register state. After this resume and reboot will occur where it will called by software request to reboot the machine [1].

The basic goal of the BIOS was to test and initialize the system, to find an input device which in general we can say keyboard, an output device that would be like monitor and a boot device, usually called hard disk. After that BIOS start executing boot process by loading 512 bytes loader from hard drive and give control to it. After that BIOS played a supporting hardware abstraction role for the devices which it was aware. The abstraction layer provided by the BIOS were extremely primitive and provided no synchronization. Due to the synchronization problem, the BIOS abstractions were polled rather than interrupt driven. The abstractions have allowed for implementation of numerous underlying architectures, all of which are compatible with one another [2].

There is another abstraction in firmware provided by the PC BIOS which is called option ROMs. An option ROM is a BIOS extension that resides on an add-in card. The option ROM serves the same purpose as device driver in a operating system which allow the base software to access peripherals that it does not intrinsically known. Early many companies producing BIOS noted that BIOS was the only software that actually ran and required to boot the system. The second point that noted is firmware was the only piece of software that was absolutely tied to platform [2].

The BIOS discussed here is implemented in two different environment for Intels next Generation Processors. The two different environments are EDK I and EDK II (also called as Native). The detail about the EDK I and EDK II will be discussed later. The implementation with EDK I and EDK II is done with the help of a flag defined

in the code. Also the driver is implemented in two different modes in which one is with the complete package of the driver which contains all the source files, header files, dependency files if any, information files etc.. while the other mode is a binary implementation of the driver in which the drivers binary file will directly used in the firmware so no need to build the driver again as it it already a built in driver with the .efi image. Binary mode will contain only .efi file of the driver and information file for that. This is also implemented by defining a flag for that [2].

At last, like BIOS has been operating system neutral throughout its life, enabling remarkable innovation in the operating system community for platform sellers. A platform developed by the same company as the operating system, allowing close, often seamless, integration between software and firmware [2].

1.2 Motivation

It is always good to do something productive which can be considered as a beneficial thing in any area for an organization. Here the objective behind the work done in the thesis is to give more flexibility to the Intel clients by giving choices on boot loaders for FSP. Whenever user will use that FSP at that time users can use any open source boot loader to boot the system and Intel will provide only that part for which it is known.

BIOS comes before OS booting so for booting any OS, boot loader is required after BIOS execution. In regular BIOS, it is not possible to change or add any functionality of silicon initialization code because of legal terms. So, to give more access to customers FSP comes into frame which is a one type of binary which provide key programming information of silicon initialization code. It provides chipset and processor initialization in a format that can be easily incorporated into many existing boot loaders. So, by using FSP customers can use any open source boot loaders like coreboot, Linux and Yacto.

Different Intel hardware devices may have different Intel FSP binary instances, so a platform user needs to choose the right Intel FSP binary release. The FSP binary should be independent of the platform design but specific to the Intel CPU and chipset complex. We refer to the entities that create the FSP binary as the FSP Producer and the developer who integrates the FSP into some platform firmware as the FSP Consumer.

So, the motivation behind this thesis work is to develop FSP such that any limitation of boot loader from customer side should not be there. From customers side, they should change any functionality by its own and they should use any open source bootloaders in upcoming processors like Kabylake and Broxton.

1.3 Problem Statement

In general, BIOS provides all the important information to the platform for initialization of silicon which is very crucial for initializing any platform. there is some information which is not directly available but if anyone want to use that it will be available with legal agreements because of proprietary information. In the case of FSP, it is a binary distribution code for particular silicon initialization code. FSP provides ready access of all the key programming information mainly which is not publicly available. The second point of FSP is just to abstract the all complexities of the initialization of Intel silicon and will give you a required number of defined interfaces. So the design goal is to provide necessary required initialization code as FSP provides only some part of products features.

The limitation of EFI BIOS is that it allows to use particular boot loader for booting the system and it becomes hard point to follow rules when one has to implement another features. So FSP contains only silicon initialization code so users can add any other functionality inside their product without much problem.

Another objective is BIOS contain complete reference code so it requires larger memory to store when FSP have very small footprint like near about 200 KB. Because of small footprint its execution time is so fast compared to regular BIOS. Whenever user want to integrate boot loader with FSP it will integrate very easily and most important thing is it is supported among all Intel Atom, Core and Xeon processors.

So as per the thesis work, we developed the FSP among all the Intel next generation processors like Kabylake and Broxton which will be release at the end of 2016. We fulfilled the clients requirement of FSP for these processors and developed an excellent and stable FSP.

1.4 Thesis Organization

The thesis work carried out during the course of time has been presented in total six chapters as follows:

Chapter 1 Introduction, in this chapter, importance of the thesis and a brief background about BIOS like how BIOS is working, how it is implementing, its pieces as POST and run time phases is presented.
The motivation and problem statement of the thesis is also mentioned in this chapter.

Chapter 2 Literature Survey, It describes the basic about Intel platform architecture, basics about BIOS, Different phases of BIOS, Different boot modules, software architecture overview, UEFI specification, about legacy BIOS and EFI BIOS. It also gives idea about EDK and EDK II platforms.

Chapter 3 Firmware Support Package, It describes what is FSP, Boot flow of FSP, Different versions of FSP, Modification in FSP for reference code in every year, Different firmware volumes of FSP, Basics about different APIs as well as functionality of that APIs.

Chapter 4 FSP Implementation Strategy, This chapter contain the execution of FSP in reference code, POST and Phase code implementation, Implementation of python script for FSP driver separation,How to execute the FSP in platform, FSP 2.0 implementation and the result produced by FSP, How to configure the platform firmware at the run time and statically, Responsiveness infrastructure of FSP for getting execution time of all drivers and at the last how to boot the OS for firmware from eMMC.

Chapter 5 Automated BIOS generation using System RDL, This is another project on which i am working. This chapter contains the overview of the automation of BIOS generation, Firmware Developement Model of systemRDL, It's usage model and implementation of the project which can help at every aspect.

Chapter 6 Conclusion and Future Scope, It contains all the conclusions related

to FSP execution and how important automated BIOS generation from system RDL. In future scope, there are some thoughts which we discussed with architects can be help more on FSP to give the flexibility to customers. From advancement side of automation for BIOS, which can give more flexibility to developers.

Chapter 2

Literature Survey

2.1 Intel Platform Architecture

Platform encompasses all required ingredients, features, capabilities, initiatives and technologies. Total four major components of platform are as follows [3]:

1. **Hardware:**
   Processors, chipsets, Communication devices, Memory, Boards, and Systems [3].

2. **Software:**
   Operating systems, Applications, Firmware and Compilers [3].

3. **Technologies:**
   Hyper Threading Technology(HTT), Intel virtualization Technology, Intel Active Management Technology(AMT) [3].

4. **Standard and Initiatives:**
   Wi-Fi, WiMAX, The Wireless Verification Program [3].

Intel platform architecture is complex with lots of components on it. Every component must work as designed and there shouldnt́ be any conflicts between the devices on it. Now a days it is come up with single chip solution which means CPU and PCH are in single die. Intel platform architecture is shown in Fig. 2.1. From Fig. 2.1, it is very clear there are many components connected with platform. Mainly at higher level there are two divisions called Northbridge and Southbridge. These two components are connected by a bus which is known as DMI. It is mostly Intel link for this bridge

connection on motherboard. By using this DMI bus north bridge and south bridge will communicate with each other [4].
Northbridge is usually a one type of hub which is used to control the high end devices like memory controller. Because of this high performance demand, north bridge is directly connected to CPU. Here shown in figure all the high end devices like graphics bus and memory bus are connected to Northbridge. High speed graphics bus and memory bus are connected by PCI Express bus with north bridge. PCI Express bus is one type of serial bus which has very high performance in terms of data transfer. In earlier Intel platform architecture, it was used PCI bus which was parallel bus so its data rate was so low which was not suit for Northbridge thats why PCI is replaced by PCI Express which gives very high speed because of point to point technology [4].

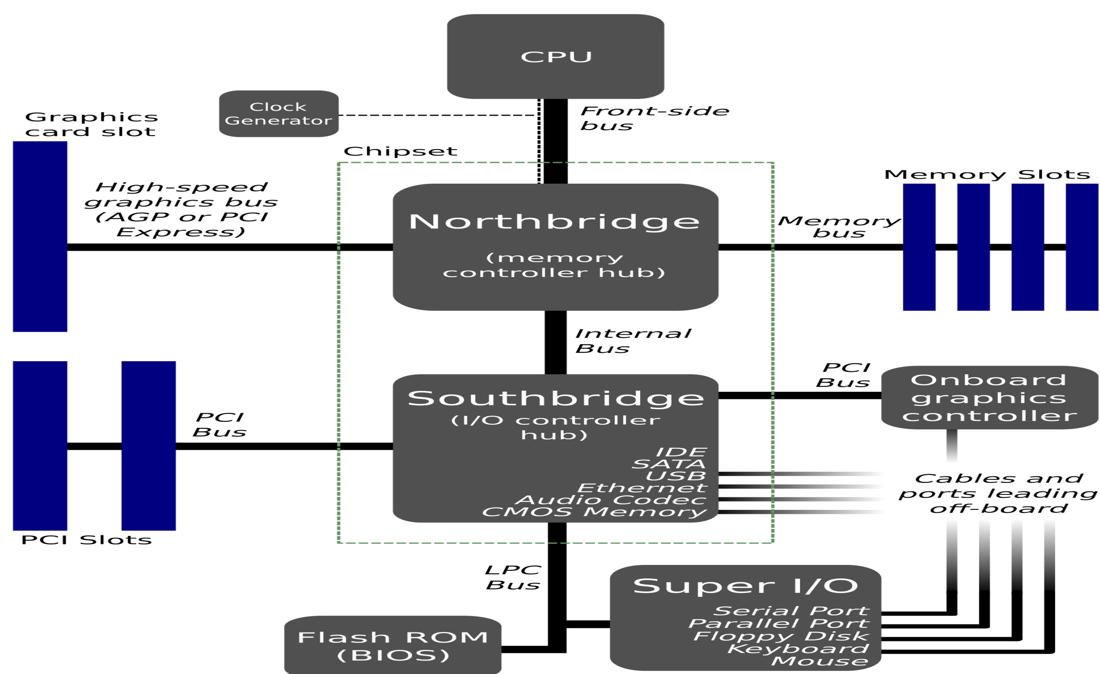


Fig. 2.1: Intel Platform Architecture [4]

Bridge which is connected by DMI with Northbridge is called Southbridge. Southbridge implements the slower performance speed compared to Northbridge because unlike Northbridge it is not directly connected to CPU. In Intel platform it is also called as I/O controller hub. It handles all of the computers I/O functions such as USB, audio, serial, the system BIOS, Interrupt controller, and the IDE channels. As shown in Fig. 2.1, all the pots like serial port, parallel port, keyboard, Mouse, etc. all ports are con-

nected with south bridge. There are PCI slots also available with Southbridge which can be used to connect more devices.

Now a days in modern platform architecture, Intel did a major change like instead of Southbridge, they put PCH block which is directly connected with CPU in a single die. PCH was found to overcome the bottleneck between processor and motherboard. As speed of processor is increasing, the data transfer rate between the CPU and motherboard would achieve full bandwidth. So by PCH it overcome because it took most of the task of the Southbridge and few roles of north bridge too by front side bus that have not been incorporated into the CPU package [4].

2.2 Platform Software Architecture

The definition of firmware on PC is instructions (with data) that are consumed by non-IA execution engines associated with a non-CPU hardware device. There are three main categories of firmware as follows [4]:

- Fixed embedded firmware: contained in ROM and hidden from platform view
- Upgradeable embedded firmware: contained in built-in non-volatile memory with default image (code/data); upgradable during life cycle
- Externally stored firmware: storage of the code/data is outside of the device package that executes the firmware. The external storage is likely in non-volatile memory form. Patches for embedded firmware falls into this category

No action is required on platform SW to support fixed embedded firmware. Examples of embedded firmware components are [4]:

- CPU microcode, uncore firmware
- ME ROM code

Upgradeable embedded firmware is presumed to be functional at platform build time. Discrete graphics card firmware is in this category. Upgrade tool is expected to be available for at least one of the users SW environment. There is no known ingredient with

firmware in this category [4].
Firmware code or data is contain by physical repository of firmware device. A physical firmware device may be divided into smaller pieces to form a multiple logical firmware device. This logical firmware device is called firmware volume. Each volume is arranged into a bunch of files. As we know the file unit is the basic unit of storage for firmware space. Many formats of files have quite different and discrete parts within it. These parts are called file sections, otherwise just sections for short. All the sections begin with one type of header that declares the type as well as the length of that section. The headers for this section must be 4 bytes aligned within the parent files image [4].
Mainly there are large number of sections, they fall into the main two big categories as follows [4]:

- Encapsulation sections
- Leaf sections

First section is encapsulation section which is essentially container that hold other sections. The sections which contained inside an encapsulation section is called as child sections. Leaf section directly contain data and mainly do not contain other sections unlike encapsulation sections [4].

2.3 BIOS Overview

BIOS is the first code run by a PC when powered on. It acts as a layer between OS and Hardware. BIOS initialize the various platform components like CPU initialization, core initialization, memory and chipset initialization etc. The BIOS must do its job before your computer can load its operating system and applications. The basic input/output system (BIOS), also known as the System BIOS or ROM BIOS, is a de facto standard defining a firmware interface.
The BIOS software is built into the PC, and is the first code run by a PC when powered on like boot firmware. The primary function of the BIOS is to set up the hardware and load and start a boot loader. When the PC starts up, the first job for the BIOS is to initialize and identify system devices such as the video display card, keyboard and mouse, hard disk drive, optical disc drive and other hardware [4].

The BIOS then locates software held on a peripheral device (designated as a boot device), such as a hard disk or a CD/DVD, and loads and executes that software, giving it control of the PC. This process is known as booting, or booting up, which is short for bootstrapping [4].

BIOS software is stored on a non-volatile ROM chip built into the system on the motherboard. The BIOS software is specifically designed to work with the particular type of system in question, including having knowledge of the workings of various devices that make up the complementary chipset of the system. In modern computer systems, the BIOS chip's contents can be rewritten, allowing BIOS software to be upgraded.

BIOS features are as follows [4]:

- It acts as a layer between OS and Hardware
- It gets your computer up and running
- Initializes the hardware like Microprocessor, memory, chipset, devices, peripherals etc

BIOS is communicating with platform and operating system when it is doing its task and after finishing its task it will work as supporting code as shown in Fig. 2.2 as follows [4]:

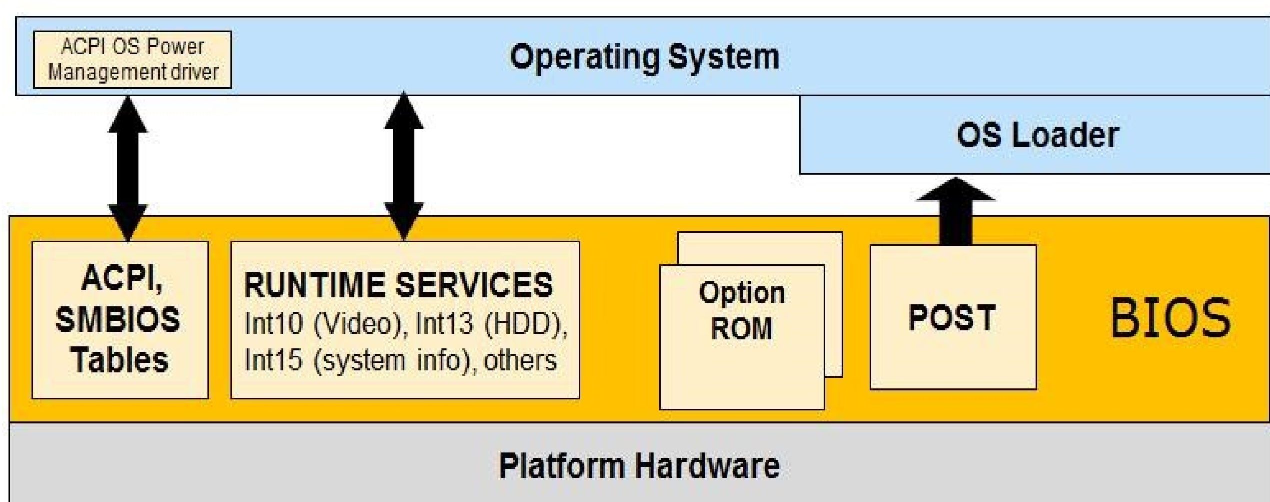


Fig. 2.2: High Level Diagram of BIOS Space [4]

From Fig. 2.2, it is clear that BIOS is medium to communicate between operating system and platform hardware. BIOS is the only software in the platform that knows all the details of the motherboard. POST is the responsible for testing system which will use OS loader to load operating system whereas ACPI and SMBIOS tables are useful for power management and to control it. Runtime services are services which generates interrupt for particular service to OS then OS give return back notification by executing that services.
Basically BIOS will provide services like:

- It gives all the power related management functionality through ACPI
- It loads and take care of control over to the OS boot loader
- It give a set of standardized regularities for the OS to use
- It fetch the motherboard and silicon environment from the OS
- It build the system to work an OS
- It give runtime services to the OS e.g. disk and video

The Advanced Configuration and Power Interface (ACPI) is one kind of a specification which was prepared to structured most of the industry related interfaces which gives robust operating system (OS)-directed motherboard configuration as well as management related to the power of both devices as well as entire systems. This is the heart for an Operating System-directed configuration and Power Management (OSPM).
Power-On Self-Test (POST) routines run very quickly after power is giving to system, by nearly all electronic devices. It includes regularities to set an primary value for internal as well as output signals and to execute all internal tests, as found by the device manufacturer. These initial conditions are also referred to as the devices state.

2.4 Legacy and EFI BIOS

Currently, Industry has migrated from Legacy BIOS to a standard and modular EFI BIOS. EFI BIOS offers new improved features and flexibility for code developers. The difference between Legacy BIOS and EFI BIOS is shown in Table 2.1.

Legacy BIOS	UEFI BIOS
This is traditional BIOS	New architecture based on EFI spec
Written in assembly code, initially designed for IBM PC-AT	C based, initially designed for itanium server systems
Interface is per-BIOS spaghetti code, not modular	Well defined module environment and interface based on EFI specification
Lives within the first 1MB of system memory	Can live anywhere in the 4GB system memory space
Uses 16bit memory access. Requires hacks to access above 1MB memory	Allow direct access of all memory via(32 bit and 64 bit) pointers
Supports 3rd party modules in the form of 16 bit option ROMs	Supports 3rd party 32/64 bit drivers
Examples: AMI core 8, Phoenix legacy BIOS	Examples: Aptio (AMI), H20 (Insyde), Tiano (Intel)

Table 2.1: Legacy BIOS and EFI BIOS [4]

Legacy BIOS is able to run different operating system, like MS-DOS, equally well on computers other than IBM. Additionally Legacy BIOS has defined OS independent interface for hardware that enables interrupts to communicate with video, disk and keyboard devices along with BIOS ROM loader and bootstrap loader. Now a days legacy BIOS is not used that much then also its two functions called system configuration and setup are using now.
UEFI was created to change the Legacy BIOS to streamline the process of booting, and it behave as the interface between operating system of computer and its platform firmware. It not only changes the most functions of BIOS, but also offers a rich extensible pre-OS environment which will give advanced boot and runtime services [4].

Fig. 2.3: UEFI Interface between BIOS and OS Loader [4]

UEFI to BIOS accomplishes the same basic task as it is act as common interface between system firmware and operating system. It has same size and performance mea-

sures as it has boot faster in less flash environment. It should be operating system neutral like it should work in variety of system including Linux and windows. Major change in between this access method of ACPI and SMBIOS are different[7].
BIOS code whatever we are using today enable for legacy BIOS as well as EFI BIOS by means of a module called CSM. Some OS not support the EFI BIO, so for that CSM module is used to run necessary BIOS code. When this mode is ON, system will boot for legacy BIOS and if this mode is OFF, system will boot to native EFI BIOS [4].
The CSM module is used to translates all the data generated under the EFI environment into the data or information required by the legacy environment as well as it will makes the legacy BIOS services available such that it will booting to the operating system and for use in runtime [4].

2.5 UEFI Specifications

Unified Extensible Firmware Interface (UEFI) is grounded in Intel initial Extensible Firmware Interface(EFI) specification, 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 systems.
The interface inside the UEFI is in the form of data tables which contains platform-related all information, after that boot as well as runtime service calls which are available to the OS loader and to the OS. By all this, it provides a complete environment for booting an OS. This specification is designed only as a pure specification for interface. As we know, the specification defines the all the set of interfaces as well as structures of that platform firmware must implement. Similarly, another way specification defines all the set of interfaces as well as structures that the OS may use in booting. The interface of UEFI and compatibility of UEFI specification is shown in Fig. 2.4 as follows [4]:

Fig. 2.4: UEFI Interface [4]

The intent of the specification is to define a way for the OS and platform firmware to communicate only information necessary to support the OS boot process. This is accomplished through a formal and complete abstract specification of the software visible interface presented to the OS by the platform and firmware. Furthermore, an abstract specification opens a route to replace legacy devices and firmware code over time. New device types and associated code can provide equivalent functionality through the same defined abstract interface, again without impact on the OS boot support code [4].
The specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market segments. At the same time, the specification allows maximum extensibility and customization abilities for OEMs to allow differentiation. In this, the purpose of UEFI is to define an evolutionary path from the traditional PC-AT-style boot world into a legacy-API free environment [4].

2.6 UEFI BIOS Boot Phases

When platform initialization occurs, at that time BIOS will pass through different four phases called Security, Pre EFI initialization, Driver execution environment and Boot device select. BIOS starts execution once power will be on. After that it execute the

phases step by step for initialization and during BDS phase, it gives the control to OS after that by using boot loader OS takes care the system and BIOS work as supporting code. As shown the flowchart in Fig. 2.5 [4].

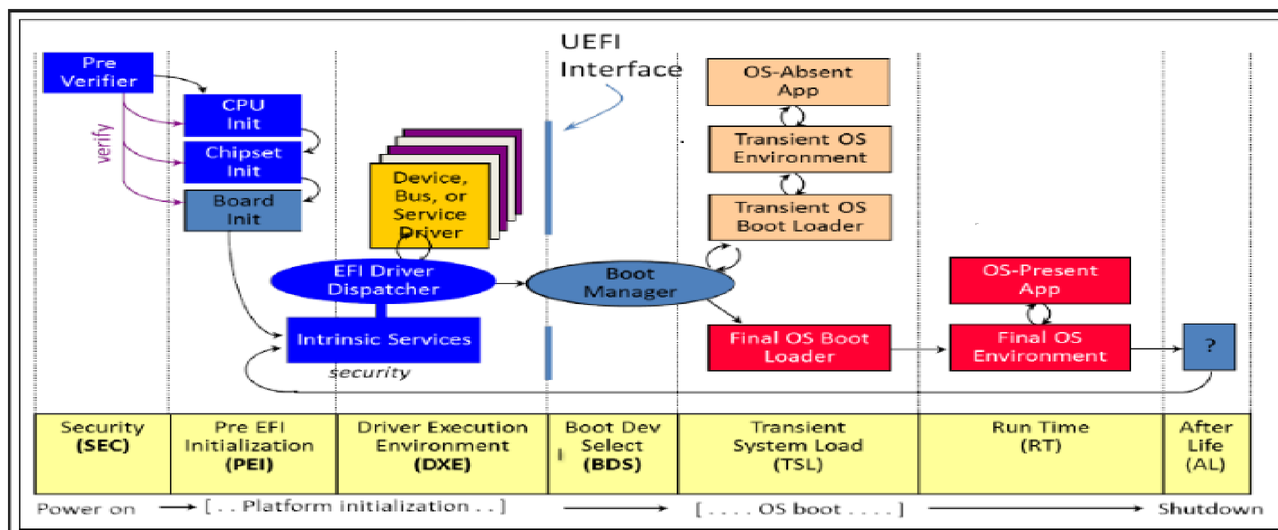


Fig. 2.5: Phases of BIOS Execution [4]

EFI BIOS is a modular code and it boots in a manner such that each phase will execute correctly. EFI Boot process is dived into four main phases which are:

- Security Phase
- PEI Phase
- DXE Phase
- BDS Phase as shown in Table 2.2.

SEC	PEI	DXE	BDS
Provides processor boot strap vector	Initializes processor	Initializes processor, cache, chipset and SMM	Run down list of selected devices.
Initializes temporary RAM using CPU cache	Detect corrupted flash and recover if corrupted	Execute PCI enumeration and initializes video, keyboard, mouse and USB legacy	Load boot image into memory
Provides optional security features	Find and initializes RAM	Initializes drivers and create table interfaces	Jump to boot image and try for next image

Table 2.2: EFI Boot Phases and Services [4]

1. **SEC Phase:**

   The Security phase or we can say SEC phase is the first phase in the PI Architectue and is used for the following:

   - It handles all platform related restart events
   - It creates a memory store for temporary base
   - It serves as the root of trust in the system
   - It passes information related to handoff to the PEI Core

   This phase contains the first code executed after power-on or reset. Not only have peripherals not been initialized, but memory may not be available. The SEC phase is not particularly suitable for software-based performance measurement. For most platforms, SECs total elapsed time can be determined by measuring from the beginning of time to the start of PEI. This is the mechanism used by EDK II. Apart from the minimum required architecturally information about handoff, this phase can also pass optional information to the next phase called PEI Core, such as the SEC Platform will give Information PPI or may be information about the health status of the processor [4].

2. **PEI Phase:**

   The above phase called the PEI phase of the PI Architecture is invoked quite early in the phase of boot flow. Specifically, after some initial processing in the Security (SEC) phase, any restarting event will called by the PEI phase. The PEI phase will operate with the platform in a nascent state at the starting phase, leveraging only on processor resources, like the cache of processor as a call stack, to release Pre-EFI Initialization Modules (PEIMs). These PEIMs are responsible for the following [4]:

   - Initializing most of original memory complement
   - Describing the memory stack data in Hand-Off Blocks (HOBs)
   - Describing the location of volume for firmware in HOBs
   - Giving control into the next phase called Driver Execution Environment (DXE) phase

Ideally, this phase is intended to be the smallest amount of code chunk to achieve the ends listed above. As such, any other reliable algorithms as well as processing should be deferred to the DXE phase of execution. This phase actually consists of two sub-phases: Prememory and Postmemory. PreMem is the state before main memory is available for use and PostMem refers to the state after main memory is usable [4].

After SEC phase transitions to PEI phase, the firmware is in the PreMem state. At this point, some temporary memory is usually available. On some IA platforms the temporary memory is actually a portion of the processors cache that has been placed in a special mode. While operating in this special mode a number of restrictions exist:

- Temporary memory may only be used for data storage, not instruction execution
- The size of temporary memory is usually small
- Temporary memory will not survive enabling of caching
- Initialized external or static variables cannot be used since they will reside within the read-only firmware device, not temporary memory. Global constants can be used, but they must be declared as CONST and treated as read-only

The PEI phase flow is shown in Fig. 2.6.

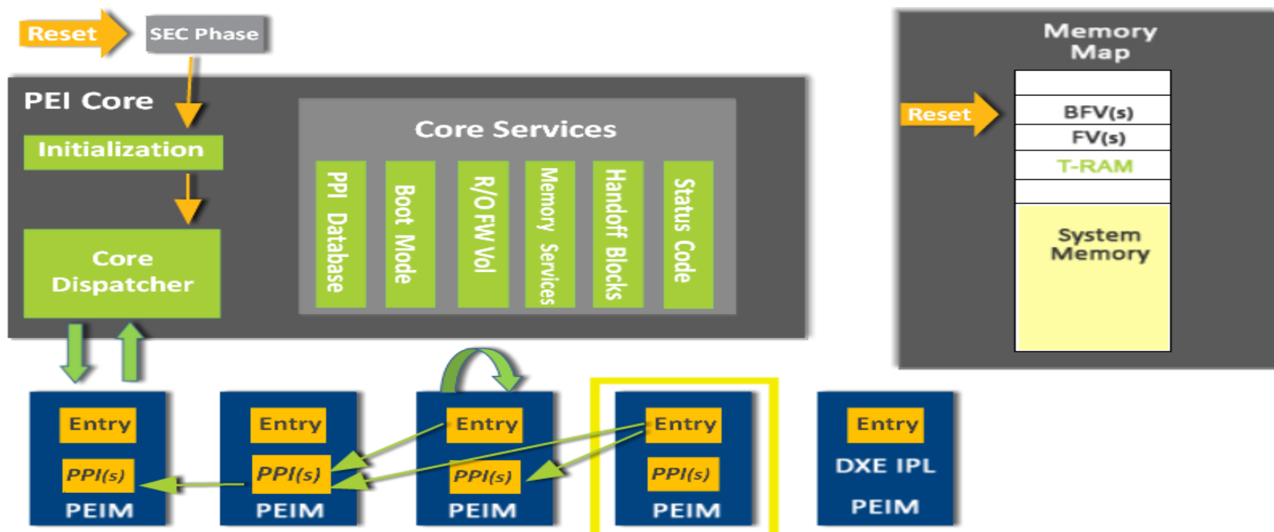


Fig. 2.6: PEI Phase Flow [4]

As shown in Fig. 2.6, it is high level flow of PEI execution in which when power on will happen SEC phase will occur and execute from memory map. During this phase T-RAM should also initialize. After that SEC invokes the PEI initialization which will call different PEI services. In next phase core dispatcher comes which finds firmware volume to execute PEIM which execute PPIs. After all PEIM execution system memory will initialize and at last final PEIM executes called DXE IPL which will do transaction into DXE phase. Table 2.3 is PEI service table as follows [4]:

Services	**Functions**
PPI	InstallPpi() LocatePpi() NotifyPpi() ReInstallPpi()
Boot Mode	GetBootMode() SetBootMode()
HOB	GetHobList() CreateHob()
Flash Volume	FfsFindNextFile()
PEI Memory	InstallPeiMemory() AllocatePool()
Status Code	PeiReportStatusCode()
Reset	PeiResetSystem

Table 2.3: PEI Services and its Functions [4]

As soon as memory has been initialized and is ready for use, these HOBs, and other PEI data stored in temporary memory, are copied into main memory and the PEI phase continues on in the PostMem state. Due to memory restrictions during

the PreMem state, some differences exist between performance measurements made during PEI and measurements made later.

This phase is used for crisis recovery as well as resuming from the S3 sleep state. For the crisis recovery, this phase should be there reside in some small, fault-avoidance block of the firmware store. Because of that, it is imperative to keep the footprint of the PEI phase as small as possible. Apart from that, for a successful resume for S3, the resume speed is of utmost importance, so the code path which will be through the firmware should be minimized. Above two boot flows also speak to the need to keep the processing as well as code paths in the PEI phase to a minimum. The implementation of this phase will be more dependent on the architecture of the processor than any other phase. In particular, behind this idea, the more resources the processor provides at its initial or nearby initial state, the richer the interface between the PEI Foundation and PEIMs [4].

3. **DXE Phase:**

   This phase also called as Driver Execution Environment is where most of the system will perform initialization. Pre-EFI Initialization which is the phase comes before DXE, is used for initializing main memory in the platform so that the this phase can be loaded and can be executed. The control of the system at the end of the PEI phase is passed to the this phase through a list of data structures created based on position-independent called Hand-Off Blocks (HOBs). By using DXE phase the amount of complexity become decrease and enable to write a code in more modular fashion. Lets see how PEI to DXE transition will happen by Fig. 2.7 [4].

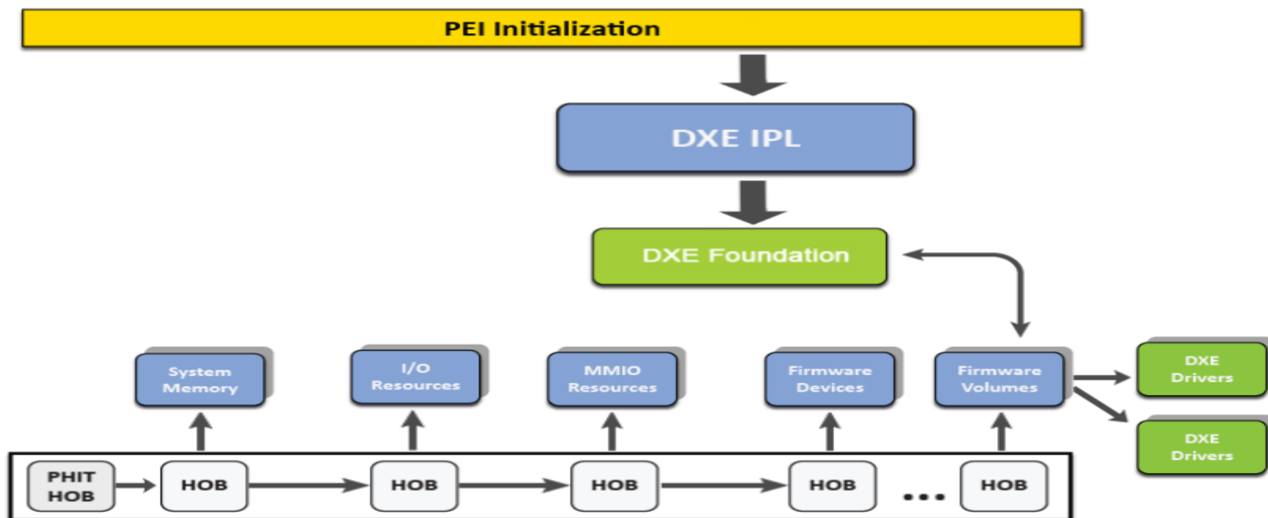


Fig. 2.7: PEI to DXE Foundation [4]

When PEI initialize it calls to DXE IPL which store the HOBs. After DXE IPL complete its initialization, it call to DXE foundation which calls firmware volumes to extract the DXE drivers. HOBs are the only blocks which pass from PEI to DXE phase. DXE foundation populate the architecture protocols. Architecture protocols are the only DXE drivers that can access the hardware directly. It means whichever DXE drivers want to communicate with hardware has to go through the architecture protocols.

There is particular DXE phase flow is also there in order to execute the particular services of DXE phase as shown in Fig. 2.8. In this DXE phase overview, DXE foundation will initialize the DXE services, once this services initializes foundation begins to dispatch the DXE drivers. DXE dispatcher looks for firmware volume which will find a priori file. A priori file contain DXE drivers which have peak of image. DXE drivers should execute in exact order what they mentioned because each DXE driver is loaded into system memory specified by DXE dispatcher. Once dispatcher complete a priori file the it will look for more firmware volumes until all DXE drivers will execute. Once all the drivers are execute then DXE dispatcher will look for a priori file and check wheather any driver still need time to execute otherwise it will leave the DXE phase and will give the control to next phase [4].

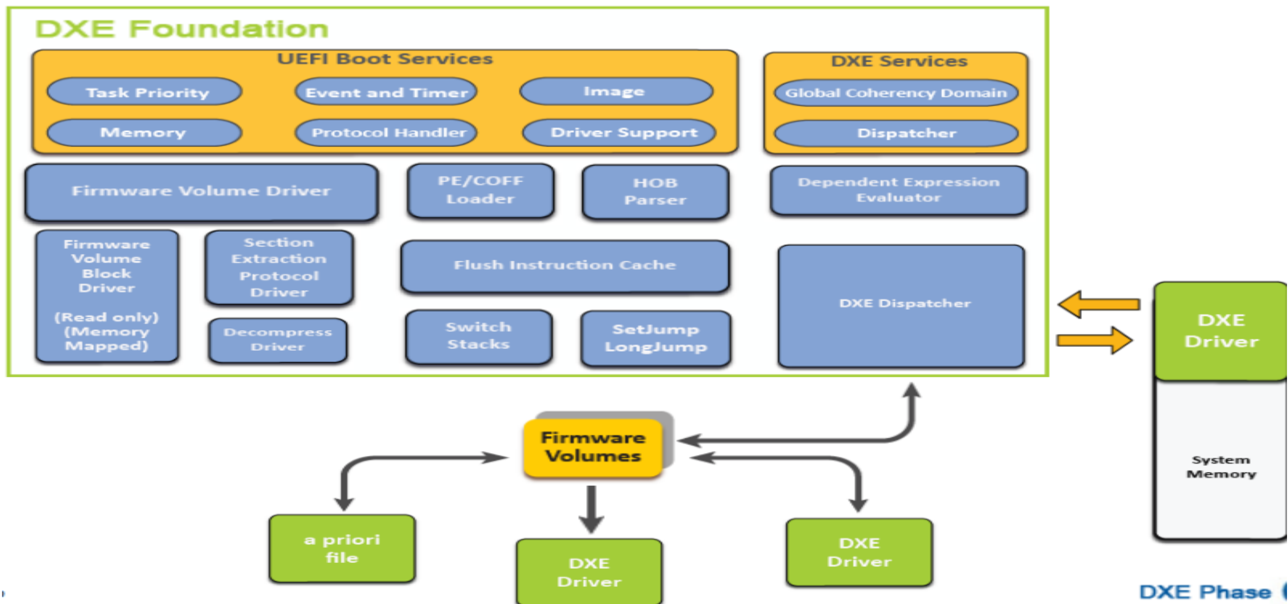


Fig. 2.8: DXE Phase Flow [4]

4. **BDS Phase:**

During Driver Execution Environment, the Boot Manager executes the DXE drivers by using DXE Dispatcher only after all the DXE drivers whose dependencies have been satisfied. After this process, control is given to the BDS phase of execution. The BDS phase is used for implementing the boot policy of platform. This boot policy gives flexibility which allows system vendors to optimize the user experience during BDS phase of execution. The Boot Manager will also support booting from a short-form device path which will starts with the first node and being a firmware volume device path.

The boot manager should must use the GUID in the volume of firmware device node to match it to a firmware volume inside the system. The GUID in the volume of firmware device path is compared with the firmware volume name GUID. If both match is made, then the volume for firmware device path can be appended to the device path of the matching volume for firmware and normal boot behavior can now be used. The BDS phase is implemented as part of the BDS Architectural Protocol.

The Driver Execution Environment Foundation gives control to this phases Architectural Protocol after all of the Driver Execution Environment drivers who have dependencies satisfied as well as loaded and executed by the DXE Dispatcher.

The BDS phase is responsible for the following [4]:

- It will initialize the console devices
- It will load the device drivers
- It will trying to attempt load and execute boot selections

If the BDS phase will not be able make progress, it goes for executing the DXE Dispatcher to see if the dependencies of any additional DXE drivers have been satisfied or not since the last time the DXE Dispatcher was invoked.
DXE dispatcher calls last DXE driver called BDS driver. In general DXE core call DXE dispatcher which first complete all the DXE and UEFI drivers and then it completes BDS entry which is implemented as driver. BDS entry could require other driver to dispatch so that BDS might recall DXE dispatcher. This scenario is shown in Fig. 2.9 [4].

Fig. 2.9: BDS Phase Flow [4]

2.7 EDK and EDK II Platform

EDK (Extensible Firmware Interface Developer Kit) was the first generation of the open source EFI development kit. EDK was a development environment designed with a functional arrangement of the components to support windows development. EDK provides access to the outside of Intel who did not have a direct license agreement with

Intel with more robust development environment. EDK was directed at companies who develop both the firmware and the drivers. EDK was eventually upgraded to EDK II which expands operating system support.
EDK II is the second generation development environment. It has two main objectives which were lacking in the first generation EDK as follows [4]:

- EDK II organizes the content in whole chunk which are added and removed as a whole

- EDK II allows compiling under multiple operating systems, including windows, Linux, Apple OS

Chapter 3

Firmware Support Package

3.1 Overview

Intel Firmware Support Package (FSP) provides key programming information for initializing Intel silicon including the processor, memory controller, chipset and certain bus interface as needed. It can be easily integrated into a firmware boot environment of the developers choice such as core boot, Wind River Vxworks, RTOS, Linux and open source firmware.

Intel found that it keeps the necessary information of programming which is crucial for silicon initialization. Some important information of programming is treated as secret information and if anyone wants then only be accessible with legal agreements. The first point for design of FSP is to provide easy access to the important information of programming that is not available everywhere. The second point of FSP is just to abstract the all complexities of the initialization of Intel silicon and will give you a required number of defined interfaces. So main design goal is to provide necessary required initialization code as FSP will provide us only some part of products features [5].

FSP is easy to adopt, economical to build and scalable to design thereby reducing time to market. As it is not a standalone boot loader, it must be integrated with host boot loader infrastructure to carry out other functions such as [5]:

- Initializing non Intel components
- Conducting bus enumeration and discovering devices in the system

- Industry Standards

When FSP will create it gives package in which it contains following:

- FSP binary
- Guide for Integration
- Updatable Product Data (UPD) or Vital Product Data (VPD) structure definitions
- File for Boot Settings (BSF)

The utility required for FSP configuration called Binary Configuration Tool (BCT) is available as a different package. By using this tool FSP configuration data can be easily changed which is there in UPD and we can change that data run time. So, by using this tool FSP can modify the setting at the run time [5].

3.2 FSP Usage Model

FSP is basically creating from complete reference BIOS only which is shown in Fig. 3.1 as follows:

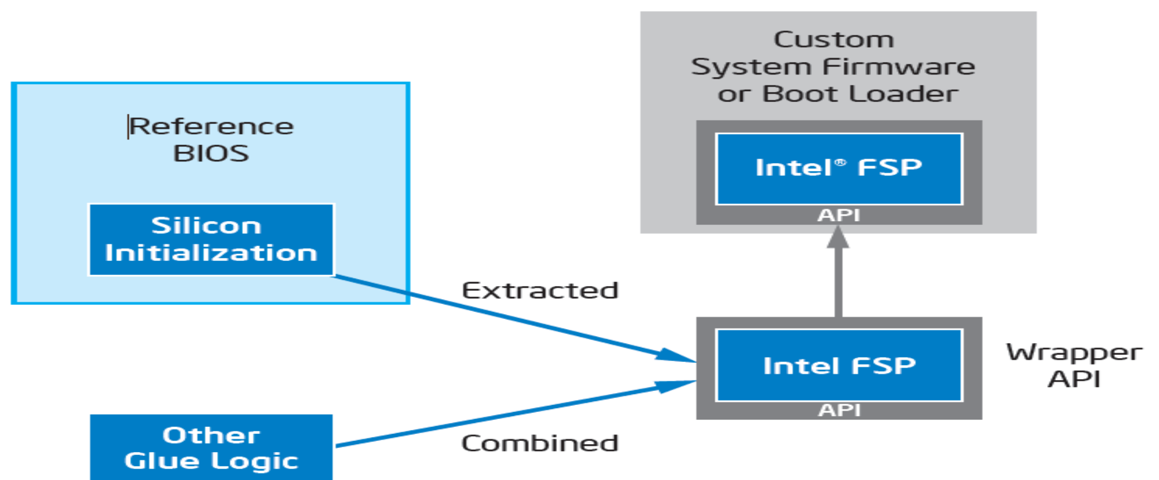


Fig. 3.1: FSP Usage Model [5]

As per Fig. 3.1, it is clear that first only silicon initialization code is require from complete reference BIOS by extracting from Intel boot loader development kit. This

Silicon initialization code is combined with FSP glue logic which contain different architecture protocols which makes wrapper API. After that we have to put that wrapper into custom system boot loader to load that FSP in the system. Basically FSP is incorporated into many existing boot loader frameworks without exposing the Intellectual Property (IP) of Intel. FSP is distributed as single binary package to the customer. In FSP, all the silicon initialization PEIMs are packaged into one single package and make one separate package which will work as FSP. After making as single package it can plugs into existing firmware frameworks which is suit for all platform because of no need of any modification required to support the FSP. Main objective of FSP is to do a binary customization of regular silicon initialization code so that anyone can use very freely without much modification with whatever existing boot loader which gives more freedom to customers [5].

3.3 FSP Code Delivery Model

At every year Intel modified its FSP code delivery model by giving change in code structure of complete reference BIOS in silicon initialization as shown in Fig. 3.2 [5].

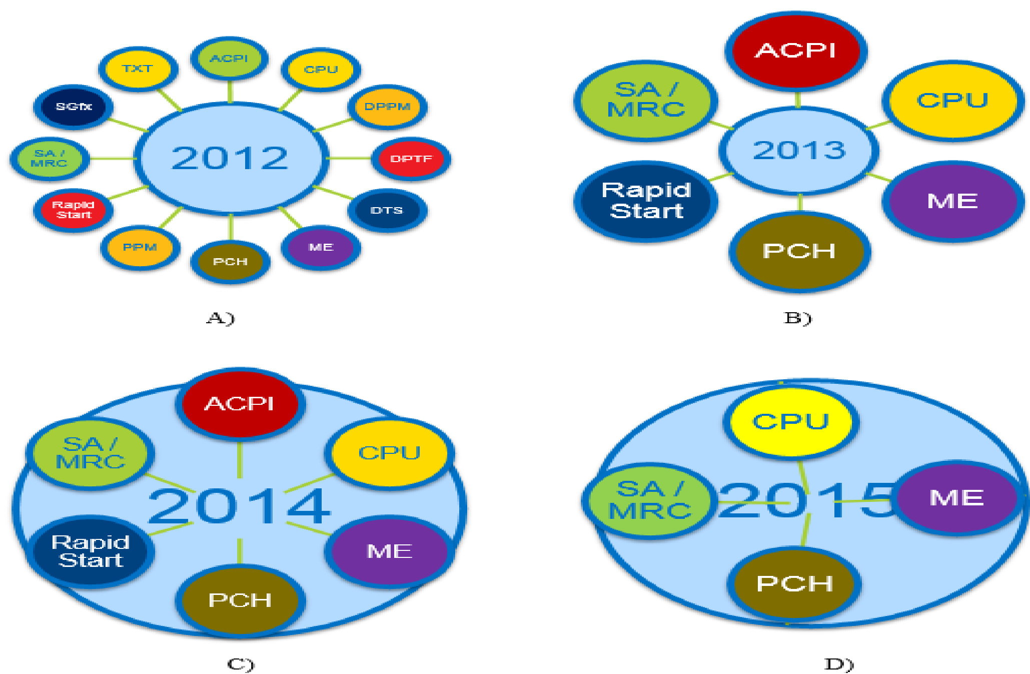


Fig. 3.2: Code Delivery Model for FSP A) 2012 B) 2013 C) 2014 D) 2015 [5]

In 2012, Intel started to implement FSP from reference code of silicon initialization. From Ivybridge platform FSP is started to execute. In this time Intel was providing total 12 different reference code packages for binary enabling strategy as well as at that time all reference code packages were individual. In the 2013, when Haswell platform came at that time Intel gave a FSP implementation as a 6 reference code packages as all are separate only. In the 2014, Intel did major changed in a FSP code format at the time of Broadwell where they made a single RC package which single RC package is of binary package but the number of reference code packages are same as of 2013. In Braswell at 2014 and Skylake at 2015, Intel gave more reduction in reference code packages as they gave only for silicon package namely South Agent, PCH, ME and CPU as a single RC package and gave platform package as sample code [5].

3.4 FSP Integration

The FSP binary can be integrated into many different boot loaders and embedded OS. Below are some required steps for the integration [5].

1. **Customizing:**
   The FSP has some sets of configuration parameters that are part of the FSP binary and can be customized by external tools provided by Intel [5].

2. **Rebasing:**
   The FSP is not Position Independent Code (PIC) and the whole FSP has to be rebased if it is placed at a location which is different from the preferred base address specified during the FSP build [5].

3. **Placing:** Once the FSP binary is ready for integration, the bootloader needs to be modified to place this FSP binary at the specific base address identified above [5].

4. **Interfacing:** The bootloader needs to add code to setup the operating environment for the FSP, call the FSP with the correct parameters and parse the FSP output to retrieve the necessary information returned by the FSP [5].

3.5 FSP Boot Flow

When binary enabling strategy completed then the next step will be the flow of booting for FSP. During FSP, we have to decide which API will come at which position and based on that have to decide design flow. There are different APIs for memory, silicon and temporary memory to execute that we should have to follow for particular execution of FSP. In boot flow every step of API is very important to execute correctly because every API is dependent on next API to work for so when previous API will execute at that time next API will start to initialize. FSP boot flow is shown in Fig. 3.3 as follows [5]:

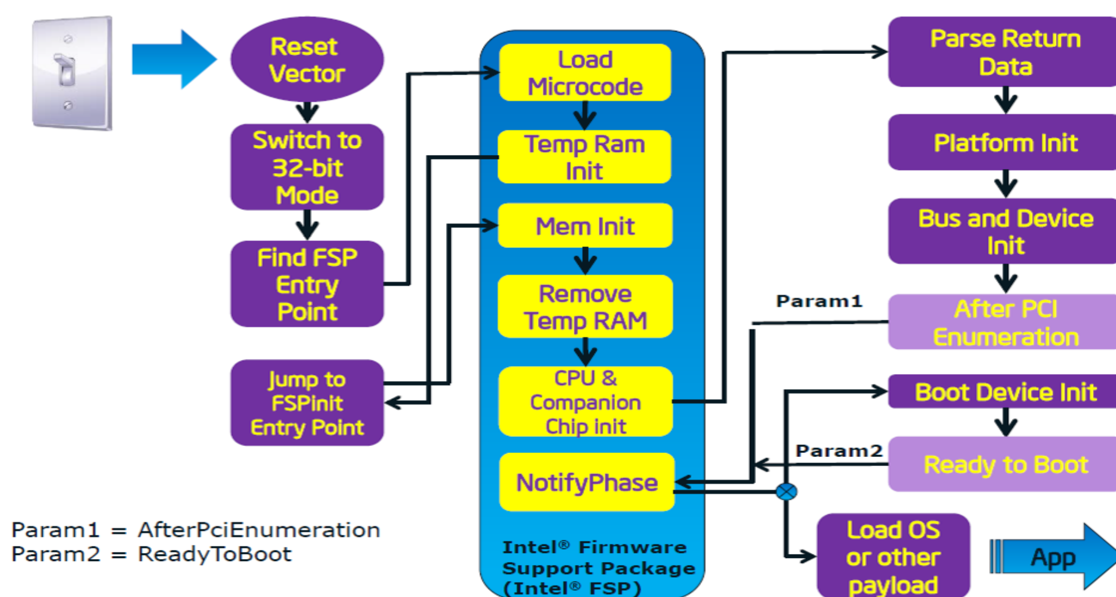


Fig. 3.3: FSP Boot Flow [5]

From Fig. 3.3, which is flow of FSP, with the FSP binary from the FSP Producer in blue and the platform code that integrates the binary, or the FSP Consumer, in purple. The FSP EAS describes both the API interface to the FSP binary that the consumer code invokes, but it also describes the hand off state from the execution of the FSP binary. The latter information is conveyed in Hand-Off Blocks. When FSP started executing it find for entry point from where it can start. After that it will initialize cache as RAM. At the next phase it initializes memory and remove temporary memory as well. After memory portion it comes to chipset initialization and initialize platform and gives con-

trol to OS after ready to boot event.

3.6 FSP Versions

Day by day when FSP becomes more popular then changes in boot flow was also required. According to demand of board or customer boot flow should be like more appropriate for customers which should give good performance. So, initially FSP had version 1.0 at the time of FSP started. After that Intel started to work on FSP 1.1 which was giving more control and flexibility to the boot loader which ultimately was good for customers. At later stage now Intel invent FSP 2.0 which have many changes over FSP 1.1 which is implemented in latest platforms and giving good performance over other versions [5].

a) FSP 1.0 and 1.1:

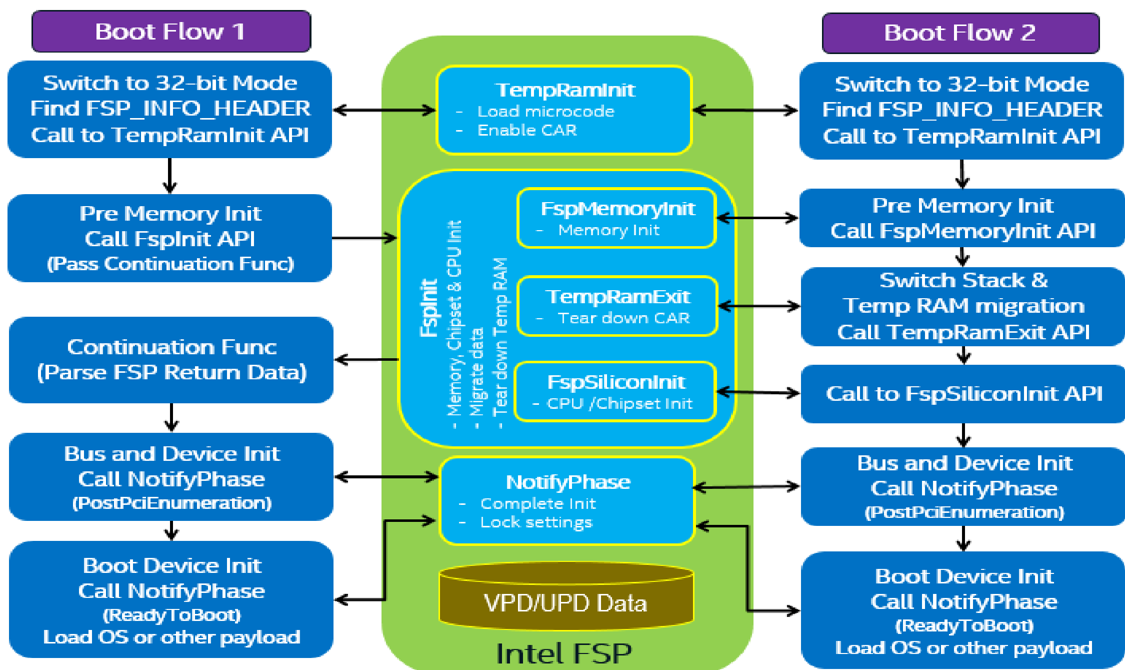


Fig. 3.4: FSP Boot flow Versions (a) 1.0 (b) 1.1[5]

From Fig. 3.4, one can see FSP 1.0 and 1.1 boot flow. As simple way its look like a same flow but changed many things. From the figure it is clear that at the TempRamInit API both are same but when we see at the next step which is MemoryInit API

at that point both version are changed. In FSP 1.0 when control comes at FSPInit API then it will go to FSP package and execute all three API called FspMemoryInit, TempRamInit and FspSiliconInit. After executing this all only control come back to boot loader. Whereas we can see that in FSP 1.1 after every API execution control come back to boot loader means after FspMemoryInit, TempRamInit and FspSiliconInit API control come back and search for boot loader and after that it will execute other phases. So by this method FSP upgraded from 1.0 to 1.1. By doing this change in FSP 1.1, Intel gave more access to boot loader over FSP so that if any change wants to do then can do it easily and give it to FSP.

In the FSP boot flow all the APIs are playing very important role during execution. Every API should call in order to execute FSP correctly by method. So now we can see one by one every API as follows [5]:

1) TempRamInit API:

The above API which is TempRamInit is called immediately after coming out of reset as well as before the memory and cache are available. This API also loads the microcode updation, enables code cache for a region which is specified by the boot loader and it sets up a temporary memory stack to be used prior to main memory stack being initialized. To execute this API, a hardcoded stack memory must be set up with the following values [5]:

- The return address of this API where the TempRamInit returns control
- A pointer for the input parameter for this API

A prototype of this API is define as shown in Fig. 3.5.

```
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_INIT) (
  IN  FSP_TEMP_RAM_INIT_PARAMS     *TempRamInitParamPtr
);
```

Fig. 3.5: TempRamInit API prototype [5]

2) FspInit API:

The above API is called after TempRamInit. This API mainly initializes the memory stack, the processors and the chipset to which is required to enable normal operation of these devices. This API accepts a pointer from the data structure that will be dependent on a platform and defined for each FSP binary. The boot loader mainly provides a continuation function as a input parameter when calling this API. After this API completes its execution, it will not return to the boot loader from where it was called but rather it will returns control to the boot loader by calling the same function which is passed to this API as an argument.
A prototype of this API is shown in Fig. 3.6 [5].

```
typedef
EFI_STATUS
(EFIAPI *FSP_INIT) (
  IN OUT  FSP_INIT_PARAMS       *FspInitParamPtr
);
```

Fig. 3.6: FspInit API Prototype [5]

3) NotifyPhase API:

The above API is used to notify the FSP about the all phases in the boot phases. This allows the FSP to take necessary actions as needed whenever different initialization

phases will execute. The phases will be dependent on platform and will be documented with the release of FSP. Recently FSP supports mainly two notify phases [5]:

- Post PCI enumeration
- Ready To Boot

A prototype of NotifyPhase API is as shown in Fig. 3.7 [5].

```
typedef
EFI_STATUS
(EFIAPI *FSP_NOTIFY_PHASE) (
  IN  NOTIFY_PHASE_PARAMS       *NotifyPhaseParamPtr
);
```

Fig. 3.7: NotifyPhase API Prototype [5]

4) FspMemoryInit API:

The above API is called after TempRamInit and initializes mainly the memory stack. This API accepts a pointer to a structure of data which will dependent on platform and defined for each and every FSP binary. This API initializes the memory subsystem portion, initializes the pointer to the HobListPtr, and returns from other to the boot loader from where it was called. Still the memory of system has been initialized in this API, the boot loader should be migrate its stack memory and data from memory to memory of system after this API.

A prototype of this API is shown in Fig. 3.8 [5].

```
typedef
EFI_STATUS
(EFIAPI *FSP_MEMORY_INIT) (
  IN OUT  FSP_MEMORY_INIT_PARAMS  *FspMemoryInitParamPtr
);
```

Fig. 3.8: FspMemoryInit API Prototype [5]

5) TempRamExit API:

The above API is called after FspMemoryInit. This API will release the temporary memory arranged by TempRamInit. This API accepts a pointer to a structure of data that dependent on platform and defined for each and every FSP binary. FspMemoryInit, TempRamExit as well as FspSiliconInit API provide an alternate method to finish the silicon initialization and provides boot loader an opportunity to get control whenever system memory is available and before the temporary memory is release.
A prototype of this API is shown in Fig. 3.9 [5].

```
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_EXIT) (
  IN OUT VOID           *TempRamExitParamPtr
);
```

Fig. 3.9: TempRamExit API Prototype [5]

6) FspSiliconInit API:

The above API is called after TempRamExit. FspMemoryInit, TempRamExit as well as FspSiliconInit API provide an different method to complete the silicon initialization. This API initializes the processor and the chipset which including the IO controllers in the chipset required to enable normal operation of such a devices. This API accepts

a pointer to a structure of data that dependent on platform and defined for each FSP binary.

A prototype of this API is shown in Fig. 3.10 [5].

```
typedef
EFI_STATUS
(EFIAPI *FSP_SILICON_INIT) (
  IN OUT VOID          *FspSiliconInitParamPtr
);
```

Fig. 3.10: FspSiliconInit API Prototype [5]

b) FSP 2.0:

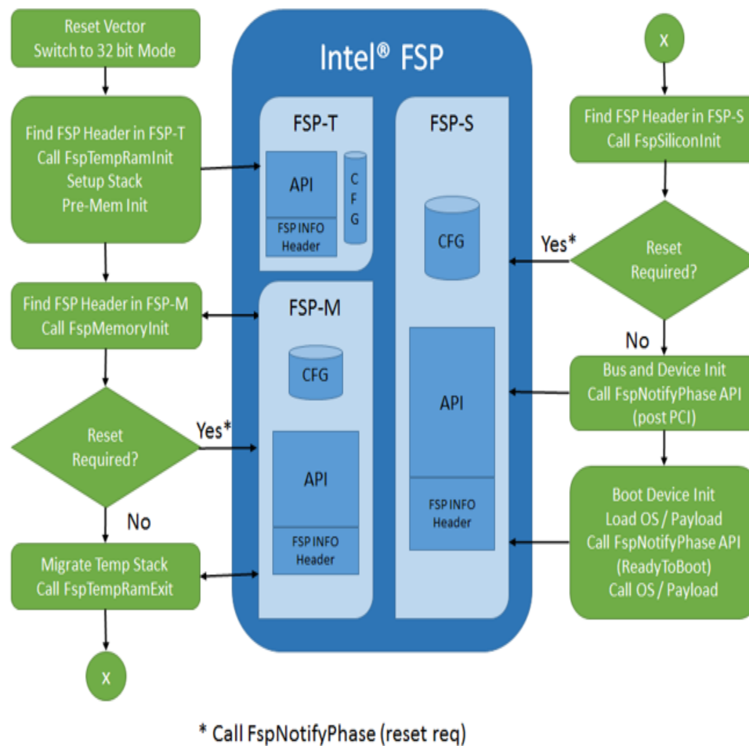


Fig. 3.11: FSP Boot Flow Version 2.0 [6]

From Fig. 3.11, It is the proposed boot flow of FSP 2.0. Here we can see that major change compared to FSP 1.0 and FSP 1.1 is it divided into main three parts called FSP-T, FSP-M and FSP-S. FSP-T contain API related to temporary memory initialization like TempRamInit API. FSP-M contain API related to main memory initialization and to come out from temporary memory like MemoryInit API and TempRamExit API. FSP- S contains information related to silicon initialization and to notify OS phase as SiliconInit API and NotifyPhase API. Forth portion is like optional which mainly used for backware compatibility with FSP 1.0 and FSP 1.1 specifications. If this component will present it comes first as first component in FSP binary. This component have an FSP_INFO_HEADER with header revision 1 or 2 and may provide the interfaces required for backward compatibility with previous FSP specifications.

Now from boot flow, when powered on first control find for TempRamInit from FSP-T and it will look for FSP_INFO_HEADER. After completion of this API control back to find FspMemoryInit API from FSP-M and control goes to this APIs header. During this phase, it checks for reset, if it required then it will come again in the same API otherwise it migrates to next API. Next to FspMemoryInit comes FspTempRamExit

API which comes in FSP-M which is used to remove temporary memory. After that FSP-S will come which have FspSiliconInit API which will initialize the silicon. After that raise a request for reset, if yes then control goes back to FspSiliconInit otherwise control goes to next phase called FspNotifyPhase API. In this API, FSP gives control to OS and check for two event called PCI enumeration and Ready to boot. Next to that payload will come and OS will boot by using boot loader [6].

Chapter 4

FSP Implementation Strategy

4.1 POST and Phase Code

The Power On Self Test is activated by the BIOS. It runs a series of checks and diagnostics on your motherboard. The objectives of the BIOS during POST are as follows [6]:

- It check for registers of CPU
- Check the unification of the BIOS with itself
- Check the different peripheral components
- Check the main memory of system
- Check and execute BIOS
- Identify, organize, and select which devices are available for booting

Actually BIOS start its POST when the reset is given to CPU. When reset will occur then first memory location which CPU tries to execute is called as reset vector. During reboot, the CPU will save this code fetch to the BIOS stored on the system flash memory. For Warm boot BIOS will come from stored location called RAM [6].

4.1.1 Why POST Code Required

The POST runs very quickly, it hardly takes 2-3 mins to boot to OS and user will normally not even noticed that its happening unless it stops in between because presence

of some faulty or some hardware is missing, When turned on the PC, it may happen system starts beeping sounds and then stopped without booting up. That is the POST telling something is wrong with the machine. Here the speaker is used because this test happens so early on, that the video is not even activated yet! Its hard to finds out where actually execution is stopped in which phase or which hardware is faulty or missing in the system.

If execution is stops in between. Some debug boards also doesnt contain serial interface which is used to take serial dumb of POST, In this situation also its really hard to find out where the execution is if its hangs in between, with help of last executed POST code its easy to debug execution is in which phase. So, by this code one can easily determine the problem by using board and we not have to give much effort for finding any error during booting process. POST code itself will tell the error for code [6].

4.1.2 How POST Status Code Works

POST look out the information present by displaying a number to the port 80 (a screen display was not possible with some failure modes). Both the number mainly one is progress indication and second is error codes were generated. If suppose failure is occurred then it will not generate a code, at that time code which was available on last POST will come to aid in diagnosing the problem. There are add-on cards also available that can be placed in to PCI slot and on which post codes can be seen else now a days with this comes inbuilt with debug board. BIOS POST code is shown in Fig. 4.1 [6].

Fig. 4.1: BIOS POST Code in Board [6]

4.1.3 Types of POST Code

POST status codes are mainly divided in to following categories:

- Debug codes
- Error codes
- Progress codes

Firstly for debug codes required for operations related to the basic nature of the information about debug. Second one is error codes which is required for operations related to exception conditions. Next one comes Progress codes which is for operations related to activities of the component classification. The values 0x000x0FFF are in general operations that are common by all subclasses in a class. After class, there are also subclass-specific operations. Out of the all subclass-specific operations, the values 0x10000x7FFF are occupied by this specification. The remaining values (0x80000xFFFF) are not defined by any of this specification and this value can assign for that range by OEMs. The merging of class and subclass operations provides the full set of operations that is given by an entity.

Table 4.1 gives the class for POST code as follows:

1)Hardware	1)Computing Unit	1)EFI_COMPUTING_UNIT
	2)User accessible peripheral	2)EFI_PERIPHERAL
	3)I/O bus	3)EFI_IO_BUS
2)Software	1)Host Software	1)EFI_SOFTWARE

Table 4.1: POST Code Classes [6]

4.1.4 Implementation

Each postcode is falls into specific subclass and class, each class has its unique value which is predefined. There are four main classes as follows:

- Software Host Software
- I/O Bus
- User-Accessible Peripherals
- Computing Unit

For Progress code,
Progress code for execution reached to DXE phase.
PEI_CPU_AP_INIT, 0x35 ,
This status falls into computing unit class and host processor subclass.
#define PEI_CPU_AP_INIT
(EFI_COMPUTING_UNIT_HOST_PROCESSOR EFI_CU_HP_PC_AP_INIT)
For this status code function call is:
REPORT_STATUS_ CODE
(EFI_PROGRESS_CODE,
EFI_COMPUTING_UNIT_HOST_PROCESSOR EFI_CU_HP_PC_AP_INIT)
Which will show value 35 on the status codes display.
For error code,
Error code for memory is installed or not:
PEI_MEMORY_NOT_INSTALLED, 0x55 ,
This status code is falls into software class and PEI foundation subclass.
#define PEI_MEMORY_NOT_INSTALLED
(EFI_SOFTWARE_PEI_SERVICE EFI_SW_PEI_CORE_EC_MEMORY_NOT_ INSTALLED)
For this status code function call is:

REPORT_STATUS_CODE

(EFI_ERROR_CODE,

EFI_SOFTWARE_PEI_SERVICE_EFI_SW_PEI_CORE_EC_MEMORY_NOT_INSTALLED

)

4.1.5 Phase Code

As we show that POST code is required for BIOS, when BIOS gives some error during booting at the same method in FSP when particular code will give error or when FSP will hang at that time it is quite difficult to find where system stucked because there are many API and its phases where system may hang and it will not boot. So, Phase code is such a utility by which we can find very easily that at which point system hang.
When Phase code started applying at that time we found that higher nibble is free during BIOS POST code except Memory Reference Code which will take complete 16 bit as DD00 to DD7F. So, we planned to put this into higher nibble API wise so that it will combine for both POST code and Phase code.

4.1.6 Implementation

When Phase code started to implement at that time first of all they give one proposal which is used to set phase code along with the POST code as shown in Fig. 4.2.

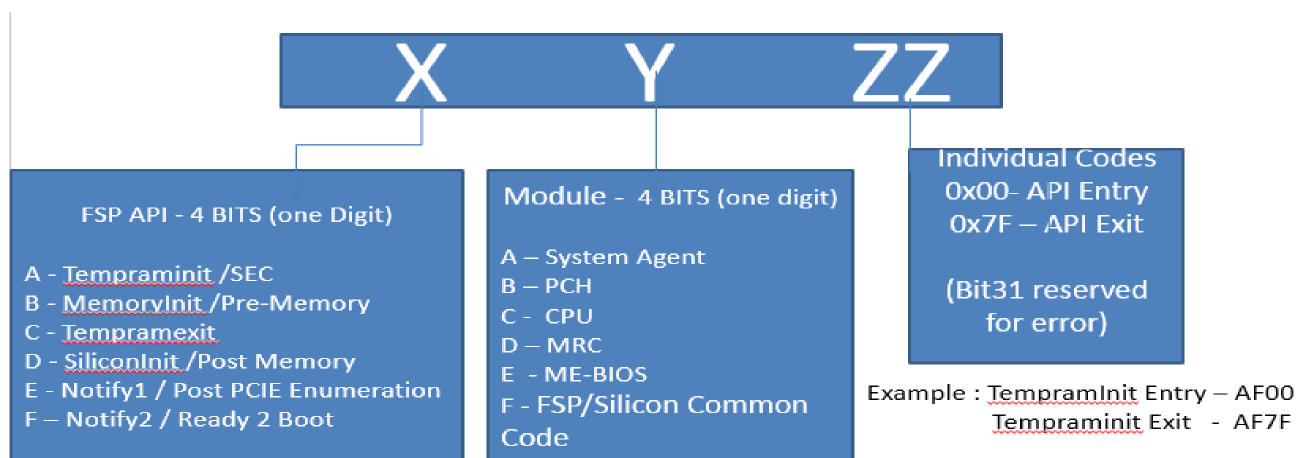


Fig. 4.2: Proposed Method for Phase Code

As Shown in Fig. 4.2, ZZ is lower two nibbles whereas X is higher nibble. For FSP, higher nibble is used for phase code and other bits are kept as it is for POST code. Mainly last two nibbles are used for API entry and exit to get the idea about which API is running. The best point about this proposal was all APIs are sequential so that POST code will work fine but the main drawback is MRC have DD00 to DD7F but according to this proposal it should be BD00 to BD7F so, we have to change this either in API or we have to change this into MRC code.

For above reason second proposal comes into frame for implementation as shown in Fig. 4.3.

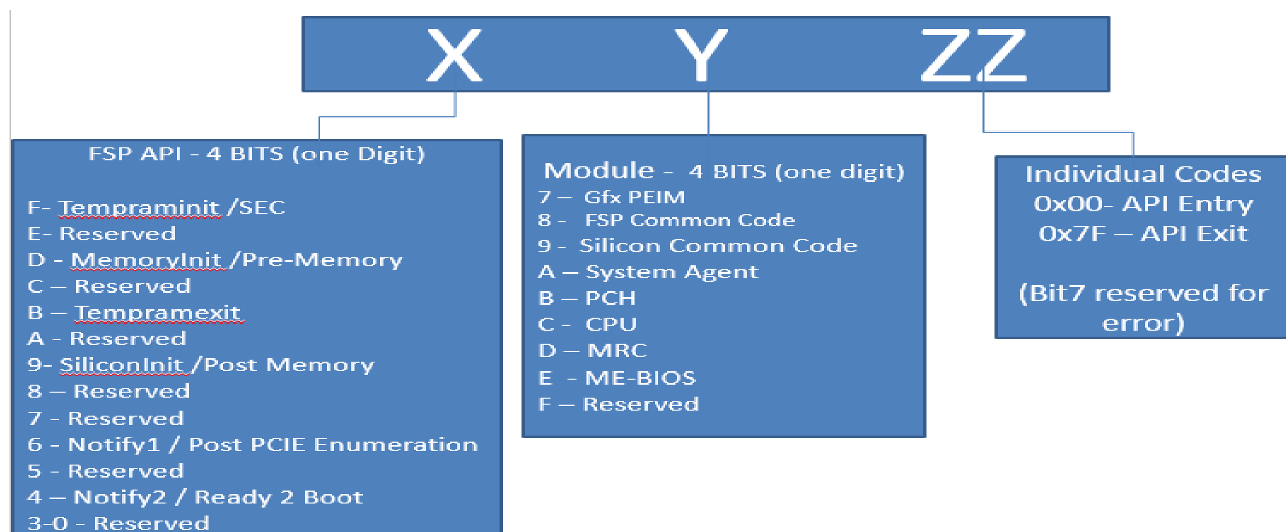


Fig. 4.3: Finalized Implementation of Phase Code

As shown in Fig. 4.3, It looks like same as previous proposal but here for FSP, higher nibble started in reverse order means first API we can start from F like that sequence wise it follows all APIs. By this method the problem of MRC also solved and Phase code and POST code we can put together. So after that all the APIs started from TempRamInit to NotifyPhase API will follow reverse hex numbers.

4.2 FSP Separation from Source Code

As we know complete BIOS required all the Phases like SEC, PEI, DXE and BDS to execute the complete BIOS. At every step BIOS will dispatch the required driver and

it will give control to next phase. Again next phase will dispatch the necessary drivers and like that complete cycle goes on. This four phases are very important to execute the regular BIOS.

After that Intel gave optimization in their code and found FSP where it will give just binary to the end user to experience flexibility. Because it is using only some drivers during execution it not required the all folders of regular BIOS as shown in Fig. 4.5.

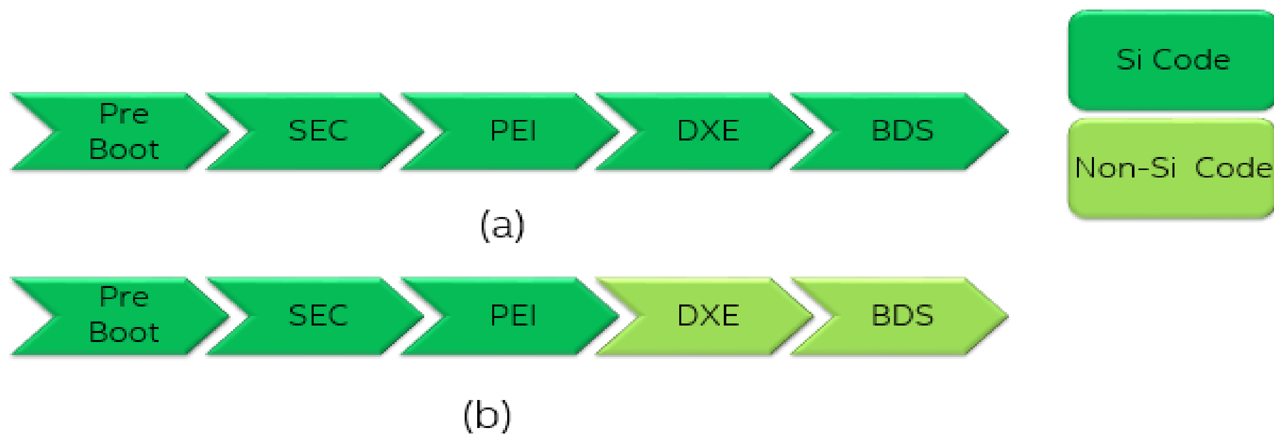


Fig. 4.4: (a) Regular BIOS Execution (b) FSP Execution

Fig. 4.4 (a) shows phase execution for regular BIOS where green color shows that it required all the phases mainly green color is for Si package required to execute. Now shown in Fig. 4.4 (b), where only first three phase pre boot, SEC and PEI are required to execute the FSP rest of the phase are not required. Here show that in FSP, only first three phases are come under Si package, rest of the phases are non-silicon package.

So phases which are not required then corresponding drivers are also not required of that phase in FSP. If we put that driver as it is then it will not do anything but unnecessarily it occupies space in reference code. So it is better to remove that driver from reference code.

For removing unnecessary driver, python script is requires for Skylake, Kabylake and Broxton where after running this script all the unnecessary drivers removed from the source code and only useful drivers are there in Silicon package for FSP.

4.2.1 Flow Chart of Script

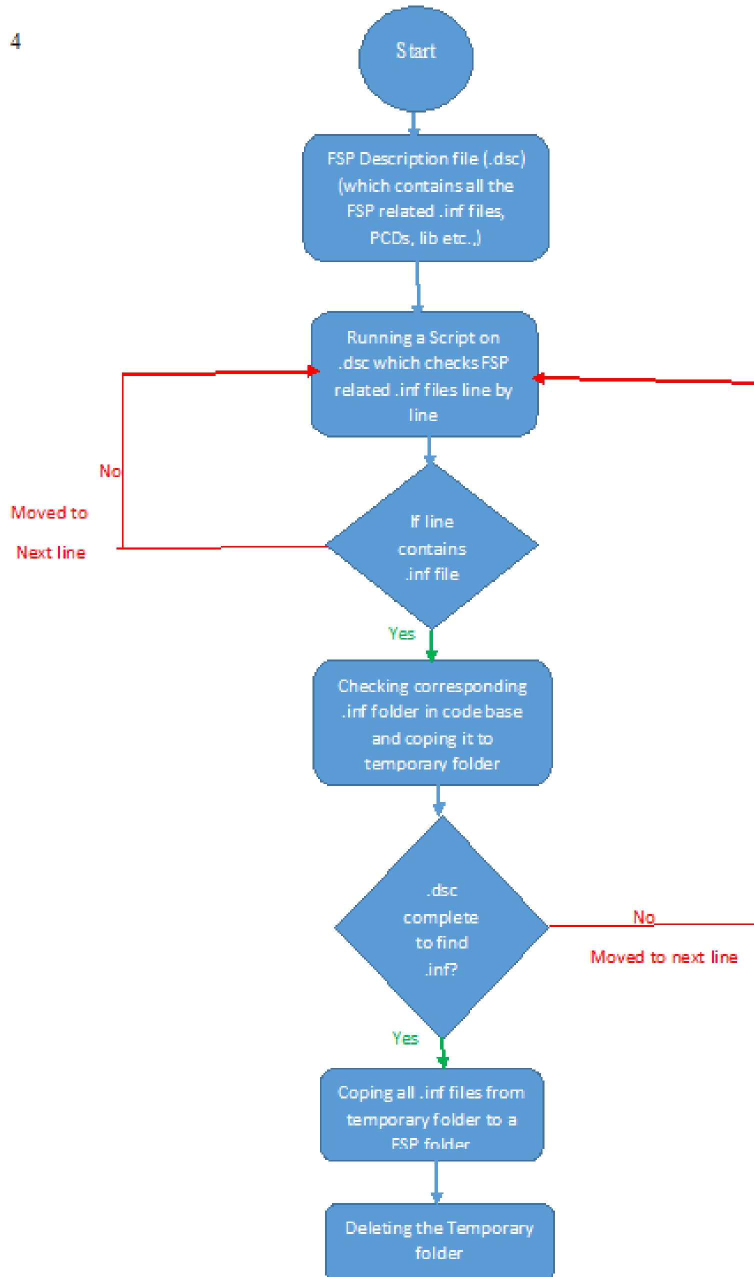


Fig. 4.5: Flow Chart of Script

As shown in flow chart, it is clear that for that script we have to find a file which gives all the information related to FSP drivers. In regular BIOS FspPkg.dsc file will give the information regarding all the FSP drivers which are there in silicon package. So, if we want to find the drivers then our script should run in FspPkg.dsc. After that for drivers regular BIOS will contain .inf extension. So inside FspPkg.dsc we have to search .inf

files and if we detect that file then we want location of that file. After extracting that .inf file, It was found location also for that file and saved in one temporary folder which is created by script. For every .inf file, it was happened same and put in respective temporary folder and saved there at last when all the .inf will complete then we have to put this files in original folder and delete the temporary folder.
We have to put this python script inside in the FspPkg of reference code. Where it will execute the script and optimization the code.

After running this python script,

- Sky lake silicon code became 24.4 MB from 69.7 MB
- Kaby lake silicon code became 26.5 MB from 72.5 MB
- Broxton silicon code became 117 MB from 216 MB

As shown in Fig. 4.6, where it shows that silicon folder before running this python script and after running the python script. There is huge amount of file is not required to build the FSP are deleted after running this python file so it occupies very less space as well as it boot very fastly compared to regular BIOS.

Fig. 4.6: Silicon Folder before Python Script

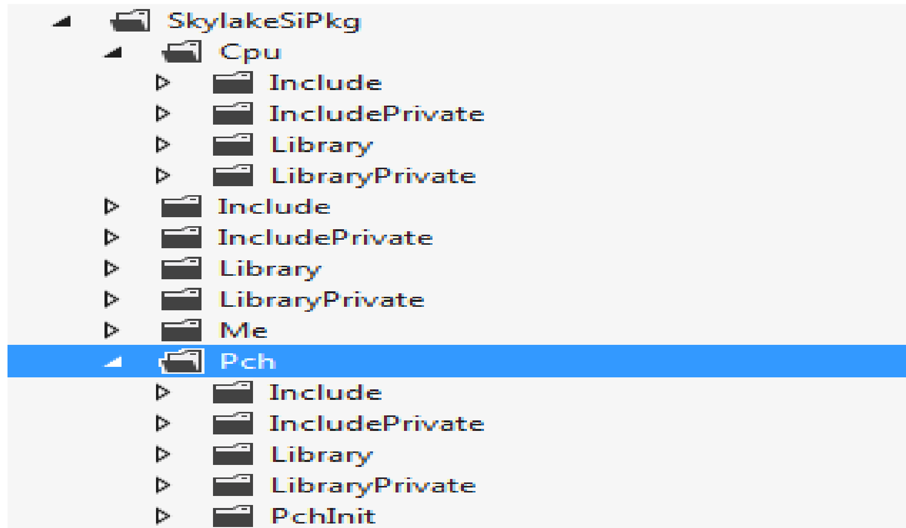


Fig. 4.7: Silicon Folder after Python Script

As shown in Fig. 4.7, it is very clear that what python code did here inside Cpu folder only it is deleted more than 3 folders and it will same case for all the folders inside the Silicon folder so that it will allow to FSP to work fastly.

Like that there are many packages in EDK II platform for building as mentioned below in Table 4.2.

Sr.	Package	Details
1	Base tool	This will give tools related to build for both EDK and EDK2. It gives miscellaneous tools such as ECC and EOT
2	EdkCompatibilityPkg	It gives security features were introduced (e.g. Authenticated Variable Service, Driver Signing, etc.)
3	EdkShellBinPkg	It gives header les and libraries which enable build the EDK module in UEFI 2.0 mode with EDK II Build
4	EmulationIntelRestrictedPkg	Binary images of the EFI Shell 1.0 for IA32, X64 and IPF
5	FatBinPkg	This package provides binary device drivers to support the FAT32 le system
6	FatPkg	This Platform le is used to generate the Binary Fat Drivers
7	IA32FamilyCpuPkg	This package supports IA32 family processors,with CPU DXE module, CPU PEIM, CPU S3 module,SMM modules, related libraries, and corresponding definitions
8	IntelFrameworkModulePkg	contain definition and module for EFI framework
9	IntelFrameworkPkg	This package provides definitions and libraries that comply to Intel Framework Specifications
10	MdeModulePkg	This package provides the modules that conform to UEFI Industry standards.
11	Mde Pkg	provide definition for MDE specification
12	Network Pkg	It gives IPv6 network drivers, PXE driver, IPsec driver, iSCSI driver and necessary applications for shell.
13	PcAtChipsetPkg	Designed to follow PcAt defacto standard
14	PerformancePkg	provide performance measurement
15	SecurityPkg	Provides TPM, secureboot , UID
16	ShellBinPkg	Provide Binary shell application which follow UEFI
17	ShellPkg	provide definition for EFI and UEFI shell
18	SourceLevelDebugPkg	provide target side modules.
19	UefiCpuPkg	provides UEFI compatible CPU modules.

Table 4.2: EDK II Reference Code Packages [4]

4.3 FSP 2.0

Before FSP 2.0, Intel was using FSP 1.1 for implementation of FSP in reference code. In FSP 1.1 Intel was used in general two firmware volume called memory and silicon. In Memory FV, all the APIs related to the memory will added. In Silicon FV, all the APIs related to the silicon will added. In this version memory related APIs are combined so it will be temporary memory as well as main memory both will include so it

was not giving good performance [4].

So for more optimization, Intel give proposal of FSP 2.0 where it give enhanced FV by separating all APIs from respective FVs. By using this, it gave excellent performance and gave better solution to the customers too [4].

Here one can see the difference between FV structure for FSP 1.1 and FSP 2.0. in Fig. 4.8.

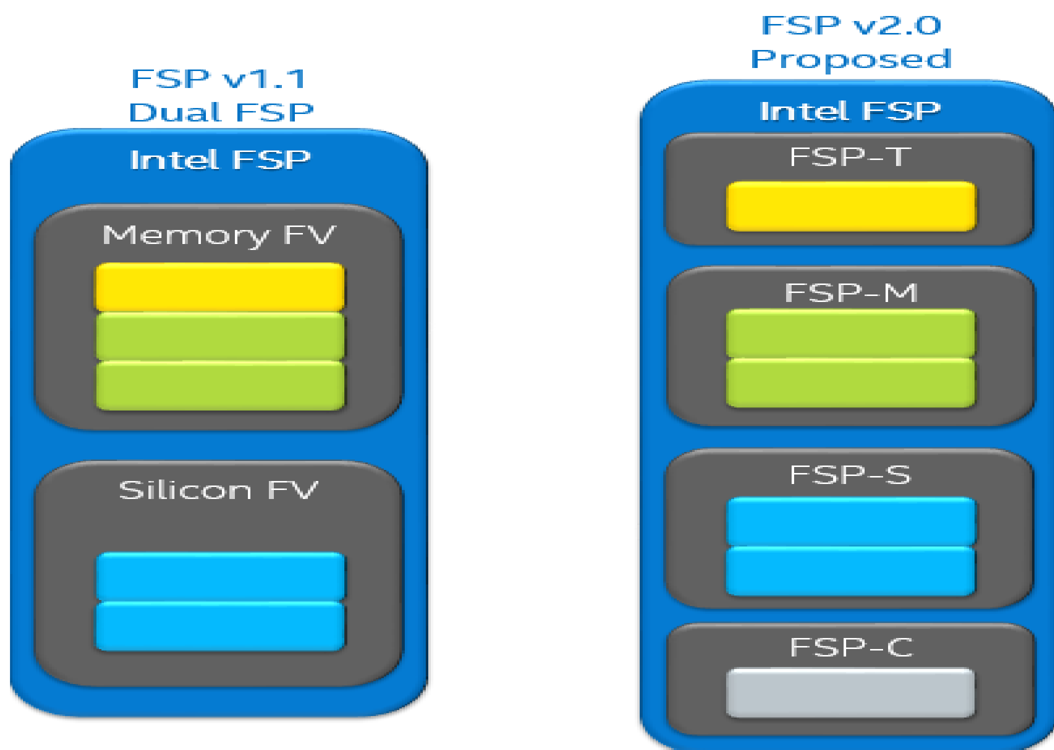


Fig. 4.8: Firmware Volume for (a)FSP 1.1 (b) FSP 2.0 [5]

In Fig. 4.8 (a), there are two firmware volume defined where yellow API is TempRamInit API, both green are FSPMemoryInit API and TempRamExit API. Last both blue APIs are FspSiliconInit API and NotifyPhase API. But shown in Fig. 4.8 (b), it is proposed diagram of FSP 2.0 where TempRamInit is comes under FSP-T. FSPMemoryInit and TempRamExit comes under FSP-M. FspSiliconInit and NotifyPhase comes under FSP-S. The last FV is added in FSP 2.0 which is compability component. If it is present in FSP 2.0 then it will execute first in the FSP binary. This component will have an FSP_INFO_HEADER with header revision 1 and 2 and may interface required for backward compatibility with previous versions [5].

As per FSP 1.1, FSP 2.2 have following changes did inside the flow.

- Updated FSP Binary format with FSP component information T, M, S, C
- Updated FSP Information Tables with FSP component identification
- Removed VPD configuration data and updated UPD configuration data
- Updated Boot Flow
- Added Reset Request status return types

So after implemented FSP 2.0 it is coming in reference code as shown in Fig. 4.9 as follows:

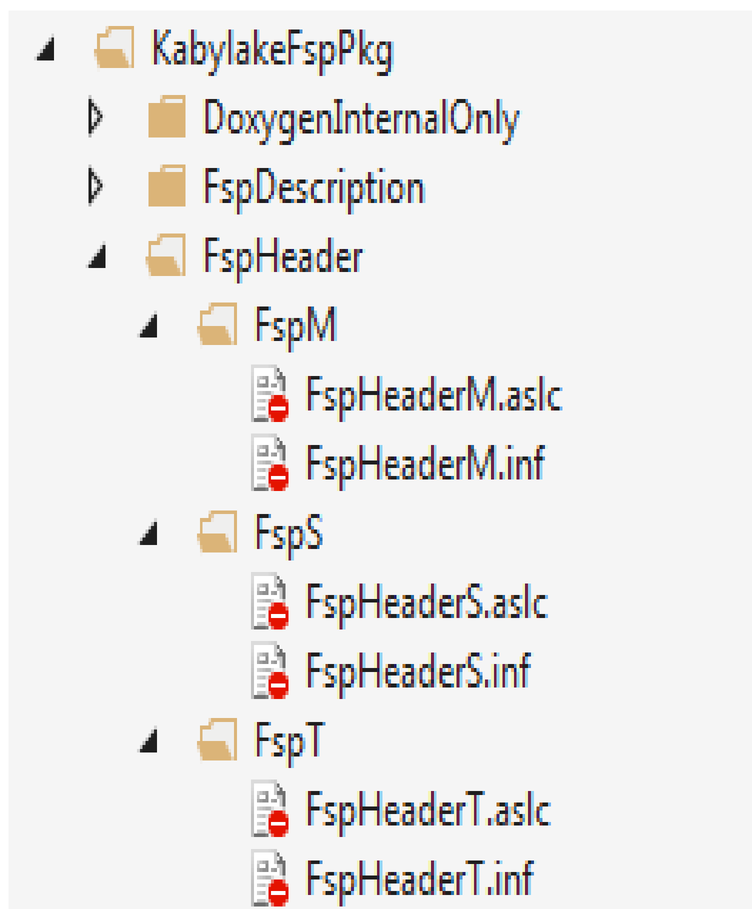


Fig. 4.9: FSP 2.0 Implementation [5]

4.4 Firmware Configuration for FSP

At the time of initialization of FSP, FSP needs some data to configure particular feature at the run time. So to support this need, it required some data region where we can keep this data because it will update at the run time and FSP will take the data from this data region.FSP will do this configuration for platform and based on the input given by user it will take the data for platform configuration and changed the value from data in configuration region. So when FSP will build the source code it will take the configuration data from this region and by this way it will get the latest platfrom configuration [6].
The data region which saves the configuration contains two sets called VPD(Vital Product Data) and UPD(Updatable Product Data). This set of regions are divided based on it's type of configuration changing for data. VPD is region where we can save the configuration data statically only means during the time of boot, boot loader can't change the data at this region. So if some user want platform configuration changes at the run time then this region will not work. UPD is a region working exactly opposite to the VPD where we can save the data dynamically and when boot loader want to change the data at the time of boot, then it can easily change the data at run time [6].
For changing the data statically inside the VPD and UPD, there is one separately customized tool called Binary Configuration Tool(BCT). In general BCT is a one utility mainly used to change all the settings of configuration resides in binary file. This enables the users to use the BCT which will customize the FSP configuration from Intel's FSP binary [6].

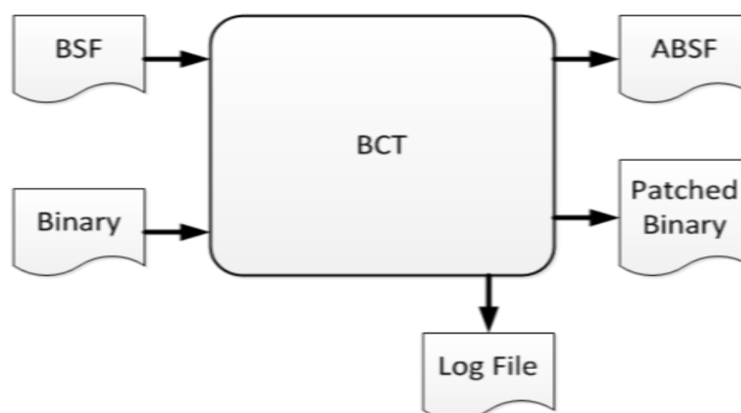


Fig. 4.10: High Level Diagram of BCT [6]

As shown in Fig. 4.10, it is giving basic understading of how BCT tool is working. BCT tool is first read the Boot Setting File(BSF) which provides a graphical interface for manipulating a setting of FSP binary. Addition to BSF, BCT can take the binary with modified configuration settings. At the output side BCT gives augmented BSF(ABSF) as well as it will give the patched binary. Augmented BSF contain the information about the setting of configuration to use the patch binary. Apart from this, BCT also gives the log file at the output side which will give the information about which configuration setting used in BSF so that it can work as feedback for next setting [6].

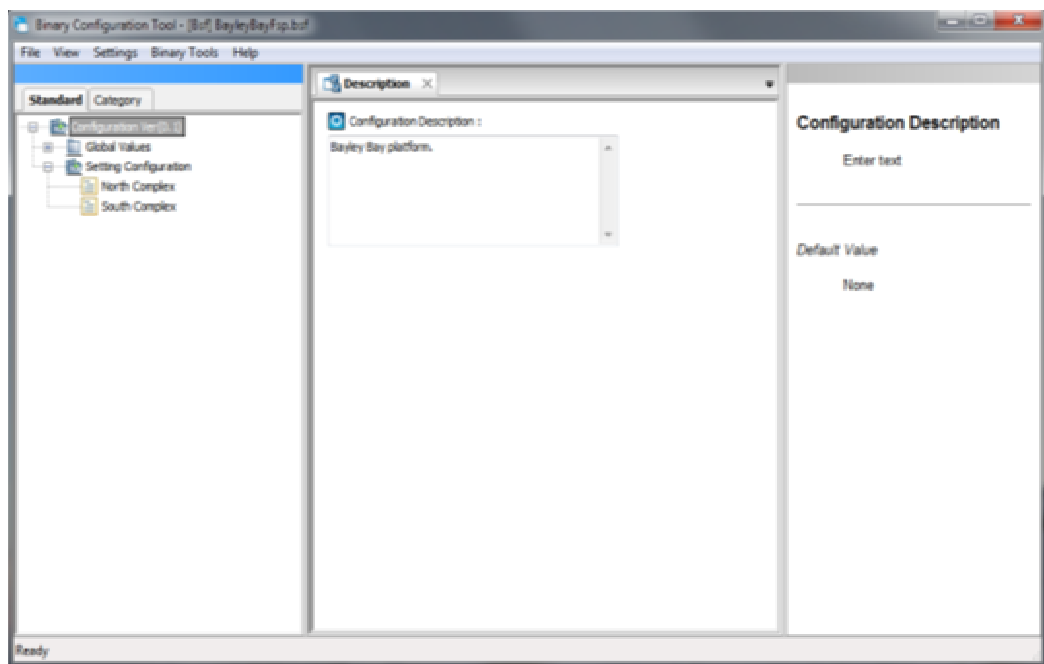


Fig. 4.11: GUI of BCT [6]

From Fig. 4.11, it is GUI for BCT. In this tool left pane is navigation pane which will be use to traverse the configuration settings tree under the standard or category tab. Right pane is help pane which will change as configuration settings by floating mouse sursor. Center pane is configuration pane which display all the settings of configuration. For Modifying the binary file, select File option and choose the file from open BSF option. After load the menu item select the file. After modified the configuration settings, patch the binary associated with BSF by select the patch under Binary tools menu [6]. Apart from static configuration, UPD data can also be modified by boot loader dynamically. For this UPD needs to be organized as a structure.

```
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_INIT) (
  IN   VOID        *FsptUpdDataPtr
);

typedef
EFI_STATUS
(EFIAPI *FSP_MEMORY_INIT) (
  IN   VOID        *FspmUpdDataPtr
  OUT  VOID        **HobListPtr;
);
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_EXIT) (
  IN  VOID             *TempRamExitParamPtr
);
typedef
EFI_STATUS
(EFIAPI *FSP_SILICON_INIT) (
  IN  VOID            *FspsUpdDataPtr
);
```

Fig. 4.12: API called to UPD [6]

As shown in Fig. 4.12, where every APIs contains the UpdDataPtr as an argument which will be initialized to point UPD. Whenever it will give NULL value then FSP will use built in UPD for FSP binary. If boot loader overrides the UPD parameters, then it will copy UPD from flash to memory and now FSP binary will initialize with this data structure [6].

- MemoryInitUpd - Add definition between these two tags

!HDR EMBED:{MEMORY_INIT_UPD:MemoryInitUpd:START}
gBroxtonFspPkgTokenSpaceGuid.DdrFreqLimit | 0x0031 | 0x01 | 0x0
!HDR EMBED:{MEMORY_INIT_UPD:MemoryInitUpd:END}

Offset
Size in Bytes
Default

- SiliconInitUpd

!HDR EMBED:{SILICON_INIT_UPD:SiliconInitUpd:START}
XXXXXXXXXXX
!HDR EMBED:{SILICON_INIT_UPD:SiliconInitUpd:END}

Fig. 4.13: Definition of UPD

If we want to use this UPD then first we have to define this UPD in FSP file which

is shown in Fig. 4.13. For every APIs there is different UPDs which need to define. In this definition it contain three fields which are offset, total number of bytes and the dafault value of this UPD.

4.5 Responsiveness Infrastructure of FSP

4.5.1 What is Responsiveness?

Responsiveness is the measurement of the time taken by particular executable. In FSP there are many drivers which will execute at the time of boot according to it's sequence. In FSP, this drivers are executing based on flow like in PEI phase, drivers which are relevant to PEI phase will execute same case will happen with the other phases as well. As we discussed earlier that in FSP 2.0 there are mainly four firmware volumes called FSP-T, FSP-M, FSP-S and FSP-C. So whenever particular FV will execute then all the drivers under this FV will execute. So we need the timing of this all drivers for different FV's of FSP which is called Responsiveness of FSP.

4.5.2 Why Responsiveness?

Once one has to get the timing of all the drivers of FSP for all FV's then we can calculate the total timing taken by FSP drivers which is needed for calculating total execution time taken by FSP. Apart from this by using responsiveness we can look the timing of drivers so that we can judge the performance of particular drivers and if by chance any driver give wrong timing then we can check that why it is giving that timing. If we have the execution timing of all the drivers then we can reduce the total execution timing of FSP by changing some drivers code.

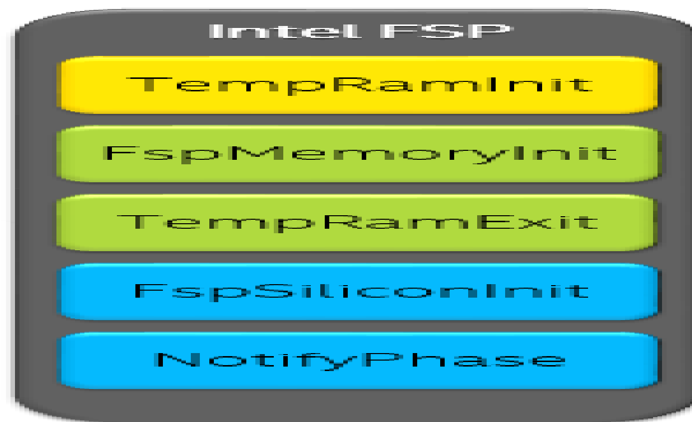


Fig. 4.14: FV's of FSP for responsiveness [7]

Fig. 4.14 shows different APIs in different FVs. TempRamInit API is executing in FSP-T. FspMemoryInit and TempRamExit are executing in FSP-M and FspSiliconInit and NotifyPhase are executing in FSP-S. So whenever particular APIs are executing in sequence then based on that drivers will execute and we have to collect the timing while execution.

4.5.3 Implementation

Responsiveness infrastructure can make by FSP binary and FSP wrapper. There are separate drivers for FSP binary and FSP wrapper. So for getting the response time for FSP wrapper, we need to save the execution time of all the drivers from platform package. For FSP binary, we need to change the drivers from FSP package. FSP binary and FSP wrapper are in sync at the time of execution and it will execute the drivers in sequence one after another [7].

Once all drivers complete its execution, we gets the response time of all drivers which we need to store somewhere to execute at the time of booting. The area of storage for this response time is called Firmware Performance Data Table(FPDT). FPDT is the type of table which provides complete information about the platform initialization performance records. This all information gives the data of performance during boot for specific work within firmware. The FPDT contains only those data that is necessary for boot process of every platform as follows [7]:

- Value of timer at the beginning of BIOS

- Switch the control to boot loader

Value of timer express in increase of 1 nanosecond. So suppose if any driver indicates the 25678 as execution time then it means it taken 25.678 microsecond to execute. For capturing the log at the time of boot, we are using one tool called intel system scope tool. This tool we need to install to os first afterwards when the FPDT log is save then during the time of boot whenever we reach to os we need to open the tool and just has to save the FPDT log. This log contains all the data which is saved under FPDT region [7].

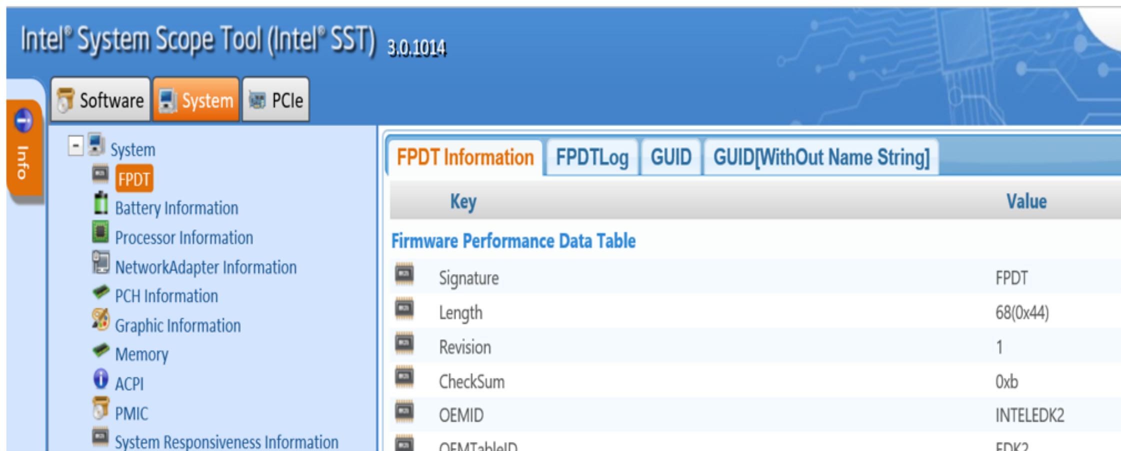


Fig. 4.15: Intel's System Scope Tool


```
    Token="0x4000-0x407f" NameString="" GUID="c912229556ed4d4e06a6508d894d3e40"/>
<Item DeltaTime="9932268" EndTime="1157078064" StartTime="1147145796" Token="Module
    Start" NameString="" GUID="9b3ada4fae564c248deaf03b7558ae50"/>
<Item DeltaTime="251308197" EndTime="1415589132" StartTime="1164280935"
    Token="Module Start" NameString="" GUID="9e1cc850673148488752667 3c7005eee"/>
<Item DeltaTime="312342197" EndTime="1735043406" StartTime="1422701209"
    Token="Module Start" NameString="" GUID="a8499e65a6f648b096db45c266030d83"/>
<Item DeltaTime="-151549068" EndTime="216925737" StartTime="368474805"
    Token="0x5021-0x5030" NameString="" GUID="a8499e65a6f648b096db45c266030d83"/>
<Item DeltaTime="-1891598914" EndTime="1586191541" StartTime="-817176841"
    Token="0x5031-0x2" NameString="" GUID="a8499e65a6f648b096db45c266030d83"/>
<Item DeltaTime="481894766" EndTime="2074999755" StartTime="1593104989"
    Token="Module Start" NameString="" GUID="3381fc3f879141e58871a960a4ed24b7"/>
<Item DeltaTime="107556" EndTime="2132914099" StartTime="2132806543" Token="0x21-
    0x40" NameString="" GUID="52c05b140b98496cbc3b04b50211d680"/>
<Item DeltaTime="5952991" EndTime="-2117616823" StartTime="-2123569814" Token="0x30-
    0x31" NameString="" GUID="52c05b140b98496cbc3b04b50211d680"/>
<Item DeltaTime="12287595" EndTime="-2096570014" StartTime="-2108857609"
    Token="Module Start" NameString="" GUID="9b3ada4fae564c248deaf03b7558ae50"/>
<Item DeltaTime="2128091978" EndTime="48830868" StartTime="-2079261110"
    Token="Module Start" NameString="" GUID="9e1cc850673148488752667 3c7005eee"/>
<Item DeltaTime="74870" EndTime="69881309" StartTime="69806439" Token="0xf000-0xf07f"
    NameString="" GUID="56ed21b6ba23429e8932376d8e182ee3"/>
<Item DeltaTime="62233969" EndTime="70028255" StartTime="7794286" Token="0xd000-
    0xd07f" NameString="" GUID="56ed21b6ba23429e8932376d8e182ee3"/>
<Item DeltaTime="260657150" EndTime="715530965" StartTime="454873815" Token="0xb000-
    0xb07f" NameString="" GUID="56ed21b6ba23429e8932376d8e182ee3"/>
```


Fig. 4.16: Result of Responsiveness

As shown in Fig. 4.16, it is showing one of the driver's execution time implementation. In that PERF_START_EX and PERF_END_EX are the macros used to calcute the

start and end timing. Fields of this macro gives the records address and GUID respectively to store properly. Figure 4.19 shows the timing of that driver in intel system scope tool where one can see that particular driver is taking approximately 6021 nanoseconds time for execution.

4.6 eMMC Boot Flow for FSP

4.6.1 Overview

After developing the FSP, when it comes at the side of booting, FSP can boot by using SSD or by using eMMC. This devices are used to fetch the OS for boting. eMMC is a multimedia card use for solid state storage. In general eMMC is used to store the data for portable device. By using MMC reader, customers can read the data by just attaching the MMC in slot. Now a days Embedded MMC(eMMC) is mainly used in a industry for storage in low cost devices. eMMC is not that much sophisticated for speed and features, so it is not find in any typical desktop and laptop computers. eMMC doesn't have firmware inside chips, multiple flash memory chips, fast interfaces and faster hardware inside it.

eMMC is low cost flash memory that is inside the phone or low end PCs to work as host for bootable device. It is quite cheaper than solid state storage called solid state drive(SSD). Solid state drive has controller which also have the firmware which support advance features like it can do read and write operation for overall memory chips in SSD, by this way it can increase the operational speed. There are multiple chips in SSD unlike the eMMC so that whenever we will write or read something in SSD it will work so faster than eMMC.

4.6.2 Implementation

At the time of eMMC booting, most of the platform firmware are stored in eMMC. While booting from SSD, platform firmware will fetch by SPI through serially but in eMMC it is happening by different way. eMMC is connected externally to the board, direct access of eMMC is not possible so for accessing the eMMC data we need to follow other procedure. As platform firmware is stored in eMMC, if we want to boot the OS from eMMC then we need to take the firmware from eMMC. For taking the firmware from eMMC, CPU will put CSE as inter mediator to fetch the data from eMMC as CPU

can't fetch the firmware directly.

When the system power on, the CSE will first fetch the block of code from eMMC and this code will store into SRAM where CPU can directly fetch this block. This SRAM will be saved it's data at the top of CPU's 4GB address space but there are many blocks of memory needs to save at that address space, SRAM is limited in size. So once CPU will start it's execution from SRAM then turn by turn CSE will transfer the complete block of code from eMMC to SRAM.

The firmware boot flow is done in three stages:

- Execute from SRAM
- Execute from CAR
- Execute from Memory

First block of code will store in SRAM but after sometime once sequence will complete data will be remove from SRAM and when temporary RAM need to implement at that time CAR will use and FSP-M will execute from CAR. Once temporary memory needs to execute and main memory will come at that time firmware will switch into memory from CAR. This switching will happen by boot loader. Once boot loader will switch the code into main memory after that it will initialize the silicon and it will boot to OS.

Chapter 5

Automated BIOS generation using System RDL

5.1 Overview

This is another project intel is driving to automate the BIOS code generation. With development of FSP project, This project is also going parallel with FSP development. As a part of this project, i have to create a BIOS code by just using RDL file and not by manually developing BIOS.
SystemRDL is kind of language which can be use to design as well as development of digital system which is complex otherwise. By using systemRDL anyone can deliver the intellectual property (IP) products used in designs. Specifically systemRDL layout support the cycle of registers from specification, verification, generation to maintenance and documentation. Here from specification registers can be arrays and memories. For systemRDL the source is description of registers from software, hardware and documentation. At the output side it will generate RTL documentation.
In today's scenario, BIOS code developers are developing the BIOS code manually after referring the BIOS specs and RTL or XML files which contains every details of registers added in latest flow. Apart from this files developers needs to refer C specs and HAS document for any change in features. This all files change platform by platform so developers need to check every time when they will develop the BIOS for different platforms. So now developers need to trace the changes for new designs and have to make changes accordingly but as it becomes time consuming and error prone which is

increase the overall system bug [].

Now if we can work on automation of generation process of BIOS, then we can solve the many problems as developers are facing during the time of development and it will become faster process from development side and it will result into Time To Market product. So basically it gave the following advantages by automated BIOS generation process as follows:

- Decrease the developement effort from silicon code
- Gave the standard format of BIOS
- Presilicon time frame utilize into design of firmware

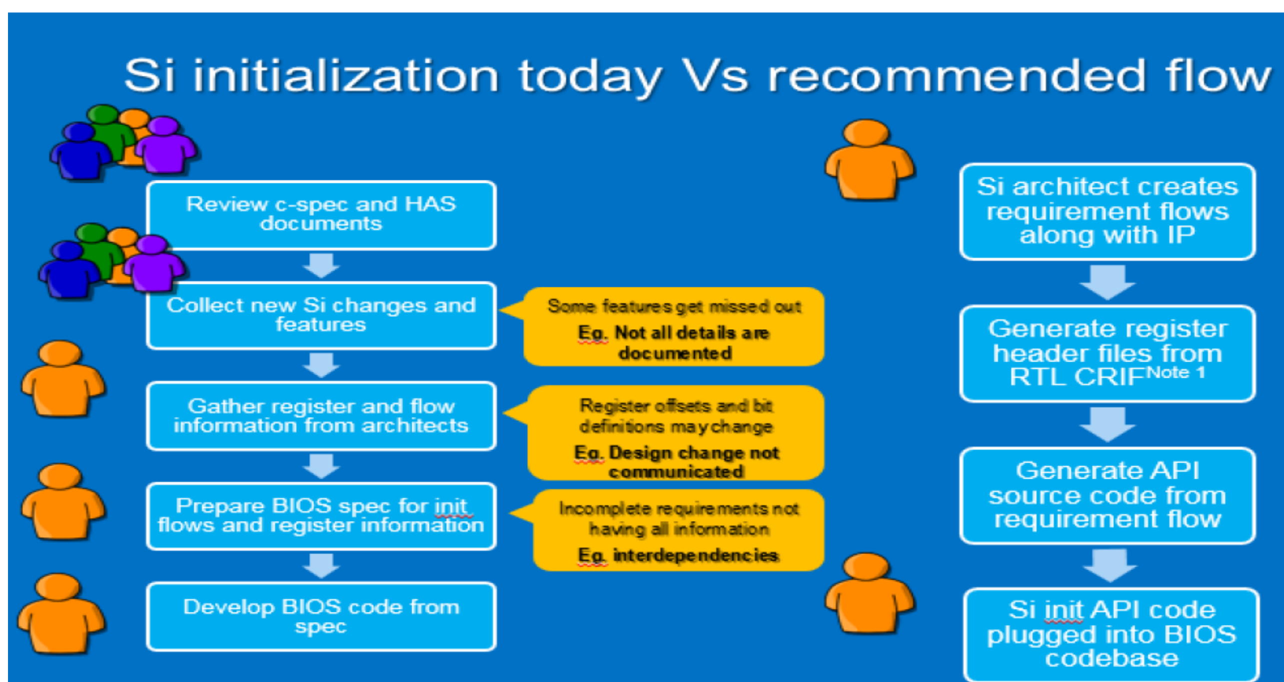


Fig. 5.1: High Level Diagram of Proposed Solution[11]

As shown in Fig. 5.1, which is giving idea about the proposed solution of today's problem. Left one is the flow we are following today in which first we have to review C specs and HAS documents to collect the changes the data at the silicon level features. Many times new features at the silicon level is not mention properly then some feature will miss out to implement. Now we have to collect the data at the register level from architects to make one flow for BIOS but sometime bit level change not communicates properly then design will not work properly. Now from this all information BIOS specs

will prepare for register information for flow initialization which will give interdependence as from BIOS specs code will develop. So finally all process will become error prone [8].

For ovecome this problem another proposed flow come into picture where we can remove this interdependence. In this flow silicon architects create the flow for specific IP in a format of XML files which gives the information at design level. Now from this RTL crif files we will generate the header files which will give the data about the registers for that IP. From this flow we will generate the source file along with header file. After at the last phase we will integrate that header file and source file into main BIOS code base [8].

5.2 Firmware Developement Model

For automate the BIOS from SystemRDL we need to follow specific model which will be different from ordinary flow to make the development automated as shown in Fig 5.2:

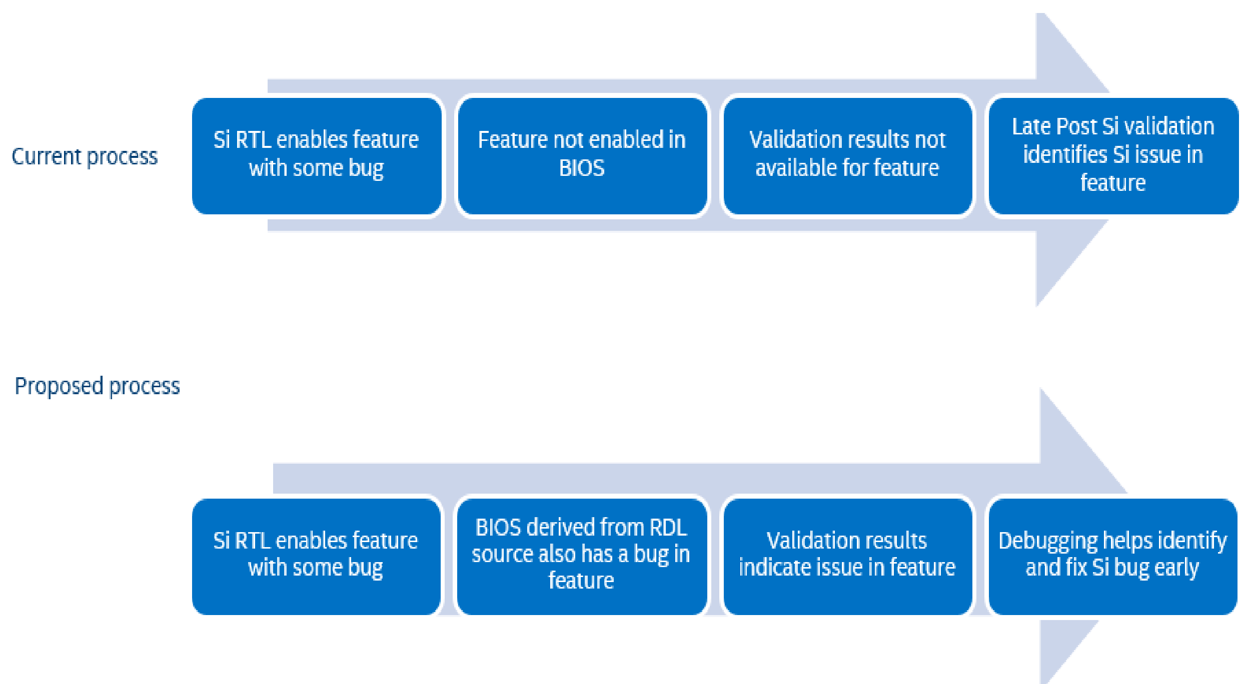


Fig. 5.2: Firmware Development Model [11]

As Shown in Fig. 5.2, which is showing two developement process namely one which we are following as of now and the second process is what we are implementing

to automate the BIOS and give the early solution to the customers. Here at the first stage both are following the same process as first we need to enable the feature from silicon RTL documentation with some bug if created. Now in current process, designer will not enable that feature as it has some bug but we are deriving our BIOS files from RTL documentation we will enable that feature with bug [11].

As designer not allowed to enable the feature so at the time of pre validation we will not have any result about that feature. In proposed process, at the time of pre validation will have some result of that feature which will also indicate the issue related to that bug for feature. Currently after pre validation we are enabling the feature and so we get the idea about that issue at post validation but at the time of our model we already get the idea about the bug at pre validation we solved that issue in pre validation and by so we can reach to customers very early [11].

5.3 System RDL Usage Model

BIOS automation will done by RDL file which will provide the sufficient information about the flow of BIOS which is shown in Fig. 5.2 as follows:

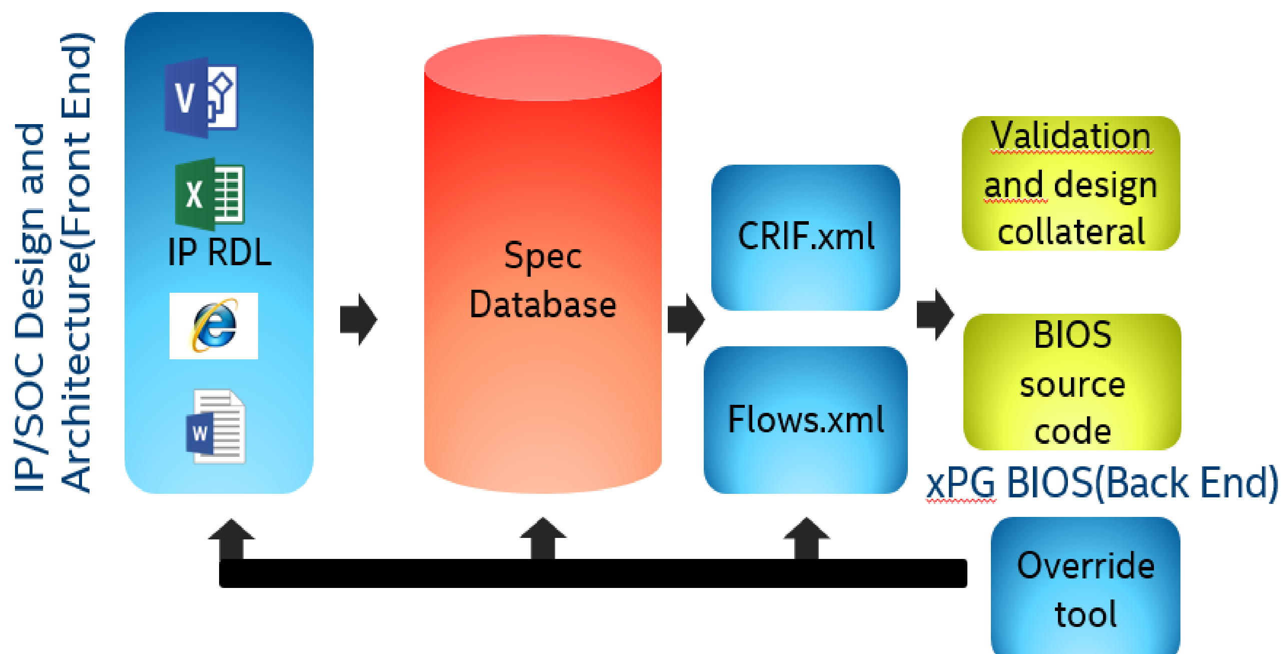


Fig. 5.3: System RDL Usage Model [8]

As shown in Fig. 5.3, it shows usage model for automated BIOS generation by system RDL. At the center there is spec database which contains all necessary specs which

will require to generate the flow. Mainly it has silicon changes specs and the register related information to generate the BIOS spec. This spec DB will be used by both front end and back end to create the complete flow. At the front end mostly designers will be there which will design a flow for automation and the User Interface will work by override tool to use the BIOS code. This override tool and the UI both are in sync from front end to back end. For this communication spec DB is not requires. Override tool will make after BIOS code generation of source file and header file which will use afterwards by the users. So by this way complete cycle creates from the front end to back end[11].

Once we completed the data gathering from front side from designers which is requires for design flow, it will collect that all information in the form of specs and it will be save in the spec DB. Now once specs will be generate, by using that specs RTL flow will be create at the back end which will be validate so that if some error will come then it can remove at that point only. After validating flow, it needs to be integrate into the main flow and check that all are working fine or not. Once that flow will work fine then based on that flow needs to make the document which will helpful for further designing as well as it will work as feedback system for next flow. From spec DB apart from RTL flow, it will generate the XML flow which will give the all details of registers level and will generate the flow[11].

There is one python script working in between the flow which will take the input as XML file this file can be named as CRIF file as well and at the output side this script will give the source file and header file for BIOS. Python script will take the XML file and first it will check all the register convention mention in the file as well as it's offset and it's value. Once it will find the information about this register then it will define that register into header file with its offset and value. This register will be use inside source file. Once all register will be define inside header file properly then script will start to make source files based on the flow mention inside XML files[11].

XML files have fields to define all the information for particular function inside flow. It has field named title where we have to give the flow name after that it ask for parameters need to pass into flow for future use of registers. Like that it contains so many fields called Step, Guard, Action, Condition etc. so based on the requirements of functions, we need to fill the all field properly and give that field to CRIF file.Now when this file

will go as an input for python script, script take a look for all this field and inside script all action is already define properly that which field should put at which place inside source file while defining function.

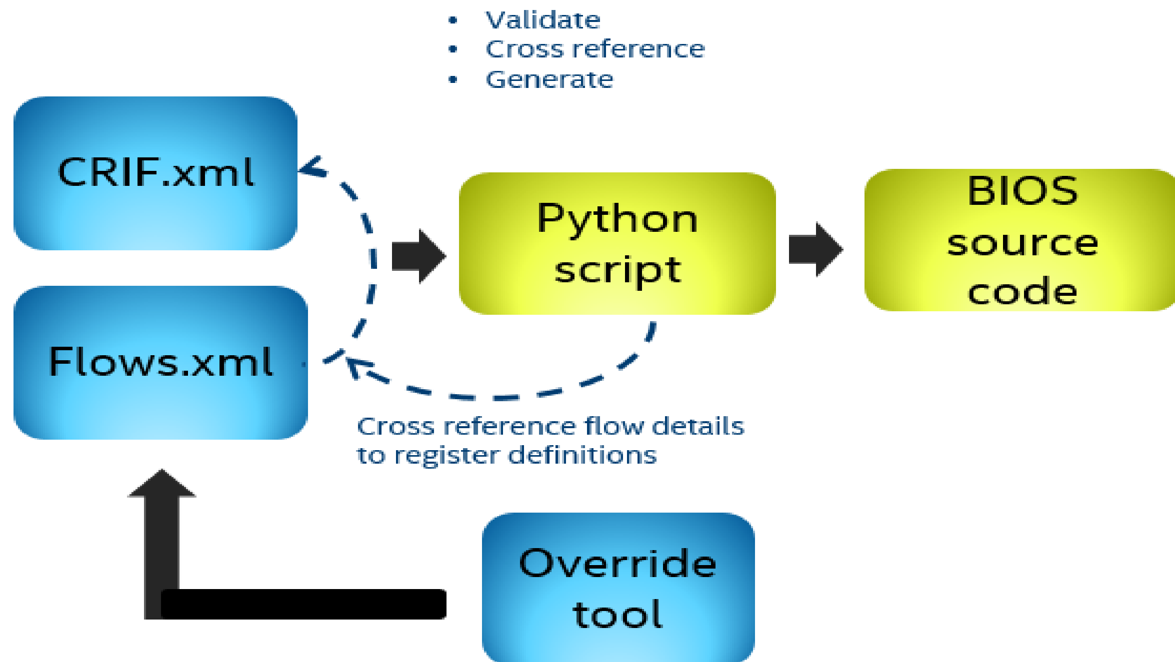


Fig. 5.4: Back End Overviewn [8]

As shown in Fig. 5.4, it is giving some more overview about the back end which is too important to work on. Here from the front end and spec database, it generates the two XML files called the flows and CRIF in which CRIF will contain all the information regarding the registers used to make the flow. Here we can see it is like feedback system where first python script will generate the BIOS source file and header file. After that script will give its feedback to XML file where flow will take this as input and it will change its flow accordingly [8].

Once flow change according to feedback, it generate the output as what flow changed and after that changed flow goes as input to the CRIF files because once flow will change it will change all the register information so that BIOS generation source and header file will generate according to the change flow otherwise if source files are not the as same as flow then BIOS integration will not work. So there is complete cycle created at the back end side so at every point complete flow from back end side will stay updated and it will work for same. At the last the override tool which is also the

part of back end can change the flow XML file which can be change manually. So if in looping some error will come then we should change the file so that we can make it proper [8].

5.4 Implementation

For implementing this project, first we requires the result from front end which will give the CRIF file and flow file. Flow file will generate from systemRDL which has basic flow generation method for all IP inside this file. SystemRDL file includes all the information regarding registers which are described by RTL files. For every RTL files it has information regarding some other register files where each register file contain some registers and this register contains the different fields of registers.

```xml
<?xml version="1.0" encoding="utf-8"?>
<ACES xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" Version="Dev" SVNRevision="Unknown" xmlns="http://intel.com/XMLSchema/ACES/1.0">
  <User>AMR\INTEL</User>
  <Timestamp>2015-10-28T23:09:59.7222077-07:00</Timestamp>
  <Data>
    <Title />
    <Description />
    <Table id="AMR_zhenkuny20151027T220942T635815805828965289">
      <Source>
        <Document>
          <Name>C:\Users\INTEL\Documents\pcie_flows.xlsx</Name>
          <Timestamp>2015-10-28T23:09:38.7676096-07:00</Timestamp>
        </Document>
      </Source>
      <Infos>
        <Info>
          <Key>Template</Key>
          <Value>State Transition</Value>
        </Info>
        </Infos>
      <Header>
        <FieldInfo>Step</FieldInfo>
        <FieldInfo>Guard</FieldInfo>
        <FieldInfo>Action</FieldInfo>
        <FieldInfo>Condition</FieldInfo>
        <FieldInfo>Next</FieldInfo>
        <FieldInfo>Notes</FieldInfo>
      </Header>
      <Records>
        <Record index="1">
          <Field>
            <Name>Step</Name>
            <Value>start</Value>
          </Field>
          </Record>
      </Records>
    </Table>
  </Data>
```

Fig. 5.5: Snippet of XML file

As shown in Fig. 5.5, it is showing the flow of XML file where it is asking for different field to fill for generating the next file in flow. Here in every register file it will ask for register file, bar number, bus number, device number, function number and registers used. Here CRIF will give as an input to python script and at the output side script will generate the requires header files and source file based on information given into flow XML file.

Once CRIF file will give as an input then at the output side we will get the CRIF.pkl file. CRIF.pkl file is the filter version of CRIF file which will contain only those type of

register information which is requires for generating source and header file. CRIF file is different for different IP and it will change its register information according to the IP. Now for generating source file and header file instead of using CRIF file we will use CRIF.pkl file because this file is already filter version of original file so we can directly get the information only which we want.

```
#include <B0D0F0RootComplexCfg.h>
#include <B0D0F0Mchbar.h>
#include <B0D8F0Gmr1Cfg.h>


  VOID gmminitapi(IN CPU_STEPPING CpuSteppingId)
  {

  UINT32                    McD8BaseAddress;
  UINT32                    McD0BaseAddress;
  UINTN                     MchBarBase;
  UINT32                    Data32_0;
  UINT32                    Data32_1;
 McD8BaseAddress = MmPciBase (SA_GMM_BUS_NUM, SA_GMM_DEV_NUM, SA_GMM_FUN_NUM) ;
 McD0BaseAddress = MmPciBase (SA_MC_BUS, SA_MC_DEV, SA_MC_FUN) ;
 MchBarBase      = MmioRead32 (McD0BaseAddress + 0x48) & ~BIT0;
 Data32_0 = MmioRead32 (McD0BaseAddress + CAPID0_B_REG);
     if (CAPID0_B_GMM_DIS_MSK_000 & Data32_0)
 {
    Data32_1= MmioRead32 (McD0BaseAddress + DEVEN_REG);

    if (DEVEN_D8EN_MSK_000 & Data32_1)

    {
        DEBUG  ((EFI_D_INFO,"\nGMM Device Disabled"));
    }
```

Fig. 5.6: Snippet of Source file

For generating the source code and the information about all IPs, other script will be used. This script will write the offset, register mask, register width etc. into header file which all are standardized and declare throughout the code which will be useful for optimization. As shown in Fig. 5.6 which is snapshot of source file generated by script but it is looking similar to the file which developers are developing. In this source file, script generating first function named GmmInitApi with the arguments specify in the CRIF file. Once function define, it will define the variable which will use by this function. After that script will start to define and use the registers one by another declare in CRIF file. This register value is already given into header file.

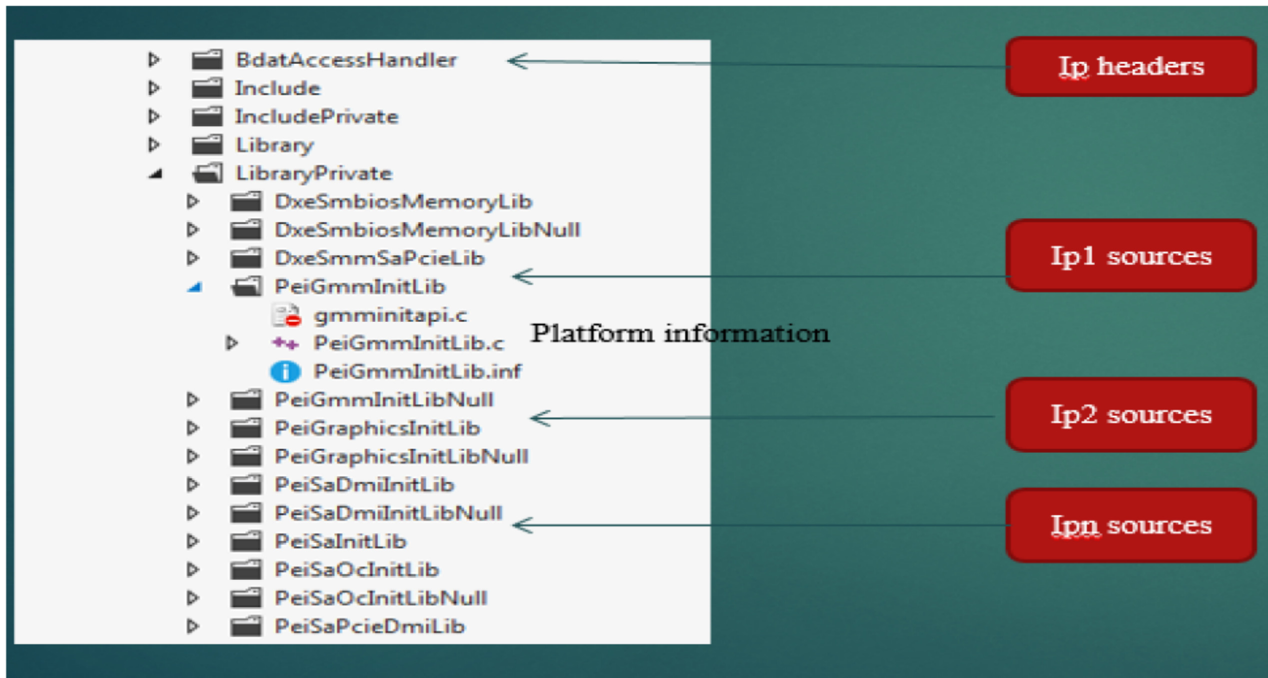


Fig. 5.7: Integration of BIOS code

As shown in Fig. 5.7, which is showing the integration of BIOS code generated by SystemRDL into the main codebase.Here in codebase header files are keeping inside Include folder whereas different IPs are stored according to the IP's basic location. For source file store , it saves in LibraryPrivate folder for main codebase.

Chapter 6

Conclusions and Future Scope

6.1 Conclusions

From this thesis work of **Firmware Support Package BIOS for Next Generation Processors**, FSP is very important binary configuration of BIOS. In general, if we use regular BIOS it requires large size of memory as well as it is taking time to boot. Instead of that if we use FSP then it contains less number of drivers than regular BIOS. So, it require less memory size as well faster boot time. FSP has become very popular because of points mentioned below [12]:

- Its interface is simple for any boot loader
- It has very small foot print around 200 KB
- It will take 200 milliseconds for execution
- Customizable configuration data region for the developers platform environment
- Supported across the Intels all intelligent system processors
- Widely adopted by Intels extensive ecosystem, integrating the Intel FSP into an array of value add boot loader solutions

By this research project of **Automated BIOS generation using System RDL**, we can use that automation of BIOS code is how much important. It not only reduce the developers effort but it reduce the error while code generation and so that it reduce the time to market.

According to flow, we are trying to solve the issue into pre silicon validation, issue at the time of SOC initialization become reduce to 30% which helped to enable the customers very early. As new features are implemented at the very early stage, stabilization of this feature becomes so faster which result into complete end to end feature validation in pre silicon and so faster launch of product than actual deadline [12].
Because of this work, Intel got alignment for overall pre silicon firmware readiness to actual pre silicon milestones which in result give all the platform productivity so higher and give chances to put the resources into right areas.

6.2 Future Scope

FSP is going good way in place of reference BIOS where it will execute the FSP in three different phases as preboot,SEC and PEI. Now a days FSP is working so faster than regular BIOS as it has advantages like:

- Royalty free solution from Intel
- Allow fast boot solution
- Easily integrated into existing firmware implementations

At present, we are developing FSP with its responsiveness infrastructure. As of now, for responsiveness, we are checking the execution timing of every APIs to get the performance of each driver. Now onwards with measuring the timing, we will also start to optimize the code. Optimization is requires to achieve the milestone set at the upper level. So for every API of FSP like FSP-T, FSP-M and FSP-S we will set one execution time and we will follow that by optimizing the code.
There are many features which FSP is supporting, so if developers want to enable or disable the feature then it can be possible by build flags. So in case if we want unbuild some features then build flags play a main role but this facility is not available for customers as they do not have code base. So, planning to have one more API called FSP-O which allows customers to have that facility.

For **Automated BIOS generation using System RDL**, As of now we are using particular SOC initialization for that IP. That SOC initialization happen by one Flow file that we need to change for different IPs according to the IP on which we are working. In main BIOS codebase there are many IPs included for complete silicon initialization.

So whenever we will start SOC initialization for different IP at that time first we need to change the flow file then only we can able to do initialization for that IP. To overcome this issue we need to do is re usability. By re usability we can use same IP initialization code flows across different segments which will reduce more efforts from designers.

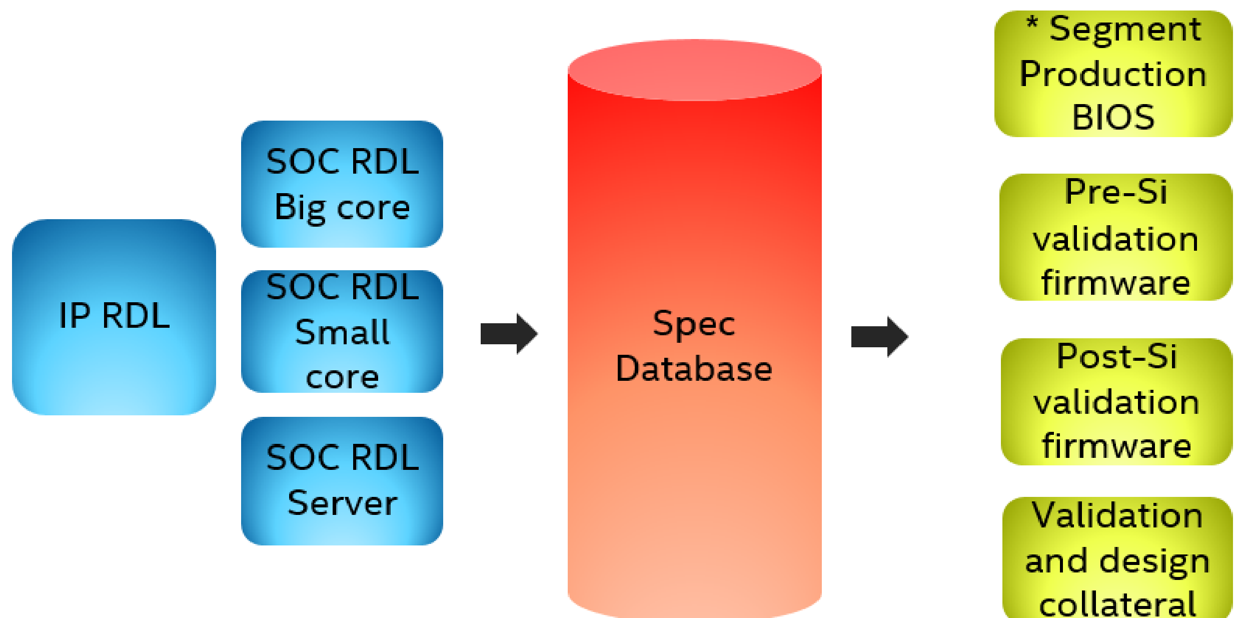


Fig. 6.1: Re usability Structure [12]

From Fig. 6.1, which giving the overview of re usability structure. In this one main RDL will generate which can be separate and use by Big core platforms, Small core platforms and server platforms for SOC initialization. Once that SOC initialization will become complete, then based on the platforms it will generate the spec database. This will give the idea that which SOC initialization happened so that at the back end side according to that design collateral for registers, pre si and post si validation and at the last segmentation of source and header file will happen [12].

Bibliography

[1] G. M. Vincent J. Zimmer, "A tour beyond BIOS using the intel firmware support package with the EFI developer kit II," *White Paper, Intel Corporation*, September 2014.

[2] S. Shah, "Intel architecture system - a handshake to begin pcie enumeration," *White Paper, Intel Corporation*, December 2014.

[3] J. MacInnis, "Implementing firmware on embedded intel architecture," *White Paper, Intel Corporation*, January 2009.

[4] M. D. Sun Jiming, "Intel firmware support package for intel architecture," *White Paper, Intel Corporation*, December 2014.

[5] P. D. Anton Cheng, "Intel fast boot - a recipe for two second pc boot," *White Paper, Intel Corporation*, October 2012.

[6] "Binary configuration tool user guide," *Intel Corporation*, June 2013.

[7] R. Gough, "Kaby lake platform bios enabling guide for usb type-c," *White Paper, Intel Corporation*, March 2016.

[8] R. H. Vincent Zimmer, Michael Rothman, "Beyond BIOS," *Intel Press*, September 2006.

[9] "Intel firmware support package, external architecture specification v1.1 draft6," *Intel Corporation*, April 2015.

[10] "Bios writers guide for UEFI 2.3.1," *Intel Corporation*, August 2012.

[11] "Advanced configuration and power interface specification 5.1," *Intel Corporation*, July 2015.

[12] "Unified extensible firmware interface specification 2.6," *Intel Corporation*, January 2016.

[13] J. Pelner, "EDK II platform configuration database entries: An introduction to PCD entries," *White Paper, Intel Corporation*, May 2011.

[14] "Web based training on UEFI BIOS phases and EDK II platform," *Intel Corporation*, August 2014.

[15] "Cannonlake SOC BIOS user guide," *Intel Corporation*, January 2016.