Design Specifications for UEFI Applications and Diagnostics

Kadin Brooks, Ivaylo Kozhuharov, Halla Tuaum

Sponsor: Intel

By Kadin Brooks kadinb@uw.edu

REVISION HISTORY

Date	Version	Sections	Editor
10/23/2022	Version 0.1	All	Kadin Brooks
10/30/2022	Version 0.2	Introduction	Kadin Brooks
10/30/2022	Version 0.3	1,3	Kadin Brooks
10/30/2022	Version 0.3	1,3	Halla Tuaum
10/30/2022	Version 0.3	1,3	Ivaylo Kozhuharov
11/27/2022	Version 0.4	1, 3, 4, 6	Kadin Brooks
11/27/2022	Version 0.4	1, 3, 4, 6	Halla Tuaum
11/27/2022	Version 0.4	1, 3, 4, 6	Ivaylo Kozhuharov
1/22/2023	Version 0.5	2,3,4,6	Kadin Brooks
1/22/2023	Version 0.5	2,3,4,6	Ivaylo Kozhuharov
2/19/2023	Version 0.6	5,6	Kadin Brooks
2/19/2023	Version 0.6	5,6	Ivaylo Kozhuharov
2/26/2023	Version 0.6	2, 8	Kadin Brooks
4/9/2023	Version 0.7	5	Kadin Brooks
4/9/2023	Version 0.7	5	Ivaylo Kozhuharov

Table of Contents

1. Introduction

The purpose of this design specification document is to be a reference for the design and implementation of Universal Extensible Firmware Interface (UEFI) applications and diagnostic tools generated by our senior design team for our sponsor, Intel Corporation. More specifically, this document contains information about the design and implementation of UEFI shell applications for CPU feature enumeration, memory topology, and SMM handler latency and size.

The UEFI describes an interface between the operating system and the platform firmware [3]. The UEFI was preceded by the Extensible Firmware Interface (EFI). The UEFI is in the form of data tables that contain platform related information and boot and runtime service calls that are available to the operating system loader and the operating system. The UEFI defines the set of interfaces and structures that platform firmware must implement in addition to the set of interfaces and structures that operating systems may use in booting.

The intended audience of this document includes engineering and computing professionals who are interested in operating systems, system firmware, and computer architecture. More specifically, it includes firmware engineers working as part of the Tianocore open source repository. The scope of design contained in this document is largely constrained to the UEFI shell, the x86 family of instruction set architectures, and Intel Core processors.

2. System Overview

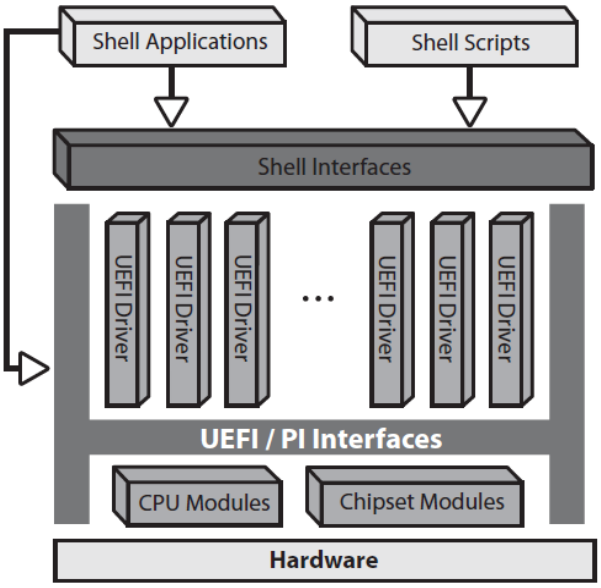

Figure 1. Architectural Relationship Between UEFI Shell and Underlying System Firmware and Hardware

The UEFI Shell is a program that uses a series of interfaces known as UEFI [2]. The programs that manifest these interfaces are based upon UEFI drivers and the system firmware. The main purpose of these programs is to initialize the state of the computer for the operating system. The UEFI Shell functions in an environment between the UEFI and applications. Applications can use the UEFI Shell APIs and other UEFI APIs. The UEFI is written using the system Platform Initialization (PI) interfaces.

The UEFI Shell is built around a platform running a BIOS that is UEFI compliant. The UEFI standards organization publishes the UEFI and PI specifications that drive the underlying architecture of the BIOS. The UEFI Shell is considered to be a BIOS extension.

This project will deliver a set of UEFI Shell applications that will diagnose the state and performance of the system CPU, Memory, and the SMM Interrupt Handler. This project is intended to function within the larger set of applications contained in the UEFI API.

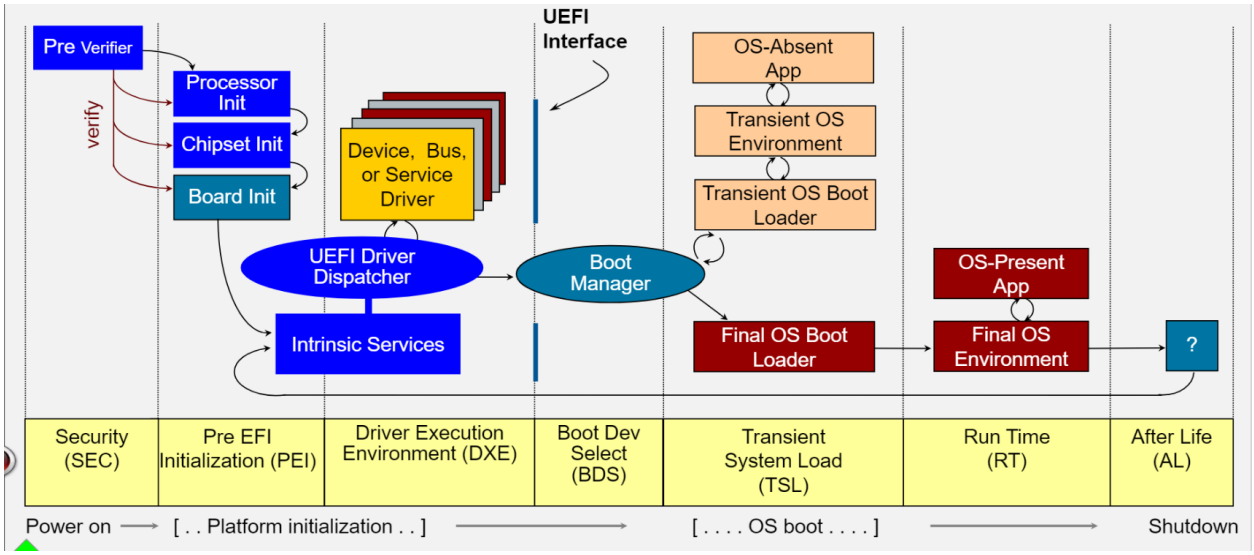

Figure 2. Role of UEFI in Platform Booting Sequence

PI compliant system firmware must incorporate six phases: security (SEC) phase, pre-efi initialization (PEI) phase, driver execution environment (DXE) phase, boot device selection (BDS) phase, run time (RT) services, and after life (AL) phase [4].

The security (SEC) phase is responsible for handling all platform restart events, creating a temporary memory store, serving as the root of trust for the system, and passing information to the pre-efi initialization (PEI) phase.

The pre-efi initialization (PEI) starts after the SEC phase. Any machine restart event will invoke the PEI phase. The PEI phase initially operates with the platform in a nascent state, leveraging only on-processor resources such as the processor cache as a call stack for dispatching pre-efi initialization modules (PEIMs). The pre-efi initialization modules are responsible for initializing some permanent memory complement, describing the memory in hand off blocks (HOBs), describing the firmware volume locations in HOBs, and passing control to the driver execution environment (DXE) phase.

Prior to the DXE phase, the PEI phase is responsible for initializing permanent memory in the platform so that the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position independent data structures called HOBs. The DXE phase includes several components consisting of the DXE foundation, DXE dispatcher, and a set of DXE drivers.

The DXE phase will pass control to the boot device selection (BDS) architectural protocol after all DXE drivers whose dependencies have been satisfied have been loaded and executed by

the DXE dispatcher. The BDS phase is responsible for initializing console devices, loading device drivers for the boot device, and loading and executing boot selections.

The transient system load (TSL) phase is primarily the OS vendor provided boot loader. The UEFI shell is executed during this phase. The runtime (RT) phase is the final OS environment where OS present applications are executed. Both the TSL and RT phases may allow access to persistent content via UEFI drivers and UEFI applications.

The after life (AL) phase consists of persistent UEFI drivers that are used for storing the state of the system during OS shutdown or restart sequences.

Tianocore EDK II is a reference implementation of UEFI by Intel. EDK stands for EFI Development Kit and is developed by the open source Tianocore community. The EDK II build environment must support development workstations running Microsoft Windows operating systems, Linux operating systems, or Apple Mac operating systems. In addition multiple compiler tool chains from Microsoft, Intel, and GCC must be supported. All provided source code must be POSIX compliant.

UEFI and PI specifications define the standardized format for EFI firmware storage devices (flash based devices and other non-volatile storage devices) which are abstracted into firmware volumes.

A firmware volume (FV) is a file level interface to firmware storage. Multiple firmware volumes may be present in a single flash device or a single firmware volume may span multiple flash devices. In all cases, a firmware volume is formatted with a binary file system. The file system used is typically the firmware file system (FFS) but other file systems may be used. All UEFI modules are stored as files in the format of firmware volumes. Some modules may be executed in place (linked at a fixed address and executed from the ROM) while other modules may be relocated when they are loaded into memory.

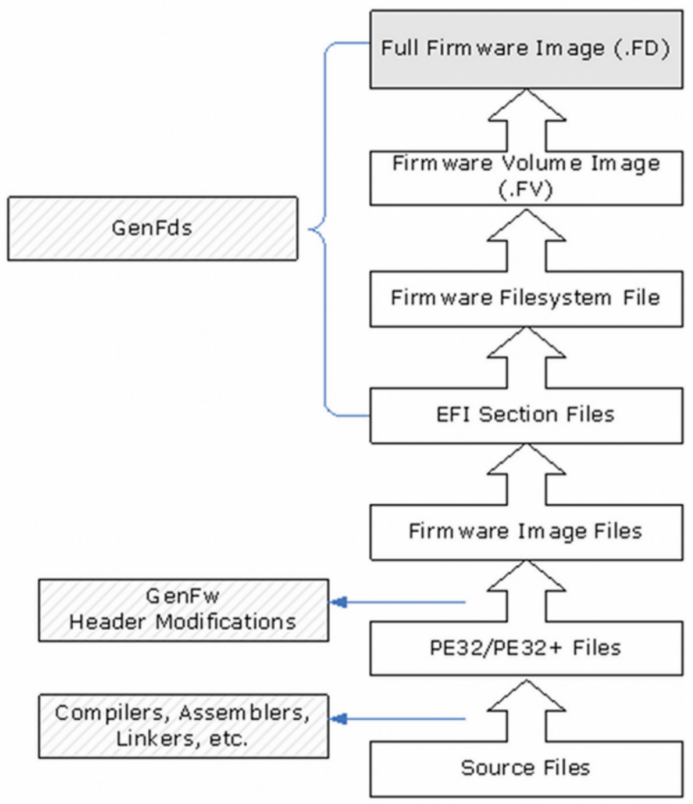

Figure 3. UEFI/PI Firmware Image Creation

There are several layers of organization in the development of a full UEFI/PI firmware image. Source files must undergo a sequence of processing steps to become a full firmware image. Figure 3 depicts the processing steps and implementation tools used to convert source code files to their corresponding firmware image.

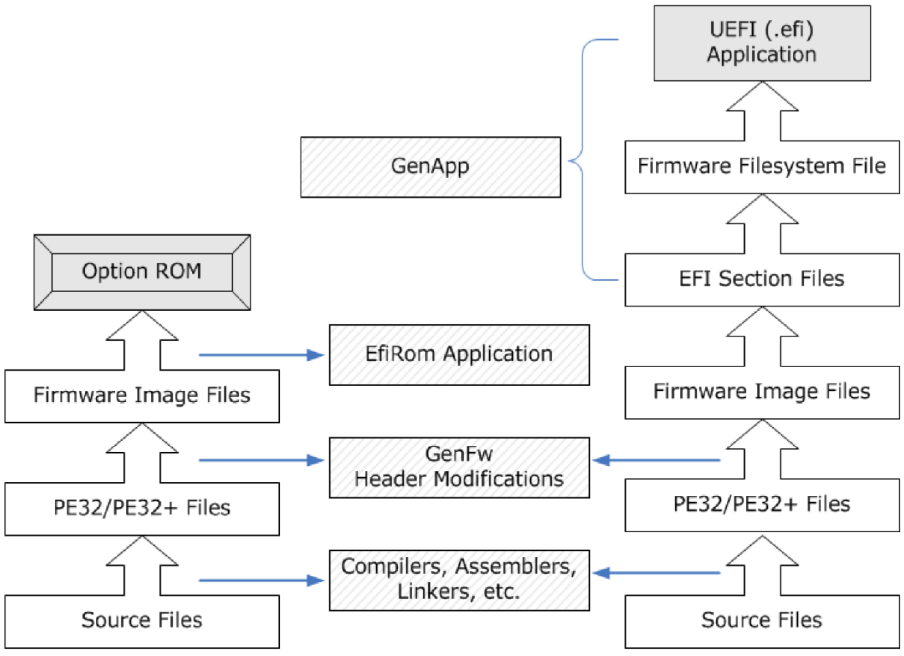

Figure 4. EFI Option ROM and UEFI Application Creation

In addition to creating images that initialize the complete platform, the firmware volume build process also supports the creation of standalone EFI applications such as OS loaders and Option ROM images that contain driver code. Figure 4 depicts the processing steps and reference implementation tools for converting source files to corresponding UEFI applications and Option ROM images.

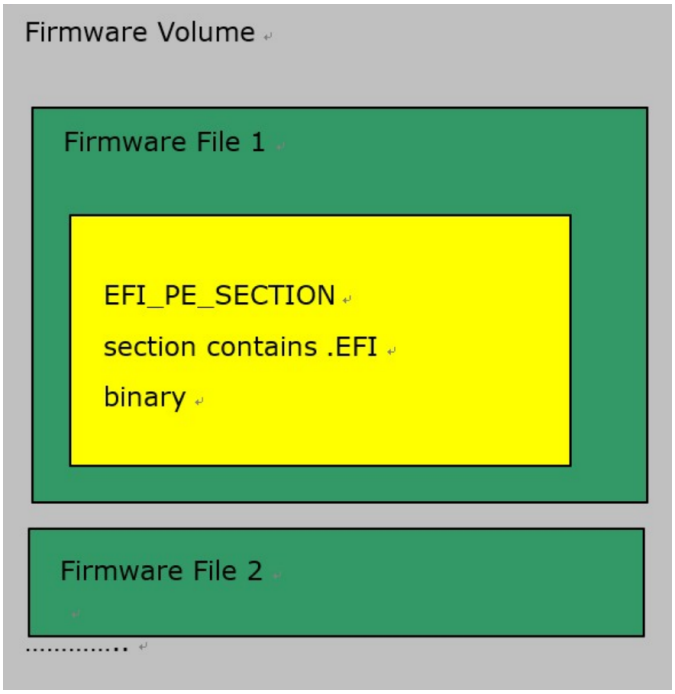

Figure 5. Firmware Volume and UEFI Module Format

UEFI modules are the smallest piece of separately compilable code or prebuilt binary. It contains a metadata file (INF) plus source code or binary. The INF file is required by the UEFI build system to describe the module's behavior.

A UEFI package is a group of zero or more modules. A package must contain a package metadata file (DEC) and possibly a platform metadata file (DSC).

A UEFI platform is a special type of package with additional metadata files. A package must contain one DSC file and zero or one FDF file. The FDF file is only required if flash output is required.

UEFI modules consist of source files or binary files and a module definition (INF) file. A INF file describes a module's basic information and interfaces. Typical UEFI modules are firmware components that are built, placed in a FFS file, and then placed into a FV image. Typically components may be a driver or application which are built into a *.efi binary file and placed into the FFS file as EFI_PE_SECTION, raw binary data, an option ROM driver that is placed into a device's option ROM, a standalone UEFI driver, a standalone UEFI application, or a library instance that is built to a library object file *.lib and statically linked to another module.

The UEFI defines many different module types. Module types are used to indicate the phase of execution of a module, the binary image generation for different types of modules, indicate the EntryPoint() or Constructor() API for different module types, and indicate the suitable library instance for different module types.

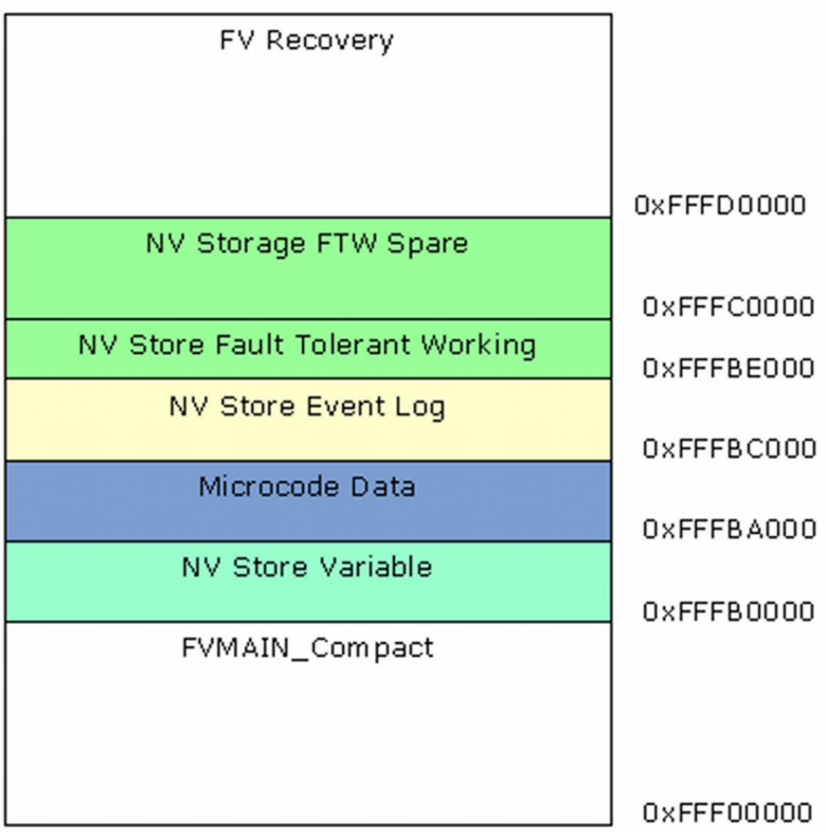

Figure 6. Typical IA32/X64 Flash Device Layout

All code starts as either C sources and header files, assembly sources and header files, UCS-2 HII strings in Unicode files, Virtual Forms Representation files, or binary data files. Per UEFI and PI specifications, the C and Assembly files must be compiled and linked into PE32/PE32+ images.

While some code is designed to be compiled and run from ROM, most UEFI and PI modules are written to be relocatable. Some modules also permit dual mode where it will execute from memory only if memory is available, otherwise they will execute from ROM. Source code is assembled or compiled and then linked into PE32/PE32+ images. The binary executables are converted into EFI firmware file sections. Each module is converted into an EFI section consisting of a Section header followed by the section data (driver binary).

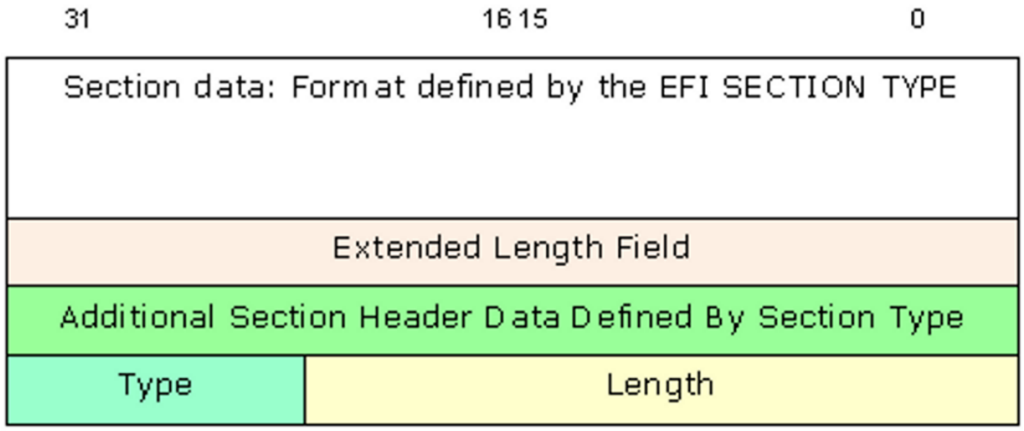

Figure 7. General EFI Section Format

Multiple EFI sections can be combined into a Firmware file (FFS) which consists of zero or more EFI sections. Each FFS consists of a FFS header plus the data.

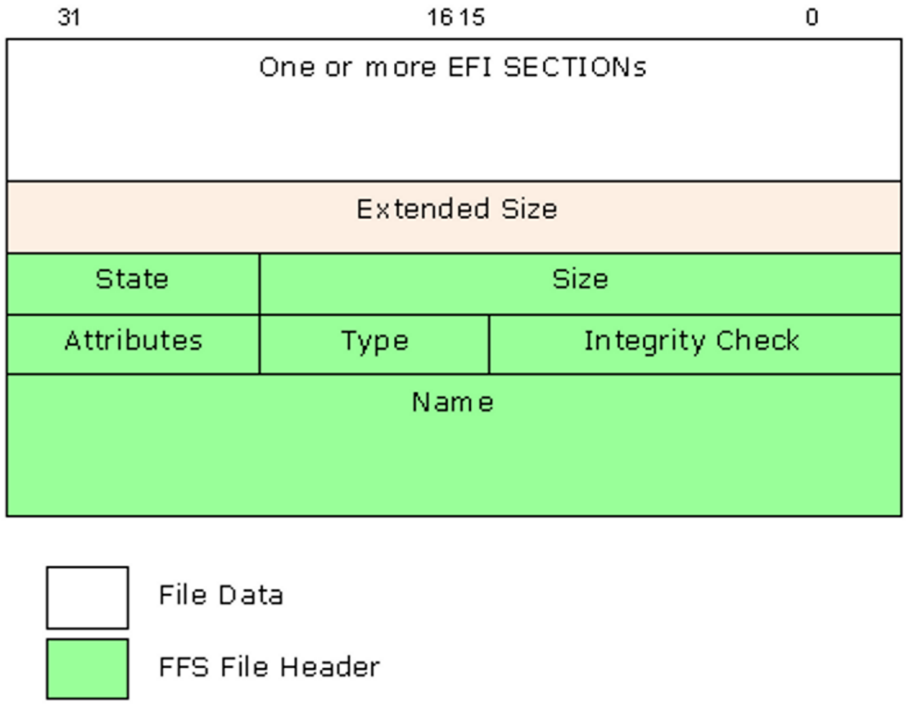

Figure 8. Firmware File (FFS) Layout for files larger than 16 Mb

One or more FFS files are combined into a Firmware Volume (FV). The format for a firmware volume is a header followed by an optional extended header, followed by zero or more FFS files.

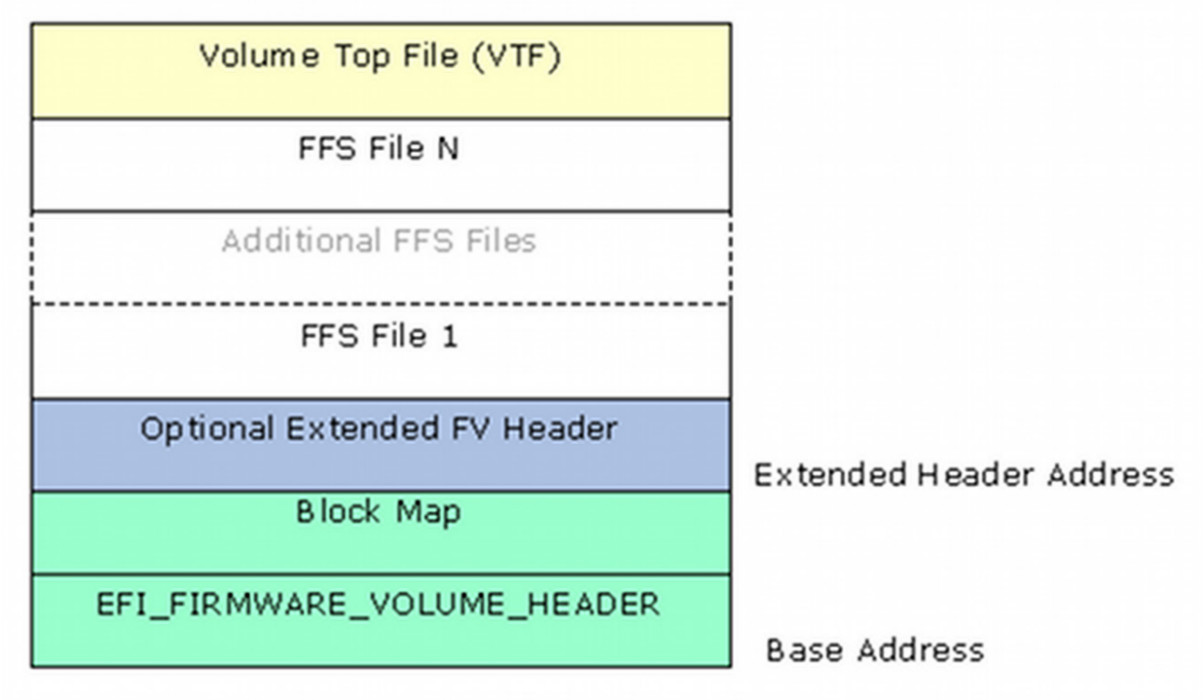

Figure 9. General Firmware Volume Layout

Creating EFI Images

EDK II modules include both libraries, drivers, and applications. Library modules are compiled and linked as static libraries. Drivers and applications are compiled to object files then linked with the static libraries they require. After the static image has been created, the resulting image is run through the dynamic linker to generate the relocatable binary images (DLL). All EFI images must be formatted PE32/PE32+/COFF.

Since UEFI/PI images are not standard executables, these dynamically linked (DLL) files must be processed to become UEFI/PI compliant images. This processing involves replacing the standard header with an EFI header that reflects the EFI_SECTION type. Prior to creating the EFI section files, PEI Foundation and PEIM images may be processed into either a terse image or have the .reloc section removed (for images that will always execute directly from ROM).

Creating a Terse Image

The DOS, PE, and optional headers must be replaced with a minimal header. The TE header will have a signature of “VZ”. Per the PE/COFF specification, at offset 0x3C in the file is a 32-bit offset from the start of the file to the PE signature which always follows the MSDOS stub. The PE signature is immediately followed by the COFF file header.

After verifying the DOS header’s magic number (0x5A3D), the PE signature is verified and the Machine type is obtained from the optional header’s subsystem field.

As a last step before creating the image, the COFF header specifies the value of the NumberOfSections in the file which needs to be backed into a single byte of the TE header.

After the header is created, then the rest of the original image is appended to the TE header.

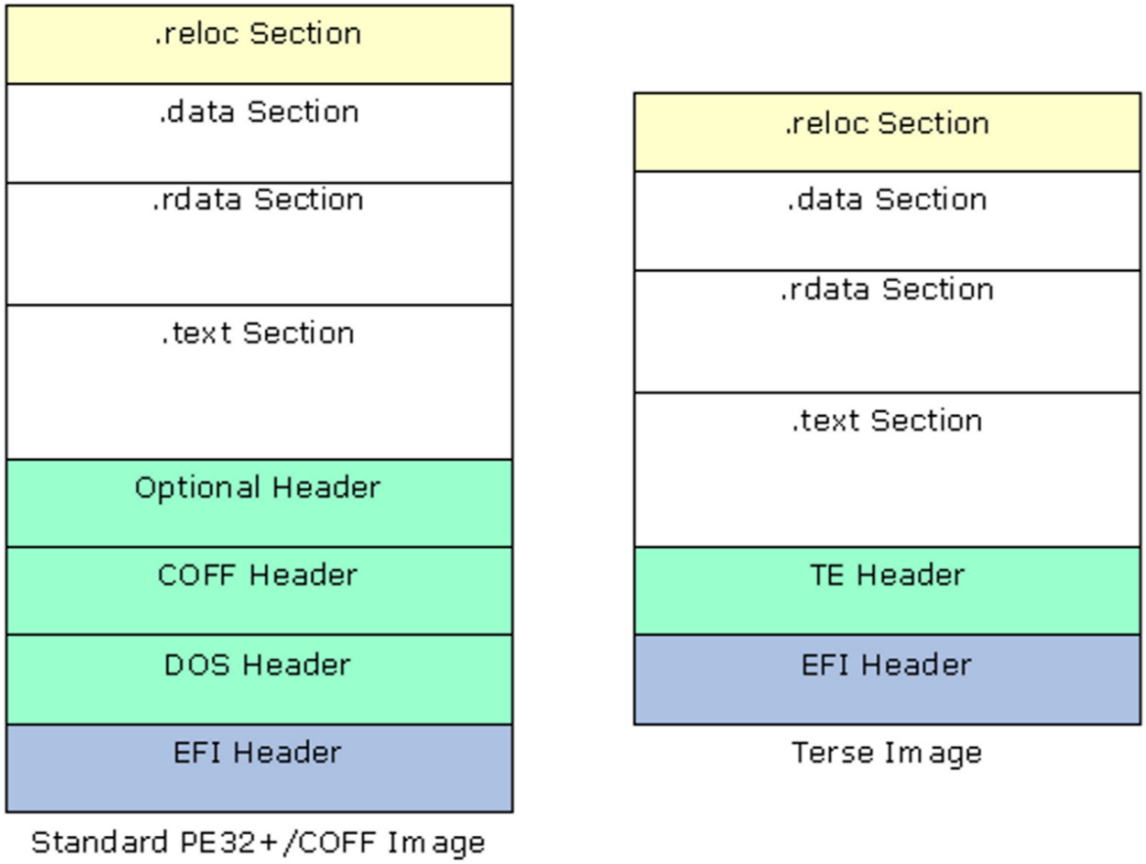

Figure 10. EFI Image Files

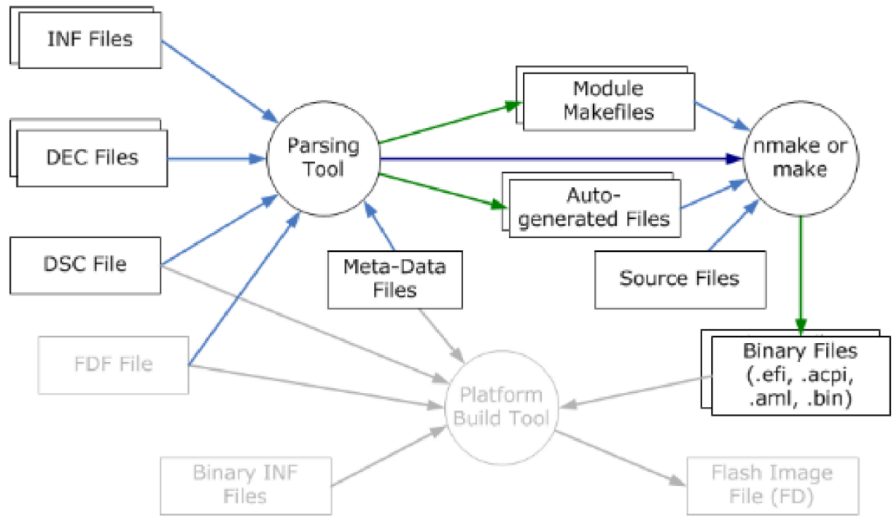

Figure 11. EDK II Platform Build Process Flow

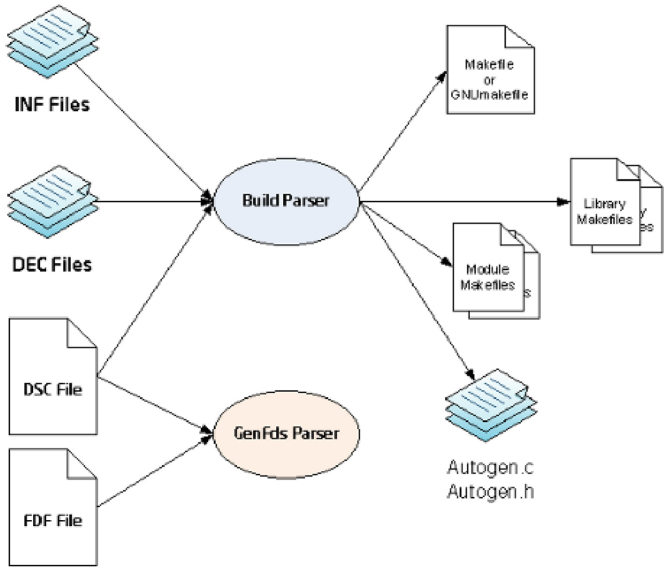

Figure 12. EDK II Parsing Data Flow

EDK II Build System

The EDK II build system produces binary images that conform to UEFI and PI specification file formats. In some cases, the tools have been extended to follow the Intel Innovation Framework for EFI Specifications. Some binary content may also follow other industry standard specifications, such as ACPI and PCI specifications.

The build system allows the setting of multiple paths that will be searched when attempting to resolve the location of EDK II packages. The build system works in the context of a platform

using the Platform Description (DSC) file to define what will get built. When building a single driver or application the DSC file is used to define what will be built along with the libraries, configuration settings, and custom build flags.

Build Process Overview

Prior to executing a build command, specific system environment variables must be initialized: WORKSPACE and EDK_TOOLS_PATH are required for all builds. Additionally, the provided EDK II tool set must be present in a directory that is in the system environment variable: PATH. The edksetup scripts provided in the root directory of the EDK II development tree will set the WORKSPACE and EDK_TOOLS_PATH environment variables as well as modify the system environment variable, PATH, to ensure that the tools can execute.

The EDK II Build Process is handled in three major stages:

Prebuild or AutoGen stage: parse metadata files, UCS-2LE encoded files, and VFR files to generate C source code files and Makefiles.

Build stage: process the source code files to create PE32/PE32+/COFF images that are processed to EFI format using NMAKE.

Post-build or ImageGen stage: process the binary, EFI format files to create EFI “FLASH” images, EFI update capsules, UEFI applications, or PCI Option ROMs.

INF files

INF files are EDK II Module Information Files. The INF files are used by EDK II utilities that parse build meta-data files (INF, DEC, DSC, and FDF files) to generate AutoGen.c, Autogen.h, and makefile files for the EDK II build infrastructure [6].

The INF meta-data file describes properties of a module. INF files generated during a build describe how the module was compiled, linked, and what platform configuration database items (PCDs) are exposed. Each module or component INF file is broken into sections similar to other build meta-data files. The purpose of a module’s INF file is to define the source files, libraries, and definitions relevant to building the module, and creating binary files that are either raw binary files or PE32/PE32+/coff format files. EDK II parsing utilities are token based which permits elements to span multiple lines. The EDK II utilities check EDK II INF files, and, if required, generate C code files based on the content of the EDK II INF.

The EDK II meta-data files use the INI file format. The INF file consists of sections delineated by section tags enclosed within square brackets. Section tag entries are case-insensitive. The content below each section heading is processed by the parsing utilities in the order that they occur in the file. The precedence for processing these section tags is from right to left, with sections defining an architecture having a higher precedence than a section which uses

common (or no architecture extension) as the architecture. Sections are terminated by the start of another section or the end of the file.

The [Defines] section of EDK II INF files is used to define variable assignments that can be used in later build steps. The [Defines] section includes many elements: INF_VERSION, BASE_NAME, PI_SPECIFICATION_VERSION, UEFI_SPECIFICATION_VERSION, FILE_GUID, MODULE_TYPE, BUILD_NUMBER, VERSION_STRING, MODULE_UNI_FILE, LIBRARY_CLASS, PCD_IS_DRIVER, ENTRY_POINT, UNLOAD_IMAGE, CONSTRUCTOR, DESTRUCTOR, SHADOW, PCI_DEVICE_ID, PCI_VENDOR_ID, PCI_CLASS_CODE, PCI_REVISION, PCI_COMPRESS, UEFI_HII_RESOURCE_SECTION, CUSTOM_MAKEFILE, SPEC, DPX_SOURCE.

The INF_VERSION element is an eight digit hexadecimal number that identifies the INF spec version. The BASE_NAME element is a single word identifier that is used for the component name. The FILE_GUID element is a GUID number that is required for all EDK II format INF files. The MODULE_TYPE element specifies the type of module. The VERSION_STRING specifies a value that will be encoded as USC-2 characters in a Unicode file for the VERSION section of the FFS. The ENTRY_POINT element is the name of the driver's entry point function.

The [Sources] section is used to specify the source files that make up the component. Directory names are required for files that exist in subdirectories of the component. All directory names are relative to the location of the INF file.

The [LibraryClasses] section is used to list the names of library classes that are required or optionally required by a component. A library class instance, as specified in the DSC file, will be linked into the component.

The [Packages] section lists all of the EDK II declaration files that are used by the component. Data from the INF and the DEC files are used to generate content for the AutoGen.c and AutoGen.h files. The locations of the packages listed in this section will be used in generating include path statements for compiler tool chains. The path must include the DEC file name and the name of the directory that contains the DEC file.

DSC files

DSC files are EDK II Platform Description (DSC) files. In order to use EDK II modules or the EDK II Build Tools, EDK II DSC and FDF files must be used [7]. EDK II tools use INI style text based files to describe components, platforms, and firmware volumes.

EDK II parsing tools contain the templates for processing files to create the component binary images from source files. EDK II Binary Modules are not required to be included in EDK II DSC files. EDK II DSC files are a list of: EDK II module INF files, EDK components, EDK libraries, EDK II library class instance mappings, and EDK II PCD entries.

Each platform DSC file is broken into sections in a manner similar to the component description (INF) files. However, while the intent of a component's INF file is to define the source files, libraries, and definitions relevant to building the component, the function of the platform DSC file is to specify the library instances, components, and output formats used to generate binary files that will be processed by other tools to generate an image that is loaded into a flash device.

In general, the parsing utilities read each line from the sections of the platform description (DSC) file, process the component, module, or library's INF file on the line to generate a makefile before continuing to the next line.

3. Requirements

This project has 3 different goals that we are trying to achieve. All of these goals will be UEFI shell applications that will be tested on the Whiskey Lake boards that have been provided by Intel. This is so that we can test these applications without interfering with the security protocols of our machines. The first goal of our project is a shell application that can read the characteristics of the CPU and display them to the user. The second goal of our project is a shell application that can provide a memory topology of the system at boot, and provide memory leak detection for the boot sequence. Our final goal for this project is an application that reports the SMM handler's latency. It will also document SMI sizes and sources for review by the user. All of these applications will have to be written into the boot image of the board so that they can access these low level functionalities. The current state of the art in UEFI shell applications is Tianocore. Tianocore is an open source repository for UEFI applications that perform similar functions to our applications. We aim to follow coding practices that are followed in the Tianocore repository to minimize the possible vulnerabilities that our applications may add to the system in case they were to be run on a bigger machine. We also have to keep to these practices in order to avoid breaking the board's boot sequence.

4. System Architecture

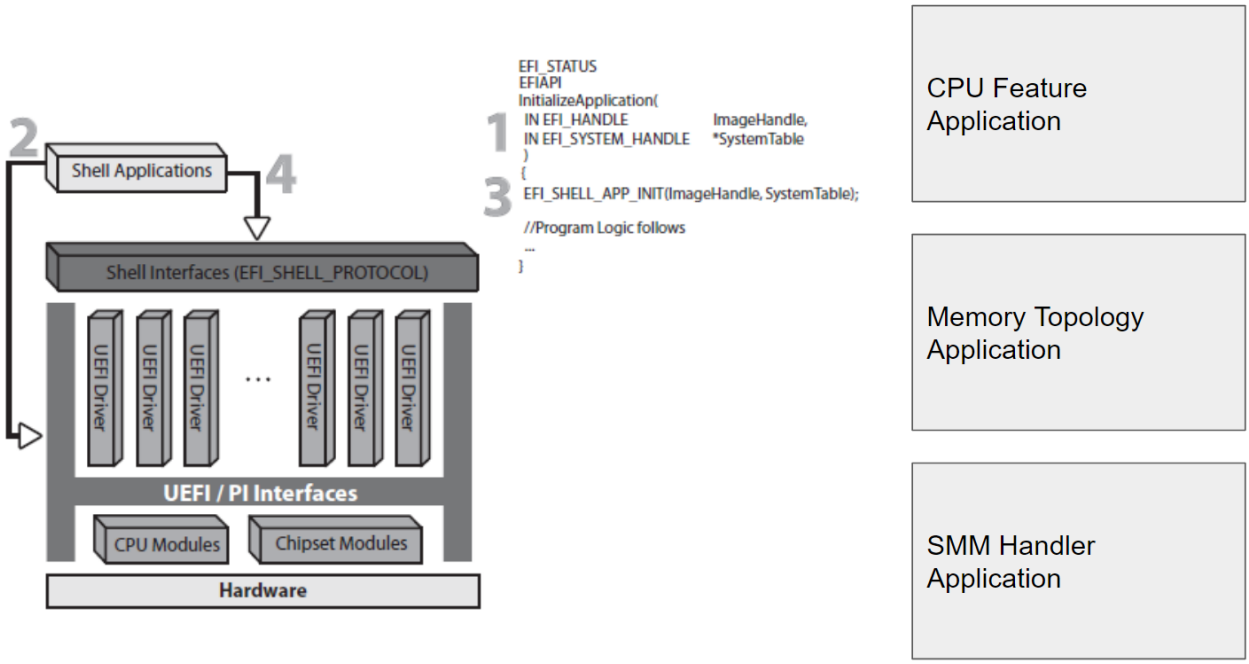

Figure 13. System Architecture for UEFI Shell Application Launch

UEFI Shell Applications:

These are UEFI compliant binary programs. These programs interact with the underlying UEFI Shell environment and are unloaded from memory after execution.

UEFI applications are EFI images of the type EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION. These images are executed and automatically unloaded when the image exits or returns from its entry point.

The INF file for a UEFI application specifies MODULE_TYPE as UEFI_APPLICATION. The [Defines] section of the INF file specifies the entry point for the application. There are two parameters for the UEFI application entry point, the ImageHandle and SystemTable. The ImageHandle refers to the image handle of the UEFI application and the SystemTable is the pointer to the EFI System Table.

A UEFI application may consume the UEFI Boot Services, UEFI Runtime Services, and UEFI System Table. A UEFI application can access the protocol interfaces produced by UEFI drivers by invoking the LocateProtocol(), HandleProtocol(), and OpenProtocol() services.

Variables in UEFI Shell applications are defined as key/value pairs that consist of the identifying information (the key) and arbitrary data (the value). Variables are intended for use as a means

to store data that is passed between the EFI environment implemented in the platform and the EFI OS loaders and other applications that run in the EFI environment. UEFI applications can read and write variables via the UEFI Runtime Services GetVariable() and SetVariable(). These services are available for UEFI applications since the Variable Architecture Protocol is installed during the DXE phase prior to the transient system load (TSL) phase where UEFI applications may be executed.

UEFI Shell:

This is a program that supports the launching and interpreting of shell applications. The Unified Extensible Firmware Interface is an interface that is supported by many machines. It allows us to create programs that run on lower levels of the operating system.

The UEFI Shell is a special UEFI application that provides the user with a command-like interface for invoking various UEFI executable applications.

UEFI Drivers:

These are UEFI compliant binary programs that follow the UEFI specification driver model. On launch, these programs remain in memory and provide protocols or services that remain resident in the system.

It should be noted that the primary difference between what someone might call a standard UEFI driver/application and a UEFI Shell application is that the latter has knowledge of the programmatic components of the UEFI Shell infrastructure. That being said, UEFI Shell applications can leverage the underlying localization support in the same manner as any other BIOS component, such as UEFI Drivers and Option ROMs.

The INF file for UEFI drivers must set the MODULE_TYPE to UEFI_DRIVER. There are several protocols that can be used in the UEFI driver entry point. The most common protocols used to separate the loading and starting/stopping of UEFI drivers include the Driver Binding Protocol, Component Name Protocol, and Driver Diagnostics Protocol. The Driver Binding Protocol provides functions for starting and stopping the driver and determining if the driver manages a particular controller. The Component Name Protocol provides functions for retrieving a human-readable name of a driver and controllers that a driver manages. The Driver Diagnostics Protocol provides functions for executing diagnostic functions on the devices that a driver manages.

UEFI drivers can use protocol services to access protocol interfaces produced by other modules. There are several boot services to handle protocols. The InstallProtocolInterface(), ReInstallProtocolInterface(), and InstallMultipleProtocolInterfaces() services are used to install protocols. The LocateProtocol(), HandleProtocol(), and OpenProtocol() services are used to retrieve protocols.

UEFI drivers can read and write variables using the UEFI Runtime Services GetVariable() and SetVariable(). When using these services, the distinction between a UEFI driver and DXE driver is that DXE drivers must explicitly specify the dependency relationship for EFI_VARIABLE_ARCH_PROTOCOL and EFI_VARIABLE_WRITE_ARCH_PROTOCOL in the [depex] section of the INF file. UEFI drivers do not have this section in their INF file.

UEFI/PI Interfaces:

A set of interfaces that consist of the Driver Execution Environment (DXE) and Boot Device Selection (BDS) that allow the UEFI Shell to interact with lower level CPU modules and Chipset modules.

CPU and Chipset Modules:

A series of executable modules and data files that are organized in flash memory. These elements contain information needed to configure the I/O devices, memory, and other system components.

System Hardware:

CPU, Memory, I/O devices, and other components of the Up Xtreme by Aaeon.

5. System Design

5.1 Hardware Design

Objectives: The purpose of the hardware tools is to provide a platform for loading and running our UEFI Shell applications.

Constraints: We are constrained to using an UP Xtreme series board with an 8th generation Intel Core CPU. We will be flashing a custom made firmware image generated using the software tools, source code files, and metadata files provided in the Intel training materials. We will use the most recent version of DediProg Software (version SF7.4.7.6) for erasing and programming the Flash memory contents of the UP Xtreme board.

Components: The hardware components of our project include the UP Xtreme board with an 8th generation Intel Core CPU, a DediProg Flash programmer, a 128 GB SSD Storage device, a 19V 7A power supply, and USB cables.

Uses and interactions: We will interface the UP Xtreme board with our personal computers using FTDI, USB cables, and jumper wires for UART serial communication with the Tera Term program for implementing a console window for the UP Xtreme board. We will interface the UP Xtreme board to the DediProg Flash programmer using the DediProg 12 pin connector cable for flashing our custom firmware image to the UP Xtreme board.

Interface: We will use the Tera Term program for generating a console window for the UP Xtreme board. We will use the DediProg program for flashing the firmware images to the board.

Resources: We used the hardware presentation slides provided by Intel for setting up the DediProg hardware and software tools and Tera Term hardware and software tools. We have a saved firmware image that has been previously tested which we can use to debug the device in case any custom firmware images our team generates cause device failures.

Subsystem Design: We will use the DediProg programmer to erase the flash memory of the device and load new firmware images to the device. We will use the Tera Term application to generate a console on our personal computers while running the UP Xtreme board, accessing the UEFI Shell, and executing the Shell executables.

5.2 Software Design

Objectives: The purpose of the software design is to design Shell applications that can be executed in the UEFI shell environment. The Shell executables will be designed from C source code files and metadata files provided in the UEFI training materials provided by Intel. We will modify these C source code files and metadata files to meet the specific requirements for the design project.

Constraints: We are constrained to using the set of software tools provided or suggested by Intel for designing our UEFI shell applications.

Composition: The software tools for this project include Tianocore EDK II, Python 3.7.x or greater for Windows, Git for Windows, Microsoft Visual Studio, NASM for Windows, Intel ACPI source language (ASL) compiler, USB Drivers, Dediprog, and Tera Term. The UEFI training materials provide the base UEFI Shell applications to be built.

Uses and interactions: We will generate the custom firmware image for the UP Xtreme board using the build_bios.py Python script provided by Intel. The Python script will process the C source code files and metadata files to generate the .efi executable Shell applications and custom firmware image for the UP Xtreme board. We will run the UEFI Shell applications from the UEFI Shell while booting the UP Xtreme board after the firmware image has been flashed to the device.

Interface: The main software interface for building the custom firmware image and .efi executables will be the build_bios.py Python script provided by Intel in addition to the modified C source code files and metadata files corresponding to each Shell application.

Resources: We will modify the base UEFI Shell applications provided in the UEFI training materials. The UEFI training materials will discuss the various steps in generating the custom firmware image and .efi executables.

Subsystem Design: The CPUID application accesses specific spots in memory that hold the information of the CPU, and translates that information into a human readable format. The second program is a shell application that finds the size of reserved memory that is used by system firmware. The last application is a SMI logging application that will list the SMI's that have occurred while it was running.

5.3 Human Interface Design

User Perspective: We will access the UEFI Shell by interrupting the boot process to the operating system on the board. Once we have started the UEFI Shell program, we will locate the location of the .efi executables on the USB flash drive connected to the UP Xtreme board using Shell commands for providing the path to the USB drive.

Interface Objectives: The Shell executables should be run from the UEFI Shell terminal. The output from the executables will be piped into output text files which will be placed on the USB flash drive and later viewed on a personal computer.

Interface Constraints: All UEFI Shell executables should be able to be stored on a FAT32 USB drive and executed from the USB drive. All UEFI Shell executable output should be piped into text files stored on the USB drive.

Sample use cases: The three UEFI Shell executables that we will design have separate functionality and use cases. The CPUID executable will be used to list detailed information on the CPU and caches by processing the output of executing each leaf of the CPUID instruction. The memory management UEFI Shell executable will be used to report the memory locations used for the firmware image and memory leaks associated with executing various Shell programs. The System Management Mode (SMM) Interrupt handler UEFI Shell executable will report the function name of the SMI Handler, the name of the dispatcher the handler is registered with, the source file name and line number, and the SMI context.

User Assumptions: The users of our deliverables will include engineers working within the Tianocore repository. The .efi executables will be intended to report information specific for the UP Xtreme board.

6. Project Bill of Materials

- **Hardware:**
  - SUT - Up Xtreme Celeron
  - Up Xtreme Power Supply and power cables
  - USB and Serial cable
  - FTDI USB 3.3V to Serial Cable
  - Test lead wire for FTDI 2 10 Pin USB Up Xtreme
  - USB Thumb drive
  - USB debug 3.0 cable
  - 400gig 3710 ssd's (2.5" sata). Or 256gb nvme ssd
  - 4 Port Gigethernet Switch 8 Port Netgear
  - 4 Port USB 3.0 hub

- **Software:**
  - VisualStudio 2019
  - Python version >3.8.x
  - Git
  - NASM version >2.15.x
  - IASL
  - EDK II
  - Simics

7. Ethical Considerations

For this project, we need to make sure that our work meets the standards provided by our sponsor. We need to acknowledge our sources whenever possible since our work will be building off previous work in the Tianocore repository. We need to assure that our UEFI Shell applications do not disrupt the functionality of the hardware resources that we will be using for this project. We do not have other ethical considerations at this time.

8. References

1. V. Zimmer, M. Rothman, and S. Marisetty, *Beyond bios: Developing with the Unified Extensible Firmware Interface*. De G Press, 2017.
2. M. Rothman, V. Zimmer, and T. Lewis, *Harnessing the UEFI shell moving the platform beyond DOS, second edition*. Boston, MA: Walter de Gruyter, 2017.
3. *Unified Extensible Firmware Interface (UEFI) Specification*, Release 2.10. UEFI Forum, Inc. From the official UEFI Forum.
4. “EDK II build specification,” *EDK II Build Specification - EDK II Build Specification*. [Online]. Available: https://edk2-docs.gitbook.io/edk-ii-build-specification/. [Accessed: 22-Jan-2023].
5. “EDK II module writer's guide,” *EDK II Module Writer's Guide - EDK II Module Writer's Guide*. [Online]. Available: https://edk2-docs.gitbook.io/edk-ii-module-writer-s-guide/. [Accessed: 22-Jan-2023].
6. “EDK II module information (INF) file specification,” *EDK II Module Information (INF) File Specification - EDK II INF Specification*. [Online]. Available: https://edk2-docs.gitbook.io/edk-ii-inf-specification/. [Accessed: 26-Feb-2023].
7. “EDK II platform description (DSC) file specification,” *EDK II Platform Description (DSC) File Specification - EDK II DSC Specification*. [Online]. Available: https://edk2-docs.gitbook.io/edk-ii-dsc-specification/. [Accessed: 26-Feb-2023].

9. Errata