Plagiarism Checker X Originality Report

Similarity Found: 5%

Date: Sunday, May 10, 2020
Statistics: 969 words Plagiarized / 18066 Total words
Remarks: Low Plagiarism Detected - Your Document needs Optional Improvement.

IP Independent Generic Firmware to accelerate Development Process Iteration Submitted by Gahan Saraiya 18MCEC10 Department of Computer Science & Engineering, Institute of Technology, Nirma University, Ahmedabad, Gujarat - 382481, India. May, 2020 IP Independent Generic Firmware to accelerate Development Process Iteration Major Project Submitted in partial ful?llment of the requirements for the degree of Master of Technologyin Computer Science & Engineering with specialization in Computer Science & Engineering Submitted by Gahan Saraiya 18MCEC10 Department of Computer Science & Engineering, Institute of Technology, Nirma University, Ahmedabad, Gujarat - 382481, India.

Declaration I hereby declare that the dissertation IP Independent Generic Firmware to ac- celerate Development Process Iteration submitted by me to the Institute of Technology, Nirma University, Ahmedabad, 382481 in partial ful?llment of the re- quirements for the award of Master of Technology in Computer Science & En- gineeringwith specialization in Computer Science & Engineering is a bona- ?de record of the work carried out by me under the supervision of Prof. Dvijesh Bhatt.

I further declare that the work reported in this dissertation, has not been submitted and will not be submitted, either in part or in full, for the award of any other degree or diploma of this institute or of any other institute or University. Sign: Name & Roll. No.: Date: Computer Science & Engineering Certi?cate This is to certify that the dissertation entitled IP Independent Generic Firmware to accelerate Development Process Iteration submitted by Gahan Saraiy a (Roll No.

18MCEC10) to Nirma UniversityAhmedabad, in partial ful?llment of the requirement for the award of the degree of Master of Technology in Com- puter Science & Engineering

with specialization in Computer Science & Engineering is a bona-?de work carried out under my supervision. The disserta- tion ful?lls the requirements as per the regulations of this University and in my opinion meets the necessary standards for submission.

The contents of this disser- tation have not been submitted and will not be submitted either in part or in full, for the award of any other degree or diploma and the same is certi?ed. Prof. Dvijesh Bhatt Dr. Priyanka Sharma Guide & Assistant Professor, Professor, CSE Department, Coordinator M.Tech - CSE (CSE) Institute of Technology, Institute of Technology, Nirma University, Ahmedabad. Nirma University, Ahmedabad Dr. Madhuri Bhavsar Dr. R. N.

Patel Professor and Head, I/C Director, CSE Department, Institute of Technology, Institute of Technology, Nirma University, Ahmedabad Nirma University, Ahmedabad. Abstract Intel System on a Chip (SoC) features a new set of Intel Intellectual Property (IP) for every generation. BIOS involves development of major individual components such as Processor, Graphics/Memory Controller, Input/Output Controller hub, Sys- tem Monitor/Management Bus, Direct Media Interface, SATA/IDE/USB, Peripheral Component Interconnect (PCI), Voltage Regulator and Advanced Con?guration and Power Interface (ACPI) for every Intel System on a Chip (SoC). Section 1. describes all the basic information required on the Intel SoC. Section 2.

involves the design of the Basic Boot Flow of the BIOS followed by Section 3. and Section 4. explains the architecture and protocols which are the concept used to build the proposed framework which is described under Section 5. to aid the development and debugging iteration for various stakeholders including but not limited to BIOS Developers, Validation Engineers, Automation team.

The framework is designed and implemented to aid the development process by eliminating longer duration of common debugging steps and providing a sophisti- cated way to build and test the various scenarios includes but not limited to Setup Options, Firmware Flashing, UEFI Variable Creation. Acknowledgements It gives me immense pleasure in expressing thanks and profound gratitude to Prof.

Dvijesh Bhatt, Assistant Professor, Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout this work. The appreciation and continual support he has imparted has been a great motivation to me in reaching a higher goal. His guidance has triggered and nourished my intellectual maturity that I will bene?t from, for a long time to come. It gives me an immense pleasure to thank Dr.

Madhuri Bhavsar, Honorable Head of Computer Science And Engineering Department, Institute of Technology, Nirma University, Ahmedabad for her kind support and providing basic infrastructure and healthy research environment. A special thank you is expressed whole heartedly to Dr. Alka Mahajan, Honorable Director, Institute of Technology, Nirma University, Ahmedabad for the unmention- able motivation she has extended throughout course of this work.

I would also thank the Institution, all faculty members of Computer Engineering Department, Nirma University, Ahmedabad for their special attention and suggestions towards the project work. Gahan Saraiya 18MCEC10 v Contents Declaration ii Certi?cate iii Abstract iv Acknowledgements v List of Figures ix 1. Introduction . . . . . 1 1.1 Uncore IP 1 1.2

Legacy BIOS and UEFI . . . . . 1 BIOS . . . . . 1 1.2.1 Background of Legacy BIOS . . . . . 2 1.2.2 Limitations of legacy BIOS . . . . . 2 1.3 Uni?ed Extensible Firmware Interface (UEFI) . . . . . 3 1.3.1 UEFI Driver Model Extension 3 1.3.2

UEFI's Role in boot process . . . . . 5 1.4 Advanced Con?guration and Power Interface (ACPI) . . . . . 5 1.4.1 Overview of ACPICA Subsystem . . . . . 5 1.4.2 Operating System Services Layer OSL . . . . . 6 1.4.3 ACPICA Subsystem Interaction . . . . . 7 1.5 Peripheral Component Interconnect Express (PCIe) . . . . . 8 1.5.1 Functional Description 8 1.5.2

BUS Performances and Number of Slots Compared . 9 1.6 Graphics Controller . . . . . 9 1.6.1 Graphics Output Protocol (GOP) . . . . . 10 1.6.2 GOP Overview . . . . . 10 1.6.3 GOP DRIVER . . . . . 10 1.6.4 GOP Integration . . . . . 11 2. Design . . . . . .

. . . . . 12 2.1 Design Overview of UEFI . . . . . 12 vi vii 2.1.1 Goal of UEFI Driver . . . . . . 13 2.2 UEFI/PI Firmware Images . . . . . 14 2.3 Platform Initialization PI Boot Sequence . . . . . 16 2.4 Security (SEC) . . . . . 16 2.5 Pre-EFI Initialization (PEI) . . . . . .

. . 17 2.5.1 PEI Services . . . . . 19 2.5.2 PEI Foundation . . . . . 21 2.5.3 PEI Dispatcher . . . . . 22 2.6 Driver eXecution Environment (DXE) . . . . . 23 2.7 Boot Device Selection (BDS) . . . . . 23 2.8 Transient System Load (TSL) and Runtime (RT) 23 2.9

Introduction Intel System on a Chip (SoC) features a new set of Intel Uncore Intellectual Prop- erty (IP) for every generation. Section 1. covers the introduction and overview of BIOS, UEFI and it’s role and major components - Advanced Con?guration and Power Interface (ACPI), Peripheral Component Interconnect Express (PCIe) and Graphics Controller. Section 2. describes the design of UEFI and the boot phases in detail.

The study of the BIOS binary structure and mapping of each components byte and alignment is described in Section 3.. Proposed work to reducing the pro- cess of build iteration described in Section 5.. 1.1 Uncore IP The Uncore encompasses system agent (SA), memory and Uncore agents such as graphics controller, display controller, memory controller and Input Output (IO).The Uncore IPs are Peripheral Component Interface Express (PCIe), Graphics Process- ing Engine (GPE), Thunderbolt, Imaging Processing Agent (IPU), North Peak (NPK), Virtualization Technology for directed-IO (Vt-d), Volume Management Device (VMD).

PCI Express abbreviated as PCI or PCIe, is designed to replace the older PCI stan- dards. A data communicating system is highly-developed via PCIe for use the trans- fer data between the host and the peripheral devices. Intel developed the hard- ware interface which allows the connection of external peripherals to a computer called Thunderbolt.

This interface not only has PCI Express (PCIe) and Display- Port (DP) combined into two serial signals but additionally provides DC power also, bundled in just one cable. “Graphics Processing Engine (GPE)”, “Integrated graph- ics”, “shared graphics solutions”, “integrated graphics processors (IGP)” or “uni?ed memory architecture(UMA)” utilize a portion of a memory of computer system in- stead of having dedicated graphics memory. GPEs can be integrated onto the moth- erboard as part of the chipset.

Guest virtual machines use Virtual Technology for Directed-IO (Vt-d), an input/output memory management unit(IOMMU), to directly use peripheral devices, such as hard-drive controllers, accelerated graphics cards and Ethernet, through interrupt remapping. 1.2 Legacy BIOS and UEFI BIOS is the governing reference which speci?es a ?rmware interface.

"Legacy" (as in Legacy BIOS) - in terms of ?rmware speci?cations it refers to an older, broadly used speci?cation. Major responsibility of BIOS is to initialize the hardware devices, loading and commencing an OS. When the system boots, the BIOS initializes and identi?es every connected system devices including keyboard, mouse, hard disk drive, solid state drive, video display card and other hardware followed by locating software stored on a boot device i.e.

a hard disk or removable 2 TABLE 1: Comparison of Legacy BIOS and UEFI Legacy BIOS EFI Programming Language used Assembly Language C Language (99%) Resources Interrupt Hardcode Mem- ory Access hardcore In- put/Output Access Divers, Handlers and Pro- tocols Processor Type x86 16 - b i t CPU Protects Mode Expand Interrupt through hook Driver to be loaded OS Communication Bridge via ACPI through runtime driver 3 r d Party ISV & IHV Support Bas Ease of Support and for Cross-Platforms storage such as USB or CD/DVD and loads and executes that software, transferring control of the system to it. This ?ow of actions is also known as "booting" or "boot strapping".

Table 1 overviews of comparisons of UEFI with legacy BIOS. 1.2.1 Background of Legacy BIOS In 1980s, IBM developed the personal computer with a 16-bit BIOS with the aim of ending the BIOS after the ?rst 250,000 products. Legacy BIOS is based upon In- tel’s original 16-bit architecture, ordinarily referred to as "8086" architecture.

And as technology advanced, Intel extended that 8086 architecture from 16 to 32-bit. Legacy BIOS is able to run different OS very well irrespective if the system is IBM or not. Additionally, Legacy BIOS also has a de?ned OS-independent interface for hardware that enables interrupts to communicate with keyboard, disk and video services along with the BIOS ROM loader and bootstrap loader, to name a few. Use of legacy BIOS is diminishing and is expected to be phased out in new systems by the year 2020.

1.2.2 Limitations of legacy BIOS With progress in technologies, the BIOS implementations were also updated with many new con?guration and power management technologies and added support 3 for many generations of Intel® architecture hardware.

Although a few of limita- tions namely, upper memory block (UMB) dependencies, PC AT hardware depen- dencies, 1 MB addressable space and 16-bit addressing mode persisted throughout the years. The need to integrate libraries of third-party ?rmware modules into a single platform solution across multiple product lines and ensuring quality of indi- vidual ?rmware modules arises in the industries.

The existing market demands to overcome inherent limitations lead towards development of a fresh BIOS architec- ture which is introduced in market as The UEFI speci?cations. One major problem with existing BIOS implementations is that since they are highly customized for a speci?c motherboard, there maintenance is dif?cult. A lot of effort is required in signi?cant porting, integration, testing and debug work of changes in component modules.

The UEFI architecture is designed to considering these limitations and to resolve them. 1.3 Uni?ed Extensible Firmware Interface (UEFI) UEFI is a replacement for legacy BIOS to act as the interface between a operating system and its platform ?rmware streamlining the booting process. It offers a rich extensible pre-OS environment with advanced boot and runtime services, replacing most BIOS functions.

Uni?ed Extensible Firmware Interface (UEFI) is grounded in Intel's initial Extensible Firmware Interface (EFI) speci?cation 1.10, which de- ?nes a software interface providing linking to an operating system and platform ?rmware. It has intrinsic networking capabilities, is designed to work with multi- processors (MP) system and also allows users to execute applications on a command line interface. The UEFI Forum board of directors consists of representatives from 11 industry leaders as described in Figure 1.

These organizations work to ensure that the UEFI speci?cations meet industry needs.

UEFI uses a different interface for runtime services and boot services but UEFI does not specify how "Power On Self Test" (POST) and Setup are implemented those are BIOS' primary functions. 1.3.1 UEFI Driver Model Extension Boot devices are accessible via a set of protocol interfaces.

The UEFI Driver Model provides a replacement for PC-AT-style option ROMs. The UEFI Driver Model was not designed to replace the high-performance OS spe- ci?c drivers but to access boot devices in the pre-boot environment, to support the execution of modular pieces of code, also known as drivers.These drivers con- trol hardware buses and devices on the platform, and also they may provide some software-derived, platform speci?c service.

The information required by the driver 4 FIGURE 1: Board of Directors of UEFI Forum developers for implementing combination of bus drivers which boot an UEFI-complaint OS are included in the UEFI Driver Model. Thus the UEFI Driver Model is designed to be generic.The UEFI Speci?cation de- scribes how to write USB bus drivers, USB device drivers, PCI device drivers, PCI bus drivers and SCSI drivers.

Additional details are provided that allow UEFI drivers to be stashed within PCI option ROMs along with maintaining the compat- ibility with legacy option ROM images. The UEFI Speci?cation is designed keeping in mind the goal of having compact driver images. However to facilitate support for multiple processor architectures, a driver object ?le for each architecture is required to be included leading to a space issue.

To resolve this issue, UEFI de?nes EFI Byte Code Virtual Machine. Ev- ery driver ?le is compiled into just a single EFI Byte Code object which is run by an UEFI Byte Code Interpreter included in the UEFI Speci?cation complaint ?rmware. Another very common method to resolve this issue is compression.

The UEFI spec- i?cation de?ned compression and decompression algorithms which may be used to reduce the size of UEFI Drivers. This information can used by OEMs, IHVs, OSVs, and ?rmware vendors for de- veloping drivers that produce standard protocol interfaces, and operating system loaders which could be utilized to boot UEFI compliant OS. 5 1.3.2

UEFI's Role in boot process During the boot process, UEFI speaks to the operating system loader and acts as the interface linking the operating system and the BIOS. The PC-AT boot environment is challenging to innovate as each new ?rmware capa- bility requires ?rmware developers to craft more complex solutions, often requiring OS developers to perform manipulation for their boot code.

Since this is a time- consuming process and also required investment of resources, the UEFI speci?ca- tion undertakes it as a primary goal to overcome this issue. 1.4 Advanced Con?guration and Power Interface (ACPI) The “ACPI Component Architecture (ACPICA)” is an implementation of a group of software components according to the ACPI speci?cation.

It is created with the goal of isolating operating system dependencies to a relatively small translation or conversion layer (the OS Services Layer). This makes the bulk of the ACPICA code independent of any individual operating system.so it can used for new operating systems with no source changes within the ACPICA code itself.

Tthe architecture include below component: • ACPICA Subsystem - independent of OS and kernel which serves the primal ACPI services like the AML interpreter and management of namespace. • ACPICA Subsystem - independent of OS and OS Services Layer for every host OS to serve OS support. • The ASL compiler/disassembler for translating the source code from ASL to AML and also disassembling the ASL source code from the binary ACPI tables if exists.

• Many ACPI utilities for running the interpreter in level 3 user space taking out the binary ACPI tables residing in the output result of ACPI Dump utility along with translating ACPICA source code to output format of Linux/Unix. Figure 2 portrays the ACPICA subsystem in relation with the device driver(s), host OS, and the ACPI hardware. 1.4.1

Overview of ACPICA Subsystem The “ACPICA Subsystem” develops the basic primal aspects of the ACPI speci?- cation. Includes an AML parser/interpreter, ACPI table and device support, ACPI namespace management, and event handling. As the ACPICA subsystem serves the 6 FIGURE 2: The ACPI Component Architecture lower level services for system, it also involves low-level services of OS like memory management, scheduling, synchronization and I/O.

To allow the ACPICA Subsystem to easily link between any operating system that engage such services, an Operating System Services Layer transforms ACPICA-to- OS requests inside the system calls publicized by the host OS. This OS Services Layer is the one and only element of the ACPICA which pertains source code which is limited to a particular host OS. 1.4.2

Operating System Services Layer OSL “OS Services Layer (OSL)” act as a request translation service for host os from OS- independent ACPICA subsystem. The OSL

develops a common subset for interfaces of OS service by utilizing the primitives usable from host OS. The OSL has to be developed afresh for each and every supported host OS.

There exists only one ACPICA Subsystem which OS-independent but there has to be a different OSL for each OS backed by the ACPICA. The whole ACPICA in relation to the host OS is portrayed in Figure 3 7 FIGURE 3: ACPICA Subsystem Architecture 1.4.3 ACPICA Subsystem Interaction ACPICA Subsystem develops a subset of external interface links that could directly summoned via host OS.

These Acpi interfaces serve the literal ACPI services for host. When OS services are needed while servicing of request of an ACPI the Sub- system makes oblique request to host OS through the ?xed AcpiOs interfaces. FIGURE 4: Interaction between the Architectural Components Figure 4 portrays the kinship and fundamental interaction linking the diverse ar- chitectural modules by screening the control ?ow among them.

Note that OS in- dependent ACPICA Subsystem could never call the host OS directly and instead it has to make call(s) to the AcpiOs interfaces inside the OSL. This serves the ACPICA code as OS-independence. 8 1.5 Peripheral Component Interconnect Express (PCIe) The “Peripheral Component Interconnect (PCI)” architecture has emerged out to be a very thriving beyond even the many more optimistic prospects. Today about every new computer system arrives equipped with at least one PCI slots.

Even though there are about to countless PCI slots are shipped globally, there does exists tons of PCI adapter cards which are available to ful?ll the all the virtual possible application needs. This fast growing generation has also raised the demand for many new higher performance interface for input-output communication to support rising technol- ogy such as ultra high bandwidth technologies like 10 - G b Ethernet, 10 - G b “Fi- breChannel”, 12 X In?niBand and many more.

A regulation which could adapt to carrying out these high performance objectives, while withholding compatibility for previous generation of PCI would beyond any doubt can offer the idealistic solution. “PCI-X 2.0” standard has been developed to meet these objectives. It is capable to serve the uninterrupted performance to feed the nearly all high-bandwidth pro- grams while at the same instance keeping up the full hardware and software back- ward compatibility for previous “PCI” and “PCI-X” generations. The PCI-X 2.0

reg- ulation establishes two new grades of speed and performance which are PCI-X 266 and PCI-X 533. These speed grades endeavors bandwidths that are twice and four- fold

that of previous generation PCI-X 133 which results to ?nally supplying band- widths which are 32 x faster compared to the older version of PCI.

It successfully succeeds to bring of additional required performance via time-proven "Double Data Rate (DDR)" and "Quad Data Rate (QDR)" mechanisms that transmit data at either twice or fourfold the base frequency of clock. As the PCI-X 2.0 conserves too many modules of previous generation PCI it's bene?ciary for terri?c amount of preceding development job.

The OS, device drivers, protocols, connectors, form factor, BIOS, electrical signals, Bus functional modal among many more original PCI modules are all greatly rendered in the new PCI-X 2.0 speci?cation. Also, many of these modules actually remains untouched in PCI-X 2.0. Due to not having the much of the dis- similarity, it enables development easier because these modules have been already designed and developed with required engineering and familiar to the developers. As a end result, risk is dramatically decreased because the time-to-market became short. 1.5.1

Functional Description The hardware which is used to implement a PCI based system consumes a software interface served by PCI BIOS feature. It has elementary use to generate operations in address spaces speci?c to PCI (PCI con?guration space). The X86 architecture allows following mode to operate as per PCI BIOS features: • Real mode 9 • Protected mode – 286 protected mode (16 : 16) – 386 protected mode (16 : 32) • Flat mode (0:32 protected mode) In the Flat mode, all the segments begun at linear address 0 and span till the end of the whole 4 - G B address space.

1.5.2 BUS Performances and Number of Slots Compared The several architectures characterized by the PCISIG. Figure 5 portrays the de- velopment of PCI bus clock frequencies and bandwidths. Its pretty obvious that the increasing the bus frequency does comprise the load in terms of electrical nodes as also to the maximum permissible connectors on a bus at that clock frequencies.

FIGURE 5: Comparison of Bus Frequency, Bandwidth and Number of Slots A "PCI express (PCIe) Interconnect" is responsible in connecting two PCI devices via PCI link. A PCI link consists if any signal pairs in each direction. These sig- nals (x1, x2, x4, x8, x12, x16, x32) known as the Lanes. A BIOS designer decides that how many Lanes implementation should be permissible based on the benchmark performance of targeted platform on a given PCI link.

Figure 5 portrays the total bandwidth values for various PCI link width. 1.6 Graphics Controller Almost every graphics controllers are merely PCI controllers only. And it is

also obvious that the graphics drivers who are responsible to control and manage these 10 graphics controllers are also PCI drivers. Note that even if the most graphics con- trollers are PCI controllers but even then the graphics controllers can also utilize many of the other buses i.e. USB buses.

Characterizes of Graphics drivers are listed below: • Follows UEFI Driver Modal • Depending on the driver managed adapter, a graphics driver could be classi- ?ed as into a single output adapter and a multiple output adapter. • For each output expected, the graphics driver has to construct child handles. • For some of the output ports and protocols (such as GOP Protocol) the graphics drivers must create child handles. • Graphics drivers are hardware-dependent (i.e.

speci?c to the corresponding chip) due to the need of initializing and managing the graphics device. Note that (IHV) has the privilege of choosing whether to support and implement all the required modules of the UEFI speci?cation. i.e., all modules might not be implemented to support on a speci?ed system con?guration which doesn't support all of the services and features understood by the needed modules. 1.6.1

Graphics Output Protocol (GOP) The "Graphics Output Protocol GOP" Driver is member of the driver of UEFI boot time which are responsible for running up the display while the bios is booting. This driver triggers displaying of logo while the bios is booting. 1.6.2 GOP Overview The GOP driver is the successor for video controller of legacy BIOS and sheers the utilization of UEFI pre-boot ?rmware without the use of CSM.

The GOP driver can be 32 - b i t, 64 - b i t, or I A - 64 with no binary support. Pre-boot ?rmware architecture of UEFI which could be either 32 - b i t or 64 - b i t has to adapt the corresponding GOP driver architecture (32 - b i t or 64 - b i t). The GOP driver could be one of the boot mode: "fastboot" (for speci?c platform optimized mode to speedup the boot time) or "generic" (the normal boot process).

1.6.3 GOP DRIVER The EFI speci?cation characterizes the "Universal Graphic Adapter (UGA)" proto- col to provide graphics that could be device-independent. However, Speci?cation of UEFI eliminated the inclusion of UGA and replaced it with its successor GOP so that VGA hardware dependencies can be removed. 11 TABLE 2: GOP Driver ?les File Name Description Format GopDriver.efi The GOP driver binary Uncompressed PE/COFF image Vbt.bin Contains Video BIOS Table (VBT) data Raw Binary Vbt.bsf BMP script ?le. Required for modifying Vbt.bin using BMP tool Text 1.6.4

GOP Integration The platform ?rmware has to align with the below listed requirements for integra- tion of GOP Driver: • Platform ?rmware has to be obedient with UEFI 2.1 or

later. • Platform must enumerate and initialize the graphics device. • Platform must allocate enough graphics frame buffer memory required to sup- port the native mode resolution of the integrated display.

• The platform has to bring forth the standard protocol EFI_PCI_IO_PROTOCOL and also the EFI_DEVICE_PATH_PROTOCOL on the graphics device handle. Ad- ditionally, the platform must produce PLATFORM_GOP_POLICY_PROTOCOL. • The platform ?rmware should not establish the legacy BIOS Video. The GOP Driver solution comprises the following ?les shown in Table 2 GOP driver ?les. Customize the VBT data ?le Vbt.bin as per platform requirements and the corre- sponding BSF ?le. Integrate Vbt.bin and GopDriver.efi ?les into the platform ?rmware image.

The process of accomplishing this step is determined by the plat- form implementer, speci?c to the platform ?rmware implementation. 12 2. Design 2.1 Design Overview of UEFI The UEFI construction is depending on the below listed primal elements: • Re-utilizing of already existing interfaces - In order to keep up assets in existing infrastructure codebase, both at the OS and ?rmware level, many different existing speci?cations which are usually implemented on platforms harmonious with supported processor speci?cations has to be developed on platforms which is able to comply with speci?cation of the UEFI.

• System partition characterizes a partition and ?le system which are to de- veloped to grant safe sharing between various different vendors and various purposes. The power to include a disjoint, shareable system partition exists a chance to gain platform value-add without importantly thriving the need for nonvolatile memory of platform. • Boot services are responsible to offer interfaces for devices and system func- tionality which could be utilized during the time of ongoing boot process.

De- vice access is abstracted by “handles” and “protocols”. This features reuse the investment in already existing BIOS codebase by persisting underlying im- plementation necessity out of the speci?cation without giving execution load to the consumer accessing device.

• Runtime services - A stripped set of runtime services is given to guarantee seize abstraction for resources of platform hardware which could be required by the OS while its conventional operations. Error! Reference source not found determines the fundamental interaction of the different component parts of an UEFI speci?cation-amenable system which are utilized to carry out platform and OS boot. From the system partition, the os loader image is retrieves by the platform ?rmware.

The speci?cation supplies for a diversity of media and mass storage device types as “disk”, “CD-ROM”, and “DVD” as well as “remote boot” via a network (also known as LAN boot or network boot). By the use of extensible protocol interfaces, there is possibility to include many other boot media types but also these would need OS loader alteration if they need to use the protocols. Once begun, the OS loader proceeds to boot the whole operating system.

To achieve this it could utilize the EFI boot services and interfaces characterized by respected speci?cations to analyze, embrace, and initialize the several platform components and the OS software which controls them. Also, for the OS loader will be capable to access EFI runtime services while in boot phase. 13 FIGURE 6: UEFI Conceptual Overview 2.1.1

Goal of UEFI Driver Majorly below are the listed motives to be expected to achieve by the UEFI Driver: • Compatible - Any driver who conformist to the speci?cation has to hold up compatibility along with the UEFI and EFI Speci?cation. Hence, the UEFI Driver Modal brings up bene?ts of the extensibility mechanisms in the UEFI Speci?cation to include the desired and necessary features.

• Simple - Any driver who conformist to the speci?cation has to be simpler to develop and maintain. The UEFI Driver Modal has to permit a driver writer to focus on the ad hoc device for which the driver is to be developed. A driver should not be related to issues correspondence to platform management or policy of platform. These circumstance should be left over for the system ?rmware.

• Scalable - The UEFI Driver Modal requires the adaptability for all kind of platforms including mobile, embedded systems, workstations, servers as well as desktop systems. • Flexibility - The UEFI Driver Modal have to have capability to enumerate over every the devices (or only the relevant devices needed to boot the OS). With the minimum device enumeration support for fast boot ability can be achieved and the full device enumeration results to provide capability for per- forming system maintenance, or system diagnostics, OS installations on any boot device exists on the system.

• Extensible - The UEFI Driver Modal must be able to unfold to succeeding bus types as and when they are characterized. 14 • Portability - Every drivers transcribed in the UEFI Driver Model has to be portable among platforms and within every founded processor architectures. • Interoperability - Every drivers has to coexist along with every other ?rmware and drivers and also do so without incurring con?icts for any resource.

• Describing hierarchies of complex bus - The UEFI Driver Modal has to be capable to

key out a all kind of bus topologies from the platforms as simple as single bus to platforms with extremely complex bus which may consists of multiple buses of different kind. • Address the issues for legacy ROM option - The UEFI Driver Modal needs to instantly come up and resolve the restrictions and regulations of legacy ROM options.

Especially, it has to be capable to build add-in device cards which supports both UEFI drivers and legacy ROM options. However, maintaining this backward compatibility, the solution with proposed method- ology should also provide a way to migrate from legacy ROM option driver to UEFI driver. 2.2

UEFI/PI Firmware Images UEFI and PI speci?cations characterize the standard format for EFI ?rmware stor- age devices which includes FLASH or any other nonvolatile storage which are sep- arated in “Firmware Volumes”. FIGURE 7: UEFI/PI Firmware Image Creation 15 Build systems should have the capability of processing ?les to construct the ?le for- mats represented by the UEFI and PI speci?cations.

The tools which are supplied as part of the “EDK II BaseTools package” processes ?les compiled via third party scripts and tools, as well as unicode ?les and text ?les in order to construct UEFI or PI amenable binary image ?le. In few cases, where UEFI or PI speci?cations don’t have an corresponding ?le format for input such as the “Visual Forms Repre- sentation (VFR)” ?les utilized to make PI compliant IFR contents, scripts, tools and documentation have been supplied which permits the user to create text ?les that are treated into formats speci?ed by UEFI or PI speci?cations. FIGURE 8: UEFI/PI Firmware Image Creation There are different layers of structure to a complete UEFI/PI ?rmware image.

These layers are exempli?ed in Figure 8. Every Shifts between layers means that a pro- cessing block that transforms or unites previously treated ?les into the another 16 higher level. Also in Figure 8 portrayed the reference tools utilized that processes the ?les to transit them among the different layers.

The layers are described in more emphasized manner in Section 3. Apart from constructing images that initialize the whole platform, the build process also sustains creation of standalone UEFI applications (such as OS Loaders) and ROM images having driver code. Figure 7 portrayed the reference implementation tools and creation processes for both the image type.

The closing feature that is backed by the EDK II build process is to packaged and distributed the founding of Binary Modules to be utilized by other governing body. Binary modules doesn’t need to distribute the source code. This shall only allow vendors

to publicize UEFI images without releasing copyrighted source code. The process of packaging allows construction of an archive ?le having multiple bi- nary ?les which can be either Firmware Image ?les or higher (FFS, EFI Section ?les, etc.). The build process would only allow insertion of such binary ?les in to the corresponding level of the build stages. 2.3

Platform Initialization PI Boot Sequence Platform Initialization PI compliant system ?rmware has to support the six phases: 1. “Security (SEC)” Phase 2. “Pre-e? Initialization (PEI)” Phase 3. “Driver Execution Environment (DXE)” Phase 4. “Boot device selection (BDS)” Phase 5. “Run time (RT)” services 6. “After Life (AL)” of system. Figure 9 describes the phases and transition in detail. 2.4

Security (SEC) “Security (SEC)” phase is the initial phase through which the boot ?ow begins. This phase is responsible for the following: • Handling restart events of every platform • Creation of a temporary memory stash 17 FIGURE 9: PI Boot Phases • Bringing the trust root in the system • Transit handoff content to next phase - the “PEI Foundation” Figure 10 portrays the Flow of the DXE phase.

The security section may have the modules with source code scripted in assembly language. Hence, some EDK II module development environment (MDE) modules can consist of assembly code. During Occurrence of this, both Windows and GCC versions of assembly language code are served in different ?les. 2.5

Pre-EFI Initialization (PEI) “Pre-EFI Initialization (PEI)” phase represented within the PI Architecture spec- i?cations brought up quite betimes in the period of boot. Especially, after about preliminary processing of the Security (SEC) phase, any system restart event will bring up the PEI phase. The PEI phase is designed to be developed in many parts and consists of: • PEI Foundation (core code) 18 FIGURE 10: SEC Phase 19 • Pre-EFI Initialization Modules (specialized plug-ins) The PEI phase at ?rst operates along the platform in a developing stage, holding only resources on processor, such as the cache for maintaining call stack and dis- patching the “Pre-EFI Initialization Modules (PEIMs)”. Figure 11 portrays the Flow of the DXE phase.

The PEI phase can not presume the convenience of amount of main memory (RAM) as DXE and hence PEI phase support is limited to: • Locates and validates PEIMs • Dispatches of PEIMs • Facilitates commuting of PEIMs • Provides handoff information content to later phases These PEIMs can be considered for accountable for: • Initializing few permanent memory complement • Characterizing the main memory in “Hand-Off Blocks (HOBs)” • Characterizing locations of the ?rmware volume in HOBs • Transit the control ?ow into next phase - the “Driver Execution Environment (DXE)” phase Figure 12

shows a diagram describes the action carried out during the PEI phase 2.5.1

PEI Services “PEI Foundation” institutes a system table for the PEI Services named as “PEI Ser- vices Table” which is viewable to every PEI Modules (PEIMs) exists on the system. A PEI Service could be de?ned as a method, command or some other potentiality manifested by the “PEI Foundation” when the requirements of that service initial- ization are ful?lled.

As the PEI phase having no permanent memory accessible until almost the end of life of the phase, all the various types of services constructed during this phase (“PEI phase”) cannot be as enrich as those constructed during later phases. A pointer referenced to PEI Services Table is sent to the entry point of each and every PEIM and also within part of each “PEIM-to-PEIM Interface (PPI)” because the location of PEI Foundation and its temporary storage memory is un- known at the time of build.

20 FIGURE 11: PEI Phase 21 FIGURE 12: Diagram of PI Operations 2.5.2 PEI Foundation The Phase PEI Foundation carries out following activity: • Dispatching of “Pre-EFI initialization modules (PEIMs)” • Maintaining and managing the boot mode • Initialization of permanent main memory • Conjure the DXE loader The PEI Foundation developed to be portable among all the various platforms ar- chitecture of a speci?ed instruction-set. i.e.

A binary for “IA-32” (32-bit Intel ar- chitecture) works across all Pentium processors and similarly “Itanium® processor family” works on all the Itanium processors. Irrespective of the processor’s micro architecture, the set of service routines un- covered by the PEI Foundation has to be the same. This consistent layered area 22 over the PEI Foundation allows PEIMs to be developed by the “C programming language” and also be compiled across any micro architecture.

The PEI Foundation responsible in providing the service classes listed in Figure 13 FIGURE 13: Services provided by PEI Foundation classes 2.5.3 PEI Dispatcher The implementation of a state machine in PEI Foundation is known as the PEI Dispatcher. It evaluates the dependency expressions (DEPEX) in “Pre-EFI initial- ization modules (PEIMs)” that are lying in the FVs being analyzed.

Dependency expressions (DEPEX) are coherent alliance of “PEIM-to-PEIM Inter- faces (PPIs)”. These expressions distinguish the PPIs that has to be available for use before invoked by a given PEIM. The PEI Dispatcher references the PPI database in the PEI Foundation to conclude which PPIs have to be installed and evaluate the dependency expression for the PEIM.

If PPI has already been installed then de- pendency expression (DEPEX) will evaluate the result to TRUE which informs PEI 23 Dispatcher that it can execute PEIM. At this very stage, the PEI Foundation han- dovers control ?ow to the PEIM with DEPEX result evaluated to TRUE. The PEI Dispatcher will exit when it has examined and evaluated the results of all the PEIMs in all of the uncovered ?rmware volumes and none of the PEIMs can be dispatched (such as the DEPEX do not evaluate from FALSE to TRUE and vice-versa). At this stage, the PEI Dispatcher can not make call to any extra PEIMs.

Control ?ow is than taken back by the PEI Foundation from the PEI Dispatcher and call to the DXE IPL PPI is made to navigate control ?ow to the "DXE phase" of execution. 2.6 Driver eXecution Environment (DXE) Before the "DXE phase" the "Pre-EFI Initialization (PEI)" phase is held liable for initializing the permanent memory on the system platform. Hence, DXE phase could be loaded and executed in the permanent memory.

At the very end of the PEI phase, state of the system is handed over to the DXE phase via utilizing the Hand-Off Blocks (HOBs). Figure 14 portrays the Flow of the DXE phase. DXE phase includes three major components as shown in Figure 15 which work among each other with the aim to initialize the platform and serve the services needed for performing OS boot. 2.7 Boot Device Selection (BDS) The "BDS Architectural Protocol" has part of implementation of the Boot Device Selection (BDS) phase.

After evaluation of all the dependencies of the DXE drivers along with their satis?ed dependencies are loaded and execution is completed by the DXE Dispatcher, the DXE Foundation transfer the control ?ow to the BDS Ar- chitectural Protocol. The "BDS Phase" held liable for: • Initialize the console devices • Load the device drivers • Attempt to load and execute the boot selection 2.8

Transient System Load (TSL) and Runtime (RT) Primarily the OS vendor provides boot loader known as The "Transient System Load (TSL)". TSL and Runtime Services (RT) phases may allow access to persistent 24 FIGURE 14: DXE Phase 25 FIGURE 15: Components of DXE Phase content, via UEFI drivers and applications. Drivers in this category include PCI Option ROMs. 2.9

After Life (AL) The After Life (AL) phase contains the persistent UEFI drivers used to store the state of the system during the OS systematically shutdown, sleep, hibernate or restart processes. 2.10 Generic Build Process All code initialized as either C sources and header ?les, assembly language sources and header ?les, Unicode ?les (UCS-2 HII strings), Virtual Forms Representation ?les or binary data (native instructions, such as microcode) ?les.

Per the UEFI and PI speci?cations, the C ?les and Assembly ?les must be compiled and coupled into PE32 or PE32+ images. While some code is con?gured to execute only from ROM, most UEFI and PI modules code are written to be relocatable. These are written and built different i.e. XIP (Execute In Place) module code is written and compiled to run from ROM, while the majority of the code is written and compiled to execute from memory, which needs the relocatable code.

Some modules may also allow dual mode, where it will execute from memory only if memory is suf?cient, otherwise it will execute from ROM. Additionally, modules may permit dual access, such as a driver that contains both PEI and DXE imple- mentation code. Code is assembled or compiled, then linked into PE32/PE32+ im- ages, the relocation section may or may not be stripped and an appropriate header 26 will replace the PE32/PE32+ header. Additional processing may remove more non- essential information, generating a Terse (TE) image.

The binary executables are converted into EFI ?rmware ?le sections. Each module is converted into an EFI Section consisting of an Section header followed by the section data (driver binary). 2.10.1 EFI Section Files he general section format for sections less than 16MB in size is shown in Figure 17.

Figure 16 shows the section format for sections 16MB or larger in size using the extended length ?eld. FIGURE 16: General EFI Section Format for large size Sections(greater then 16 MB) FIGURE 17: General EFI Section Format (less then 16 MB) 2.11 Cross Compatibility of CPUs Whenever customer try to change the default Intel motherboard CPU with different Intel silicon chip which won't works.

The speci?c CPU Chip initialization varies for each generation. So, the board designs should be designed is such a way that speci?c generation CPU should support., if we change the CPU with a different Intel Board it will not even boot, because the BIOS doesn't support for other Silicon Initialization for other CPUs. 27 So, we are integrating the runtime detection of the silicon during the Pre-Extensible Firmware Initialization (PEI) phase.

So, within single Integrated Firmware Image (IFWI) should support the Multi Generation CPUs which is never tried before. Each silicon has a ?xed register from which the CPU generation can be identi?ed., so the BIOS should read that register and program in such a way the is CPU1 is in Platform it should support the CPU1 Features like PCIe, Graphics & DMI.,

if the CPU1 is replaced with CPU2 then it should support the CPU2 speed. That should be taken care by the BIOS. FIGURE 18: Cross Compatibility Design Figure 18 shows the general view of the Cross Compatibility of CPUs. BIOS is the part of Integrated Firmware Image which resides at the End of the Bi- nary table.

The CPU swap can only occur in Specially designed Intel Designed Board only. Mainly because for each and every feature it required some hard- ware(H/W) requirements. If that H/W requirement not present. Then It will boot but doesn't support the Maximum speed. Figure 19 shows the BIOS role for identifying the CPUs during PEI phase. As the number of Feature increases in the Silicon BIOS size also increases, usu- ally the BIOS size varies from Platform to Platform and CPU to CPU.,

as we are integrating the Compatibility the BIOS size obviously increases. The structure of IFWI is Shown in Figure 20 28 FIGURE 19: BIOS Support for Cross Compatibility FIGURE 20: Integrated Firmware Image 29 3. Architecture of BIOS Firmware 3.1

Overview If you interpret BIOS image as close look then it is nothing but the ?le system which is made in a byte format to be read by low level programming language which is most ef?cient method to store the data or content. The concept of initialization of Platform includes the execution of this BIOS image which is stored on the every SoC The components which plays role in Platform Initialization are listed below: • Firmware Volume (FV) - consists of one or more ?rmware ?le systems • Firmware File System (FFS) which consists of one or more ?rmware ?les • Firmware File - Encapsulated section or leaf section • Reference Layout of Binary • Pre-EFI Initialization (PEI) PEIM to PEIM Interfaces (PPIs) • Driver Execution Environment (DXE) Protocols 3.2

Design of Firmware Storage Design of ?rmware storage elaborates the way that how ?les needs to be stored and accessed in nonvolatile storage environment. Implementation of any ?rmware has to support and follow the standard structure for PI Firmware Volume and the structure of FFS. Firmware Device - a persistent physical repository consisting data and/or ?rmware code.

Typically it is a component of ?ash but may also be any other type of persistent storage. Singular physical ?rmware device can be partitioned in to multiple other smaller pieces to form many other logical ?rmware devices from it and vice-versa. Flash devices are most usual nonvolatile storage mechanism for ?rmware vol- umes.

Often, ?ash devices are partitioned into many sectors or blocks of potentially differing sizes, each along with various runtime characteristics. In the design of Firmware File System (FFS), several observed unique qualities of ?ash devices are listed below: 30 •

Erase operation processed on the basis of sector-by-sector. After ensuring, every bits of sector return their erase value1. • Write operation can be performed on a bit-by-bit basis. i.e.

In case erase value is 0 then bit value 0 can be changed to 1. • Only by performing erase operation on the whole ?ash sector, non-erase value can change to erase value. • Capable of enable/disable reads and writes to individual ?ash sectors or the entire ?ash. • Operations like writes and erases are much longer than reads operation.

• Many times places restraints on the trading operations that can be executed while a write or erase is in progress. 3.3 Firmware Volume (FV) The BIOS image is consisting of one or more logical ?rmware devices known as a Firmware Volume (FV). Firmware Volume is the very basic and ef?cient logical storage mechanism for data and/or code.

If you consider ?le system as a basic unit then ?rmware volume is unionized into these one or more ?le system units. Table 3 describes attributes in each ?rmware volume. Apart from this Firmware volumes also consisting of few more information about the correspondence between OEM ?le types and a GUID. 3.4

Firmware File System (FFS) The logical data payload within ?rmware volume is known as a Firmware File Sys- tem (FFS) which illustrates the structure of ?les and free space (if any). To af?liate a driver to ?rmware volume every ?rmware ?le systems contains a globally unique ID (GUID). Firmware ?les consists of code or raw data or both. Attributes of ?les are described in Table 4.

Integrity check and staged ?le creation are some extra attribute formats which might spotted in some ?rmware volume. Firmware File Sections are unit which unionized in a standard fashion to form certain ?le types for the ?le data. OEM ?le types (described in detail in Figure 21) enables to support non-standard ?le types.

1either all 0 or all 1 31 TABLE 3: Firmware Volume Attributes Attribute Description Name each volume has a unique identi?er name having UEFI Glob- ally Unique Identi?er (GUID). Size describes total size of all data (includes all information like headers, ?les and free/reserved space) Format describes type of Firmware File System (FFS) which is union-ized in construction of the volume.

Memory is Mapped or not? some volumes may requires to be memory-mapped which de- termines whether the entire content of the volume can ap- pear at once in the processor’s memory address space. Sticky Write? Speci?es whether or not special erase cycles requires in order to change value of bits into an erase value from non-erase value

Erase Polarity In case a volume supports Sticky Write, then after processing an erase cycle every bits in the volume will return to this value (0 or 1) Alignment A volume is required to be aligned on some power- of-two (2 x) boundary such that m i n i m u m >= highest ?le alignment value.

Enable/Disable Read capable status Decides whether to keep volumes as hidden from readable or not Enable/Disable Write capable Status Decides whether to keep volumes as hidden from writable or not Lock Capable/Status Volumes could also have their locking mechanism Read-Lock Capable/S- tatus Volumes could also have the power to lock their read status Write-Lock Capa- ble/Status Volumes could also have the power to lock their write status 32 TABLE 4: Firmware Files Attributes Attribute Description Name each volume has a unique identi?er name having UEFI Glob- ally Unique Identi?er (GUID). Name of the File(s) has to be unique within a same ?rmware volume.

Type Type of the individual ?le which can be Normal, OEM, De- bug, FV Speci?c. More ?le types information are described in Figure 21. Alignment Every data of ?le to be aligned on some power-of-two (2 x) boundary such that these boundaries are founded depending on the alignment of ?rmware volume. Size Describes size of each ?le which consists of data of size zero or more bytes PEI phase is responsible to serve the ?le related services which are carried out using PEI Service Table.

On the Other hand the EFI_FIRMWARE_VOLUME2_PROTOCOL services which are attached to a volume's handle (ReadFile, ReadSection, WriteFile and GetNextFile) are responsible to carried out ?le related services in DXE phase. 3.4.1 Firmware File Types If you consider an application with ?le name such as XYZ.exe, in which content format of XYZ.exe is implied by the ".exe" in the ?le name.

Based on the situa- tion of operation, this extension normally signals the contents of XYZ.exe. The PI Firmware File System characterizes the contents of a ?le that is returned by the ?rmware volume interface. Firmware File System of the Platform Initialization dictates an enumeration of many ?le types.

For example, the type EFI_FV_FILETYPE_RAW implies that the ?le is a RAW Binary Data. In the same way, ?les with the type EFI_FV_FILETYPE_SMM_CORE supports MM traditional mode . 3.5 Firmware File Section Firmware File Section is individual distinct unit of certain ?le types which has following attributes: 33 FIGURE 21: Firmware File Type Attribute Description Type Each section has type Size describes size of the section 34 However as many as types of sections are present, they eventually fall in one of the below broadly described categories: • Encapsulation section - logical storage consisting of the one or more sec- tion.

The child section(s) which are lying within the encapsulation section (parent section) can be another encapsulation section or a leaf section which are also called relative peers to each other. An encapsulation section never consists of data in itself; however it is just a container that ultimately ends in leaf section(s). Files which are stacked with section can be imagined as tree consisting of nodes (encapsulation section) and leaves (leaf section).

The root which can be interpreted as the ?le image itself may have a discrete number of sections. Sections that exist in the root have no parent section but are still considered peers. • Leaf Sections - Contrary to the encapsulation section, leaf section does con- tain data and only data within it.

Type of section de?nes which kind of data is stored within the leaf section. As illustrated in Figure 22, the root which we interpret as the ?le image has two encapsulation sections which are E0, E1 and one leaf section which is L3. E0 which is the ?rst encapsulation section possessing three child node which are all leaves (L0, L1, and L2).

E1 is the another encapsulation section which possesses only two children, where one of them is encapsulation (E2) and the another is the leaf (L6). E2 which is the very last encapsulation section consists two children which are both leaves only (L4 and L5). With the help of FfsFindSectionData, services related to section are populated with the help of PEI Service Table in the PEI phase.

On the other hand ReadSection which is attached to service protocol EFI_FIRMWARE_VOLUME2_PROTOCOL responsible to populate services related to section during the DXE phase. 3.6 Firmware File Section Types Subjective types of section are described in Table 5. 3.7 PI Architecture Firmware File System Format Basic encoding of binary used for PI ?rmware ?le, ?rmware volume and ?le sys- tem is illustrated in this section.

Development which carries out the non-vendor ?rmware ?les or ?rmware volumes to be enclosed into the system must have the standard formats. These sections also describes the way features of the standard format mapped into the standard interfaces of DXE and PEI. 35 TABLE 5: Types of Section Name of Section Value Details EFI_SECTION_COMPRESSION 0 x1 non-leaf section contain- ing compressed section(s) within EFI_SECTION_GUID_DEFINED 0 x2 non-leaf section which only to be used while in process of build and not for execu- tion EFI_SECTION_DISPOSABLE 0 x3 non-leaf section which only to be used while in process of build and not for execu- tion EFI_SECTION_PE32 0 x10 Image executable of PE32+ EFI_SECTION_PIC 0 x11 Code independent of posi- tion EFI_SECTION_TE 0 x12 Image of Terse Executable

EFI_SECTION_DXE_DEPEX 0 x13 Expression for DXE driver dependency EFI_SECTION_VERSION 0 x14 version of the section - tex- t/numeric EFI_SECTION_USER_INTERFACE 0 x15 Human readable and eas- ily interpretable name for driver EFI_SECTION_COMPATIBILITY16 0 x16 16-bit exe of DOS fashion EFI_SECTION_FIRMWARE_VOLUME_IMAGE 0 x17 PI Firmware Volume Image EFI_SECTION_FREEFORM_SUBTYPE_GUID 0 x18 Raw data with GUID in header to de?ne format EFI_SECTION_RAW 0 x19 Raw data EFI_SECTION_PEI_DEPEX 0 x1 b Expression for PEI driver dependency EFI_SECTION_SMM_DEPEX 0 x1 c Leaf section which deter- mine the order of dispatch for the MM Traditional driver in SMM.

36 FIGURE 22: Example File System Image The standard format of ?rmware ?le and volume also brings in extra dimensions and potential that are used to assure the unity of ?rmware volume. The standard format is unionized by three different levels: ?rmware volume, ?rmware ?le system, and ?rmware ?le. The guided formatting of ?rmware volume (Figure 23) made of two parts: The FV header and FV data.

Header of FV describes every attributes mentioned in “Firmware Volumes” in Table 3. This header also has GUID which identi?es for- mat of the ?rmware ?le system utilized to orchestrate data in the ?rmware volume. The “?rmware volume header” is compatible with every other ?rmware ?le systems except the PI Firmware File System.

“Firmware File System format” explains the way the ?rmware ?les and free space are conceived inside the ?rmware volume. However on the other hand “Firmware 37 FIGURE 23: The Firmware Volume Format File format” describes how ?les are organized. The ?rmware ?le format made of two parts: the ?rmware ?le header and the ?rmware ?le data. 3.7.1

Firmware Volume Format The PI Architecture Firmware Volume format key outs the binary structure of a ?rmware volume. The ?rmware volume format possesses a FV header followed by the FV data. The FV header is represented by variable EFI_FIRMWARE_VOLUME_HEADER. The format layout of the FV data is described by a GUID. Valid ?les system GUID values are EFI_FIRMWARE_FILE_SYSTEM2_GUID and EFI_FIRMWARE_FILE_SYSTEM3_GUID. 3.7.2

Firmware File System Format The PI Architecture Firmware File System is a binary design of logical ?le storage within ?rmware volumes. It is a ?at ?le system in which there is no rendering of any directory hierarchy structure. Each and every ?les lies directly in the root of the storage. Files are stored end to end without any directory entry to explain which ?les are present.

Parsing the information stored in a ?rmware volume to ?nd a itemization of ?les exists needs the complete walk through over the ?rmware volume in and out. Firmware File System GUID The ?rmware volume header has a unique data ?eld for the ?le system GUID. The two valid FFS ?le systems are de?ned by the GUID values in variable EFI_FIRMWARE_FILE_SYSTEM2_GUID and EFI_FIRMWARE_FILE_SYSTEM3_GUID.

In case of the FFS ?le system, if it does allows ?les larger than 16 M B along with backward compatibility EFI_FIRMWARE_FILE_SYSTEM2_GUID then EFI_FIRMWARE_FILE_SYSTEM3_GUID is used. 38 Volume Top File known as VTF is a ?le that has to be presented such that the very last byte of ?le is also the very last byte of the ?rmware volume. Irrespective of type of the ?le, a VTF have to have GUID for the ?le name which is declared as variable EFI_FFS_VOLUME_TOP_FILE_GUID.

Driver cide if Firmware ?le system has to be exposed of this GUID and in?x an alignment pad ?le as and when needed to assure that the VTF is situated correctly at the top of the ?rmware volume. Length and alignment of File requirements needs to be coherent with the top of volume so that a write error does occurs and the unwanted ?rmware volume modi?cation can be prevented. 3.8

Firmware File Format (FFS) Every FFS ?les begins with its header data that is aligned on an 8 - b y t e (which is power-of-two 23) boundary with respect to the origin of the ?rmware volume. FFS ?les consists of the below parts: • Header • Data When a zero-length ?le is created without any data it still has to have header and will consume minimum 24 b y t e s of space. The data (if any) exists in ?le then it immediately conjugated after the header.

How the data within a ?le is formed can be identi?ed by the “Type” ?eld in the header which can be either of EFI_FFS_FILE_HEADER and EFI_FFS_FILE_HEADER2. Figure 24 exempli?es the typical layout of a (i.e. EFI_FFS_FILE_HEADER) “PI Archi- tecture Firmware File” (= 16 M b). Figure 25 exempli?es the typical layout of “PI Architecture Firmware File” (> 16 M b). 3.9

Firmware File Section Format Storage Data format mechanism of section is described in this section. Each indi- vidual section starts with a section header which is followed by the data de?ned using the section type. Section headers are always aligned at 4 - b y t e boundaries with respect to the start of the ?le image.

In case of padding required between the section then to achieve the 4 - b y t e alignment as de?ned, every bit value of padding is set to zero. There some section types which are variable in terms of data length and are more precisely represented as data

streams instead of data structures. Irrespective of type of the section, all section headers starts with a 24 - b i t inte- ger telling the section size, and 8 - b i t section type.

The format of the rest of the 39 FIGURE 24: Layout representation of FFS File Header (= 16 M b) FIGURE 25: Layout representation of FFS File Header 2 layout for ?les (> 16 M b) section header and data is de?ned by the section type. If size of the section size is 0 x F F F F F F then the size is de?ned by a 32 - b i t integer that follows the 32 - b i t section header.

Figure 26 and Figure 27 shows the typical layout of a section data format. 3.10 File System Initialization To ensure unity of the ?le system it is mandatory to maintain state byte of each ?le correctly such that it won't compromised even in case of power failure during operation on any FFS.

It is desired that an FFS driver produces an instance of 40 FIGURE 26: Section Header Format when s i z e < 16 M b FIGURE 27: Section Header Format of when s i z e = 16 M b using Extended Length ?eld Firmware Volume Protocol so that every normal ?le operations carried out in that context. Every ?le operations has to follow all the rules of creation, update, and deletion mentioned in this speci?cation to avoid corruption of the ?le system. 3.11 Traversal and Access to Files The Security (SEC), PEI, and early DXE code needs to be capable to traverse the FFS such that it's read and execute operation on ?les carried out before a write- enabled DXE FFS driver is started it's execution so that the FFS may not have any inconsistencies because of any kind of previous system failure. Hence, it has to follow a set of rules to assert the credibility of ?les before using them.

It is not incumbent on SEC, PEI, or the early read-only DXE FFS services to make any effort to perform recovery or modi?cation the ?le system. If any case exists where execution can not continue because of inconsistencies in ?le system, a recovery boot must be initiated. As there is one mutual exclusiveness that the SEC, PEI, and early DXE code can affect without instantiating a recovery boot.

This condition can be summoned by 41 any previous system failure such as power failure that come along while a ?le up- date on a previous boot. In such case, a failure can cause two ?les with an identical ?le name GUID to coexist within the same ?rmware volume where one of them will have the EFI_FILE_MARKED_FOR_UPDATE bit set to its state ?eld but are going to be otherwise totally valid ?le. The another ?le may be in unknown state of building up to and including EFI_FILE_DATA_VALID.

All ?les used preceding to the initialization of the write-enabled DXE FFS driver must be ?ltered with this test prior to their use. If this condition is observed, it's tolerable to

trigger a recovery boot and allow the recovery DXE to perform the completion of update. Note There's no ascertain for redundant ?les once a ?le found in the EFI_FILE_DATA_VALID state.

The condition where two ?les in same ?rmware volume coexist having the same ?le name GUID and both are within the EFI_FILE_DATA_VALID state cannot occur if the set of rules for creation and update are strictly followed. 3.12 File Integrity and State File corruption, no matter the cause, must be detectable in order to carried out ap- propriate steps for ?le system repair.

File corruption can come from various sources but broadly falls into three categories listed below: • Any general failure • Failure on erase • Failure on write A general failure is characterized to be evidently random corruption of the storage media. This corruption can occur because of the design problem or obsolete storage media i.e.

This type of failure can be as perceptive as replacing any single bit inside the ?le content. Using a good design of system along with reliable storage media, general failures can be avoided. However, the FFS enables catching of this kind of failure. An erase failure happens when a block erase of ?rmware volume media isn't com- pleted because of any system failure i.e. power failure.

As the erase operation is not outlined, it is likely that most of the implementation of FFS that allow ?le write and delete operations will also develop a mechanism to rectify deleted ?les and unite free space. In case the operation is not carried out successfully, the ?le system can be left out in a state which is not consistent. Likewise, a write failure takes place when a ?le system write is in motion and is left incomplete because of any system failure i.e. power failure.

This type of failure can lead the ?le system to be in an inconsistent state. 42 All of these failures can be traced while FFS initialization is in progress hence de- pending on the cause of the failure, many recovery schemes can be carried out. 43 4. System Management Mode (SMM) 4.1

Overview On "IA-32 processors", System Management Mode (SMM) is a manner of operation which is different from ?at modal so as from protected mode of the DXE and PEI phases. It is outlined as a real mode environs with 32 - b i t data bus access and its carried out in effect to either with a speci?ed interrupt type or with the System Management Interrupt (SMI) pin.

Note that Operation mode of SMM is OS inde- pendent mode and is discrete

operational mode, however it may also lies in both within and OS runtime. FIGURE 28: SMM Framework Architecture 4.2 System Management System Table (SMST) System Management System Table (SMST) is core mechanism of SMM handler to pass information and enabling activity.

SMST table allows access to service based on the SMST which also known as SMM Services. Driver can only use SMM services during execution inside context of the SMM. EFI_SMM_BASE_PROTOCOL.GetSmstLocation() service used to discover the address of SMST. SMST is a set of potentiality exported for utilization by any driver that is loaded into SMRAM.

It's similar to the EFI System Table except that by design it's ?xed set of services and data and also doesn't acknowledge to the resiliency of an EFI protocol interface. 44 SMM infrastructure component of Framework provides SMST, which manages: • Dispatching drivers inside SMM • Allocation of memory (SMRAM) • Switching the framework in and out of the applicable SMM of the processor 4.3 SMM and Available Services 4.3.1

SMM Services As EFI runtime drivers have their constraints, similarly the model of SMM frame- work will have them too. Especially, dispatch of drivers in SMM won't be capable of using any core protocol service. However SMST-based services, called SMM Ser- vices allows the drivers to be access using an SMM identical of the EFI System Table, but services of the core protocol won't grantees the availability in runtime.

As an alternative, the complete mass of EFI Boot Services and EFI Runtime Ser- vices can be available while the driver loading or "constructor" phase. With utilizing the visibility of constructor, SMM driver is capable to leveraging rich set of EFI service to perform: • Marshall interface for EFI services. • Observing EFI protocols which are populated by other SMM drivers while in constructor phase.

For drivers while not in SMM and during the initial load inside SMM, EFI protocol database becomes quite useful by utilizing design. Available services which are SMST-based includes: • Negligible, blocking kind of the device Input/Output protocol • Memory allocator from SMRAM These services are exposed through the entries present in the System Management System Table (SMST). 4.3.2

SMM Library During constructor phase of SMM driver inside SMM, all additional service within SMM Library (SMLib) are uncovered like EFI protocols. i.e., An identical status code in SMM is merely an EFI protocol with interface referencing an SMM-based driver service. To avoid error or information of progress during runtime, other SMM drivers

also locates this SMM based status code. 45 4.4 SMM Drivers 4.4.1

Process to Load Drivers in SMM Process to load driver modal in SMM is merely a DXE SMM runtime driver having a DEPEX (dependency expression) having at least EFI_SMM_BASE_PROTOCOL. This kind of dependency is essential as the DXE runtime driver which is planned for SMM will utilize the EFI_SMM_BASE_PROTOCOL to load up itself again in SMM and re-execute its entry point. Also, other SMM-loaded protocols permitted to be stayed in the DEPEX of speci?ed SMM DXE runtime driver.

Principle of the DXE Dis- patcher is to verifying if the GUIDs to be consumed by the protocols does exists in the database of protocol and capable to identifying if the driver can be loaded or not. After formerly loaded in SMM the DXE SMM runtime driver becomes capable uti- lize only minor set of services. While in its constructor entry point, the driver can use EFI Boot Services as it executes within space of boot service and SMM.

In secondary entry point in SMM driver is capable to perform: • Registration of an interface - In the formal protocol database, naming the SMM occupant interfaces with future-loaded SMM drivers • Registration with the SMM codebase - For a callback hook in effect to an SMI pin stimulation or an SMI based interrupt message from outside of SMM Code (i.e.

a boot service, runtime agent) After this constructor phase in SMM, the SMM driver needs not be rely on any other boot services as the mode of operation to carrying out execution can move away from these services. Many EFI Runtime Services could possess the majority of their execution shifted into SMM and viewable runtime portion simply becomes a proxy which merely utilizes the EFI_SMM_BASE_PROTOCOL to perform callback in SMM to carry out services.

By Possessing a proxy which allows for a modal of sharing code blocks of error handling, like services for ?ash access and also the EFI Runtime Services GetVariable() or SetVariable(). 4.4.2 SMM Drivers for IA-32 In SMM the “IA-32 runtime drivers” can not called as on the image the action by SetVirtualAddress() is performed. Hence, code segment that requires to be ac- cessible among SMM and EFI runtime needs to be migrated in SMM. 46 4.4.3

"Itanium® Processor Family" SMM Drivers From “Platform Management Interrupt (PMI)” the runtime drivers for the “Ita- nium® processor family” are called as if each of them is kind of “Position Indepen- dent Code (PIC) runtime driver”. 4.5 SMM Protocols System Architecture of SMM broke in to below two parts: • “SMM Base Protocol” - exposed by the processor.

This protocol is liable to perform: – To initialize the state of processor – Registration of the handlers • “SMM Access Protocol” - interprets the speci?c enabling and locking mecha- nism that an IA-32 memory controller may allows during execution in SMM. (Not needed for “Itanium® processor family”) 4.5.1 SMM Protocols for IA-32 Figure 29 shows the SMM protocols which are published for an IA-32 system. FIGURE 29: Protocols Published for IA-32 Systems 47 4.5.2

SMM Protocols for "Itanium®-Based Systems" Figure 30 shows the way SMM protocols are published for an “Itanium®-Based system”. FIGURE 30: Protocols Published for "Itanium®-Based Systems" 4.6 SMM Dispatcher and infrastructure SMM Code segment lies within the SMM Dispatcher. Major Role of SMM Dis- patcher is to give the control mode to the SMM handlers in a systematic method- ology.

SMM Infrastructure Code aids to drive communication for SMM to SMM. SMM handlers are PE32+ images. 4.7 Initializing SMM Phase The SMM driver for the Framework is fundamentally a enrollment transport mode to dispatch the drivers in outcome to the: • System Management Interrupts for “IA-32” • Platform Management Interrupts (PMIs) for “Itanium® processor family” 48 4.8

Relation of "System Management RAM (SMRAM)" to con- ventional memory Figure 31 shows relationship between SMRAM and main memory in IA-32. Where SMRAM is isolated secure part inside the conventional main memory. FIGURE 31: SMRAM kinship with conventional memory 4.9 Execution Mode of SMM on Processor SMM is acceded asynchronously with the ongoing main ?ow of program.

SMM was primitively developed to be clear to the OS and provide a power management facility more transparent. Preboot agents are responsible to initiate alternate uses of SMM which are: • Applicable Workaround for SoC exaggeration • Logging of error(s) • Security for the platform A SMI can be launched by energizing either the SMI logic pin via dedicated on the board or by utilizing the local APIC.

“Itanium® architecture” possess no independent separate mode for processor for the tractability of interruption however it does supports “Platform Management In- terrupt (PMI)” which indeed is a maskable interruption. However, there is this an- other way to enter PMI using a interrupt message on local “Streamlined Advanced Programmable Interrupt Controller (SAPIC)”.

This architecture informs a techniques to load modules of needful code segment that substantiate the functionality speci?ed above. The internal representation of protocol

which enables the loading of images of various handler and runs in normal memory of boot-services. Only the handlers does to run in SMRAM. 49 4.10 Accessing Platform Resources As par policy outcome process of the execution of SMM handlers is reasonably pre- vents from accessing conventional memory resources.

Hence, there does not exist any ease binding technique such as a call or trap interface to render the services in preemptive bid of non-SMM state. Besides, SMM Services - the library of service which abides a sub set of the core EFI services, i.e. device input-output protocol, memory allocation and others. Also, execution mode of SMM driver has the equivalent structure as per the EFI criterion - namely a components which executes under boot services and it could perhaps run in runtime mode. When ExitBootServices() invoked, the mechanism of an unregister event occurs. 50 5. Proposed Work In general to generate BIOS image (*.rom ?le), compilation of XYZ.c

(source code) has to be done, this compilation not only involves compilation of DXE driver, PEI driver, EFI Application but also includes pre-processing checks, compression of raw ?les which takes huge amount of time depending on the system con?guration. Im- plementation of this project aids in reduction of this compilation time. 5.1

Stake holders The proposed work is applicable but not limited to below stake holders: • BIOS development team : main development group in contributing BIOS ?rmware, this is the only stake holder who are having access to the BIOS development environment and access to the source code of the complete BIOS ?rmware • Validation team : performs various validation on developed BIOS image • Automation team : brings various integration and validation automation to module(s) • Other Development team who wishes to ease the debugging process 5.2

Issues The Proposed work is capable of mitigating below issues: • Generation of BIOS image - includes compilation of whole source code • Time complexity - took enormous amount of time to generate the BIOS image • Accessing and modifying BIOS Setup Option(s) remotely • Firmware Flashing of BIOS remotely • Updating CPU microcode • Summarizing changes among BIOS image • Avoiding exposing the source code support for OEM to ?ll their OEM informa- tion • Avoid setting of BIOS development platform for stake holders which are not meant to be the BIOS developer • Runtime BIOS Support for temporary UEFI variable creation 51 5.3 Requirements 5.3.1

Software Requirements • Visual C/C++ binaries • Python 3 • Visual Studio Code (IDE) • Memory Access Interface - supported mechanism to communicate over target memory 5.4 Development Process of Modules Framework development process is driven by implementation of independent mod- ules which can serve functionality and having

?exibility to integration to the frame- work. 5.5 Module: Setup Knob modi?cation 5.5.1

Processing Unsigned debug BIOS Before Releasing the BIOS ?rmware for public use, those are signed for security and integrity purpose, however the debug BIOS which are used Pre-release to test and verify all the functional features until all the requirements are met. Every SoC system which are under test known is SUT are con?gured in such a way that it supports debug BIOS.

The proposed framework is designed to simulate the process of SUT in terms of processing BIOS binary similar to SUT performs it after ?ashing BIOS ?rmware on SoC. Processing the debug BIOS can be classi?ed in to two ways: 1. Applying changes directly to the SUT 2. Applying changes on to the BIOS image At the high level the ?ow for both the above classi?cation remains the same but will be differentiated at the backend support.

An additional driver is attached with BIOS ?rmware to aid the framework to be able to apply changes directly to the SUT. 52 5.5.2 Additional Tech Stack Used Below are the listed technologies consumed in development of this module in addi- tion to the already speci?ed requirements in Section 5.3 • Tkinter • XML • JSON 5.5.3 Flow of the module Figure 32 describes the ?ow of setup knobs modi?cation on the System Under Test (SUT).

The iteration of the development could be reduce in two ways: 1. Processing Debug/Unsigned BIOS in section 5.5.1 2. Processing Firmware individually in section ?? 5.5.4 Screenshots of Module As a PoC for the framework, this section shows snapshots of the working module to mimic the setup options of BIOS, however as a simulation framework, it also provides quite more features which are not available in the actual BIOS due to memory limitation.

Figure 33 shows the prompt asked to user to select basic con?gurations before launching the module of framework. Con?gurations available to select are: • Working Mode (options to be selected as in ?gure 34) – online - to work on SUT and require to select valid access method for online mode from menu – offline - to work on BIOS binary • Access Method - selecting valid access method for working on SUT • Publish all? - Boolean options to decide whether to evaluate DEPEX or not.

Table 6 describes the interpretation of each button action on speci?c condition as remarks if applicable 53 FIGURE 32: Flow of Setup Knobs Modi?cation 54 FIGURE 33: Menu to Select initial con?guration for work FIGURE 34: Available work mode for the system: Online and Of?ine 5.5.5 Outcome of Module • The module is capable of cross

platform usage. • The module can work with all the platform binary and SUT.

• A communication bridge as a driver in BIOS ?rmware to aid the framework run directly on SUT is implemented. • Generic solution is provided for end-user while running any of the classi?ca- tion listed in 5.5.1. 55 TABLE 6: Interpretation of buttons on Virtual Setup Page GUI Button Interpretation Push Changes Apply changes to system if online mode else apply changes to 'bin' ?le View Changes View saved changes in new window Exit Exit the GUI Reload Reload the GUI Discard Changes Discard any change made, any value if mod- i?ed are restored to current value Load Defaults Restore to default values and revert any changes made • Simulating the information from system or binary image is provided as native GUI application. • Real time sync with simulation framework is supported. • Seamless Integration of any new features or modules in framework is made possible. 5.6

Module: Parsing Figure 35 represents the overview of the BIOS as a File system which is interpreted and parsed from the BIOS image. Detail architecture of the same is explained in Section 3. 5.6.1 Additional Tech Stack Used Below are the listed technologies consumed in development of this module in addi- tion to the already speci?ed requirements in Section 5.3 • Decompression binaries • XML • JSON 56 FIGURE 35: Overview of BIOS image as a File System 57 5.6.2

Flow of the module Figure 36 describes the ?ow of the Parsing module. The Initial part is performed by user who is responsible to select valid memory interface to work. Note that some memory interface are supported by the module which requires additional hardware and software setup which are considered to be the part of dependency of interface itself which is not in the scope of the module.

When User select valid Interface the module will determine whether user is on Target SUT or on the local BIOS image. If user is working on SUT with valid memory interface and privileges then BIOS image will be parsed from the memory. FIGURE 36: Flow of Parser As on both the cases BIOS Image is available to act on, the module will start the parsing of the BIOS image as interpretation described in Figure 35. It parses All the valid ?rmware volumes only till the end of BIOS image (skips the free space or ?rmware volumes with invalid signature and GUID).

Decompression of ?le system under the ?rmware volume if any is handled by the module too, for the decompres- sion of ?le system it uses the binary for decompression technique available to public i.e. lzma, tianocore, brotli etc. 5.6.3 Outcome of Module • Human Readable interpretation of BIOS image is provided. • Possible to debug the BIOS via setup knobs comparison. • Lookup of order of the module in BIOS image as readable

?le system is also possible.

• Veri?cation of integration of module via GUID can be done. • Extracting and storing ?le system or module of BIOS image by GUID • Summarizing changes of two BIOS image 58 5.7 Module: Runtime UEFI variable Creation Each variable in BIOS has a scope for each variable where Runtime support is one of the attribute, to simply state the run time variable one can interpret it as the variable which will be available during and after the completion boot ?ow (while OS is running).

Such a variable require special access mechanism, which is carried out by the System Management mode SMM described in Section 4.. Earlier Challenges are described as below: • Providing and maintaining native driver support from BIOS for creation of UEFI variable • Setting of Build environment for non-BIOS development team Note: As all the variable created at runtime the scope of such variable are limited to the ?ashing of the BIOS. i.e.

when BIOS is ?ashed/re-?ashed or updated, those variable won't be available on the SUT. 5.7.1 Additional Tech Stack Used Below are the listed technologies consumed in development of this module in addi- tion to the already speci?ed requirements in Section 5.3 • Flask • Ajax • jQuery • Javascript • HTML/CSS • XML • JSON 5.7.2 Flow of the module The Flow of the module is described in section 5.7.3

along with screenshots which is easier to interpret the ?ow chart in Figure 37. 59 FIGURE 37: Flow of Nvar Web GUI 5.7.3 Screenshots of Module Whenever the User launches the module the home page screen to select valid com- munication interface will appear as displayed in Figure 38. This is the crucial stage as if valid interface for communication is not selected one may not be able to use the functionality of the service.

After selection of valid Interface one may operate the desired options listed in navi- gation bar which are: Figure 39 lists the variable created under the current session which is to be applied Figure 40 displays form which allows user to create Variable, where user needs to specify the name of the variable with certain restriction of input ?eld.

To identify and lookup the Variable the GUID is required which is automatically generated by the module with required format, however if user wishes then they can modify the GUID. 60 FIGURE 38: Home Page to Create UEFI Variable TABLE 7: Navigation Bar Action Button Interpretation Create Variable Opens a form to create new Variable as in Figure 40 Display Created Variable lists out created variable as in Figure 39 Generate XML Generate XML from the stored session database as in Figure 47 Save XML Saves the generated XML on the storage de- vice Save to SUT Applies the Pending changes

action (Cre- ate/Delete/Modify) to the SUT View JSON View the stored session database in the json format as in Figure 48 61 FIGURE 39: Variables created or exists on SUT FIGURE 40: Create new UEFI Variable on SUT Figure 41 opens the list of the options if created and allows to edit their current val- ues too. However one can also add the new option to the Variable.

It allows user to create various types of options under the variable which are oneof type as in Figure 43, string type as in Figure 44, numeric type as in Figure 45 and the checkbox type which allows user to toggle the option value in as Boolean interpretation. Common ?elds for creating options including its name, type, description and size.

If user wants to change the value set for the variable created while creation of option as described in Figure 42 forms one can actually modify the value. The highlighted prompt in Figure 43 allows user to create the choices for the option where one of the multiple values to be selected as a result, By clicking Add Option 62 FIGURE 41: Options listed under Variable FIGURE 42: Edit the Existing Option Created under Variable SUT button user can create choices and under drop down menu besides Value, user can select default value to be selected for the option.

Option type string as in Figure 44 allows user to create a option which accepts minimum and maximum characters to be supported in the string as well as the default string value to be selected. To set the numeric input for the option, minimum and maximum value along with the default value to be set as in Figure 45 For the future use one can create a reserved space under the UEFI variable as in Figure 46 63 FIGURE 43: Create New Option(s) under Variable - Oneof Type FIGURE 44: Create New Option(s) under Variable - String Type Figure 47 shows the XML which is generated from the existing and newly created variables and options under it.

Figure 48 represents session data of existing and newly created data (if any) as json 5.7.4 Outcome of the module • Enables creation of UEFI variable from OS layer. 64 FIGURE 45: Create New Option(s) under Variable - Numeric Type FIGURE 46: Create Reserved Space for future use under Variable • Lifts headache of maintaining variable creation from BIOS development for individuals. 65 FIGURE 47: Generate XML SUT FIGURE 48: Generate XML SUT 66 6.

Future Scope of Work Few implementation modules of Section 5. are not well developed at production launch which a slight modi?cation and standard checks have to be performed to make the modules qualify for production level. Also as the release of production other stuff to be maintained is user guide, FAQs and other "how-to" articles to help out others to ease in using the framework.

Along with the enhancing of existing modules there will still be exercise to analyze existing system to explore more use cases which are taking a longer time for every build iteration for the system. Few of the possible use cases to study and decide the feasibility of implementation would be: • Development and testing of individual driver component rather than building the whole BIOS image • AI powered Search Engine to enhance the ?ndings of FAQs for relevant exist- ing queries and articles • Automating the initial BIOS Environment Setup • Platform independent easy installation setup for the framework 67

INTERNET SOURCES:

<1% - https://in.linkedin.com/in/parth-panchal-9a1446168
<1% - https://arxiv.org/pdf/1909.06493
<1% - http://researchspace.ukzn.ac.za/xmlui/bitstream/handle/10413/11824/Emser_Monique_2013.pdf?sequence=1
<1% - https://www.researchgate.net/profile/Devharsh_Trivedi/publication/277131825_Near_Field_Communication/links/55621e2b08ae8c0cab3332ea/Near-Field-Communication.pdf?origin=publication_detail
<1% - https://www.reddit.com/r/uwo/comments/gedi14/is_an_honour_specialization_in_computer_science/
<1% - https://vit.ac.in/sites/default/files/ProformaforsubmissionofMPhildissertation.pdf
<1% - http://shodhganga.inflibnet.ac.in/bitstream/10603/4382/3/03_declarations.pdf
<1% - https://c.mql5.com/forextsd/forum/224/Automated%20Stock%20Market%20Trading%20System%20using%20Machine%20Learning.pdf
<1% - http://www.ijareeie.com/volume-2-issue-6
<1% - https://in.linkedin.com/in/rutvik-patel-191150a9
<1% - https://www.intel.co.jp/content/dam/altera-www/global/ja_JP/pdfs/literature/rn/cv_hps_rn.pdf
<1% - https://www.toptenreviews.com/best-driver-update-software
<1% - https://es.scribd.com/document/96578137/Mother-Board-INTEL-D945GNT-TechProdSpec
<1% - https://www.globalspec.com/Industrial-Directory/system_on_a_chip_%28soc%29
<1% - https://www.velvetjobs.com/resume/development-intern-resume-sample
<1% -

http://tudr.thapar.edu:8080/jspui/bitstream/10266/5057/1/801632024_Manpreet_CSED_2018.pdf
<1% -
https://www.researchgate.net/profile/Parth_Shah8/publication/278937384_Automated_Stock_Market_Trading_System_using_Machine_Learning/links/5587d16808aef58c03a05c03.pdf
<1% - https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2889244/
<1% -
http://static6.arrow.com/aropdfconversion/678fa8a71a245a8053ab757cd4bc4d22dadbf363/975950277507591399402353-ds.pdf
<1% -
http://cesg.tamu.edu/wp-content/uploads/2012/01/RUIA-THESIS-2015-final-1.pdf
<1% -
https://www.ibm.com/support/knowledgecenter/9080-M9S/p9eab/p9eab_980_slot_details.htm
<1% -
https://software.intel.com/en-us/forums/developing-games-and-graphics-on-intel/topic/754892
<1% -
https://edk2-docs.gitbooks.io/edk-ii-build-specification/content/2_design_discussion/23_boot_sequence.html
<1% -
http://www.lannerinc.com/phocadownload/software/SDK-page/LEC-3031/BIOS/LEC-3031%20BIOS%20User%20Manual%20V1.0.pdf
<1% - https://patents.google.com/patent/US20040268107A1/en
<1% - https://blogs.coreboot.org/blog/category/uefi/
<1% -
https://docs.oracle.com/en/engineered-systems/private-cloud-appliance/2.4/admin-2.4.2/admin-pca-update-firmware.html
<1% -
https://sites.google.com/site/uefiforth/bios/uefi/platform-initialization-pi-specification/volume-3-shared-architectural-elements/2-firmware-storage-design-discussion/2-2-pi-architecture-firmware-file-system-format
<1% - https://www.amd.com/system/files/TechDocs/32559.pdf
<1% -
https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf
<1% - https://www.fda.gov/media/71610/download
<1% -
https://www.sec.gov/Archives/edgar/data/79731/000119312503054118/dex991.htm

<1% - https://archive.org/stream/springer_10.1007-b116432/10.1007-b116432_djvu.txt
<1% - https://idoc.pub/documents/3236-1430zxrg924j
<1% - http://0x0atang.github.io/files/asplos19_hix.pdf
<1% - https://www.sciencedirect.com/science/article/pii/S0167839620300042
<1% - https://ark.intel.com/content/www/us/en/ark/products/199345/intel-xeon-gold-6238r-processor-38-5m-cache-2-20-ghz.html
<1% - https://www.manualslib.com/manual/651589/Medion-Notebook.html
<1% - https://www.researchgate.net/publication/318360748_PCIe_BUS_A_State-of-the-Art-Review
<1% - https://www.aja.com/solutions/thunderbolt
<1% - https://hakzsam.wordpress.com/tag/intel-vt-d/
<1% - https://www.manualslib.com/manual/788923/Winmate-I771.html
<1% - https://www.manualslib.com/manual/77755/Intel-Se7520jr2.html
<1% - https://www.computerhope.com/jargon/b/bootdevi.htm
<1% - http://mason.gmu.edu/~fzhang4/paper/iocheck-esorics14.pdf
<1% - https://www.inspirit.net.in/books/misc/Upgrading%20and%20Repairing%20PCs.pdf
<1% - https://en.wikipedia.org/wiki/Secureboot
<1% - https://forum.osdev.org/viewtopic.php?f=1&t=23923
<1% - https://sites.google.com/site/uefiforth/bios/uefi/uefi/1_introduction/1-1-uefi-driver-model-extensions
<1% - https://www.electronicdesign.com/technologies/dev-tools/article/21800617/the-migration-from-legacy-bios-to-uefi-firmware
<1% - https://unix.stackexchange.com/questions/560027/centos-8-not-booting-going-to-dracut-resume-mode
<1% - https://techlibrary.hpe.com/docs/iss/proliant_uefi/UEFI_Edgeline_103117/GUID-09B4FCA8-D750-408D-9CEA-B78118768C33.html
<1% - https://www.deepdyve.com/lp/association-for-computing-machinery/energy-efficient-sensing-with-the-low-power-energy-aware-processing-qeaBKxRkzQ
<1% - https://www.kernel.org/doc/ols/2005/ols2005v1-pages-59-76.pdf
<1% - https://acpica.org/about
<1% - http://gauss.ececs.uc.edu/Courses/c4029/doc/acpica-reference.doc
<1% - https://www.scribd.com/document/19214137/Linux-Linux-Symposium-Procv1

<1% - https://genode.org/documentation/release-notes/16.05
<1% - https://acpica.org/sites/acpica/files/acpica-reference_18.doc
<1% - http://dictionnaire.sensagent.leparisien.fr/EPCI/en-en/
<1% -
https://www.pcisecuritystandards.org/documents/PCI-DSS-v3_2-SAQ-P2PE-rev1_1.docx
<1% -
http://pdf.cloud.opensystemsmedia.com/vita.opensystemsmedia.com/Various.Feb04.pdf
<1% - https://en.wikipedia.org/wiki/PCI_Local_Bus
<1% - https://www.slideshare.net/element14/xio2200a-pci-express-to-1394a-chip
<1% -
https://www.intel.com/content/dam/doc/guide/uefi-driver-graphics-controller-guide.pdf
<1% -
https://centos.pkgs.org/6/forensics-x86_64/fmem-kernel-modules-1.6-1.19.noarch.rpm.html
<1% - https://www.ijraset.com/fileserve.php?FID=8243
<1% - https://www.iceteks.com/news/search.php/PHP
<1% - https://www.haiku-os.org/docs/userguide/en/bootloader.html
<1% - https://bbs.archlinux.org/viewtopic.php?id=232197
<1% - https://cvit.iiit.ac.in/images/ConferencePapers/2017/DocUsingDeepFeatures.pdf
<1% -
https://edk2-docs.gitbooks.io/edk-ii-build-specification/content/2_design_discussion/22_uefipi_firmware_images.html
<1% - https://quick-geek.github.io/articles/440052/index.html
<1% -
https://mafiadoc.com/beyond-bios-developing-with-the-unified-extensible-firmware-_59f0339d1723dd05c8778003.html
<1% -
https://rog.asus.com/forum/showthread.php?91778-Q-Code-67-quot-CPU-DXE-Initiailzation-is-started-quot
<1% - https://www.slideshare.net/vedantsrivastava/unmanned-aerial-vehicle-for
<1% -
https://fosdem.org/2020/schedule/event/firmware_culisfu/attachments/slides/3709/export/events/attachments/firmware_culisfu/slides/3709/FOSDEM_2020_Intel_Capsule_Update.pdf
<1% - https://it-atnet.blogspot.com/2009/10/
1% -
https://edk2-docs.gitbooks.io/edk-ii-build-specification/content/2_design_discussion/25_generic_build_process.html
<1% - https://link.springer.com/article/10.1007%2Fs40069-013-0064-x

<1% -
https://stackoverflow.com/questions/52397267/most-efficient-method-to-store-3d-indexed-coordinate-data-pair-in-a-dynamic-var
<1% -
https://www.mssqltips.com/sqlservertip/2548/using-bit-columns-with-nulls-when-three-options-exist-in-sql-server/
<1% - https://www.sciencedirect.com/science/article/pii/S2405844015303364
<1% - https://www.dostips.com/DosCommandIndex.php
<1% - https://s.campbellsci.com/documents/es/manuals/loggernet%20-%20475.pdf
<1% - https://docs.oracle.com/cd/B19306_01/appdev.102/b14249/adlob_intro.htm
<1% -
https://ccna-course-training-institute-delhi.blogspot.com/2010/11/what-is-ripv2how-to-configure-rip-in.html
<1% - https://www.informit.com/articles/article.aspx?p=376123
<1% -
https://sites.google.com/site/uefiforth/bios/uefi/platform-initialization-pi-specification/volume-3-shared-architectural-elements/2-firmware-storage-design-discussion/2-2-pi-architecture-firmware-file-system-format/2-2-1-firmware-volume-format
<1% - https://docs.microsoft.com/en-us/dotnet/standard/io/file-path-formats
<1% - https://www.msbtechnology.com/faq/select-firmware/
<1% - https://link.springer.com/article/10.1007/s40903-015-0018-5
<1% - http://homepage.divms.uiowa.edu/~stramer/S39/lec3class.pdf
<1% - http://abacus.bates.edu/~ganderso/biology/resources/writing/HTWtablefigs.html
<1% -
https://archive.org/stream/MANUALFORFFSMACHINEAztec/MANUAL%20FOR%20FFS%20MACHINE_Aztec_djvu.txt
<1% - https://www.math.columbia.edu/~woit/QM/fall-course.pdf
<1% - https://www.guru99.com/dbms-transaction-management.html
<1% -
https://github.com/MicrosoftDocs/windows-itpro-docs/blob/master/windows/security/threat-protection/windows-defender-system-guard/system-guard-secure-launch-and-smm-protection.md
<1% -
https://www.researchgate.net/publication/295010710_Booting_an_Intel_System_Architecture
<1% - http://esec-lab.sogeti.com/feeds/rss.xml
<1% -
https://docplayer.net/18646000-A-tour-beyond-bios-supporting-an-smm-resource-monitor-using-the-efi-developer-kit-ii.html
<1% -

https://www.scribd.com/document/134914424/Intel-EFI-System-Management-Mode
<1% - https://flylib.com/books/en/4.491.1.31/1/
<1% - https://patents.google.com/patent/US7171624B2/en
<1% - http://www.patentsencyclopedia.com/class/000555743
<1% - http://www.women.jo/DetailsPage/JNCW_Ar/TendersAr.aspx?ID=3116
<1% - https://simp.readthedocs.io/_/downloads/en/6.2.0-0/pdf/
<1% -
https://www.blog.qualitypointtech.com/2010/01/software-testing-quiz-questions-and.html
<1% -
https://www.researchgate.net/publication/327948197_A_machine_learning_approach_to_generate_test_oracles
<1% - https://www.tutorialspoint.com/jboss_fuse/what_is_fuse.htm
<1% - https://www.aafp.org/fpm/2004/0300/p53.pdf
<1% -
https://software.intel.com/en-us/articles/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent
<1% - https://www.schneier.com/blog/archives/2014/01/nsa_exploit_of.html
<1% - https://docs.oracle.com/cd/E18727_01/doc.121/e13408/T335476T429683.htm
<1% -
https://docs.oracle.com/middleware/1221/soasuite/develop/GUID-83DAC40D-A3FE-4530-8832-9C51FA63DE9D.htm
<1% -
https://studynotesunisa.co.za/wp-content/uploads/2019/01/Book-answer-summaries-1-10.docx