Chapter 1
Introduction

Less but better.

—Dieter Rams

To most users, a computer is represented by the operating system that they're using and nothing more. However, unbeknownst to most basic users, there are a large number of components that must work in concert to go from where the user presses the power button, the hardware is initialized, the boot target is discovered, to where the operating system is launched.

There are two major phases of platform initialization between when a user turns a computer on and the computer has completed its initialization: the first phase is what might be called the "pre-OS" stage where the platform's hardware is initialized and made usable, and the second phase is when the boot target is launched, which oftentimes would be the target operating system.

The early phase of platform initialization is primarily focused on the launching of a target. This target almost always is an operating system, but it doesn't always have to be. Sometimes, activities such as bare-metal provisioning, diagnostics, personality migration, scripting and others are accomplished through an intermediary execution environment known as a UEFI shell.

Much of this book will further explain the intricacies of how one uses the UEFI shell to accomplish the aforementioned activities, but this being an introductory chapter, it seems reasonable to give a slight background on what the UEFI shell actually is and from where its roots evolved.

What is UEFI?

Historically, the BIOS (Basic Input Output System) was a black box. In other words, there was very limited exposure to how it worked and the interoperability associated with the BIOS and the rest of the system was limited at best.

The purpose of a BIOS was very simple: its role was to discover and initialize the hardware, run any tests that were required to ensure the hardware was in working order, and ultimately launch the boot target.

The goals for today's BIOS have not significantly changed. What has changed is that the "black box" nature of BIOS has been opened so that the industry can normalize the interfaces associated with BIOS technology and leverage it in many ways that were previously not possible.

In 2005, the UEFI (Unified Extensible Firmware Interface) Forum was established. The forum itself was formed with several thoughts in mind:

DOI 10.1515/9781501505751-001

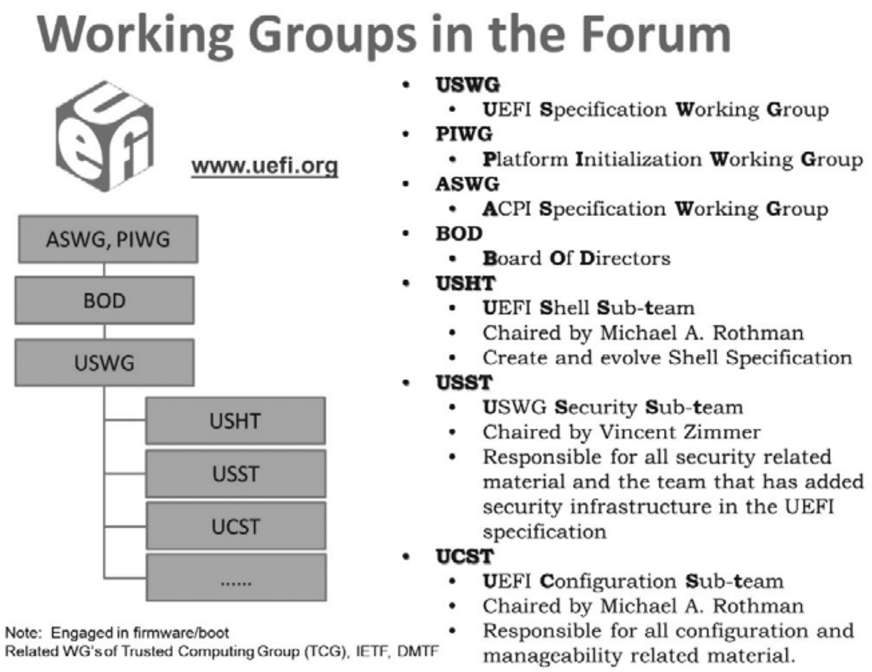
Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

- The UEFI Forum is established as a Washington non-profit Corporation
  - Develops, promotes, and manages evolution of the UEFI Specification
  - Continue to drive low barriers for adoption

- The Promoter members for the UEFI forum are:
  - AMD, AMI, Apple, Dell, HP, IBM, Insyde, Intel, Lenovo, Microsoft, Phoenix

- The UEFI Forum has a form of tiered Membership:
  - Promoters, Contributors, and Adopters
  - More information on the membership tiers can be found at: www.uefi.org

- The UEFI Forum has several work groups:
  - Figure 1.1 illustrates the basic makeup of the forum and the corresponding roles.



Working Groups in the Forum

- **USWG**
  - **U**EFI **S**pecification **W**orking **G**roup
- **PIWG**
  - **P**latform **I**nitialization **W**orking **G**roup
- **ASWG**
  - **A**CPI **S**pecification **W**orking **G**roup
- **BOD**
  - **B**oard **O**f **D**irectors
- **USHT**
  - **U**EFI **S**hell **S**ub-**t**eam
  - Chaired by Michael A. Rothman
  - Create and evolve Shell Specification
- **USST**
  - **U**SWG **S**ecurity **S**ub-**t**eam
  - Chaired by Vincent Zimmer
  - Responsible for all security related material and the team that has added security infrastructure in the UEFI specification
- **UCST**
  - **U**EFI **C**onfiguration **S**ub-**t**eam
  - Chaired by Michael A. Rothman
  - Responsible for all configuration and manageability related material.

Note: Engaged in firmware/boot
Related WG's of Trusted Computing Group (TCG), IETF, DMTF

Figure 1.1: Forum group hierarchy

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

- Sub-teams are created in the main owning workgroup when a topic of sufficient depth requires a lot of discussion with interested parties or experts in a particular domain. These teams are collaborations among many companies who are responsible for addressing the topic in question and bringing back to the workgroup either a response or material for purposes of inclusion in the main working specification. Some examples of sub-teams that have been created are as follows:
  - UCST – UEFI Configuration Sub-team
    - Chaired by Michael Rothman
    - Responsible for all configuration-related material and the team has been responsible for the creation of the UEFI configuration infrastructure commonly known as HII, which is in the UEFI Specification.
  - UNST – UEFI Networking Sub-team
    - Chaired by Vincent Zimmer
    - Responsible for all network-related material and the team has been responsible for the update/inclusion of the network-related material in the UEFI specification, most notably the IPv6 network infrastructure.
  - USHT – UEFI Shell Sub-team
    - Chaired by Michael Rothman
    - Responsible for all command shell-related material. The team has been responsible for the creation of the UEFI Shell specification and continues to maintain the contents as technology evolves.
  - USST – UEFI Security Sub-team
    - Chaired by Vincent Zimmer
    - Responsible for all security-related material and the team has been responsible for the added security infrastructure in the UEFI specification.

With the UEFI Specification, we now have programmatic interfaces to features and functionality that we never had before in previous generations. This allows third parties to create UEFI-compatible software that can run on platforms which are UEFI-compliant.

In Figure 1.2, we see a very high level illustration of the general software flow during platform initialization.

The normal process would be to launch the operating system loader, which is a UEFI-compliant item, and it in turn will initialize the operating system.

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

However, there are cases where the user or system administrator would choose to launch other components such as a UEFI Shell. This of course leads to a natural question, "What is a shell?"

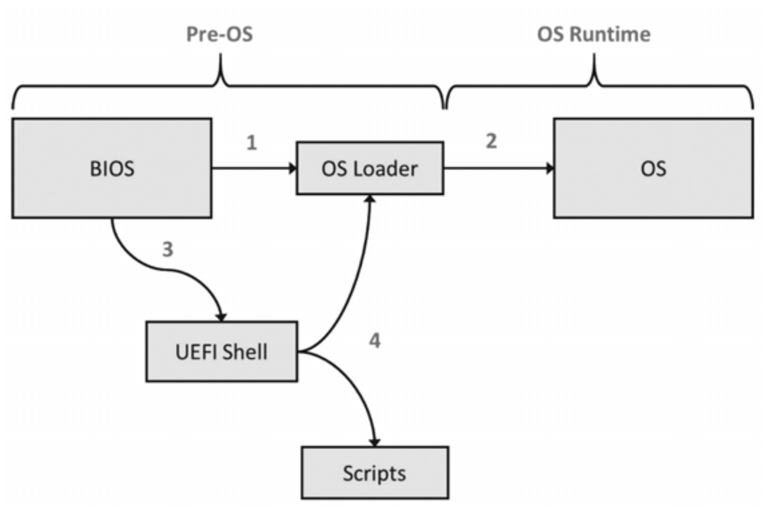


Figure 1.2: High-Level Platform Initialization Flow

What Do We Mean by *Shell?*

At its most basic, a shell is a way of exposing an interface to a user. This can be a graphics interface such as one that leverages icons, mouse clicks, and animation, or a more rudimentary interface such as a CLI (Command-Line Interface), which requires a user to type commands to a command processor for it to respond.

It should be pointed out that from a programmatic point of view, a shell also provides interfaces from which an application can interact with the underlying programmatic abstractions (interfaces).

This book will cover a wide variety of the underlying APIs (both programmatic and script-based) as well as functional capabilities. For reference, there are two very pertinent specifications that come out of the UEFI forum (www.uefi.org).

- **UEFI specification**: The specification that covers a wide variety of programmatic interfaces, many of which are associated with interacting with a platform's hardware and other platform policy. This is the specification that forms the basis of what is known as "UEFI compatibility."
- **Shell specification**: The specification that also describes programmatic interfaces that are exposed by a UEFI Shell compatible environment. In addition to

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

the typical programmatic interfaces, the specification also describes a large series of commands that form the basis of the UEFI Shell's scripting language.

A Short History of the UEFI Shell

The UEFI Shell had very humble beginnings. Its roots lie with the birth of the PC and the advent of CPM/DOS.

For those who can recall the predominant operating system of the 1980's, it was part and parcel of the original IBM PC and was very much ubiquitous for users of computers in that era. The command-line interface and many of the commands that are very familiar to users (e.g., copy, delete, echo) owe their origins to the days of DOS (Disk Operating System).

In those days, DOS was the boot target. The expectations of the user were a bit more humble than they are today, and to give a relative comparison of the complexity between now and then, bear in mind that the first DOS with all of its utilities and kernel fit within 150K worth of code, while most modern operating systems may have an on-disk size of one gigabyte or more.

The main goal of DOS was to be able to launch applications, utilities, and execute scripts.

DOS exposed limited standardized APIs to access the underlying platform, so the complexity associated with what one could do through the command-line interface was also fairly limited. However, with the advent of UEFI and the myriad interfaces that it exposed, the possibilities became fairly broad. For instance, within UEFI we have provided abstractions to access networking devices, graphical components, storage devices, and a multitude of other things. The possibilities of what a third-party application or script can do is much broader than was ever possible in the earliest of operating systems.

It should be noted that in many UEFI-compliant platforms, the UEFI shell and its underlying abstractions are all contained on the platform's embedded non-volatile storage (e.g., FLASH device) and can execute even without a boot media target. This is something that the platforms of old did not provide. On a UEFI-compliant system, you could potentially have a rather robust environment of a complete network stack, UEFI shell, and a modern programming environment with memory managers and a driver model. This powerful environment now allows for some of what will be described in ensuing chapters, such as bare-metal provisioning and advanced diagnostics.

Brief Overview of the UEFI Shell

The UEFI shell consists of two parts: a set of APIs and a command-line interface.

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

UEFI Shell APIs

The set of APIs abstract the command line and file I/O aspects of the system. For example, the command-line APIs allow UEFI Shell programs to read the command line.

There are also a variety of APIs that deal with the shell environment, such as getting the current setting for the PATH or other meaningful environment variables.

Many of these APIs are used primarily in support of scripting commands that are covered in later chapters in this book. These interfaces make a shell application much simpler to write, because they hide some of the complexity that is associated with the underlying UEFI environment. For example, there are many Shell Protocol interfaces having to do with reading and writing files on various storage media. The reason for this is because when the Shell Specification was created, we wanted to limit the amount of underlying UEFI knowledge someone must have before being an effective user of the shell. That means if it was possible to provide an abstraction that simplified writing an application, then we provided that interface. Ultimately the Shell Specification reflects what was deemed the most practical and useful abstractions to facilitate the use by new users of the shell.

Command Line Interface Features

With any command-line interface, there must be a command processor; the component that interprets what it was asked to do and then goes ahead and does it.

The command-line processor is the logic that parses whatever a user or script sends to it and is essentially the interpreter of the “shell language” that the UEFI Shell speaks. Whether it’s how hyphens are treated, wild card support, I/O redirection, or quoted strings, all of that complexity is handled by the shell and is something that can be leveraged by text-based scripts or by shell applications that in turn want to leverage the command-line interface.

As to the format of the shell applications that a third-party may create, these are all built to be compliant with what a standard UEFI application would look like. In other words, the UEFI Shell uses the same executable program format as does its underlying software layers: PE/COFF. PE/COFF is not a pure binary image. Instead, it is a series of variable length data structures that allow the UEFI Shell to load programs at arbitrary addresses in memory via a process known as relocation. PE/COFF was chosen because it is well known in the industry and produced by a wide variety of compiler/linker sets across the operating systems most developers use.

The UEFI Shell defines a scripting language. This language is similar to programming languages but operates at a higher level. The language allows for looping, conditional execution, and data storage and retrieval via environment variables. The scripting language is unique to the UEFI Shell but similar enough to other shells that learning it shouldn’t be difficult.

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

The UEFI Shell is designed for a variety of environments. To meet all of the requirements, different levels of command support are specified. In the most minimal, there is space for one user application. The shell is simply used to kick that application off. In richer versions, you'll find batch commands to control automation. Again, the user may never interact with the UEFI Shell; instead, the shell is useful to manage the order of execution of programs. In the most full featured versions of the UEFI Shell, like the ones you might be developing applications on and for, you'll find the standard commands like dir (ls), copy, a minimal full screen editor, and the like.

Why a Shell at all?

It's a reasonable question to ask, "With advanced computers and operating systems, why do we even need something as simple as a shell anymore?"

Oddly enough, the answer to that question is in the question. The UEFI Shell has persisted for one clear reason: because it is simple and useful.

Consider for a moment that the UEFI Shell does not even require a platform to have a bootable drive. No operating system may be on the machine, yet you can run the shell and compatible applications. These shell programs can reach out and touch anything on the platform and test it. They can potentially connect to a remote server and pull down gigabytes of data to provision the platform. The shell can be used as an aid in the manufacturing line to test the components on the system before they even leave the factory.

The UEFI Shell requires no platform-level customization. It requires no drivers beyond those included in the shipping system. This means that as the UEFI Shell is used it becomes less and less likely to be the culprit of bugs introduced as a part of the system. It becomes an island of consistency in an ocean of variability.

The UEFI Shell is, in the end, useful because it is small and not intrusive, just as its cousins are useful because they are large and all-encompassing.

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM

Brought to you by | De Gruyter / TCS
Authenticated
Download Date | 3/23/17 8:01 PM