GOP (Graphics Output Protocol) Configuration Driver for BIOS

Major Project Report

Submitted in partial fulfilment of the requirements
for the degree of

Master of Technology

In

Electronics & Communication Engineering

(Communication Engineering)

By

Digant H. Solanki
(11MECC16)

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2013

GOP (Graphics Output Protocol) Configuration Driver for BIOS

Major Project Report

Submitted in partial fulfilment of the requirements
for the degree of

Master of Technology
In
Electronics & Communication Engineering
(Communication Engineering)

By

Digant H. Solanki
(11MECC16)

Under the guidance of

Mr. Bimod Narayanan
BIOS Engineer,
Intel Technology India Pvt. Ltd.
Bangalore

Prof. Sachin H. Gajjar
Electronics & Comm. Branch,
Institute of Technology,
Nirma University, Ahmedabad

Electronics & Communication Engineering Branch
Electrical Engineering Department
Institute of Technology
Nirma University
Ahmedabad-382 481
May 2013

Declaration

This is to certify that

1. The thesis comprises my original work towards the degree of Master of Technology in Communication Engineering at 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.

- Digant H. Solanki

Certificate

This is to certify that the Major Project entitled **"GOP (Graphics Output Protocol) Configuration Driver for BIOS"** submitted by **Digant H. Solanki (11MECC16)**, towards the partial fulfillment of the requirements for the degree of Master of Technology in Communication Engineering, Nirma University, Ahmedabad is the record of work carried out by him under our supervision and guidance. In our 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 our knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Date: Place: Ahmedabad

Prof. Sachin H. Gajjar
Guide

Dr.D.K.Kothari
Program Coordinator

Dr. P.N.Tekwani
Head of EE Department

Dr.K Kotecha
Director, IT

Acknowledgements

I would like to express my gratitude and sincere thanks to Dr. P.N.Tekwani, Head of Electrical Engineering Department, and Dr. D. K. Kothari, Coordinator of M.Tech Communication Engineering program for allowing me to undertake this thesis work and for his guidelines during the review process.

I am deeply indebted to my thesis supervisors Prof. Sachin H. Gajjar, Department of Electronics & Comm. Engineering, Nirma University and Mr. Bimod Narayanan, BIOS Engineer at Intel India Technology Pvt. Ltd. for their constant guidance and motivation. I also wish to thank Mr. Piyush Sharma, Manager, Intel India Technology Pvt. Ltd., Mr. Sudhakar Otturu, Mr. Supriyo Choudhury, Mr. Satya Yarlagadda and all other team members at Intel for their constant help and support. Without their experience and insights, it would have been very difficult to do quality work.

I wish to thank my friends of my class for their delightful company which kept me in good humor throughout the year.

Last, but not the least, no words are enough to acknowledge constant support and sacrifices of my family members because of whom I am able to complete the degree program successfully.

- Digant H. Solanki
11MECC16

Abstract

BIOS stands for Basic Input Output System.On a very high level, it initializes the hardware like processor, chipset, peripherals etc. and then it gives control to the OS through boot manager. The BIOS must do its job before computer can load operating system and applications.Currently Intel has migrated from legacy BIOS to EFI BIOS which is based on EFI Specification. It has standard, modular environment and have many advantages over legacy BIOS.EFI is a standard or a specification which has different phases named as SEC (Security), PEI (Pre EFI Initialization), DXE (Driver Execution Environment) and BDS (Boot Device Selection). There is a provision of legacy BIOS in EFI BIOS code because not all the OSes are EFI compatible.

In legacy BIOS, VBIOS is used for the display functionalities like: to initialize Intel graphics Hardware and boot to appropriate attached display devices by means of a data called as configurable header whereas in EFI BIOS this has been replaced by GOP (Graphics Output Protocol) by its own other configurable header.For different platforms, different configurable headers are required. Configurable Headers are blocks of data inside VBIOS. In EFI BIOS, GOP driver uses the configurable header for display purpose.Configurable header is a binary file which is currently configured manually by Intel's Configurable tool. In this document, one EFI based DXE driver is discussed which is targeted to configure the Configurable header for BIOS such that it can eliminate the need to configure it manually for different platforms.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

In this thesis report, GOP Configuration Driver for BIOS is discussed.The driver is totally based on C language and it is developed using a Microsoft Visual Studio 2008 as an IDE. The driver is intended to eliminate the human intervention. Graphics Output Protocol (GOP) which is used for the EFI aware operating system is the replacement of Video BIOS (VBIOS). GOP Driver takes care of the display functionalities in the native mode (EDK II) whereas the VBIOS does the same in legacy BIOS.

GOP driver performs its task of display functionalities by means of a block of data which is called as Configurable header. The driver gets the configurable header according to the platform and performs the display functionalities defined in the configurable header. It is included in the BIOS code as a binary file only. For different platform different configurable headers are required.Intel uses its internal configurable Tool to configure the configurable header for different platforms.

The GOP Configuration Driver objective here is to configure the configurable header and eliminate the need of Intel's configurable tool to configure the header manually.The driver will get the header using Platform Gop Policy protocol, copied to some memory location and get the pointer and the new configurable header address. After getting the pointer to the copied header, driver will update the header as needed and update the Platform Gop Policy such that the GOP driver will now get the updated header. Setup options are created in the BIOS menu which are equivalent to the setup options in configurable tool. The BIOS options in the BIOS menu are patched to the non volatile memory and that are updated in the Platform Gop Policy by the means of GOP Configuration Driver.

The driver discussed here is implemented in two different environment for Intel's next Gen Processor.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 driver's 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.

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 if is done for that. Here the motivation behind the work done in the thesis is to eliminate the human intervention to update the display functionalities using Intel's Configurable tool in BIOS development for different platforms.

Video BIOS, used legacy BIOS mode, is responsible for the display or graphics functionalities for the Intel's mobile as well as desktop platforms. VBIOS does its task using interrupt calls with GPU (Graphics Processing Unit). There is a two way communication between VBIOS and GPU using the INT 10 and INT 15 call for graphics functionalities. It is not good always to use interrupt calls so in EFI BIOS, industry has introduced Graphics Output Protocol (GOP) to replace the functionality of VBIOS.

GOP does the same functionality as VBIOS using protocol and its GUIDs instead of interrupt calls used in VBIOS. Protocols and GUIDs are introduced in the new EFI framework for the BIOS. It is easy to maintain the communication between GOP and GPU using protocol and its GUIDs than with the interrupt calls used in VBIOS. More will be discussed on protocol and GUIDs later. GOP makes the display functionality possible with the help of configurable header which will be passed to the GPU.

The limitation with the GOP implementation in EFI BIOS is that there is only a one way communication between GOP and GPU only. There is no link in the BIOS code from GPU to GOP which can be used to configure the GOP if it is there. So GOP has to provide different configurable headers for different platforms and BIOS code must contain more than one configurable header for the display functionality.

The motivation behind this thesis is to make a driver in the BIOS code such that it can configure the header according to the need for different platforms and it is named as GOP Configuration Driver for BIOS. This driver will now make a second way communication link from GPU to GOP and will make it possible to configure the GOP functionalities by modifying the header.This driver is not available for the end users but this driver will be used by the different OEMs like Dell, Lenovo etc...

1.3 Terminology

ACPI	Advanced Configuration and Power Interface
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
EDK	EFI Development Kit
EFI	Extensible Firmware Interface
FFS	Firmware File System
FV	Firmware Volume
GOP	Graphics Output Protocol
GUID	Globally Unique Identifier
HOB	Hand Off Block
INT	Interrupt
OEM	Original Equipment Vendor
OSPM	Operating System-directed configuration and Power Management
PCD	Platform Configuration Database
PEI	Pre EFI Initialization
PI	Platform Initialization
POST	Power On Self-Test - Chipset initialization code
ROM	Read Only Memory
UEFI	Unified Extensible Firmware Interface
VBE	VESA BIOS Extensions
VESA	Video Electronics Standard Association
VGA	Video Graphics Adapter
VBIOS	Video BIOS

Table 1.1: Terminology used

1.4 Problem Definition

The Graphics Output Protocol (GOP) is enabled by UEFI driver to support graphic console output in the pre-OS phase. GOP is designed to be lightweight and to support the basic needs of graphics output prior to Operating System boot. The ultimate goal of GOP is to replace legacy VGA BIOS and eliminate VGA HW functionality.

A configurable header file represents a single platform configuration. Each unique configuration may be saved as a different configurable header file for different platform. As the platform changes, one has to configure a header using Intel's configurable tool. This tool has many set up option for a header configuration. This tool will generate a binary file for any particular requirement and this binary file has to be merged in the system BIOS code which will be used by a GOP driver for its graphics output.

One limitation here is that BIOS code is not able to configure a header as the platform changes. One has to configure a header using configurable tool as per the requirement and then again have to merge the new configurable header into the BIOS code. In this document, one DXE driver, called GOP Configuration Driver for BIOS, is discussed which can accomplish the task to configure the header which is a block of data used for display functionalities. This driver will have a code such that it can configure some specific header data and it is possible to use a same header for different platforms.

1.5 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2, *Literature Survey,* describes the basics about BIOS, UEFI specification and the difference between legacy BIOS and EFI BIOS. It also tells about the UEFI Driver Model.

Chapter 3, *DXE Overview,* presents the concepts about DXE phase which is useful to make a DXE driver. In this chapter, DXE Phase overview, DXE Foundation, DXE Dispatcher and a classification of DXE drivers is discussed.

Chapter 4, *GOP Configuration Driver,* covers the difference between GOP vs. VBIOS and purpose of the driver , its flow diagram, issues faced and resolver and finally it describes the high level understanding of every files used in the driver.

Chapter 5, *Driver Implementation,* describe how the driver is implemented using two different environment and also with the two different modes using flags. It also covers the overview, Difference and similarities between EDK I and EDK II environments.

Finally , **In Chapter 6,** conclusion and scope for future work is presented.

Chapter 2

Literature Survey

2.1 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 ('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.[9]
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.[9]

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.[9]

BIOS features are as follows:

- 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.

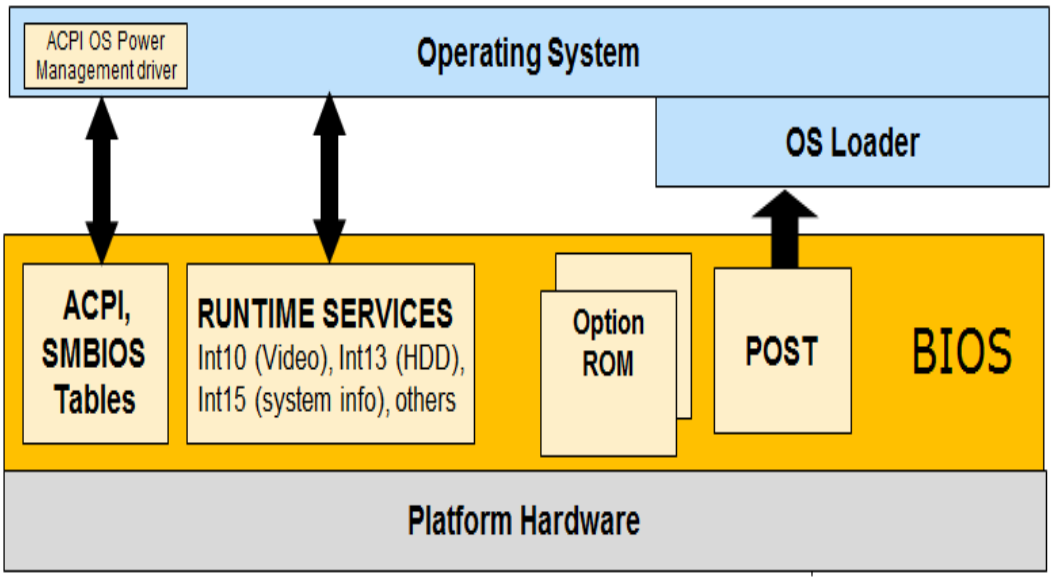


Figure 2.1: BIOS Overview

- Provides Power Management functionality through ACPI
- Loads and hands control over to the OS boot loader
- Provides a set of standardized routines for the OS to use
- Abstracts motherboard and silicon specifics from the OS
- Prepares system to run an OS
- Provides runtime services to the OS e.g. disk and video

The Advanced Configuration and Power Interface (ACPI) is a specification which was developed to establish industry common interfaces enabling robust operating system (OS)-directed motherboard device configuration and power management of both devices and entire systems. ACPI is the key element in Operating System-directed configuration and Power Management (OSPM).[3]

Power-On Self-Test (POST) refers to routines run immediately after power is applied, by nearly all electronic devices. POST includes routines to set an initial value for internal and output signals and to execute internal tests, as determined by the device manufacturer. These initial conditions are also referred to as the device's state.

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 Figure 2.2.

Legacy BIOS	UEFI BIOS
This is the 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	Allows direct access of all memory via (32-bit and/or 64 bit) pointers
Supports 3rd party modules in the form of 16 bit Option ROMS	Supports 3rd party 32/64 bit drivers
No built in boot/test environment	Built-in boot/test via EFI - Shell
Only supports 16-bit runtime services such as INT10, INT13, etc	New runtime interfaces and supports legacy OSs and 16-bit legacy devices
Example of Legacy BIOS: AMI core 8, Phoenix legacy BIOS	Standardized implementations: Aptio (AMI), H2O (Insyde), Tiano (Intel)

Figure 2.2: Legacy BIOS vs. EFI BIOS

BIOS code used today supports legacy BIOS as well as EFI BIOS by means of a CSM module. All the OS are not compatible with the EFI BIOS; CSM module is used to run appropriate BIOS code. When CSM mode is ON, system will boot to legacy BIOS and if CSM mode is OFF, system will boot to native EFI BIOS.

The CSM translates the information generated under the EFI environment into the information required by the legacy environment and makes the legacy BIOS services available for booting to the operating system and for use in runtime.

2.2 UEFI Specification

2.2.1 Overview

This Unified Extensible Firmware Interface (hereafter known as UEFI) Specification describes an interface between the operating system (OS) and the platform firmware which is shown in the figure 2.3. UEFI was preceded by the Extensible Firmware Interface Specification (EFI).

The interface is in the form of data tables that contain platform-related information, and boot and runtime service calls that are available to the OS loader and the OS. Together, these provide a standard environment for booting an OS. This specification is designed as a pure interface specification. As such, the specification defines the set of interfaces and structures that platform firmware must implement. Similarly, the specification defines the set of interfaces and structures that the OS may use in booting.[1]

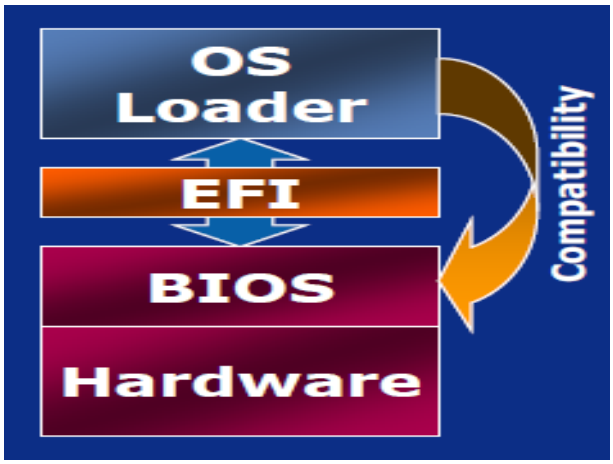


Figure 2.3: EFI Interface [4]

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.[1]

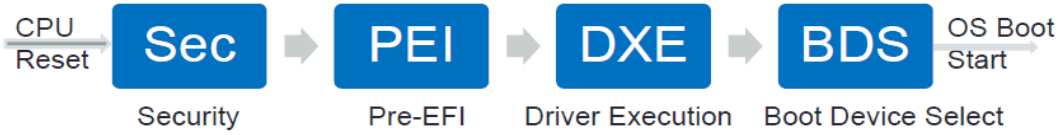
The specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The selection of protocol interfaces can evolve over time to be optimized for various platform market 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.[1]

2.2.2 UEFI Boot Phases

EFI BIOS is a modular code and it boots in a manner shown in Figure 2.4. EFI Boot process is dived into four main phases which are:

- Security Phase
- Pre EFI Initialization Phase
- Driver Execution Environment Phase
- Boot Device Selection Phase

Each phases with its services are shown in the figure 2.4



Sec	PEI	DXE	BDS
-Provides uP boot strap vector -Initializes temporary RAM using CPU cache -Provides optional security features	-Init Proc -Update ucode -Initialize chipset -Detect corrupted flash and recover if corrupted -Find and initialize RAM	-Initialize uP, cache, -Initialize more c/s -Initialize SMM -Execute PCI enumeration -Initialize video, kbd, mouse -Initialize USB legacy: -Initialize drives -Create table interfaces (ACPI, SMBIOS, ...)	-Run down list of selected devices. -For each, try to find boot image -Load boot image into memory -Jump to boot image -If fail try next

Figure 2.4: EFI Boot Phases and services

Security phase:

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

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

In addition to the minimum architecturally required handoff information, the SEC phase can pass optional information to the PEI Core, such as the SEC Platform Information PPI or information about the health of the processor.

Pre-EFI Initialization (PEI) Phase:

The Pre-EFI Initialization (PEI) phase of the PI Architecture specifications (hereafter referred to as the PI Architecture) is invoked quite early in the boot flow. Specifically, after some preliminary processing in the Security (SEC) phase, any machine restart event will invoke the PEI phase.The PEI phase will initially operate with the platform in a nascent state, leveraging only on processor resources, such as the processor cache as a call stack, to dispatch Pre-EFI Initialization Modules (PEIMs). These PEIMs are responsible for the following:

- Initializing some permanent memory complement.
- Describing the memory in Hand-Off Blocks (HOBs).

- Describing the firmware volume locations in HOBs.
- Passing control into the Driver Execution Environment (DXE) phase.

Philosophically, the PEI phase is intended to be the thinnest amount of code to achieve the ends listed above. As such, any more sophisticated algorithms or processing should be deferred to the DXE phase of execution.
The PEI phase is also responsible for crisis recovery and resuming from the S3 sleep state. For crisis recovery, the PEI phase should reside in some small, fault-tolerant block of the firmware store. As a result, it is imperative to keep the footprint of the PEI phase as small as possible. In addition, for a successful S3 resume, the speed of the resume is of utmost importance, so the code path through the firmware should be minimized. These two boot flows also speak to the need to keep the processing and code paths in the PEI phase to a minimum. The implementation of the PEI phase is more dependent on the processor architecture than any other phase. In particular, the more resources the processor provides at its initial or near initial state, the richer the interface between the PEI Foundation and PEIMs.

Driver execution Environment (DXE) Phase:

The Driver Execution Environment (DXE) phase is where most of the system initialization is performed. Pre-EFI Initialization (PEI), the phase prior to DXE, is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position-independent data structures called Hand-Off Blocks (HOBs). HOBs are described in detail in the Platform Initialization Hand-Off Block Specification. The more about the DXE phase will be discussed in the next chapter.

Boot Device Selection (BDS) Phase:

The Boot Manager in DXE executes after all the DXE drivers whose dependencies have been satisfied have been dispatched by the DXE Dispatcher. At that time, control is handed to the Boot Device Selection (BDS) phase of execution. The BDS phase is responsible for implementing the platform boot policy.This boot policy provides flexibility that allows system vendors to customize the user experience during this phase of execution.

The Boot Manager must also support booting from a short-form device path that starts with the first node being a firmware volume device path. The boot manager must use the GUID in the firmware volume device node to match it to a firmware volume in the system. The GUID in the firmware volume device path is compared with the firmware volume name GUID. If a match is made, then the firmware volume device path can be appended to the device path of the matching firmware volume and normal boot behavior can then be used.

The BDS phase is implemented as part of the BDS Architectural Protocol. The DXE Foundation will hand control to the BDS Architectural Protocol after all of the DXE

drivers whose dependencies have been satisfied have been loaded and executed by the DXE Dispatcher. The BDS phase is responsible for the following:

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

If the BDS phase cannot make forward progress, it will reinvoke the DXE Dispatcher to see if the dependencies of any additional DXE drivers have been satisfied since the last time the DXE Dispatcher was invoked.

2.2.3 Boot modules

Platform

This module touches almost all components on the mother board including CPU and PCH. It does the Board detection, Processor Power management including C-states, P-states, throttling, Thermal reporting and Implement some security features related to processor.

PCH

PCH module is used to provide following services. Intel HD Audio, SMBUS, SPI, SATA, Legacy interrupt, System Management Interrupts, System reset, Timers, USB, Display Link Etc.

Memory

This module runs in PEI phase in 32bit mode.It supports detection and initialization of memory modules and complies with the requirements in the BIOS specification.

ME

Me is management engine, it provides centralized administration, also responsible for vpro technology, integrated clock control etc.

Security

It provides trusted execution environment also provide security related function create cryptographic keys, windows bitlocker.

System Agent

This is uncore part mainly responsible for graphics also initializing System Memory, initializing Power Management, internal Graphics, internal Audio, and PCI Express, modifying SA register default values for optimal performance, SMRAM initialization.

2.3 UEFI Driver Model Goals [2]

The UEFI Driver Model has following goals:

- Compatible

  Drivers conforming to this specification must maintain compatibility with the previous EFI and UEFI Specification.

- Simple

  Drivers that conform to this specification must be simple to implement and simple to maintain. The UEFI Driver Model must allow a driver writer to concentrate on the specific device for which the driver is being developed. A driver should not be concerned with platform policy or platform management issues. These considerations should be left to the system firmware.

- Scalable

  The UEFI Driver Model must be able to adapt to all types of platforms. These platforms include embedded systems, mobile, and desktop systems, as well as workstations and servers.

- Flexible

  The UEFI Driver Model must support the ability to enumerate all the devices, or to enumerate only those devices required to boot the required OS. The minimum device enumeration provides support for more rapid boot capability, and the full device enumeration provides the ability to perform OS installations, system maintenance, or system diagnostics on any boot device present in the system.

- Extensible

  The UEFI Driver Model must be able to extend to future bus types as they are defined.

- Portable

  Drivers written to the UEFI Driver Model must be portable between platforms and between supported processor architectures.

- Interoperable

  Drivers must coexist with other drivers and system firmware and must do so without generating resource conflicts.

- Describe complex bus hierarchies

The UEFI Driver Model must be able to describe a variety of bus topologies from very simple single bus platforms to very complex platforms containing many buses of various types.

- Small driver footprint

  The size of executables produced by the UEFI Driver Model must be minimized to reduce the overall platform cost. While flexibility and extensibility are goals, the additional overhead required to support these must be kept to a minimum to prevent the size of firmware components from becoming unmanageable.

- Address legacy option rom issues

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

2.4 GUIDs [2]

A UEFI programming environment provides software services through the UEFI Boot Services Table, the UEFI Runtime Services Table, and Protocols installed into the handle database. Protocols are the primary extension mechanism provided by the UEFI Specification. Protocols are named using a GUID.

A GUID is a unique 128-bit number that is a globally unique identifier (a universally unique identifier, or UUID). Each time an image, protocol, device, or other item is defined in UEFI, a GUID must be generated for that item. The example below shows the structure definition for an EFI_GUID in the EDK II along with the definition of the GUID value for the EFI Driver Binding Protocol from the UEFI Specification.

Protocol services are registered in the handle database using the GUID name of the Protocol and Protocol services are discovered by looking up Protocols in the handle database using the GUID name associated with the Protocol to perform the lookup operation.

UEFI fundamentally assumes that a specific GUID exposes a specific protocol interface (or other item). Because a protocol is "named" by a GUID (a unique identifier), there should be no other protocols with that same GUID. Be careful when creating protocols to define a new, unique GUID for a new protocol. Put another way, the GUID forms a contract: If the UEFI Driver finds a protocol with a particular GUID, it may assume that the contents of the protocol are as specified for that protocol. If the contents of the protocol are different, the driver that published the protocol is assumed to be in error.

```
///
/// 128 bit buffer containing a unique identifier value.
/// Unless otherwise specified, aligned on a 64 bit boundary.
///
typedef struct {
  UINT32  Data1;
  UINT16  Data2;
  UINT16  Data3;
  UINT8   Data4[8];
} GUID;

///
/// 128-bit buffer containing a unique identifier value.
///
typedef GUID                      EFI_GUID;

///
/// The global ID for the Driver Binding Protocol.
///
```

Figure 2.5: GUID Structure[2]

```
#define EFI_DRIVER_BINDING_PROTOCOL_GUID \
  { \
    0x18a031ab, 0xb443, 0x4d1a, {0xa5, 0xc0, 0xc, 0x9, 0x26, 0x1e, 0x9f, 0x71 } \
  }
```

Figure 2.6: An example of GUID Definition[2]

```
///
/// Global ID for the Component Name Protocol
///
#define EFI_COMPONENT_NAME2_PROTOCOL_GUID \
  {0x6a7a5cff, 0xe8d9, 0x4f70, { 0xba, 0xda, 0x75, 0xab, 0x30, 0x25, 0xce, 0x14 } }

typedef struct _EFI_COMPONENT_NAME2_PROTOCOL  EFI_COMPONENT_NAME2_PROTOCOL;

///
/// This protocol is used to retrieve user readable names of drivers
/// and controllers managed by UEFI Drivers.
///
struct _EFI_COMPONENT_NAME2_PROTOCOL {
  EFI_COMPONENT_NAME2_GET_DRIVER_NAME      GetDriverName;
  EFI_COMPONENT_NAME2_GET_CONTROLLER_NAME  GetControllerName;

  ///
  /// A Null-terminated ASCII string array that contains one or more
  /// supported language codes. This is the list of language codes that
  /// this protocol supports. The number of languages supported by a
  /// driver is up to the driver writer. SupportedLanguages is
  /// specified in RFC 4646 format.
  ///
  CHAR8                                    *SupportedLanguages;
};
```

Figure 2.7: An example of Protocol Definition and Declaration[2]

In some ways, GUIDs are can be viewed as contracts. If a UEFI Driver looks up a protocol with a certain GUID, the structure under the GUID is well defined. If the GUID is duplicated, this 1:1 mapping breaks. If a GUID is copied and applied to a new protocol, the users of the old protocol call the new protocol expecting the old interfaces or vice versa. Either way, the results are never good.

2.5 Protocols and handles[2]

The extensible nature of UEFI is built, to a large degree, around protocols. Protocols serve to enable communication between separately built modules, including drivers.

Drivers create protocols consisting of two parts. The body of a protocol is a C-style data structure known as a protocol interface structure, or just "interface". The interface typically contains an associated set of function pointers and data structures.

Every protocol has a GUID associated with it. The GUID serves as the name for the protocol. The GUID also indicates the organization of the data structure associated with the GUID. Note that the GUID is not part of the data structure itself. The example below shows a portion of the Component Named 2 Protocol definition from the UEFI Driver Model chapter of the UEFI Specification. Notice that the protocol data structure contains two functions and one data field.

Protocols are gathered into a single database. The database is not "flat." Instead, it allows protocols to be grouped together. Each group is known as a handle, and the

handle is also the data type that refers to the group. The database is thus known as the handle database. Handles are allocated dynamically. Protocols are not required to be unique in the system, but they must be unique on a handle. In other words, a handle may not be associated with two protocols that have the same GUID.

Chapter 3

Driver Execution Environment

3.1 Overview

The Driver Execution Environment (DXE) phase is where most of the system initialization is performed. Pre-EFI Initialization (PEI), the phase prior to DXE, is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position independent data structures called Hand-Off Blocks (HOBs).

There are several components in the DXE phase like:

- DXE Foundation
- DXE Dispatcher
- A set of DXE drivers

The DXE Foundation produces a set of Boot Services, Runtime Services, and DXE Services. The DXE Dispatcher is responsible for discovering and executing DXE drivers in the correct order. The DXE drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for system services, console devices, and boot devices. These components work together to initialize the platform and provide the services required to boot an operating system. The DXE phase and Boot Device Selection (BDS) phases work together to establish consoles and attempt the booting of operating systems.[5]

The DXE phase is terminated when an operating system is successfully booted. The DXE Foundation is composed of boot services code, so no code from the DXE Foundation itself is allowed to persist into the OS runtime environment. Only the runtime data structures allocated by the DXE Foundation and services and data structured produced by runtime DXE drivers are allowed to persist into the OS runtime environment.Figure 3.1 shows the phases that a platform with Framework firmware will execute.[5]

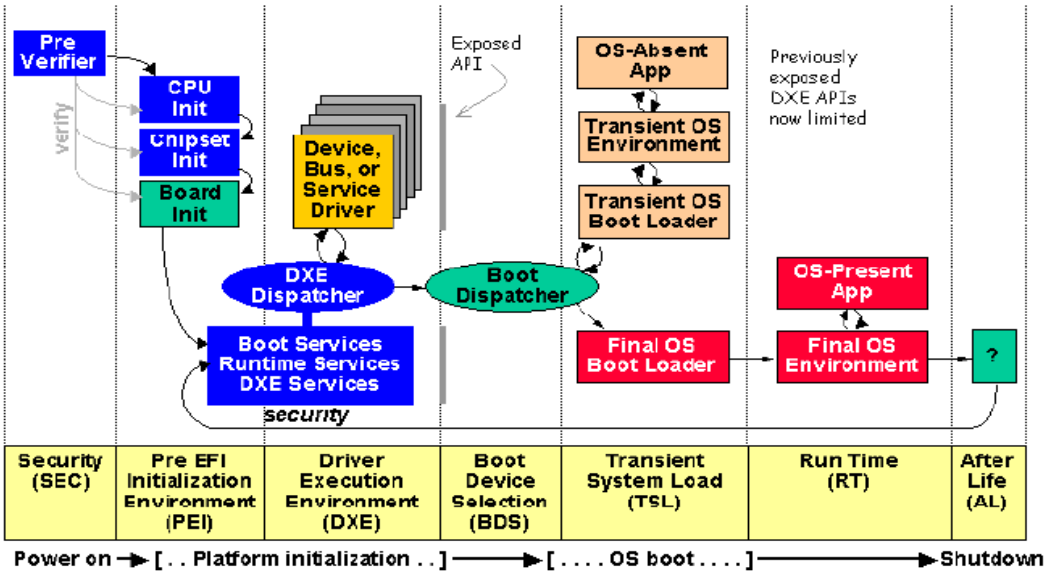


Figure 3.1: EFI Boot sequence[5]

The DXE phase does not require a PEI phase to be executed. The only requirement for the DXE phase to execute is the presence of a valid HOB list. There are many different implementations that can produce a valid HOB list for the DXE phase to execute.

3.2 EFI System Table[5]

The EFI System Table is passed to every executable component in the DXE phase. It contains a pointer to the following:

- EFI Boot Services Table
- EFI Runtime Services Table

It also contains pointers to the console devices and their associated I/O protocols. In addition, the EFI System Table contains a pointer to the EFI Configuration Table, and this table contains a list of GUID/pointer pairs. The EFI Configuration Table may include tables such as the DXE Services Table, HOB list, ACPI table, SMBIOS table, and SAL System table.

The EFI Boot Services Table contains services to access the contents of the handle database. The handle database is where protocol interfaces produced by drivers are registered. Other drivers can use the EFI Boot Services to look up these services produced by other drivers.Figure 3.2 shows the components of EFI system Table. All of the services available in the DXE phase may be accessed through a pointer to the EFI System Table.

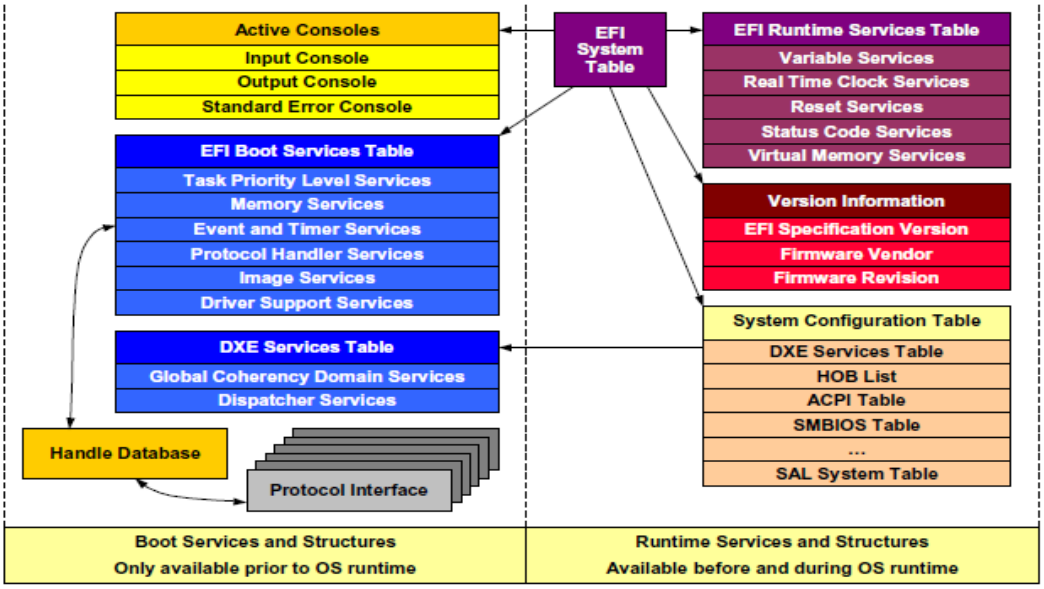


Figure 3.2: EFI System Table and its services[5]

3.3 DXE Foundation[5]

The DXE Foundation is designed to be completely portable with no processor, chipset, or platform dependencies. This lack of dependencies is accomplished by designing in several features:

- DXE Foundation depends only upon a HOB list for its initial state

  This means that the DXE Foundation does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE Foundation.

- The DXE Foundation does not contain any hard-coded addresses.

  This means that the DXE Foundation can be loaded anywhere in physical memory, and it can function correctly no matter where physical memory or where Firmware Volumes (FVs) are located in the processor's physical address space.

- The DXE Foundation does not contain any processor-specific, chipset-specific, or platform specific information.

  Instead, the DXE Foundation is abstracted from the system hardware through a set of DXE Architectural Protocol interfaces. These architectural protocol interfaces are produced by a set of DXE drivers that are invoked by the DXE Dispatcher.

The DXE Foundation must produce the EFI System Table and its associated set of EFI Boot Services and EFI Runtime Services. The DXE Foundation also contains the

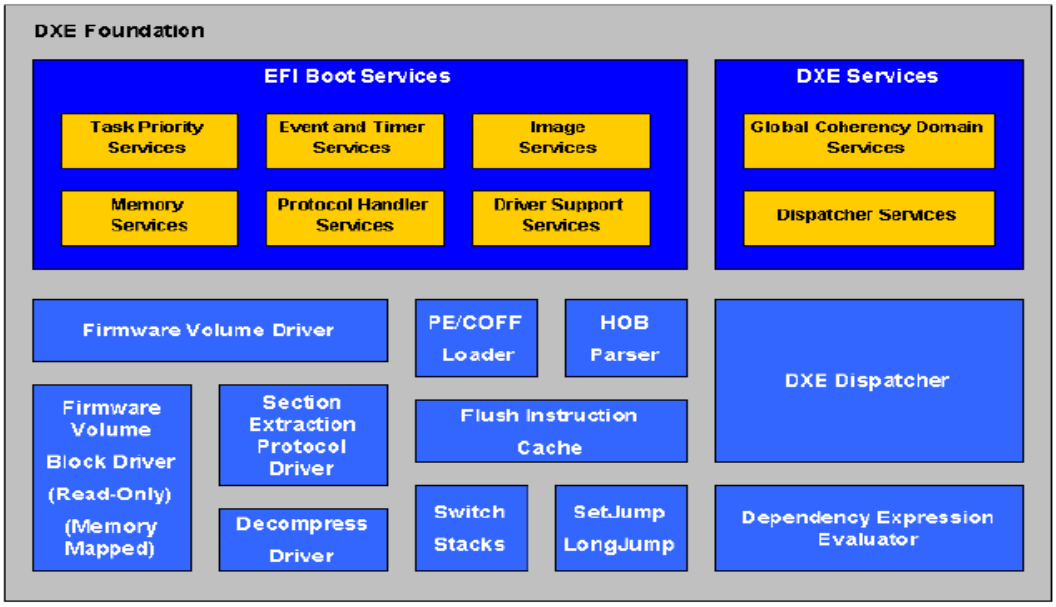


Figure 3.3: DXE Foundation components[5]

DXE Dispatcher whose main purpose is to discover and execute DXE drivers stored in FVs. Figure 3.3 shows the components of DXE Foundation.

The execution order ofDXE drivers are determined by a combination of the optional a priori file and the set of dependency expressions that are associated with the DXE drivers. The FV file format allows dependency expressions to be packaged with the executable DXE driver image. DXE drivers utilize a PE/COFF image format, so the DXE Dispatcher must also contain a PE/COFF loader to load and execute DXE drivers. Below diagram shows the components of DXE Foundation used in EFI.

3.4 DXE Dispatcher[5]

After the DXE Foundation is initialized, control is handed to the DXE Dispatcher. The DXE Dispatcher examines every firmware volume that is present in the system. Firmware volumes are either declared by HOBs, or they are declared by DXE drivers. For the DXE Dispatcher to run, at least one firmware volume must be declared by a HOB.

The DXE Dispatcher is one component of the DXE Foundation. This component is required to discover DXE drivers stored in firmware volumes and execute them in the proper order. The proper order is determine by a combination of an a priori file that is optionally stored in the firmware volume and the dependency expressions that are part of the DXE drivers. The dependency expression tells the DXE Dispatcher the set of services that a particular DXE driver requires to be present for the DXE driver to execute. The DXE Dispatcher does not allow a DXE driver to execute until all of the DXE driver's dependencies have been satisfied. After all of the DXE drivers have been loaded and executed by the DXE Dispatcher, control is handed to the

BDS Architectural Protocol that is responsible for implementing a boot policy that is compliant with the EFI Boot Manager.

The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes. Some DXE drivers may depend on the services produced by other DXE drivers, so the DXE Dispatcher is also required to execute the DXE drivers in the correct order. The DXE drivers may also be produced by a variety of different vendors, so the DXE drivers must describe the services they depend upon. The DXE dispatcher must evaluate these dependencies to determine a valid order to execute the DXE drivers. Some vendors may wish to specify a fixed execution order for some or all of the DXE drivers in a firmware volume, so the DXE dispatcher must support this requirement.

In addition, the DXE Dispatcher must support the ability to load "emergency patch" drivers. These drivers would be added to the firmware volume to address an issue that was not known at the time the original firmware was built. These DXE drivers would be loaded just before or just after an existing DXE driver.

Finally, the DXE Dispatcher must be flexible enough to support a variety of platform specific security policies for loading and executing DXE drivers from firmware volumes. Some platforms may choose to run DXE drivers with no security checks and others may choose to check the validity of a firmware volume before it is used, and other may choose to check the validity of every DXE driver in a firmware volume before it is executed.

3.5 DXE Drivers

The DXE architecture provides a rich set of extensible services that provides for wide variety of different system firmware designs. The DXE Foundation provides the generic services required to locate and execute DXE drivers. The DXE drivers are the components that actually initialize the platform and provide the services required to boot an EFI-compliant operating system or a set of EFI-compliant system utilities. There are many possible firmware implementations for any given platform. Because the DXE Foundation has fixed functionality, all the added value and flexibility in a firmware design is embodied in the implementation and organization of DXE drivers.

There are two basic classes of DXE drivers:

- Early DXE Drivers
- DXE Drivers that follow the EFI Driver Model

Additional classifications of DXE drivers are also possible. All DXE drivers may consume the EFI Boot Services, EFI Runtime Services, and DXE Services to perform their functions. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed.

Classes of Drivers:

1. Early DXE Drivers:

The first class of DXE drivers is those that execute very early in the DXE phase. The execution order of these DXE drivers depends on the following:

- The presence and contents of an a priori file
- The evaluation of dependency expressions

These early DXE drivers will typically contain basic services, processor initialization code, chipset initialization code, and platform initialization code. These early drivers will also typically produce the DXE Architectural Protocols that are required for the DXE Foundation to produces its full complement of EFI Boot Services and EFI Runtime Services. To support the fastest possible boot time, as much initialization should be deferred to the DXE drivers that follow EFI Driver Model. The early DXE drivers need to be aware that not all of the EFI Boot Services, EFI Runtime Services, and DXE Services may be available when they execute because not all of the DXE Architectural Protocols may be been registered yet.[2]

2. DXE Drivers That Follow the EFI Driver Model:

The second class of DXE drivers is those that follow the EFI Driver Model. These drivers do not touch any hardware resources when they initialize. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the Boot Device Selection (BDS) phase to connect the drivers to the devices that are required to establish consoles and provide access to boot devices. The DXE drivers that follow the EFI Driver Model ultimately provide software abstractions for console devices and boot devices, but only when they are explicitly asked to do so.[2]

The DXE drivers that follow the EFI Driver Model do not need to be concerned with dependency expressions. These drivers simply register the Driver Binding Protocol in the handle database when they are executed, and this operation can be performed without the use of any DXE Architectural Protocols. DXE drivers with empty dependency expressions will not be dispatched by the DXE Dispatcher until all of the DXE Architectural Protocols have been installed.

Additional Classification

DXE drivers can also be classified as the following:

- Boot service drivers
- Runtime drivers

Boot service drivers provide services that are available until the ExitBootServices() function is called. When ExitBootServices() is called, all the memory used by boot service drivers is released for use by an operating system.[1]

Runtime drivers provide services that are available before and after ExitBootServices() is called, including the time that an operating system is running. All of the services in the EFI Runtime Services Table are produced by runtime drivers.

The DXE Foundation is considered a boot service component, so the DXE Foundation is also released when ExitBootServices() is called. As a result, runtime drivers may not use any of the EFI Boot Services, DXE Services, or services produced by boot service drivers after ExitBootServices() is called.[1]

Chapter 4

GOP Configuration Driver

4.1 GOP vs. VBIOS

The Graphics Output Protocol (GOP) is enabled by UEFI driver to support graphic console output in the pre-OS phase.GOP is designed to be lightweight and to support the basic needs of graphics output prior to Operating System boot. The ultimate goal of GOP is to replace legacy VBIOS and eliminate VGA HW functionality.

In legacy VBIOS, INT10 and INT15 call scheme is used as a communication between platform and GPU. INT10 is a call from VBIOS to GPU and INT15 is a call from GPU to VBIOS. This INT communication scheme is messy and hard to maintain so it is been replaced by the UEFI protocol in GOP.

In GOP, UEFI Protocols are used as a communication link between GOP driver and GPU for the display functions. Each Protocol is identified by its GUID as discussed. In this, BIOS doesn't have to maintain the INT kind of call for display functionalities but it has to make things work based on the GUIDs of the protocol as per the need in EFI native mode.

Major differences between GOP driver vs. legacy VBIOS[6]

- Accessed through UEFI protocols vs. Interrupts and VGA/VBE interface
- Boot only services vs. both boot and OS run-time services
- Written in C instead of x86 assembler

Figure 4.1 and 4.2 show the Legacy boot mode and EFI Boot mode respectively. In Legcy Boot, VBIOS will take the responsibility of display functionalities whereas in EFI Boot, GOP will take do the same.

Compatibility Support Module (CSM) code is required for an implementation of the Intel Platform Innovation Framework for EFI. The CSM provides compatibility support between the Framework and traditional, legacy BIOS code and allows booting a traditional OS or booting an EFI OS.[8]

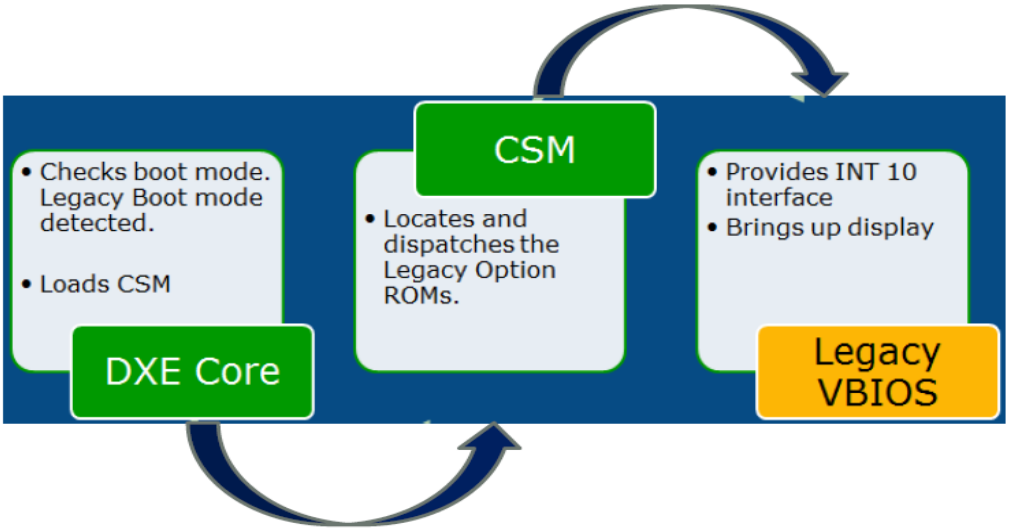


Figure 4.1: Legacy Boot

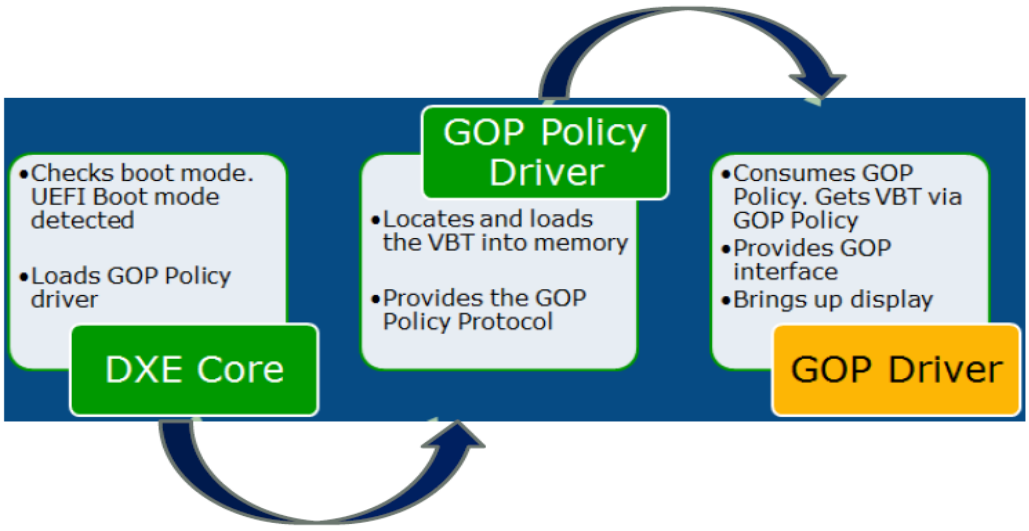


Figure 4.2: EFI Boot

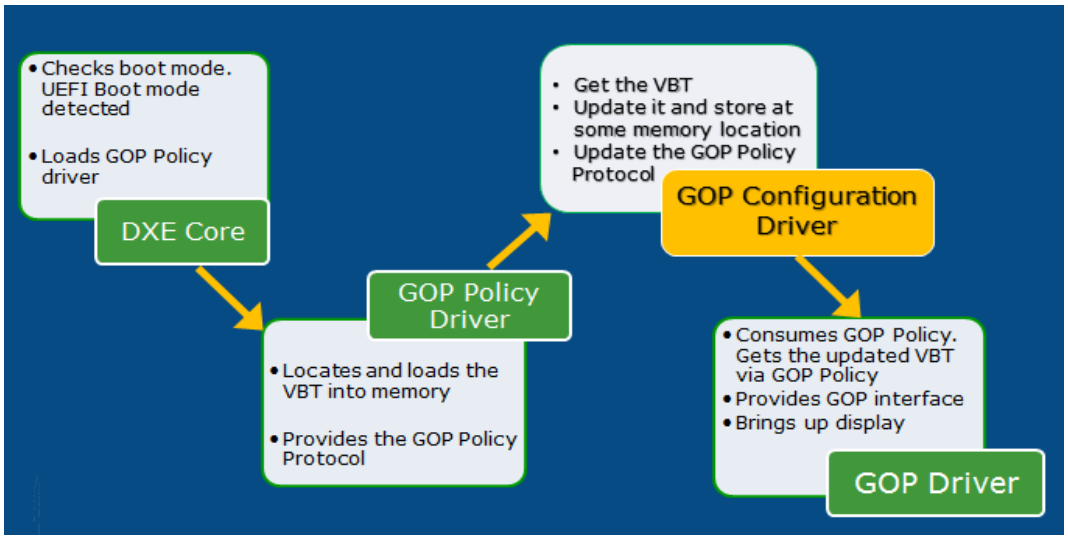


Figure 4.3: EFI Boot with GOP Configuration Driver

After the introduction of GOP Configuration driver in BIOS, the scenario shown in Figure 4.2 will be replaced as shown in the Figure 4.3

4.2 Objective of the Driver

As discussed, there is a GOP driver within BIOS code which is responsible for the display functionalities for Intel based mobile and desktop board. It provides the required display functionalities through configurable header which is a block of data. Configurable header is a binary file which gets generated by the Intel's Configurable tool. This tool has set up options for the display functionalities which can be modified manually and finally the configurable header will get generated for those particular display functionalities for a particular platform.To be generic, BIOS code have more than one headers and GOP driver will choose the correct header based on the platform.

The limitation with the GOP driver here is that there is only a one way communication between GOP and GPU. GPU to GOP communication is not there currently and that means GOP is enable to configure the header. So,*it is an objective of this project to make a two way communication between GOP and GPU by means of a GOP Configuration Driver which will configure the header according to the BIOS setup options for different platforms.*

4.3 Hardware/Software used

To develop the driver discussed here in the thesis, some sort of hard wares and soft wares require which is discussed here in this topic. Whole BIOS code is developed under the framework defined by the UEFI Specification and which require an IDE (integrated development environment) to develop the code.

Figure 4.4: DediProg SF600[10]

Hardware required here are:

- CRB (Customer Reference Board)

  This is the reference board used for the internal purpose of the organization and this will be provided to the OEMs (Original Equipment Vendor) like Dell, Lenovo etc. In this thesis, mobile board as a CRB is used throughout. This CRB contains is like a typical mother board used here for the internal testing purpose.

- Dediprog SF600

  It is the hardware which is used for flashing the BIOS in flash memory. The SF600 shown in figure 4.4 is a high speed "in System Programming" programmer to update the SPI Flash soldered on board (In Circuit Programming) or in the socket adaptor (Off line programming). The programmer is easily controlled by the computer DediProg Software through the USB bus offering friendly interface and powerful features to users.[10]

- USB to Serial Cable

  It is used to take a debug log for verification of the code.

Soft wares that required here are:

- Microsoft Visual Studio 2008

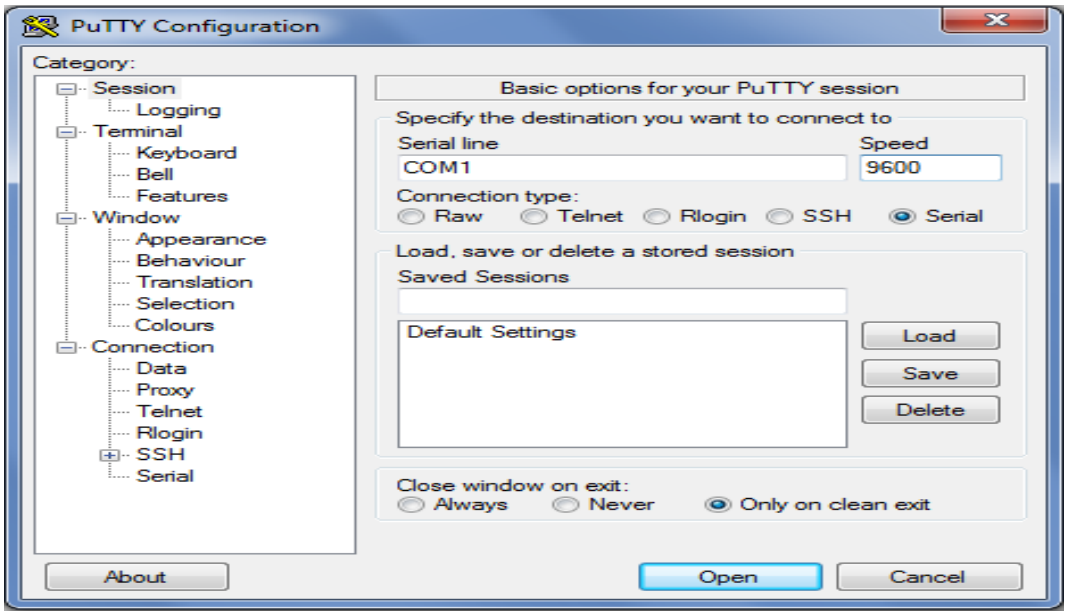


Figure 4.5: Putty Configuration Window

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop console and graphical user interface applications. Visual Studio supports different programming languages by means of language services, which allow the code editor and debugger to support (to varying degrees) nearly any programming language, provided a language-specific service exists. This IDE is used to develop a driver discussed here and also the whole EFI BIOS code. EFI BIOS code is arranged in different modules also called packages and it is the actual concept of EFI to make the code modular. The code is developed under the EDK environment which will be discussed later.[9]

- Putty

  It is a free and open source terminal emulator application which can act as a client for the SSH, Telnet, rlogin, and raw TCP computing protocols and as a serial console client. Purpose of putty here is to use as a serial console client. It is used to take a debug log on a particular COM port. Putty configuration window is shown below in Figure 4.4 [9]

EDK (EFI Development Kit)

The EDK (EFI Development Kit) is the open-source component of the "Framework", Intel's implementation of the EFI Specification, which was developed under the project code named "Tiano". The EDK is essentially a container for the Framework's Foundation code and sample drivers. The EDK is also a development kit for developing, debugging, and testing EFI and Framework drivers, EFI Option ROMs, and EFI Applications for use in the Framework environment.Currently the industry is migrating to the new EDK II which is a modern, feature-rich, cross-platform firmware develop-

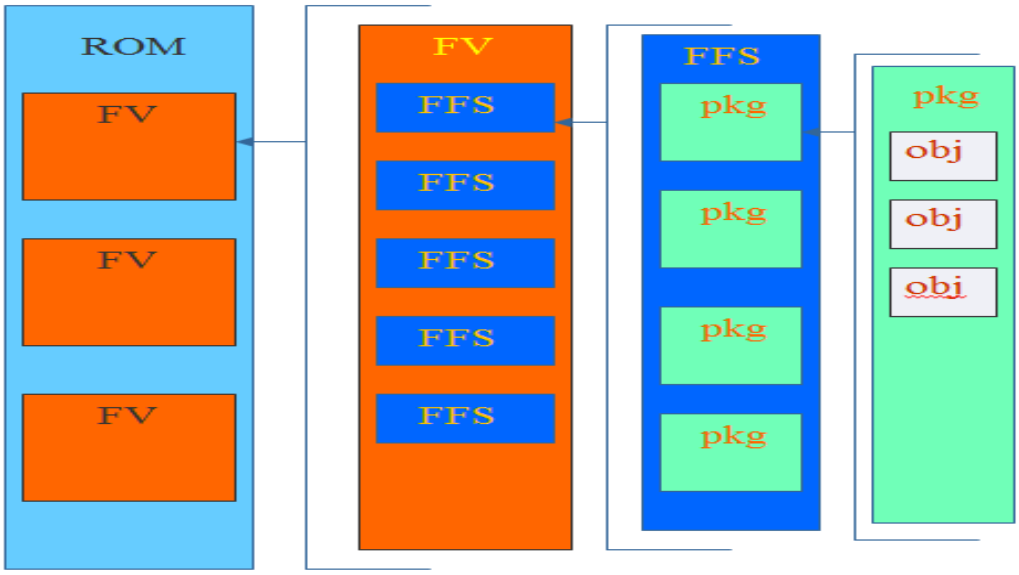


Figure 4.6: UEFI image system

ment environment for the UEFI and PI (Platform Initialization) specifications.

The driver discussed here is developed under the EDK environment using the Microsoft Visual Studio as an IDE. Using the MS visual studio as an IDE for BIOS development, one has to build the code to generate the .Rom image of BIOS. This image will reside in the SPI chip on the platform which is a Flash memory on the platform where the actual BIOS code sits in the form of .rom image. This can be understood from the Figure 4.5 which shows the UEFI image system.

UEFI specification defines the standardized format for EFI firmware storage devices (FLASH or other non-volatile storage) which are abstracted into "Firmware Volumes". Build systems must be capable of processing files to create the file formats described by the UEFI specification. The tools provided as part of the EDK II BaseTools package process files compiled by third party tools, as well as text and unicode files in order to create UEFI compliant binary image files.[7]

As shown in the Figure 4.5, object file created by compiling a BIOS code will be packed into different packages. These different packages will be combined into FFS called Firmware File System. Multiple EFI Sections are combined into a Firmware file (FFS) which consists of zero or more EFI sections. Each FFS consists of a FFS header plus the data.

Framework Firmware File System (FFS) is a binary layout of file storage for firmware volumes. It is a flat file system in that there is no provision for any directory hierarchy; rather, files all exist in the root directly. Files are stored end to end without any directory entry to describe which files are present.

A Firmware Volume (FV) is a file level interface to firmware storage. Multiple FVs may be present in a single FLASH device, or a single FV may span multiple FLASH devices. An FV may be produced to support some other type of storage entirely, such as a disk partition or network device.A firmware device is a persistent physical repository that contains firmware code and/or data. A single physical firmware device may be divided into smaller pieces to form multiple logical firmware devices and a logical firmware device is called a firmware volume.

Finally the .rom image will be created with multiple Firmware Volumes. This .rom image of BIOS code is going to be flashed in the SPI flash memory on the platform.

4.4 Flow Diagram

In this section, flow diagrams for the full fledge GOP configuration driver are presented to get a high level understanding of every step used to make this driver. Flow diagrams are divided in to three phases as below:

- Creating setup options in the BIOS Menu which are going to be configured in the configurable header. This is shown in Figure 4.7
- Creating the Policy Protocol which will be updated by the driver based on the setup options. This is shown in Figure 4.8
- GOP Configuration which will configure the header. This is shown in Figure 4.9

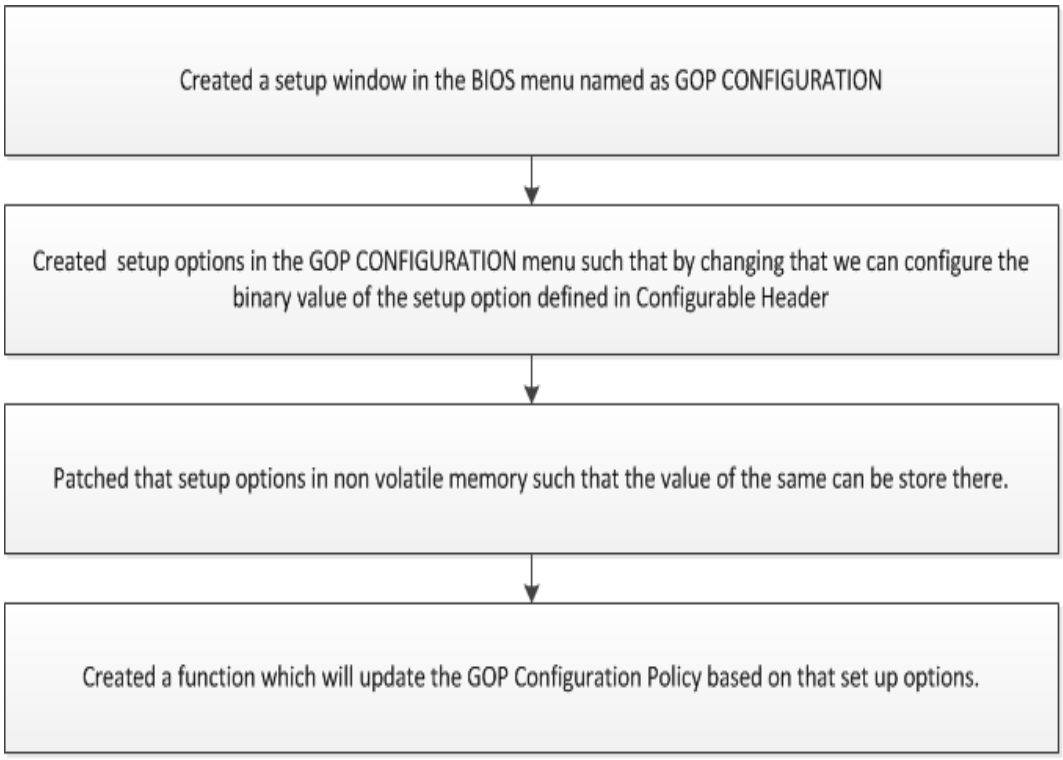


Figure 4.7: Flow Diagram to create setup options in BIOS

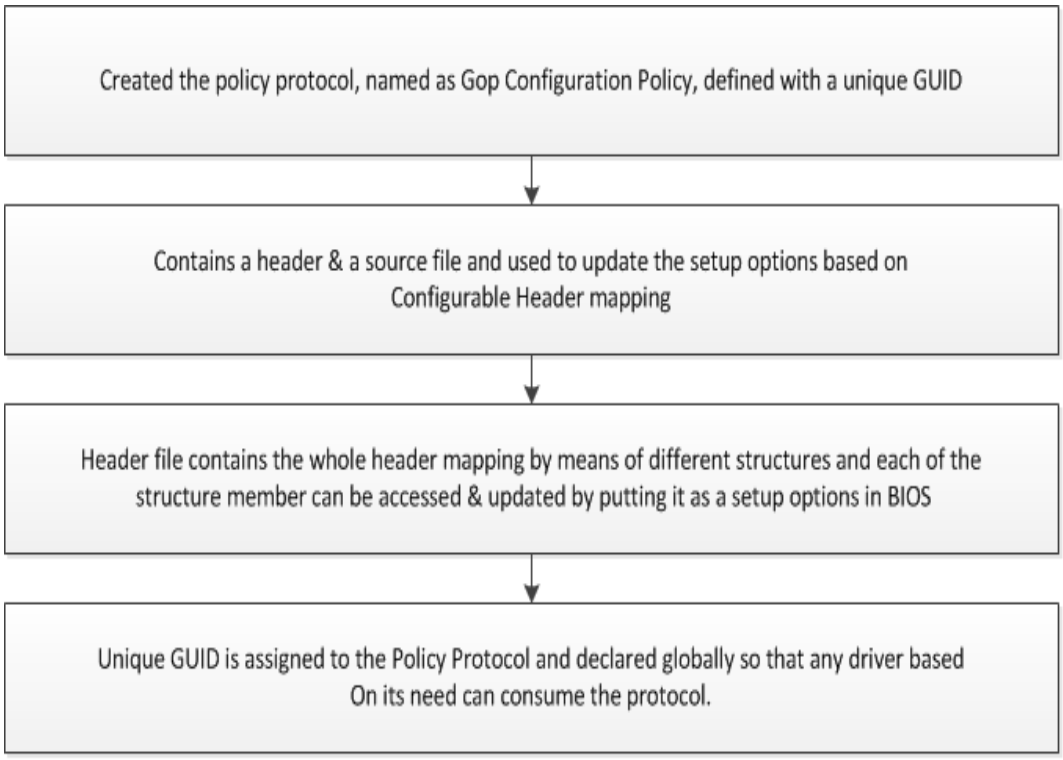


Figure 4.8: Flow Diagram of GOP Config Policy

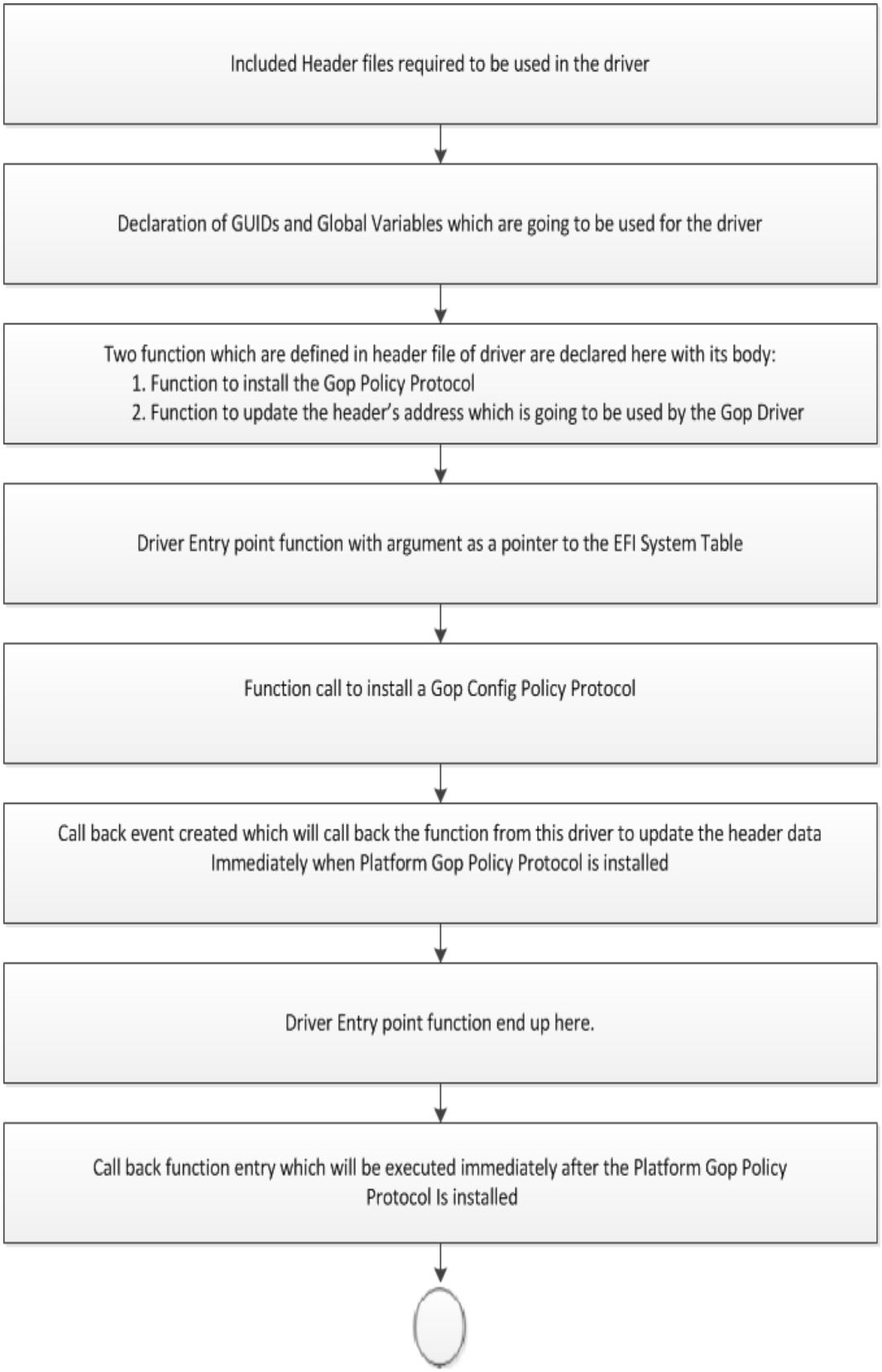

Included Header files required to be used in the driver
Declaration of GUIDs and Global Variables which are going to be used for the driver
Two function which are defined in header file of driver are declared here with its body:
1. Function to install the Gop Policy Protocol
2. Function to update the header's address which is going to be used by the Gop Driver
Driver Entry point function with argument as a pointer to the EFI System Table
Function call to install a Gop Config Policy Protocol
Call back event created which will call back the function from this driver to update the header data Immediately when Platform Gop Policy Protocol is installed
Driver Entry point function end up here.
Call back function entry which will be executed immediately after the Platform Gop Policy Protocol Is installed

Figure 4.9: Flow Diagram of GOP configuration Driver

4.5 Issues Faced and resolved

To make a full fledge driver is not so easy deal rather it's all depend on your experience, your understanding and your command over coding. There are some issued faced up to now to make this driver working as per the expectation.

This driver objective is to configure the header and put that updated header for the GOP driver to use for display. So it is obvious to make this driver executed before the GOP driver. Also GOP configuration driver has to get the original header and then have to update it so it should execute after the driver which is used to get the header from firmware volume.

Problem 1:Execution Order flow

Intention here is to make a GOP Configuration Driver independent so there is not any dependency expression for the driver. Because of this the driver is loaded sometime before Platform Gop Policy or sometime after that. Platform Gop Policy is the policy protocol which is used to get the header data from firmware volume. This policy protocol is used inside the driver to get the header data. To fulfil the driver objective execution flow ideally required in the BIOS is as follows:

- Platform Gop Policy
- Gop Configuration Driver
- GOP Driver which uses the configurable header.

As the Gop Configuration Driver is independent, it was not possible to get the exact flow all the time. that was the major problem faced during that time.

Solution 1:

To resolve this issue, callback event function inside the driver is created.Callback event function is used inside the driver where actual logic to update the configurable header data is kept. This callback event set here such that it will call the function immediately after the installation of Platform Gop Policy. And Gop Policy Protocol is added as a dependency expression for Platform Gop Policy. So the execution flow will always look like:

- Gop Configuration Driver
- Platform Gop Policy
- Callback event function which will update the header
- GOP Driver which will use the updated header.

So, thats how the problem of execution order was solved.

Problem 2: Porting a Driver into EDK II

After the driver implemented in EDK I, that has to be ported to EDK II also. The driver is already planned for the next Gen Intel Processor targeted in 2014. There are some changes needs to be done while porting the driver from EDK I to native. The major change is to make a new information file as per the EDK II INF file specification. Also the libraries for native implementation are different then of the EDK I. Problems here faced are to include the proper libraries because that was making errors of undefined or unexpected tokens used in the source code of driver. Problems also faced in the EDK II information file which was causing the linking errors because of the changed libraries in EDK II and also some other concepts which should be followed strictly in the EDK II fashion.

Solution 2:

To resolve the errors while porting, it was necessary to go through the EDK II INF file Specification. Form there the idea was clear how the INF file should be in EDK II fashion. Also searched for the libraries replaced in EDK II and tried with that and make the driver to work properly as it was in EDK I.

Other issues are like small things that were missed out and that lead to the failure of build for the BIOS code. These things were like not included the proper files, not given a proper path in the information file, declared the variable as global if that has to be used locally and also to where to put the function which update the policy for the driver etc..

4.6 Code walkthrough

In this section, the GOP configuration driver is discussed with the high level detail of all the files defined in the driver. But before discussing the driver into detail, its good to have a overview of the BIOS code packages and directories.

Whole BIOS code in is divided in to the different packages with some EDK I and some EDK II fashion. All the packages are described briefly in the Table.2 below.

In Figure 4.10, the actual driver hierarchy in the BIOS code is showed. The driver contains the different files and directories are also discussed briefly in this section. That will clear the idea about the working and distribution of the driver.

	Package	Description
1	Basetool	Provides build related tools for both EDK and EDK2. Also contains miscellaneous tools such as ECC and EOT
2	PlatformPkg	Intel's Platform specific code (all in Native).
3	RefcodePkg	Intel's Silicon specific Reference code.
4	Build	Created when source is built and contains binary images.
5	CryptoPkg	Several security features were introduced (e.g. Authenticated Variable Service, Driver Signing, etc.)
6	EdkCompatibilityPkg	Provides header files and libraries that enable you to build the EDK module in UEFI 2.0 mode with EDK II Build.
7	FrameworkModulePkg	Intel Framework Module Package contains the definitions and module implementation which follows Intel's UEFI & EFI Framework Spec.
8	FrameworkPkg	This package provides definitions and libraries that comply to Intel's UEFI & EFI Framework Specifications.
9	MdePkg	provides all definitions(including functions, MACROs,structures and library classes) and libraries instances, which are defined in MDE Specification. It also provides the definitions (including PPIs,PROTOCOLs,GUIDs). EFI1.10,UEFI2.3.1,PI1.2 and some Industry Standards.
10	MdeModulePkg	This package provides the modules that conform to UEFI Industry standards.
11	NetworkPkg	Provides IPv6 network stack drivers,IPsec driver, PXE driver, iSCSI driver and necessary shell applications for network configuration.
12	PcAtChipsetPkg	This package is designed to public interfaces and implementation which follows PcAt defacto standard.
13	PerformancePkg	Add performance measurement capability to code.
14	R82014CrescentBayPlatPkg	Intel's platform specific reference code (not all in Native).
15	RomImages	Created when source is built and contains the .rom & .bin images of BIOS.
16	SecurityPkg	This package provides functionalities like TPM, User identification (UID), secure boot and authenticated variable.
17	SourceLevelDebugPkg	This package contains the source code to include the Source code debugger agent into a platform.
18	UefiCpuPkg	This Package provides UEFI compatible CPU modules and libraries.

Table 4.1: Intel's BIOS code with different Packages

Now here the driver is discussed in the detail by explaining the each of the file and directory briefly. The main folder created for the driver under BIOS code is named as "GopConfig".

GopConfig

- This folder is a complete package of a full fledge DXE GOP Configuration Driver for BIOS.
- This contains two sub folders named DXE and GopConfigPolicy and one library file named as GopConfigLib.inf
- Details about all the folders and file are discussed here on a high level.

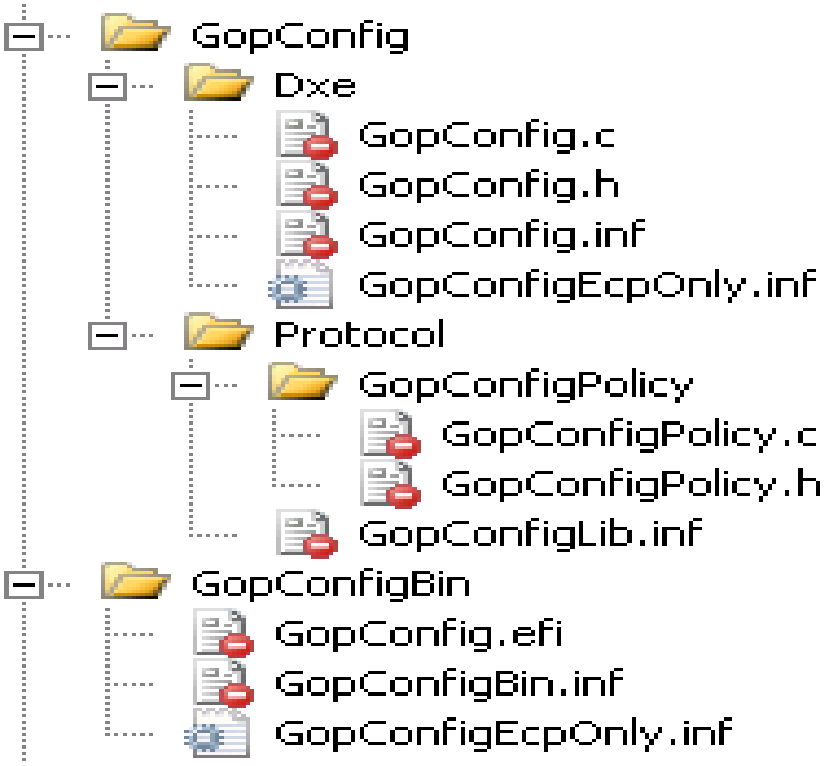


Figure 4.10: Driver Hierarchy

GopConfigPolicy

This folder contains a policy protocol which the GopConfig DXE driver uses. GopConfigPolicy folder contains two files.

1. **GopConfigPolicy.h**

- This is a header file in which a GUID is given to the Protocol and declared that protocol as an extern so that other driver can use it through its GUID.

- This file contains a whole header structure. This header structure is divided into different small structures based on the different functionalities .
- Apart from the header structure, this file contains the defined protocol structure which contains the different variables which are mapped to the setup option variables created in the BIOS setup menu.

2. GopConfigPolicy.c

- In this file, protocol GUID is declared as a global variable which will be used to refer the protocol for installing, uninstalling and any other function related to that.

GopConfigLib.inf

This is an information file used for the GopConfigPOlicy Protocol and it will create a library for the protocol so that other driver can use this protocol by including this library file.

DXE

1. GopConfig.h

- This is a header file in which the GopConfigPolicy.h and other header files are included such that driver can use the EFI defined BaseTools and library sets.
- Functions which are going to used in the driver source file GopConfig.c are declared here like function to Install the GopConfigPolicy Protocol, header callback function, function to get the header etc...

2. GopConfig.inf

- This is an information file in EDK II fashion which tell about the BASE_NAME of the driver, INF version, File GUID which is nothing but the driver's GUID, Module type as DXE_DRIVER as it is a DXE driver, Driver entry point etc... This section in the file is named as [Defined] section.
- This file tell about the sources to be built like GopConfig.c and Gopconfig.h here. Also it has sections which tell about the packages it is using to refer some protocol and libraries.
- Also the protocols the driver consumes or produces are declared here.For this driver PlatformGopPolicy protocol and GopConfigPolicy protocol are declared here.

- If any dependency is there that can be defined directly in to this file by declaring its GUID under the [Depex] section. There should not any separate file created for the dependency in EDK II. Here the driver is totally independent so there is not any dependency declared here.

3. GopConfigEcpOnly.inf

- This is an information file in EDK I fashion which is almost different than EDK II. Here instead of declaring a packages like in EDK II, hard coded paths are given for the protocol and other libraries used by the driver.
- This file defines a driver entry point and dependency source for the driver. It also defines a file GUID and component type like boot service driver, run time driver etc. Here the component type defined is BS_DRIVER that means it is a Boot Service Driver.In EDK II fashion the component type is classified in a more specific way than EDK I.
- Dependency file should be defined separately in EDK I fashion and it has to be declare in the information file of that driver as a DEPEX_SOURCE. Here there is no dependency for this driver is declared or defined in the information file as the driver is independent.

4. GopConfig.c

- This file contains almost all the functionalities of driver. In this file all the required header files are included and also the protocols GUIDs are declared which would be used in the driver locally or globally.
- As the driver is using the GopConfigPolicy Protocol, there is a function defined here which will be called by the driver initialization function to install the protocol. The other protocol, PlatformGopConfig Protocol is used here to get the header so this protocol is located here before using it inside the driver.
- Driver entry function will call the function to install the GopConfigPolicy protocol and then an event is created which will call the callback function where the header data is getting updated.
- The event is created such that it will call the callback function immediately after the installation of the PlatformGopPolicy protocol occurred.
- In callback function, after locating the PlatformGopPolicy Protocol, the driver will get the header using Get header data function declared in GopConfigPolicy protocol and defined in PlatformGopPolicy protocol, copy the header to some memory location and get the pointer to the new copied header.
- The copied header then updated using GopConfigPolicy Protocol for the defied setup options i BIOS menu as well as variables that are patched to that inside the policy protocol.

- After updating the header data, PlatformGOPPolicy protocol will be updated such that the GOP driver will use the configured header instead of the old header.

GopConfigBin

This folder is for binary implementation of the GOP Configuration Driver for BIOS. This contains binary file of the driver along with its information files for EDK I & EDK II. Each of the file it contains is discussed briefly below.

1. GopConfigBin.efi

- This is a binary file (.efi) of the driver.
- This file is being generated under the "Build" folder from the complete package of the driver (including all the files of driver) when the code is built
- When the source file of the driver is complied and built, it will generate object file, .efi file and other files with that.
- Binary file (.efi) is copied from the "Build" folder into the "GopConfigBin" folder for binary implementation of the driver.

2. GopConfigBin.inf

- This is a EDK II information file for the binary driver. Here the same BASE_NAME, Component type, Module Type and Driver entry point is defined under the [Define] section as of the GopConfig.inf file except the INF version.
- As this the binary implementation of the driver, as a source file GopConfig.efi is declared under the [Source] section.
- Also this binary file (GopConfig.efi) is nothing but the driver itself which is already built, there is no any packages or protocol or any libraries should be defined as in the case of GopConfig.inf.

3. GopConfigBinEcpOnly.inf

- This is a EDK I information file for the binary driver. Here also the same BASE_NAME, Component type, Module Type and Driver entry point is defined under the [Define] section as of the GopConfigEcpOnly.inf file except the INF version.
- Similar to the GopConfigBin.inf, because this is the binary implementation of the driver, GopConfig.efi is declared under the [Source] section.
- Also this binary file (GopConfig.efi) is nothing but the driver itself which is already built, there is no any hard coded path for protocol or any libraries should be defined as in the case of GopConfigEcpOnly.inf.

Chapter 5

Driver Implementation

Basically the driver is written in C language using Microsoft Visual Studio as an IDE. The driver is implemented in two environments and also with the two different modes.

Two different implementation of the driver here are:

- EDK Environment, also named as EDK I Environment
- EDK II Environment also called as Native mode.

And two different modes in each of the environments are:

- Driver with a source file, header file, information file etc.
- Driver with the binary file (.efi file) and the information file for that only.

5.1 EDK I Overview[11]

The EDK is the open-source component of the "Framework", Intel's implementation of the EFI Specification, which was developed under the project code named "Tiano".

The EDK is essentially a container for the Framework's Foundation code and sample drivers. The EDK is also a development kit for developing, debugging, and testing EFI and Framework drivers, EFI Option ROMs, and EFI Applications for use in the Framework environment.

5.2 EDK II Overview[11]

EDK II is a modern, feature-rich, cross-platform firmware development environment for the UEFI and PI specifications. The EDK II project is the response to the EFI community's request for a better build and version tracking environment for UEFI and PI development. The main difference between the EDK II to the original EDK is the Enhanced Build Environment of the EDK II. The advantages of the Enhanced Build Environment include:

- Operating System independence
- Flexibility in choosing the compiler and assembler tools
- The ability to generate working code using open source build tools and applications
- Enhanced development and build capability of modules and module packages
- Use of build configuration tools and data sets to provide flexible process
- Online source control allows users to contribute code and become participants.

The EDK II enhanced build environment is a significant departure from the build environment of the original EDK. There are many new concepts and features in the EDK II, which have altered the environment. If you are familiar with the original EDK these changes will be obvious, while the benefits of them will be apparent to everyone. The new structure, along with package owners is documented in the EDK II Packages.

It is important to note, that the compiled results of the EDK II are equivalent to the original EDK, the changes are in the build environment and only affect the sources at that level. Any differences in the code files are only to support the changes in the build environment, once the modules are created, they are functionally identical.

The EDK II is classified at a development level project and the EDK is an official level project. The EDK is still intended to be used for volume production and shipments while the EDK II is being further refined through additional development.

5.3 Difference between EDK I & EDK II[11]

Mainly the build architecture is very different. Parts of the differences are that the build description files (.dsc .inf, etc) have been enhanced and are different. However, the EDK II build understands EDK build description files so that EDK II build can include EDK source code. There are also build tool differences and EDK II supports more OSes and more tool chains. Other differences besides the build include that EDK II has different, richer libraries (MDELIB, etc) which makes some of the source code very different. EDK II also has the package concept so that the directory and file layout is different. EDK II also uses Platform Configuration Database (PCD) for parameterization and fix-up binary support etc. EDK II supports newer UEFI/PI specifications than EDK. In addition, EDK II has compatibility with EDK style sources through the EdkCompatiblityPkg (ECP). This is possible because the ECP will have binary compatibility for EDK through its libraries and thunk code. Additionally, EDK II is designed to work with Doxygen to generate design level specifications.

In addition, since EDK II supports the later versions of the UEFI and PI Specifications, there are newer protocols that will be part of EDK II that do not exist in

EDK. Thus if there is a desire to use the latest protocols there may be a need to use EDK II instead of EDK.

Differences Summary:

- Build infrastructure and Build tools
- Packages / directory structure
- Rich Libraries
- Platform Configuration Database (PCD)

5.4 Similarities within EDK and EDK II Environment

The similarity is that code as far as the Platform Initialization boot execution phases are similar. EDK and EDK II have similar Sec-PEI-DXE-TSL and runtime phases. Also any of the protocol interfaces that are supported in both EDK and EDK II will be the same. Thus since EDK II understands EDK build description files the EDK II build can include EDK source code though the use of the ECP.

5.5 Implementation using Flags

In the BIOS code the driver is implemented in both of the environment using both of the modes by means of flags.The Flags used in the code are:

- ECP_ENABLE Flag
- GOP_BIN Flag

The purpose of the ECP is to provide backwards compatibility for EDK I style source modules that assume UEFI 2.0/Framework 0.9. The ECP uses a number of techniques that allow these EDK I source modules to execute correctly when run on top of the EDK II. This preserves customers' existing investments in EDK I modules by allowing them to be used "as is" in EDK II based firmware. This also allows customers to plan when/if their exiting modules will be ported to EDK II without the use of ECP.

ECP_ENABLE

ECP Flag is used to tell whether the driver is building in EDK I or in EDK II environment. To build the driver, information file has to be defined in the description file (.dsc) and firmware volume definition file (.fdf). ECP flag is defined such that it will modify the information file name based on the value it has. If it is true it will add prefix "EcpOnly" at the end of a name of information file the build tool will take the EDK I information file from the source. If the ECP flag is false, it will not add any prefix to the name of the information file and the build tool will take the EDK II

information file from the source. So driver in the BIOS code now have two different information file in its folder. One for EDK I with the prefix "EcpOnly" and the other with the same name but without prefix.

GOP_BIN

GOP_BIN flag is used to tell whether the driver will be built from the source and header files defined for that or directly the binary file (.efi) of the driver will loaded. The concept behind this is to not give the driver code to the OEM (Original Equipment Vendors) like Dell, Lenovo etc To do this, instead of the whole driver source code only the binary file of the driver will be included in the Reference code which will be given to the OEMs.

When the BIOS code is built with the driver source code, different files (like .obj, .efi, .lib etc...) will be generated for that. Form that .efi file is the binary file of the driver. That .efi file of the driver will be copied from that to the other folder then of the actual folder of the driver and an information file for that will be created. Using that information file the binary implementation is possible as that information file will provide the information to the build tool that it has to take the .efi file of the driver directly instead of building the driver source code.

Chapter 6

Conclusion & Future Scope

6.1 Conclusion

This thesis shows that the GOP configuration driver for BIOS will eliminate the need of Intel's configurable tool to some extent which is used to configure the header manually. GOP Configuration driver in DXE phase will take care of configuring the header as per the requirement for platform.

This driver will get the header, copy it to some memory location, get the pointer of the copied header, update the copied header according to the need, update the Platform Gop Policy such that GOP driver will now use the updated copied header for the display functionalities. That's how this driver makes two way communications possible between GOP and GPU.

The driver is developed using EDK I & EDK II environment and driver is implemented directly using binary file and also using the complete package with source file, header file and other files required to load the driver. This implementation is accomplished using flags. Binary implementation is targeted for the OEM (Original Equipment Vendor) who uses the Intel's BIOS reference code but not able to see the driver code.

6.2 Future Scope

Driver is ready with some setup options in BIOS menu which are same as in Intel's Configurable tool to configure the header data. One can expand the driver by adding more setup options in the BIOS menu as per the requirement from OEMs.

In future if the header would be enhanced by some more data, one has to read the script file to understand and check for the modification done and then accordingly has to change the offsets used to get the every variable or data of header which are being modified in the driver.

References

[1] Unified Extensible Firmware Interface Specification Version 2.3.1, April 6, 2011.

[2] Driver Writer's Guide for UEFI 2.3.1, Version 1.01, August 2012.

[3] Advanced Configuration and Power Interface Specification, Revision 5.0, December 6, 2011

[4] UEFI Overview by Michael A. Rothman May 16th 2007.

[5] Intel Platform Innovation Framework for EFI Driver Execution Environment Core Interface Specification (DXE CIS), Version 0.9, September 16th, 2003

[6] Replacing VGA, GOP implementation for UEFI, UEFI Summer Plug fest-July 6-9, 2011Presented by AMD.

[7] EDK II Build Specification, June 2012.

[8] Intel Platform Innovation Framework for EFI Compatibility Support Module Specification, Revision 0.97, September 4, 2007

[9] http://en.wikipedia.org

[10] http://www.dediprog.com/SPI-Flash-Programmer/SF600

[11] http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=Welcome