Chapter 3

UEFI: FROM RESET VECTOR TO OPERATING SYSTEM

What we call the beginning is often the end. And to make an end is to make a beginning. The end is where we start from.—T.S. Eliot

Vincent Zimmer, Michael Rothman and Robert Hale

Abstract In PCs, the firmware that sits at the reset vector is called a BIOS. The BIOS has increased in size, complexity, and extensions apace with the complexity and richness of PCs. The increases have finally reached the point that no amount of patching will fix the old architecture. The new architecture, known as the Unified Extensible Firmware Interface (UEFI) [UEFc, ZRH] and Platform Initialization (PI) [UEFb] keep the learning's of the last years but impose a modern software engineering structure that supports the basic requirements of system initialization, configuration, and abstraction of boot devices, but which is also designed to be extensible enough to address the new features of hardware to come.

Keywords: BIOS, Firmware, UEFI, EFI, Boot ROM

3.1 Introduction

Time was when computers were humans who did computations. Then computers became machines that filled rooms, requiring constant attention by specially trained operators. Automation all but completely replaced the human computers.

The machines in those early days were finicky and crashed often. Rebooting was a time consuming and complex process requiring the operator to flick switches and load tapes in a complex order. An untold number of discoveries, and an even larger number of hours of effort later, computers are personal, ubiquitous and, at least relatively, easy to use.

As a part of that evolution, automation has all but completely replaced the human operators. Untrained end-users now seem to do what operators trained for years to do in the past, and on systems that are arguably vastly more complex. The simple boot ROM, auto-loader, or, in the PC world, BIOS, has gone a long way to replace the operator, and, in a real sense, carried the PC revolution on its back.

Boot ROMs were at first quite small and simple, if for no other reason because ROMs were quite small and expensive. Minicomputers in the early 1980s still booted from 512 byte ROMs. In 2008 the average BIOS ROM is 512 K bytes or 1024 times as large as its minicomputer counterpart. The first BIOS for the IBM* Personal Computer was 8 K bytes. Servers can use BIOS of over well over 4 MB, four times the size of the PC's entire address space.

This chapter is about what has prompted the growth in BIOS. It is also about the next generation of BIOS, how it is architected, and some of the subtle capabilities it provides. With the specifications now well over 2000 pages in length, it cannot do so in any great detail. Instead, we try to give some idea of the underlying motivations, design decisions, and possibilities inherent in this fundamental part of modern computers.

3.2 The Ever Growing Ever Changing BIOS

The system BIOS (the BIOS on the motherboard) was initially divided into two parts, POST ("for Power On Self Test") and Run-time. POST gains control via the reset vector and is responsible for the initialization, testing, and configuration of on-board devices.

The problem of 'drivers' for add-in cards was solved most elegantly by providing 'Option ROMs', pieces of code that allow the BIOS to access the cards' devices. These option ROMs were non-volatile memory components which typically resided on the hardware of the add-in card itself. In implementation, however, Option ROMs were always second class citizens, with difficulty getting control after initialization and even accomplishing fundamental tasks like allocating memory.

The final function for POST is to load the initial bootstrap loader for the Operating System and hand control to it. The intent was then that the OS would use the BIOS as an abstraction for accessing on-board devices using the BIOS run-time interfaces.

The BIOS design was quite successful in its initialization role. The run-time abstractions were at a low enough level that they proved to be only really useful so the OS could access boot devices (video, keyboard, hard drive) before it had loaded its own drivers.

Over the years, requirements changed, initially prompting a set of marginally related BIOS extensions and finally prompting the definition of a completely new architecture.

3.2.1 The Evolution of the Power-On Self-Test

POST's goal is to provide the OS with a functioning fully configured platform. That is certainly not what the BIOS finds at the reset vector. The BIOS typically finds little other than the basic processor and ability to access its own memory functional. The rest must be initialized.

System initialization comprises three basic parts: finding the devices, configuring the devices, and testing the devices. BIOS devices traditionally cover a different set of components than an operating system: the chip that generates the frequencies used on the rest of the motherboard is a device to a BIOS but is typically invisible to an OS.

In the early days of BIOS, finding devices (enumeration) was not supported since it required additional hardware support that was too expensive at the time. As the hardware prices have decreased and end user support costs increased, device enumeration became more important. Modern buses including PCI, PCIe, ATA, SCSI, and USB have mechanisms which BIOSs use to locate and configure devices.

The BIOS is responsible for configuring the chips in the chipset. Chips are designed for basic functionality at reset. They require configuration to adapt them to each other, the board, and add-in devices and, in some cases, to work around bugs in the Silicon. Given the extremely high cost of hardware changes, BIOS workarounds are strongly preferred. This requirement, as much as any, motivated the BIOS to move from ROMs to Flash. Flash can be reprogrammed in the field, reducing recalls and improving support.

Given the high integration of modern PCs the utility of most BIOS testing has disappeared: The fact that the BIOS runs proves that most of the motherboard is working. It is more common to test connections to peripheral devices. Oddly, RAM is treated as a peripheral until it is initialized and tested.

The most complex and ever changing enumeration requirement is RAM. In early days, RAM initialization could occur within the first several hundred instructions and typically took less than a page of assembly code. Today's RAM must be located and its characteristics discovered (using the System management bus (SMBUS) [SMBUS]) and described to the chipset using algorithms that require thousands of lines of C code. This added complexity does allow for use of faster memory and adapting memory speeds to board and chipset.

The complexity of the BIOS routines stems from the nature of the hardware/chipset. For example, fast paged-mode Dynamic Random Access Memory (DRAM) in the mid-90's was initialized with maybe 50 lines of code. The algorithm was simple: a table had the five or six settings of the memory controller, the BIOS would attempt each setting and test to see if it "worked" (e.g., no aliasing, bits written could be read back, etc.). A modern memory controller for double data-rate DRAM, such as DDR3, may take several thousand lines of code to read serial eeprom data from the memory part, train the analog channel/compensation, and finally program the memory. So BIOS typically tracks the complexity of the hardware. And add to this complexity comprehending the requirements of various vendors who alternately provide the chipset, central processing unit (CPU), system board, clock generator chip, DRAM modules, SMBUS controllers, etc.

As time has passed, the BIOS has taken on other requirements. The BIOS is now responsible for describing the board it resides on to the operating system and applications that manage the system. Items described include board type, asset numbers, and how the board's power management features are to be accessed.

3.2.2 Run Time Evolution

The initial interfaces which abstracted peripherals were creatures of their time. They were real mode (8 or 16 bit). They mirrored the devices of their day: the maximum address space supported on a hard drive was 540 MB and the maximum memory space supported was one megabyte. In the intervening years, the industry has adopted a series of specifications, some clean and elegant, some not so, to extend the interfaces where required while retaining backwards compatibility.

3.2.3 Software Engineering

The remarkable thing about most BIOS is the amount of code that is reused. BIOS developers routinely expect to reuse over 95% of the code from platform to platform. This is a result of careful design and diligent monitoring as well as schedules that permit no alternative. A byproduct of this reuse has been a high degree of consistency and compatibility. Code is written once, tested once, and reused millions of times. Code that is known to change (chipset, processor) is isolated and cocooned with interfaces.

Traditionally, BIOS have been developed in assembly language and managed using commonly available source code control systems. Custom tools are used to manipulate the results to fit in the parts.

The BIOS run-time, while limited and archaic has been consistent enough that many generations of operating systems have booted and continue to boot on what are essentially extensions to the same interfaces. It thus may come as a surprise that there is no consistency internally between different vendors BIOS. The internal structures and architecture are almost completely incompatible. It is impossible for a company to provide a single set of code that works unmodified when integrated into all system BIOS.

3.3 Time for a Change

By around 2000, it became clear to many in the industry that the interfaces that had served us well for 20 years had become obsolete. Thirty two bit operating systems were hobbled by booting using 8 bit interfaces; 64 bit operating systems would simply not work. There was simply no way to fit the required code in the single 512 byte boot sector BIOS allowed for the first stage OS boot loader.

Again, the industry could have defined yet another patch on the existing system to allow for extensions to the 512 bytes and have lived for another year or two. But then the 64 bit operating systems would be using 8 bit interfaces to load subsequent boot stages. It became clear that something more modern was required. Existing alternatives were examined and, one by one, rejected.

A modern set of interfaces had to be defined that met the needs of the Operating System community and the system developers. There was also substantial agreement that the never quite solved problems of the option ROM vendors should be resolved by this new solution.

3.3.1 EFI and UEFI

The new set of interfaces was known as the *Extensible Firmware Interface* or EFI for short. The U (for *Unified*) was added a few years later, when an industry forum took over ownership of the specification.

The fundamental structures in EFI define extensibility, acknowledging that technology will evolve. Software engineering advances in the intervening 20 years were embraced by creating what amounts to an object architecture. This architecture was designed to be usable by all classes of systems from deeply embedded/handheld platforms to mission-critical, large, scalable servers. Basic services such as memory allocation and resource management were made a part of the core set of services. The fundamentals have the feel of an embedded non-preemptive real-time system.

Traditional interfaces, such as those for various types of peripherals, were defined using the extensible interfaces as will be defined in future years for new devices not yet thought of. Importantly, this enables option ROMs to become full members of the system. EFI embraces the idea of driver-like interfaces that exist only during boot and a much more minimal 'run-time' set of calls.

A new disk partitioning methodology was also defined which allows for greatly expanded number of larger sized partitions than what had previously been available.

The interfaces have been extended to support more devices (iSCSI for example) and more advanced features such as the Human Interface Infrastructure (HII), which supports mechanisms to support user and remote configuration of system devices with all of the localization and similar support expected of a modern system.

Backwards compatibility is a hard thing to grow out of. We do not expect to see the last of the old "legacy" BIOS interfaces disappear for many years. We are now on our way with UEFI.

3.4 UEFI and Standardization of BIOS

The BIOS evolved from crisis to crisis with small groups forming to address a need and driving industry adoption. EFI has been more encompassing and specification driven from the start. To continue that model, and to achieve input and buy-in from the industry, the Unified EFI Forum ("UEFI") was created in 2005. This organization now owns the UEFI specification, covering the interfaces between Boot Firmware, OS, and Option ROMs, and the Platform Initialization specifications, covering common interfaces between the reset vector and UEFI.

At the highest level, the UEFI Specification covers all the data one might expect to be described for launching a boot target. However, when digging into a little more detail, a reader quickly realizes that the UEFI specification covers many concepts that one might expect in a modern operating system. Ultimately, the intent of the UEFI specification is to address the issues in the pre-operating system environment that are known today, but also to provide sufficient extensibility to the described infrastructure so that the underlying architecture should be able to easily adapt to changing technology.

3.4.1 Providing Interface Extensibility

In the UEFI programming environment, the interfaces which a firmware component (i.e. UEFI Driver) would provide are commonly known as "protocols". The protocols describe the parameters and data which are exchanged when programmatically interacting with a UEFI driver. To ensure that there is a uniform interpretation of these interfaces, the specification clearly defines a contract which associates a 128-bit globally unique identifier (GUID) with the described interface description. When bound together, a GUID and an interface (itself nothing more than a something like a C struct) form a protocol.

An example of installing an instance of the EFI_SERIAL_IO_PROTOCOL can be shown in the following example from the ISA serial driver in the open source EFI Developer Kit (EDK) [EDK]:

```
#define EFI_SERIAL_IO_PROTOCOL_GUID\
{0xBB25CF6F, 0xF1D4, 0x11D2, 0x9A, 0x0C, 0x00, 0x90, 0x27,
 0x3F, 0xC1, 0xFD}

EFI_GUID gEfiSerialIoProtocolGuid =
EFI_SERIAL_IO_PROTOCOL_GUID;
typedef struct _EFI_SERIAL_IO_PROTOCOL {
    UINT32 Revision;
    EFI_SERIAL_RESET Reset;
    EFI_SERIAL_SET_ATTRIBUTES SetAttributes;
    EFI_SERIAL_SET_CONTROL_BITS SetControl;
    EFI_SERIAL_GET_CONTROL_BITS GetControl;
    EFI_SERIAL_WRITE Write;
    EFI_SERIAL_READ Read;
    EFI_SERIAL_IO_MODE *Mode;
} EFI_SERIAL_IO_PROTOCOL;

EFI_STATUS EFIAPI SerialControllerDriverStart (
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE Controller,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath
)
// Routine Description:
// Start to management the controller passed in Arguments:
//    This - pointer to the EFI_DRIVER_BINDING_PROTOCOL
//              instance.
//    Controller - handle of the controller to test.
//    RemainingDevicePath - pointer to the remaining portion
//                          of a device path.
// Returns: EFI_SUCCESS - Driver is started successfully
{
EFI_STATUS Status;
EFI_INTERFACE_DEFINITION_FOR_ISA_IO *IsaIo;
SERIAL_DEV *SerialDevice;
UINTN Index;
UART_DEVICE_PATH Node;
EFI_DEVICE_PATH_PROTOCOL *ParentDevicePath;
EFI_OPEN_PROTOCOL_INFORMATION_ENTRY *OpenInfoBuffer;
UINTN EntryCount;
EFI_SERIAL_IO_PROTOCOL *SerialIo;
SerialDevice = NULL;
```

```
// Get the Parent Device Path
Status = gBS->OpenProtocol (
    Controller,
    &gEfiDevicePathProtocolGuid,
    (VOID **) &ParentDevicePath,
    This->DriverBindingHandle,
    Controller,
    EFI_OPEN_PROTOCOL_BY_DRIVER
);
if (EFI_ERROR (Status) && Status != EFI_ALREADY_STARTED) {
    return Status;
}

// Grab the IO abstraction we need to get any work done
Status = gBS->OpenProtocol (
    Controller,
    EFI_ISA_IO_PROTOCOL_VERSION,
    (VOID **) &IsaIo,
    This->DriverBindingHandle,
    Controller,
    EFI_OPEN_PROTOCOL_BY_DRIVER
);
if (EFI_ERROR (Status) && Status != EFI_ALREADY_STARTED) {
    goto Error;
}

// Issue a reset to initialize the COM port
Status = SerialDevice->SerialIo.Reset(&SerialDevice->SerialIo);

// Install protocol interfaces for the serial device.
Status = gBS->InstallMultipleProtocolInterfaces (
    &SerialDevice->Handle,
    &gEfiDevicePathProtocolGuid,
    SerialDevice->DevicePath,
    &gEfiSerialIoProtocolGuid,
    &SerialDevice->SerialIo,
    NULL
);
```

Since the underlying UEFI environment is designed for extensibility, this means that UEFI provides mechanisms for agents to advertise UEFI protocols for use by others. For instance, participation in this level of interaction is not limited to components which were shipped with a platform. It is fully expected that add-in devices (e.g. RAID, network, etc.) when plugged into the system will have the capacity to export their own interfaces which the firmware can use.

Figure 3.1 shows a timeline of events where the UEFI subsystem will discover and launch drivers regardless of if the driver was present during platform

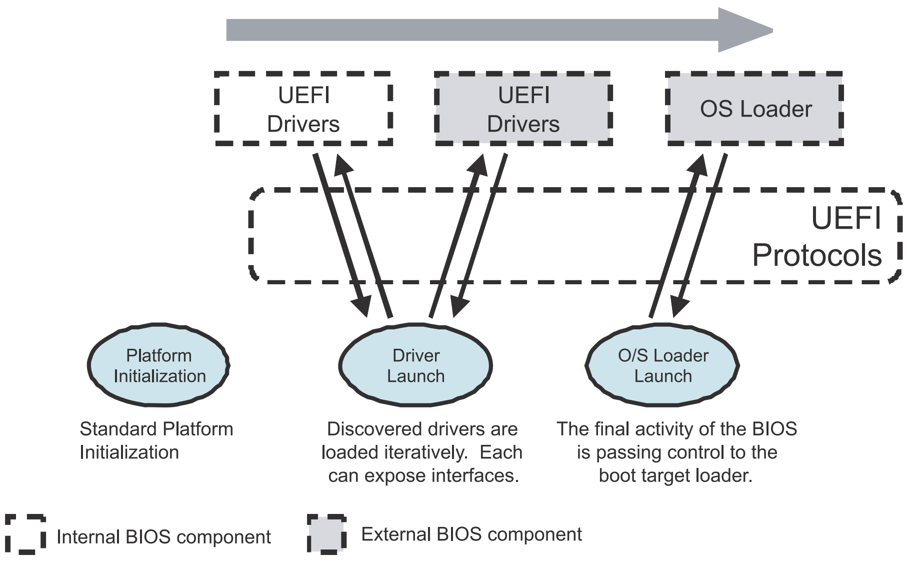

UEFI Drivers
UEFI Drivers
OS Loader
UEFI Protocols
Platform Initialization
Driver Launch
O/S Loader Launch
Standard Platform Initialization
Discovered drivers are loaded iteratively. Each can expose interfaces.
The final activity of the BIOS is passing control to the boot target loader.
Internal BIOS component
External BIOS component

Figure 3.1. BIOS component interaction.

construction or not. One thing to note which is somewhat surprising to many is that even the operating system (O/S) loader (which is provided by the O/S) is a UEFI-compliant driver which directly uses the underlying UEFI protocols.

The drivers involved in the boot process need to ensure a substantial level of cooperation since a platform may find itself launching an add-in device's driver which exposes Block I/O abstractions to storage media. The platform may then discover an O/S vendor's O/S loader, which in-turn will use a variety of other exposed BIOS abstractions. These standard interfaces make it possible for a large contingent of BIOS components to be constructed and run in a portable fashion on various UEFI codebases.

3.4.2 The Boot Processing Logic

The UEFI specification defines a variety of Non-volatile random access memory (NVRAM) variables which are intended to indicate platform policy. These variables encompass various configuration related settings (e.g. current language to use, current console to use, etc.) as well as variables associated with what drivers to load during the initialization process.

Table 3.1 enumerates the variables commonly associated with the boot process. The listed variables can be placed into two subcategories, one which is related to being a final boot target, and the other being an auxiliary driver which is not intended to be a final boot target. The primary distinction is that

Boot####	A boot load option
BootOrder	An ordered boot load option list
BootNext	The boot option for the next boot only
Driver####	The driver load option
DriverOrder	An ordered driver load option list

Table 3.1. Variables associated with boot processing.

the Boot* oriented variables are anticipated to be the final item(s) launched by the underlying BIOS.

It should be noted that some of the variables above have some #### notation included in their name. The #### represents a unique number in printable hexadecimal representation using the digits 0–9, and the upper case versions of the characters A-F. The #### will always be four digits so small numbers will use leading zeros.

Both the Boot#### and Driver#### variables contain data which relates to "where" the driver is located. The location associated with these drivers is described using something known as a device path.

A device path is a means of describing a programmatic path to a particular device. With the aforementioned boot processing variables, sufficient information can be understood from the variable content so that all the enumerable buses can be discerned and the location of the driver can be determined.

When an operating system is installed, one of the normal processes it undertakes would be for it to add a reference to its O/S loader as a Boot#### variable. The BootOrder determines which Boot#### variables are to be executed and in what order, and BootNext is used when across the next platform reset and only on the next platform reset a particular driver needs to get launched first.

3.4.3 The UEFI System Partition

Since the O/S loader is typically a UEFI compatible driver, the ability for the underlying UEFI infrastructure to find the O/S loader is required. UEFI codebases are required to have the ability to interpret certain basic file systems (e.g. FAT32), but most modern operating systems have evolved to using other file systems which the UEFI subsystem may not be able to interpret. UEFI defines the concept of a system partition which can be used by vendors to store UEFI drivers.

In Fig. 3.2, we illustrate how a platform partition may be dedicated as a UEFI system partition, including the Logical Block Addresses (LBA) of the disk media. In addition, this partition is required to have been formatted using the FAT (File-Allocation-Table) file system so that items stored in this repository can easily be retrieved by the UEFI BIOS. Platforms with UEFI-based

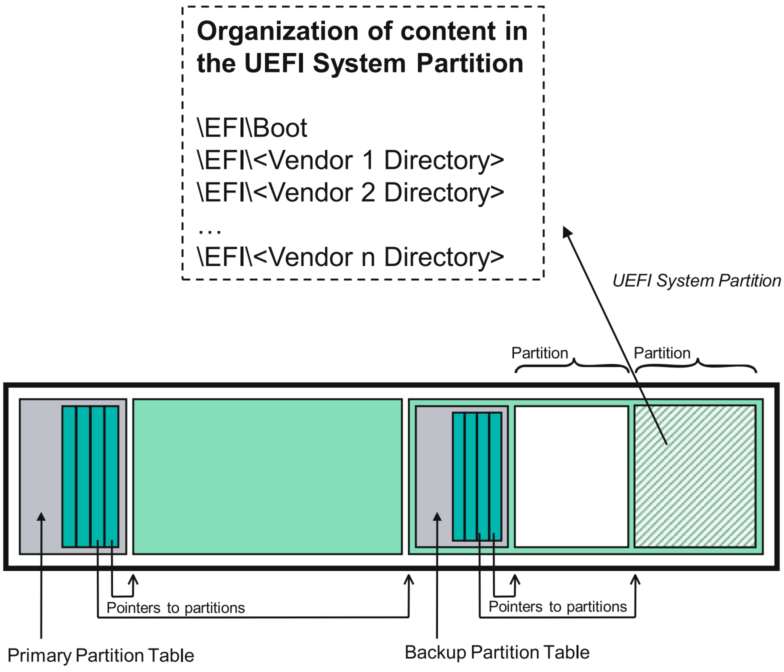


Figure 3.2. GUID Partition Table Scheme.

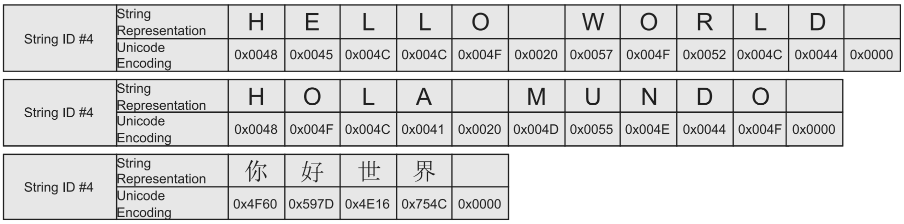
BIOS must support FAT12, FAT16, and FAT32 variants of the FAT file system. What variant the system partition is formatted with is defined by the size of the partition itself.

One very common usage of the system partition is for the O/S to store its O/S loader; however it is very reasonable to envision a usage where various other UEFI compatible platform utilities are placed in this same area. With various different parties vying for the usage of this common repository, it was envisioned that there might be file naming conflicts. To try to address this situation as much as possible, a registry was constructed [UEFa] so that different vendors could place material on the system partition without as much of a concern about file name collisions.

3.4.4 Advances in Configuration Infrastructure

The ability to advance interoperability of various BIOS components were not limited solely to items which are largely invisible to the end-user. Even though BIOS has largely been a user-invisible technology, which in other words could be phrased, "In almost all situations, BIOS should not be noticed by the user aside from a possible splash screen", the expansion of the capabilities associated with the BIOS as well as a robust programming environment has anticipated that user-visible solutions based on the underlying BIOS would come about. These solutions would encompass solutions that are pretty standard operations such as configuration of both platform and add-in devices, as well as possible other solutions which might provide user interfaces.

To facilitate the acceptance of these solutions, several areas were addressed to simplify shipping BIOS solutions in a global market. One area had to do with localization of text. The configuration infrastructure that was put in place into the most recent versions of UEFI has the support for string tokenization. This made it so that drivers could much more easily support multiple languages in a given string reference. In lieu of hard-coded references to strings a program would now simply be able to reference a string via its string number. The simplicity in this enables that there is no special software that needs to be written by users of UEFI systems to support multiple languages. As the example in Fig. 3.3 illustrates, one simply references String #4, and based on the current language setting of the UEFI system, the appropriate language would be retrieved.



String ID #4	String Representation	H	E	L	L	O		W	O	R	L	D	
	Unicode Encoding	0x0048	0x0045	0x004C	0x004C	0x004F	0x0020	0x0057	0x004F	0x0052	0x004C	0x0044	0x0000

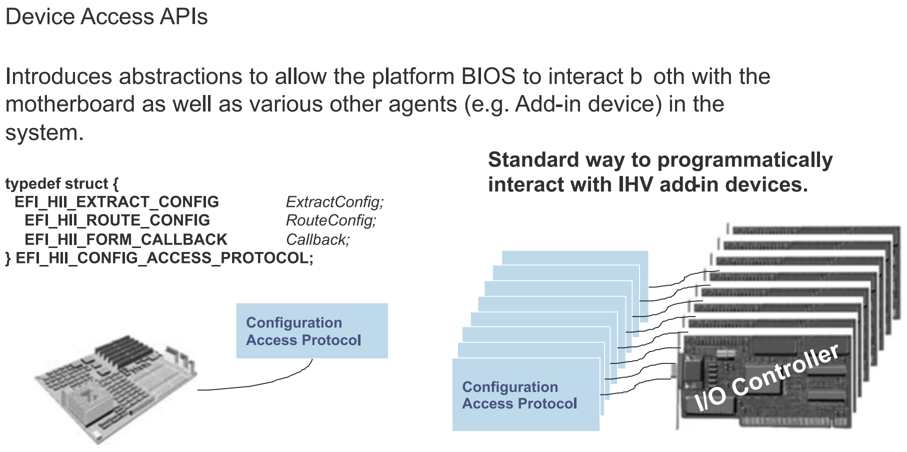
String ID #4	String Representation	H	O	L	A		M	U	N	D	O	
	Unicode Encoding	0x0048	0x004F	0x004C	0x0041	0x0020	0x004D	0x0055	0x004E	0x0044	0x004F	0x0000

String ID #4	String Representation	你	好	世	界	
	Unicode Encoding	0x4F60	0x597D	0x4E16	0x754C	0x0000

Figure 3.3. Example of string tokenization.

In addition to strings, the ability for a platform to display characters has always been fixed by the platform vendor. This posed issues with the ability for third party vendors to provide strings which might not be displayable by the platform being executed on. The UEFI configuration infrastructure introduces a means by which character glyphs can be introduced by third parties so that these glyphs can be used to help render what would previously had been undisplayable strings.

The interaction between the BIOS and the add-in devices has always been a black box operation in that there was no programmatic interaction between the components. There was also no way for the BIOS to discern any information from the device aside from what was described by the bus that the device was plugged into. In UEFI, the configuration infrastructure enables devices to provide configuration access protocols which can facilitate a variety of interaction that was formerly impossible. In addition, since this infrastructure also demands that configurable devices contribute their content (e.g. strings, forms, etc.) in a standard form, the BIOS can now proxy user interface functional-


Device Access APIs

Introduces abstractions to allow the platform BIOS to interact b oth with the motherboard as well as various other agents (e.g. Add-in device) in the system.

```
typedef struct {
  EFI_HII_EXTRACT_CONFIG      ExtractConfig;
   EFI_HII_ROUTE_CONFIG       RouteConfig;
   EFI_HII_FORM_CALLBACK      Callback;
} EFI_HII_CONFIG_ACCESS_PROTOCOL;
```

Standard way to programmatically interact with IHV add-in devices.
Configuration Access Protocol
Configuration Access Protocol
I/O Controller

Figure 3.4. Add-in devices can now be programmatically configured.

ity for the device as well as make the content portable so that it can be used remotely, locally, or even in the O/S.

Figure 3.4 describes the EFI_HII_CONFIG_ACCESS_PROTOCOL published by each configurable I/O controller driver. Given these protocols, a single platform device manager can interact with each device and provide a consistent user interface (or remote access) to the devices.

3.5 Framework, Foundation, and Platform Initialization

EFI solved the more visible issues in the firmware. The issues of interoperability inside the BIOS architecture were no less profound but much more isolated.

Intel took the lead in defining what it hoped to be a unified architecture which is now owned by the UEFI Forum and known as the Platform Initialization (PI) specification. As its basis, PI uses the same structures and core services as found in EFI. The architecture was then defined backwards from the Operating System towards the reset vector. Phases were defined to own the reset vector, manage the system up to the point RAM was initialized, RAM-resident initialization, boot device selection, and the run-time.

To address the transition from legacy to EFI, the PI can support multiple boot modes, including a module known as the Compatibility Support Module (CSM), which allows PI to boot into legacy Operating Systems using those same interfaces defined in 1980.

3.5.1 Platform Initialization Versus UEFI

It is the purview of the UEFI Platform Initialization, or what we shall refer to as the "PI", to describe these building blocks. To that end, we will describe

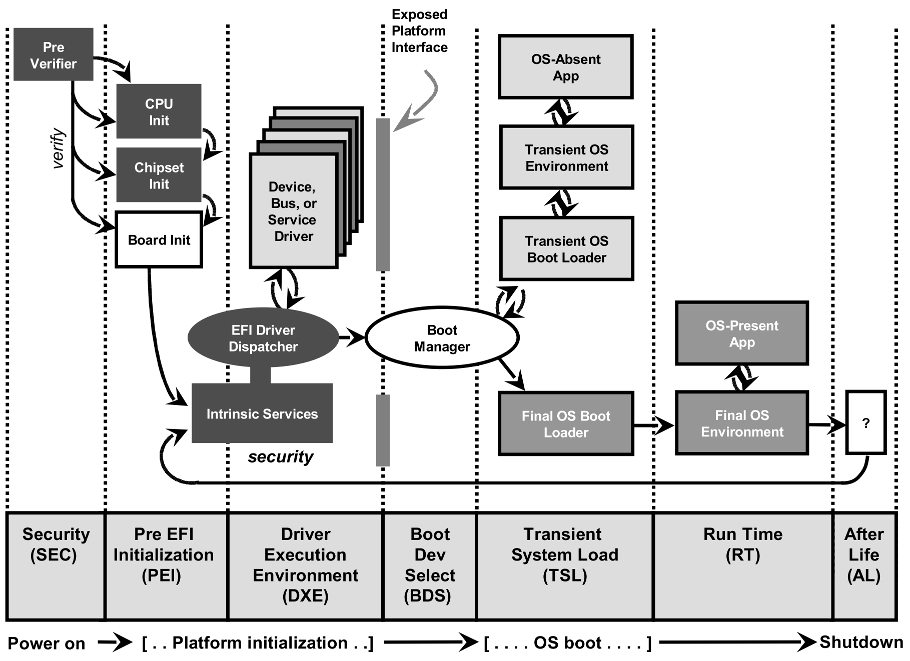


Figure 3.5. Temporal view of the system.

both a temporal and spatial view of the system. The temporal view of the PI boot is shown in Fig. 3.5. The spatial view is shown in Fig. 3.6.

What is of interest in the temporal view is that the boot flow of the machine is broken up into phases, of which there are several of interest, including the security (SEC) phase , the Pre-EFI Initialization (PEI), and the Driver Execution Environment (DXE).

3.5.2 SEC: The Security Phase

On common aspects of all processors and platforms is that they restart in a given fashion. On x86, the location is 4G – 16 bytes, for example. At this point, there is typically no initialized memory and a very rudimentary machine state. The most notable feature is the lack of memory for a heap and call-stack.

The SEC phase stands for "Security". The intent of this moniker was to describe the first location in PI where a system root of trust in BIOS could be implemented, although without platform hardware enhancements, this is not the case in most SEC construction today.

For PI, the SEC is responsible for preparing to invoke the PEI environment & the creation of temporary memory. On Intel-based systems, in order to avoid the cost of custom SRAM or other early memory store, we configure the

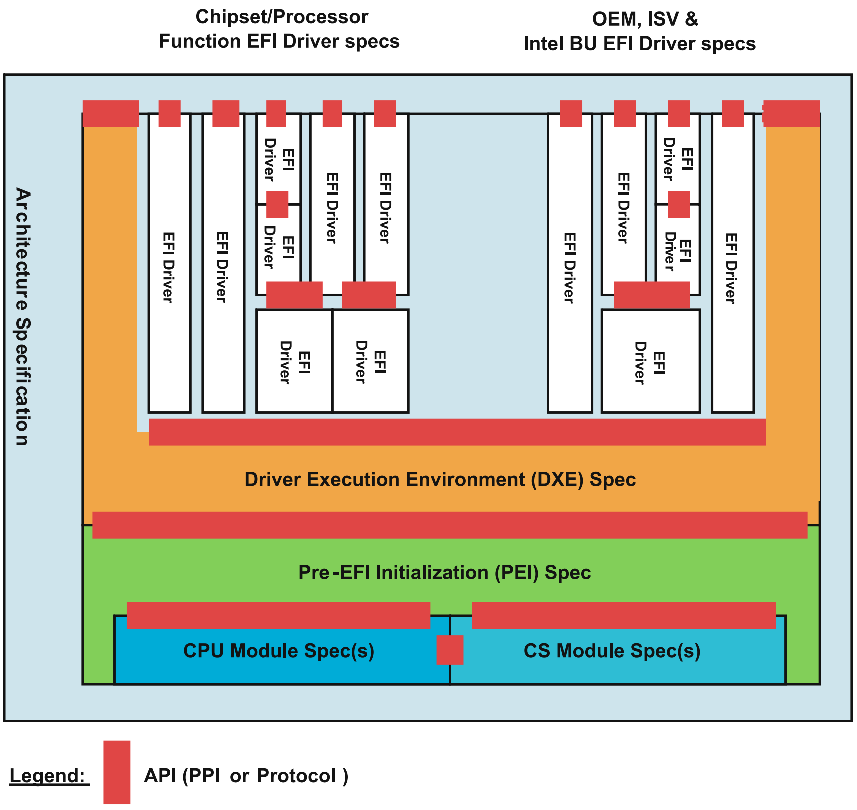

Chipset/Processor Function EFI Driver specs
OEM, ISV & Intel BU EFI Driver specs
Architecture Specification
EFI Driver
Driver Execution Environment (DXE) Spec
Pre-EFI Initialization (PEI) Spec
CPU Module Spec(s)
CS Module Spec(s)
Legend:
API (PPI or Protocol)

Figure 3.6. View of the PI modules.

processor cache as a temporary memory (or "cache-as-RAM" / CAR) store for the stack and heap. This small, early stack, in addition to putting the processor in the PEI-prescribed mode of execution (e.g., 32-bit protected mode on IA-32). The SEC "executes-in-place" (XIP) from the firmware store.

The SEC's ultimate goal is to put the machine in the state prescribed by the UEFI PI specification. This includes a single thread of hardware execution, some call-stack with a minimum size, and passing a possibly non-zero list of data structures into the PEI core.

The SEC is a single component of the portion of the flash part where the processor passes control upon a machine restart. For the PI bindings of x64, IA-32, and Itanium, this is near the end of the flash volume with the SEC core aligned to end at 4 Gbyte. Alternate architectures, such as ARM, instead expect to pass control to firmware at address zero. To accommodate both, there is a specific aspect of the firmware file system, namely the "Very Top File", that is required to be at either the "beginning" or "end" of the firmware store.

3.5.3 Firmware File systems, HOB, Boot Modes, and Capsules

The initial phase of execution, SEC, described how to pass off control to the second phase, namely PEI, but introduced terms such as "firmware volume" and "firmware file". In PI there are certain data structures and capabilities that will span all phases of PI execution. As such, before progressing toward the PEI and DXE descriptions, a description of these common elements is in order.

The firmware file system of PI includes volumes, files, and sections. The volume is the outermost container and is akin to a partition on disk. Within the volume there can be a plurality of firmware files. And finally, within the file can be a series of sections. The actual encoding of the file system is important for direct discovery of modules and data structures in the early, execute-in-place execution flow.

The file system can describe the literal binary encoding of the firmware volume and files in the storage, or they can be abstracted by API's in the PEI and DXE phase of execution.

Another important object is the Hand-Off-Block (HOB). The HOB is an in-memory list of data structures that are created by various PEI modules and consume by the DXE core and DXE modules. Some of the HOB's are required, such as a description of the memory resource map and the location of additional firmware volumes containing DXE drivers, or they can be vendor/domain specific data that an early PEIM needs to convey to a later DXE driver.

A capsule is a firmware volume file that is described by a particular HOB in PEI. Capsules are used to convey an update from a runtime environment back into the PI phases.

The boot-mode is a value that describes the type of machine restart, including manufacturing mode, the Advanced Configuration and Platform Interface (ACPI) [ACPI] S5 mechanical restart, or a system flash update. PEI phase typically detects and operates upon the boot mode.

3.5.4 PEI: The Pre-Initialization Phase

The Pre-EFI Initialization (PEI) phase of execution is the portion of the UEFI PI infrastructure that receives control from SEC and commences execution, like SEC, in XIP. The PEI core is the component that receives control from the SEC. The PEI core expects to have some RAM (typically from cache)-based stack that is described by the SEC hand-off, along with the location of the "Boot firmware volume", or the firmware volume that may contain other PEI modules in addition to the PEI core. The PEI Core is an executable image, such as PE/COFF or a reduced subset, that has its relocations fixed-up for XIP operation.

Given the stack and a pointer to a firmware volume, the PEI core, in turn, uses integrated read-only firmware volume and file system capability to search for PEI Modules (PEIM). A PEI module, like the PEI core, is an XIP executable image in the firmware volume. The PEI module exists in a firmware file that describes the file type as designating a "PEIM".

The PEI modules can be delivered by various business interests, including the processor manufacturer, chipset vendor, and the system board manufacturer. The PEIM's expose capabilities to other PEIM's via something referred to as a PEIM-to-PEIM Interface (PPI).

The firmware file, in addition to the PEI executable image, may also contain a firmware file system section referred to as a dependency expression. The dependency expression (DEPEX) represents a binary-encoded data structure that uses Reverse Polish Notation (RPN) to describe which PPI's are required by a given PEIM prior to its execution.

The ultimate rationale for the PI PEI phase of execution is to do the minimum amount of work in order to discover some permanent, main memory that is sufficient to pass control to subsequent phase of execution. By "permanent" we mean any physical memory initialized in PEI cannot be relocated to some other portion of the address space by a later phase of execution (e.g., DXE). This is the case because the final action of PEI is to invoke a PPI referred to as the "DXE IPL", or the "Driver Execution Environment Initial Program Loader". DXE IPL will discover the DXE core file in its volume, load, and pass control to the DXE core with the HOB list. If the memory were to "move" during DXE, the DXE core and any memory allocations in PEI that were marked as requiring preservation into the operating system runtime (e.g., AcpiNvs memory type) would be violated.

3.5.5 DXE: The Driver Execution Environment

The Driver Execution Environment, or DXE, is the phase of execution that received control from PEI. The input parameterization of PEI includes the HOB list mentioned before. A picture of the required HOB's is shown in Fig. 3.7.

The DXE core initially provides the UEFI system table and a series of memory only services since this file is ostensibly portable across any microarchi-

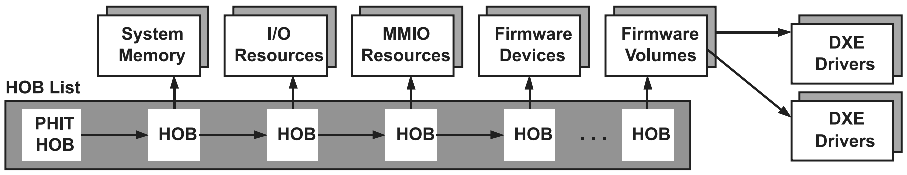


Figure 3.7. Hand-off blocks into DXE.

tecture for which it has been compiled. As such, the DXE core needs to be parameterized by details of the particular platform, including interrupt management, time keeping, UEFI variable management. This is provided by a series of DXE drivers in the firmware volume. The DXE drivers are also PE/COFF executables, but unlike most PEI modules, since the DXE phase commences with system memory, the DXE core and drivers can be decompressed and execute from main memory. Because of the performance and space-savings of this capability, the DXE drivers host higher-level, more algorithmically complex operations.

Figure 3.8 describes the UEFI system table and the associated DXE architectural protocols [UEFb] that provide the platform-specific implementation of some of the UEFI services. For example, the UEFI service SetVariable() has an associated Variable Write architectural protocol (AP) (whose instance is one of the blocks in Fig. 3.8). The use of the architectural protocols is akin to a platform hardware abstraction layer (HAL) that allows for modifying only the respective AP in response to differing platform needs.

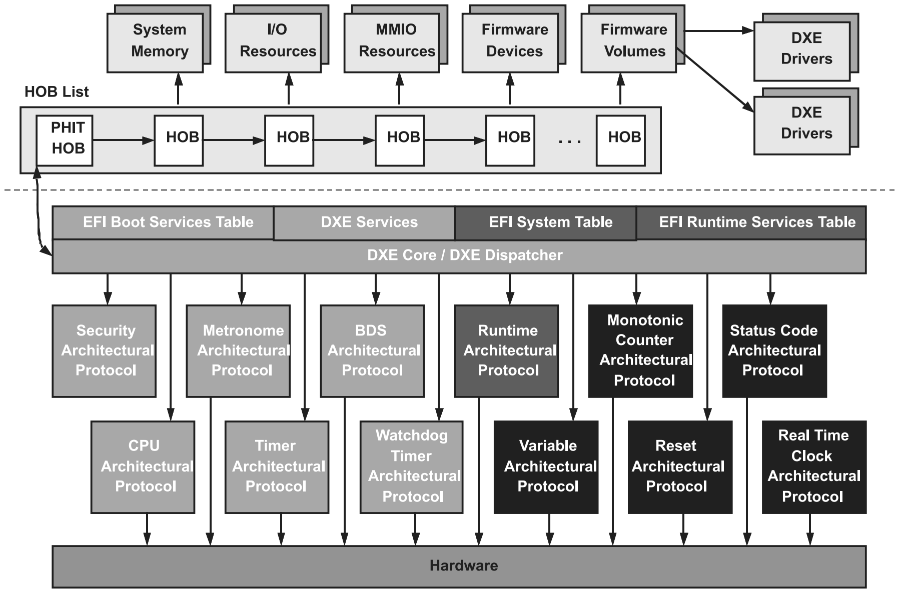

System Memory
I/O Resources
MMIO Resources
Firmware Devices
Firmware Volumes
DXE Drivers
DXE Drivers
HOB List
PHIT HOB
HOB
HOB
HOB
HOB
...
HOB
EFI Boot Services Table
DXE Services
EFI System Table
EFI Runtime Services Table
DXE Core / DXE Dispatcher
Security Architectural Protocol
Metronome Architectural Protocol
BDS Architectural Protocol
Runtime Architectural Protocol
Monotonic Counter Architectural Protocol
Status Code Architectural Protocol
CPU Architectural Protocol
Timer Architectural Protocol
Watchdog Timer Architectural Protocol
Variable Architectural Protocol
Reset Architectural Protocol
Real Time Clock Architectural Protocol
Hardware

Figure 3.8. DXE interfaces.

The operations hosted in the DXE phase include initialization of the I/O buses, such as the Peripheral Component interconnect (PCI) [PCI], creation of the System Management BIOS (SMBIOS) [SMBIOS] tables, and provide the implementation of the UEFI or conventional PC/AT BIOS. The former provision of the UEFI interfaces is by having the DXE core, whether through its memory-only services, or via a DXE call abstracted to an alternate interface, provides a fully compliant set of API's.

Unlike PEI, which refers to the interfaces it uses as PPI's, the DXE drivers expose to each other GUID'd API's that are the same as UEFI protocols. Namely an API and/or data set named by a GUID.

During DXE phase of execution, though, all UEFI interfaces (or BIOS, for that matter) are initially available. The DXE drivers uses the same binary encoding of DEPEX's as PEIM's in order to allow the DXE core to orchestrate discovery and dispatch of DXE drivers. DXE expands upon the AND, OR, and NOT opcodes of the PEI DEPEX with additional operators BEFORE, AFTER, and SOR in order to support the richer dispatch model of DXE.

Once the DXE core has dispatched all of its drivers and is ready to "boot" a subsequent pre-OS environment, whether it be the UEFI or PC/AT BIOS, the DXE core invokes the Boot Device Selection (BDS) service. The BDS implements the "boot manager" capability of the UEFI specification or provides a DXE call-down into the BIOS Boot Specification (BBS) [PhTe] capabilities. The BDS is the first opportunity to expose a user interface/splash screen. The BDS also orchestrates the behavior under various boot modes while in DXE. As such, the BDS represents the system board manufacturers specific business needs and look-and-feel. Unlike other DXE drivers that may be provided by chipset or processor vendors, the BDS is most likely heavily modified for a given manufacturer.

Again, DXE provides the subsequent pre-OS execution environment for UEFI or BIOS boots, but it is responsible for machine state construction and hand-off. DXE is typically only extensible for the system board manufacturers and doesn't admit execution of third party modules, such as PC/AT option ROM's or UEFI drivers. The interposition of any foreign content, such as a capsule firmware volume for an OS-present update utility, will typically only be executed/exposed when it is shown that the capsule was produced by the system board manufacturer (e.g. cryptographically signed).

Also, since the BDS is the last DXE component and bridges the gap into PC/AT or UEFI execution, each of which supports 3rd party adapter ROM's on cards or disk, the BDS is the last opportunity to lock down the system board resources. This includes locking the SMRAM or the block-lockable flash.

In addition to the DXE phase of execution, DXE registers components for other process modes and/or machine states. The other modes include the System Management Mode (SMM) on x64 and Platform Management Interrupt (PMI) / Machine Check Architecture (MCA) of the Itanium processor family. Each of these 2 machine phases have specific protocols that allow for loading DXE drivers into System Management RAM (SMRAM) or OS-reserved memory for SMM and PMI, respectively.

In addition, DXE provides drivers to publish data tables and other services, such as ACPI, SMBIOS, and the Itanium System Abstraction Layer (SAL) System Table (SST).

The transition to UEFI and PI from today's BIOS or proprietary boot solutions represents a seismic transition for the industry. But after the development effort and transition costs have been overcome, the "extensibility" of both UEFI and PI will offer a platform for future innovation.

References

[ACPI] ACPI. Advanced Configuration and Power Interface. www.acpi.org

[EDK] EDK. EFI Developer Kit. www.tianocore.org

[PCI] PCI. Peripheral Component Interconnect. www.pcisig.org

[SMBIOS] SMBIOS. System Management BIOS. www.smbios.org

[SMBUS] SMBUS. System Management Bus. www.smbus.org

[PhTe] Phoenix Technologies. BIOS Boot Specification. www.phoenix.com/NR/rdonlyres/56E38DE2-3E6F-4743-835F-B4A53726ABED/0/specsbbs101.pdf

[UEFa] UEFI. UEFI Registry. www.uefi.org/specs/esp_registry

[UEFb] UEFI. Unified Extensible Firmware Interface Platform Initialization Specifications, Volumes 1–5, Version 1.1. November 5, 2007. www.uefi.org

[UEFc] UEFI. Unified Extensible Firmware Interface Specification—Version 2.1. January 23, 2007. www.uefi.org

[ZRH] V. Zimmer, M. Rothman, and R. Hale. *Beyond BIOS: Implementing the Unified Extensible Firmware Interface Specification with Intel's Framework*. ISBN 0-9743649-0-8, Intel Press, September 2006. www.intel.com/intelpress/sum_efi.htm