

Harnessing the UEFI Shell

Moving the Platform Beyond DOS

MICHAEL ROTHMAN,
VINCENT ZIMMER, TIM LEWIS



Michael Rothman, Vincent Zimmer, Tim Lewis

Harnessing the UEFI Shell

Moving the Platform Beyond DOS

Michael Rothman
Vincent Zimmer
Tim Lewis

Harnessing the UEFI Shell

Moving the Platform Beyond DOS

Second Edition



ISBN 978-1-5015-1480-7
e-ISBN (PDF) 978-1-5015-0575-1
e-ISBN (EPUB) 978-1-5015-0581-2

Library of Congress Cataloging-in-Publication Data

A CIP catalog record for this book has been applied for at the Library of Congress.

Bibliographic information published by the Deutsche Nationalbibliothek

The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data are available on the Internet at <http://dnb.dnb.de>.

© 2017 Walter de Gruyter Inc., Boston/Berlin
Printing and binding: CPI books GmbH, Leck
♾ Printed on acid-free paper
Printed in Germany

www.degruyter.com

To my wife Sandi, for having infinite patience in allowing me to find the “spare” time for this endeavor, and to my sons Ryan and Aaron, who keep me grounded in what life is really about. Also to my grandfather Joseph, who instilled the love of learning and set an example I strive to promote for the next generation.

—Mike Rothman

To the three beautiful women in my life: my wife Jan, and my daughters Ally and Zoe.
—Vincent J. Zimmer

To my wife Helen, always bright and beautiful, and to my kids, curious Shannon, caring Brahms, and courageous Miriam, always the joy of my life.

—Tim Lewis

Preface

A man is rich in proportion to the number of things he can afford to let alone.

—Henry David Thoreau

This is a book about a computer program that was never intended to be what it is. The first version of what is now the UEFI Shell was created to facilitate the debugging of early parts of EFI. It was never intended to see a customer. It was a simple expedient tool to speed up an EFI developer's job. Its escape to the rest of the world was, in retrospect, inevitable because it was more valuable than we realized. It became popular enough that, in the eyes of many, it was EFI. It was not, and it is not. EFI (now UEFI) is a commonly agreed upon set of interfaces between operating systems, BIOS, and option ROMs. The shell is, in many ways, simply another operating system that sits on top of EFI.

In the next few years of “add what we need” the shell became more and more valuable. Badly suffering from too many hands and not enough guidance, it became complex enough to warrant serious effort, a seriously out of control adolescent if there ever was one. In the end, it has become valuable enough to warrant the creation of an industry specification and adoption throughout the industry. It is becoming a basis of computer component validation, computer validation, and manufacturing, system testing, and applications. Pretty good for something originally intended as a throwaway piece of code to test some EFI drivers.

The Shell started its life in 1999 or 2000 (we don't exactly remember) so it is comparatively a newcomer. Yet, it is in many ways a throwback to (at least what now) seems a much simpler time, say 1970 or so. It doesn't run protected code, have a swap file or a registry, or even a GUI. As far as we know, it doesn't even have a virus scanner.

We've discovered there is still a place for a small, simple, developer's environment that provides enough resources and support for complex programs without getting in the way of applications that need to (or at least think they need to) “own the system”.

The Book

The first part of this book introduces the basic concepts: history (Chapter 1), UEFI, the underlying operating environment (Chapter 2), basics of the shell (Chapter 3), and basic benefits of a pre-OS shell (Chapter 4). The second part of the book reviews some of the ways the UEFI Shell is used today, ranging from manufacturing (Chapter 5), provisioning (Chapter 6), configuration management (Chapters 7), and diagnostics (Chapter 8). The third part of the book reviews useful tips for batch script programming (Chapter 9) and application programming (Chapter 10). Chapter 11 discusses

how the shell can be useful to debug UEFI drivers, ironically the shell's original purpose. Appendixes cover security considerations, UEFI Shell library descriptions, and provide brief descriptions of the shells commands and APIs.

Contents

Preface — vii

Chapter 1 Introduction — 1

- What is UEFI? — 1
- What Do We Mean by *Shell*? — 4
- A Short History of the UEFI Shell — 5
- Brief Overview of the UEFI Shell — 5
- UEFI Shell APIs — 6
- Command Line Interface Features — 6
- Why a Shell at all? — 7

Chapter 2 Under the UEFI Shell — 9

- Shell and UEFI — 9
- Evolution and Revolution — 13

Chapter 3 What Is the UEFI Shell? — 15

- What Is Contained in the UEFI Shell? — 16
- What Kind of Shell Do You Have? — 16
- What!? No Shell? No Problem! — 17
- Programmatic Shell Environment — 19
- Using UEFI Shell Commands — 20
- Interactive Shell Environment — 22
- Scripting — 22
- Program Launch — 24
- File-System Abstractions — 29
- Shell Script Resolves into a UEFI Firmware Action — 31

Chapter 4 Why We Need an Execution Environment before the OS — 33

- Evolution of a Machine — 33
- The Platform Initialization Flow — 34
- UEFI Transitions — 36
- States of a Platform — 38
- Readiness of UEFI — 41
- Migration Using the UEFI Shell — 44
- Going Forward — 45

Chapter 5 Manufacturing — 47

- Throughput — 47
- Manufacturing Test Tools — 49

Hardware Access with Manufacturing Tools —	50
Converting Manufacturing Tools —	53
Conclusion —	54

Chapter 6 Bare Metal Provisioning — 55

Provisioning with the UEFI Shell —	55
UEFI Networking Stack —	56
Securing the Network —	58
Speeding Up the Network —	62
Example of Putting It Together —	62
Summary —	68

Chapter 7 Configuration of Provisioned Material — 69

Initialization Timeline —	69
Configuration Infrastructure Overview —	71
Using the Configuration Infrastructure —	72
Driver Model Interactions —	73
Provisioning the Platform —	75
Configuring through the UEFI Shell —	76
Basic Configuration —	76
Advanced Configuration Abilities —	79

Chapter 8 The Use of UEFI for Diagnostics — 85

Types of Diagnostics —	85
SMBIOS Table Organization —	87
SMBIOS Structure Table Entry Point —	88
Table Organization Graph —	88
Structure Standards —	89
Structure Evolution and Usage Guidelines —	90
Text Strings —	90
Required Structures and Data —	91
Features —	91
User Interface Design —	92
Design Guide —	92
Usage —	93
Examples —	93
Architecture Design —	94
Data Structure —	95
SMBIOS_STRUCTURE_TABLE —	95
SMBIOS_HEADER —	97
SMBIOS_STRUCTURE_POINTER —	98
STRUCTURE_STATISTICS —	99

Source Code for the Utility — 100

Summary — 105

Chapter 9 UEFI Shell Scripting — 107

Hello, World! — 108

Echo — 108

Echo All Parameters — 109

Echo All Parameters (Improved Version) — 110

Concatenate Text Files — 112

List Only Selected “ls” Information — 113

Install Script — 115

How to Make a Shell Script Appear as a Boot Option — 119

Chapter 10 UEFI Shell Programming — 121

A Simple UEFI Shell Application: HelloWorld — 121

The Source File: HelloWorld.c — 121

The Component Information (.inf) File — 123

A Simple Standard Application: HelloWorld2 — 124

The Source File: HelloWorld2.c — 124

The Component Information (.inf) File: HelloWorld2.inf — 125

Read Keyboard Input in UEFI Shell Scripts: GetKey — 126

The Source File: GetKey.c — 127

The Component Information (.inf) File: GetKey.inf — 137

The Build Description (.dsc) File — 139

Calculate Math Expressions: Math — 139

The Source File: Math.c — 140

The Component Information (.inf) File: Math.inf — 154

Convert ASCII to Unicode and Back: UniCodeDecode — 154

The Source File: UniCodeDecode.c — 155

The Component Information (.inf) File — 163

Chapter 11 Managing UEFI Drivers Using the Shell — 165

Testing Specific Protocols — 166

Loading and Unloading UEFI Drivers — 167

Load — 168

LoadPciRom — 168

Unload — 169

Connecting UEFI Drivers — 169

Connect — 169

Disconnect — 170

Reconnect — 170

Driver and Device Information — 171

Drivers	171
Devices	172
DevTree	172
Dh -d	173
OpenInfo	173
Testing the Driver Configuration and Driver Diagnostics Protocols	174
DrvCfg	174
DrvDiag	174
Debugging Code Statements	175
POST Codes	177
Post Card Debug	178
Text-Mode VGA Frame Buffer	179
Other Options	179

Appendix A Security Considerations — **181**

UEFI Shell Binary Integrity	181
Overview	181
Signed Executable Overview	182
Digital Signature	183
Signed Executable Processing	185
Signed Executable Generation Application (SignTool)	185
UEFI Load Image	186
SignTool	186
Build Environment	186
Example usage	187

Appendix B Command Reference — **189**

Command Profiles and Support Levels	189
Command List	189
Standardizing Command Output	192
Command Details	193
alias	193
attrib	194
bcfg	194
cd	196
cls	197
comp	197
connect	198
cp/copy	199
date	199
dblk	200
del	200

devices — 200
devtree — 201
dh — 201
dir/ls — 202
disconnect — 202
dmem — 203
dmpstore — 204
drivers — 204
drvcfg — 205
drvdiag — 206
echo — 206
edit — 207
eficompress — 207
efidecompress — 207
exit — 207
for — 208
getmtc — 209
goto — 209
help — 209
hexedit — 210
if — 210
ifconfig — 214
ifconfig6 — 214
load — 215
loadpciorom — 216
ls — 216
map — 217
md — 218
mem — 218
memmap — 218
mkdir — 219
mm — 219
mode — 220
mv — 220
openinfo — 220
parse — 221
pause — 221
pci — 221
ping — 222
ping6 — 222
reconnect — 223
reset — 223

rm —	224
seremode —	224
set —	225
setszie —	226
setvar —	226
shift —	227
smbiosview —	227
stall —	228
time —	228
time —	229
touch —	229
type —	230
unload —	230
ver —	230
vol —	230

Appendix C Programming Reference — **233**

Script-based Programming —	233
Parameter Passing —	233
Redirection and Piping —	234
Return Codes —	235
Environment Variables —	236
Non-Script-based Programming —	237
Shell Protocol —	238
Shell Parameters Protocol —	240

Appendix D UEFI Shell Library — **241**

Functions —	241
File I/O Functions —	241
Miscellaneous Functions —	242
Command Line Parsing —	243
Text I/O —	244
String Functions —	244
ShellCloseFile() —	245
ShellCloseFileMetaArg() —	246
ShellCommandLineCheckDuplicate() —	246
ShellCommandLineFreeVarList() —	247
ShellCommandLineGetCount() —	247
ShellCommandLineGetFlag() —	248
ShellCommandLineGetValue() —	248
ShellCommandLineGetRawValue() —	249
ShellCommandLineParseEx() —	250

ShellCopySearchAndReplace()	251
ShellConvertStringToUint64()	252
ShellCreateDirectory()	253
ShellDeleteFile()	254
ShellDeleteFileByName()	254
ShellExecute()	255
ShellFileExists()	257
ShellFileHandleReturnLine()	257
ShellFileHandleReadLine()	258
ShellFindFilePath()	259
ShellFindFilePathEx()	260
ShellFindFirstFile()	260
ShellFindNextFile()	261
ShellFlushFile()	262
SHELL_FREE_NON_NULL()	263
ShellGetCurrentDir()	263
ShellGetEnvironmentVariable()	264
ShellGetExecutionBreakFlag()	265
ShellGetFileInfo()	265
ShellGetFilePosition()	266
ShellGetFileSize()	266
ShellHexStrToUintn()	267
ShellInitialize()	268
ShellIsDecimalDigitCharacter()	268
ShellIsDirectory()	269
ShellIsFile()	269
ShellIsFileInPath()	270
ShellIsHexaDecimalDigitCharacter()	270
ShellIsHexOrDecimalNumber()	271
ShellOpenFileByDevicePath()	271
ShellOpenFileByName()	273
ShellOpenFileMetaArg()	274
ShellPrintEx()	275
ShellPrintHelp()	276
ShellPrintHiiEx()	277
ShellPromptForResponse()	278
ShellPromptForResponseHii()	279
ShellReadFile()	281
ShellSetFileInfo()	282
ShellSetFilePosition()	283
ShellSetEnvironmentVariable()	284
ShellSetPageBreakMode()	285

ShellStrToIntn() — **285**

ShellWriteFile() — **286**

StrnCatGrow() — **287**

Data Structures — **288**

Format Strings — **288**

Shell Parameters — **289**

Index — **291**

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:

- 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

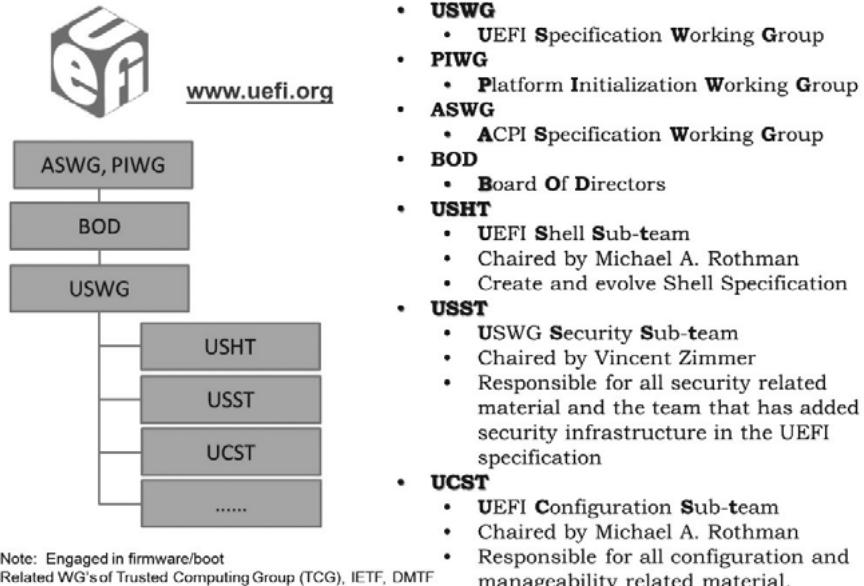


Figure 1.1: Forum group hierarchy

- 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.

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?”

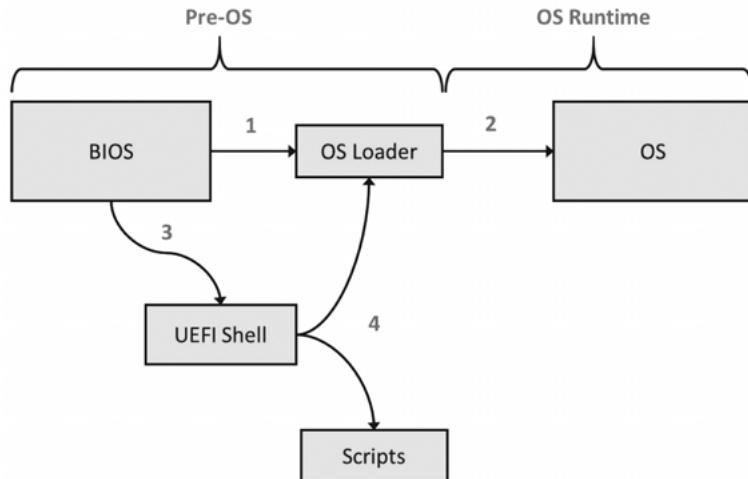


Figure 1.2: High-Level Platform Initialization Flow

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

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.

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.

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.

Chapter 2

Under the UEFI Shell

You cannot create experience. You must undergo it.

—Albert Camus

The UEFI Shell provides an interactive command-line environment. The shell includes such facilities as scripting and a hierarchical set of command profiles. This allows for usages spanning in-situ field diagnostics to a full provisioning environment. As the UEFI Shell is just a distinguished UEFI application, it bears mentioning some of the underlying capabilities UEFI upon which the shell depends.

Shell and UEFI

The UEFI Shell is launched as a UEFI application. UEFI applications are formatted as PE/COFF executables and are loaded into memory and relocated in memory via the `UEFI LoadImage()` service and invoked via the `UEFI StartImage()` service. The UEFI Shell, like any application, has two input parameters. These input parameters include an image handle and a pointer to the UEFI System Table, respectively.

To begin with the first argument, the image handle is an opaque data object that can be used with the UEFI protocol services to discover information, such as other protocols associated with this handle, via querying UEFI boot services, such as locate and handle protocols. Implementations such as EDKII typically have the image handle as a pointer to the underlying data structure defining the handle, but this is an implementation specific art and not guaranteed across other implementations. This handle can be used to discover other protocols associated with an application, such as the loaded image protocol. The loaded image protocol is something of a self-pointer with respect to the image that describes where the information is loaded in memory and other properties.

A protocol is typically published by the UEFI core or a UEFI driver. A relationship of a protocol to the UEFI Shell is shown in Figure 2.1:

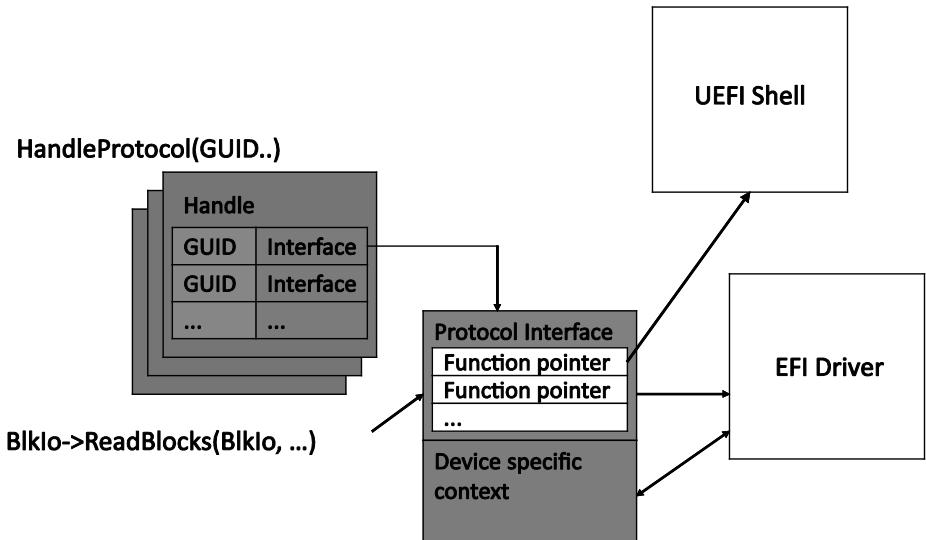


Figure 2.1: Protocol database

The second argument, or a pointer to the UEFI System Table, is important in that it provides function pointers to the boot and runtime services, along with pointers to other GUID-annotated tables. This service set, such as the boot services, includes the protocol services mentioned above. The boot and runtime services are a set of base capabilities that any UEFI conformant system needs to supply to an application. Beyond the accessor functions of the protocol database, there are boot services for memory allocation, events, and other system capabilities. The latter are termed ‘boot services’ since they expire at the `ExitBootServices()` event. The UEFI Shell itself is a boot services application, so it can leverage all of these capabilities. The relationship of the UEFI Shell to the UEFI API’s are described by the UEFI specification, this relationship is shown diagrammatically in Figure 2.2.

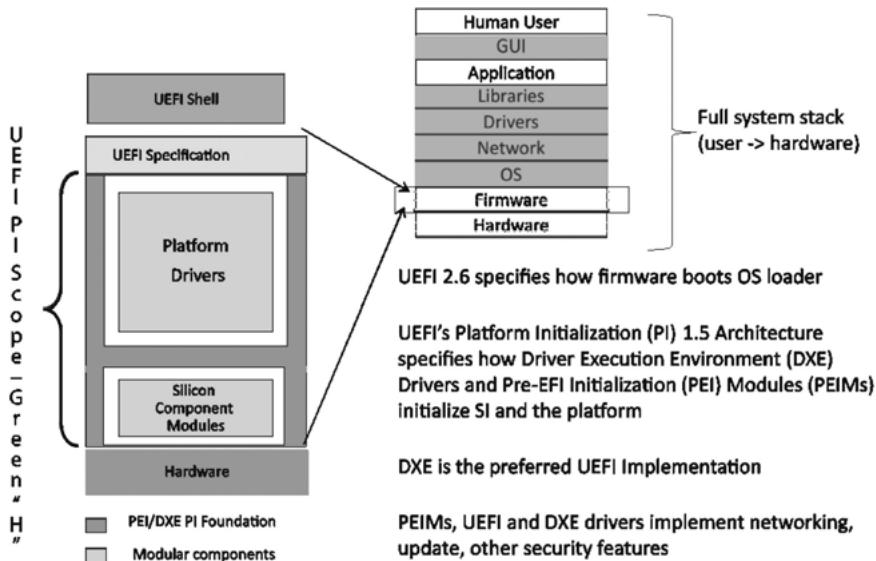


Figure 2.2: UEFI Shell relative to the underlying PI implementation

Beyond the boot services there are another set of services called ‘runtime,’ which include UEFI variables, time, and monotonic counter support. The runtime APIs are purposely kept simple because these APIs must be callable after `ExitBootServices()` and share the machine hardware at runtime with the hypervisor or operating system.

A more detailed layering of the UEFI shell and the underlying implementation is shown in Figure 2.3. A conformant shell implementation should only depend upon the interfaces and data objects defined in the main UEFI specification. As such, the platform-specific firmware can include but is not limited to things such as infrastructure based upon the UEFI platform initialization (PI) specification.

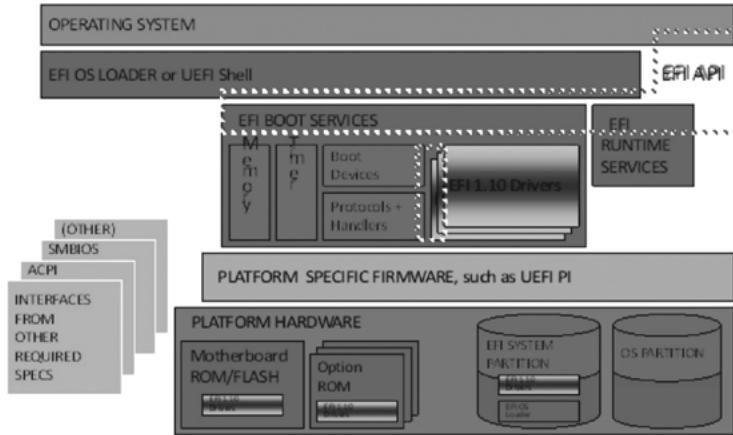


Figure 2.3: Shell API layering

There are instances, though, where a UEFI Shell application may choose to access PI interfaces. These include diagnostics that want to access all the application processors in a symmetric multiprocessor (SMP) system. An example of such an API includes the `EFI_MP_SERVICES` protocol from volume 2 of the PI specification published by the UEFI Forum. In leveraging an underlying PI API, however, the UEFI Shell application is limiting its potential portability since a UEFI implementation, including the UEFI Shell, may not have PI-conforming firmware underneath.

And finally, the cessation of UEFI boot services is shown in Figure 2.4.

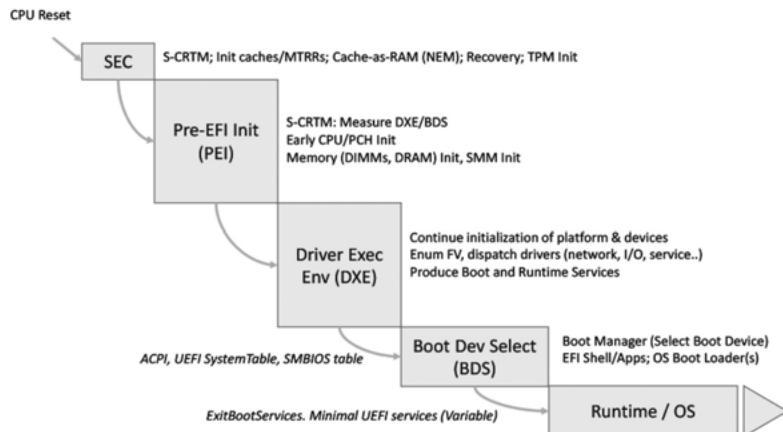


Figure 2.4: Boot flow

What this means is that like many boot service capabilities, the UEFI Shell ceases to be operational once the main operating system or hypervisor takes over.

Evolution and Revolution

The UEFI Shell is both evolutionary and revolutionary. The evolutionary aspect includes many simple firmware environments that have a command-line monitor to send commands at the low end. At the high end, many RISC workstations have an ability to drop into a command-line environment. As such, having a bare-metal environment wherein raw system resources can be ‘peeked’ and ‘poked’ (e.g., use the MM and PCI shell commands), is important for system maintenance.

The revolutionary aspect of the UEFI shell is that it provides an editing, networking, and other capabilities often found in an operating system. This allows a universal environment for diagnostics and other OS-absent applications.

Chapter 3

What Is the UEFI Shell?

Try to be like the turtle—at ease in your own shell.

—Bill Copeland

With the advent of an environment like UEFI, it would stand to reason that a common concept like a shell would arise. Conceptually, a shell is built “around” some aspect of a rather complex system and provides simplified abstractions for users to gain access to the underlying infrastructure. These users could be pieces of software (such as scripts and applications) or they could be humans interacting with the shell in an interactive manner.

A platform running a BIOS that is UEFI-compliant is what might be characterized as the “rather complex item” that a UEFI Shell is built around. The UEFI standards organization (www.uefi.org) publishes the UEFI and PI specifications, which drive the underlying architecture of the BIOS that runs in many of today’s platforms. This same organization has published a UEFI Shell Specification intended to guide what one can expect from a compliant UEFI Shell environment.

This chapter talks about various concepts, such as how the UEFI Shell is abstracting the underlying UEFI-compatible BIOS infrastructure, how certain concepts such as localization are accomplished within the shell, and the various manners in which a user can interact with the shell. It should also be noted that one of the most common uses of a shell today is to launch programs and/or scripts to enable some automated processing to occur. In many cases, DOS was a very common base for such types of activity (Figure 3.1).

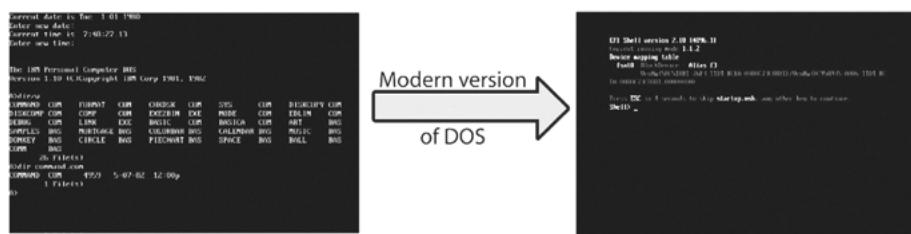


Figure 3.1: The program launching and script support of the UEFI Shell is providing an alternative to what users of DOS and other shells have been using for a very long time.

The UEFI Shell is unusual in that it is not a shell that is a client of an operating system, but is actually considered a BIOS extension. This puts the shell on par with components that traditionally would be launched prior to an operating system such as an

add-in device's option ROM. Where the UEFI Shell is launched from is largely irrelevant, but for many platform vendors, the underlying feature set and size are important considerations since in some cases the shell may actually be contained in the platform's FLASH device.

What Is Contained in the UEFI Shell?

With the consideration that size and features are important to the platform vendor, the features that are provided by a UEFI Shell are likely even more important to the users of the shell. With this in mind, the concept of having varying levels of UEFI Shell support became very important along with the ability for a client of the UEFI Shell to determine what support was being provided.

What Kind of Shell Do You Have?

The concept that a UEFI Shell can vary its support can be worrisome to some, but suffice it to say that this support is both predictable and easily dealt with. The shell is composed of two primary classes of contents:

- *Programmatic Shell Environment*. This environment is guaranteed to remain available regardless of what underlying shell level is supported by a platform that purports to support the UEFI Shell. It is composed of the calling interfaces that shell applications can use.
- *Script Shell Environment*. This environment is the one that supports the launching and interpreting of shell scripts. The biggest variation that one might witness between shell support levels is the enumeration of commands that are supported in a given support level.

The shell contains an environment variable known as the *shellsupport* variable. This variable can be used by shell applications as well as shell scripts to determine what the underlying UEFI Shell's function support is.

In Table 3.1, the various levels of shell support are listed. This illustrates how at its simplest, the shell may be used strictly for purposes of shell applications to be launched (no scripting services). At level 1, basic scripting support is introduced, while level 2 simply adds a few more commands and functionality. In level 3, the concept of being “interactive” is introduced. For people who are familiar with the “C:” prompt from DOS, this interactive mode is similar in concept. Whereas in level 2, when a script was finished processing, the shell would terminate, in level 3, the shell provides a mode that allows the user to type at the UEFI Shell prompt.

Table 3.1: UEFI Shell Levels of Support

Level	Name	Execute () / Scripting/ startup.nsh	PATH?	ALIAS?	Interactive?	Commands
0	Minimal	No	No	No	No	None
1	Scripting	Yes	Yes	No	No	for, endfor, goto, if, else, endif, shift, exit
2	Basic	Yes	Yes	Yes	No	attrib, cd, cp, date*, time*, del, load, ls, map, mkdir, mv, rm, reset, set, timezone*
3	Interactive	Yes	Yes	Yes	Yes	alias, date, echo, help, pause, time, touch, type, ver, cls, timezone

Note: * Noninteractive form only

- *Execute ()/Scripting/startup.nsh.* Support indicates whether the `Execute ()` function is supported by the `EFI_SHELL_PROTOCOL`, whether or not batch scripts are supported, and whether the default startup script `startup.nsh` is supported.
- *PATH.* Support determines whether the `PATH` environment variable will be used to determine the location of executables.
- *ALIAS.* Support determines whether the `ALIAS` environment variable will be used to determine alternate names for shell commands.
- *Interactive.* Support determines whether or not an interactive session can be started.

What!? No Shell? No Problem!

In many usage cases, bootable media is used to launch scripts or other utilities. Historically, the common components for bootable (removable) media were a floppy disk with DOS on it, some scripts, and possibly some executable utilities. DOS itself had some inherent limitations associated with a relatively weak API set compared to more modern environments, limited access to certain memory ranges, and other miscellaneous issues with more modern hardware environments. With the advent of UEFI systems, the same infrastructure can be launched as was done before (a DOS bootable image), but with relatively no discernable advantage—it simply preserves what was previously working. However, many users of bootable media (such as manufacturing operations, diagnostics, and so on) are actively porting their DOS solutions so that they can leverage the underlying UEFI BIOS environments.

Coupling UEFI-based BIOS with the UEFI Shell, a user can achieve a true advancement in what was done in prior solutions since any of the prior limitations associated with the DOS environment have been eliminated. In fact, since the infrastructure within which the UEFI Shell runs is robust, the utilities that are launched can fully leverage all of the UEFI BIOS APIs as well as the UEFI Shell infrastructure APIs in addition to running various sets of UEFI Shell scripts.

In some situations a user's shell requirements are not compatible with what the platform currently supports. For those who are trying to provide solutions (utilities, scripts, and so on) that leverage the UEFI Shell and its environment, there are three situations to consider:

- *When the built-in UEFI Shell does not meet the solution's requirements.* If the UEFI Shell's *shellsupport* level is insufficient for the solution provider's needs, a copy of a UEFI Shell might need to be carried with the solution itself.
- *When there is no built-in UEFI Shell.* There may be cases where the platform does not have a UEFI Shell built in as part of its feature set. With this in mind, the solution provider will want to carry a copy of a UEFI Shell along with its solutions carried on the provider's media.
- *When the platform is not compatible with UEFI.* Even though UEFI BIOS is being adopted in a rapid manner in the industry, some platforms will have no underlying UEFI support. To address this situation, Intel has provided to the open source community something known as the Developer's UEFI Emulation (DUET). DUET is designed to provide a UEFI environment on a non-UEFI pre-boot system. This is achieved by creating an UEFI file image for a bootable device, and then "booting" that image as a legacy boot. On this same bootable device/media a solution provider can, in addition to providing UEFI emulation, provide a copy of the shell environment as well as any other material the solution provider desires.
 - This DUET infrastructure is made available for download on the companion website associated with this book as well as being made available on the open source website www.tianocore.org.

Figure 3.2 illustrates three common usage scenarios for the UEFI Shell. The first is when the platform contains all the needed support for the script/utility solution, the second is when the underlying platform shell support is insufficient, and the third is when the platform is not UEFI-compatible.

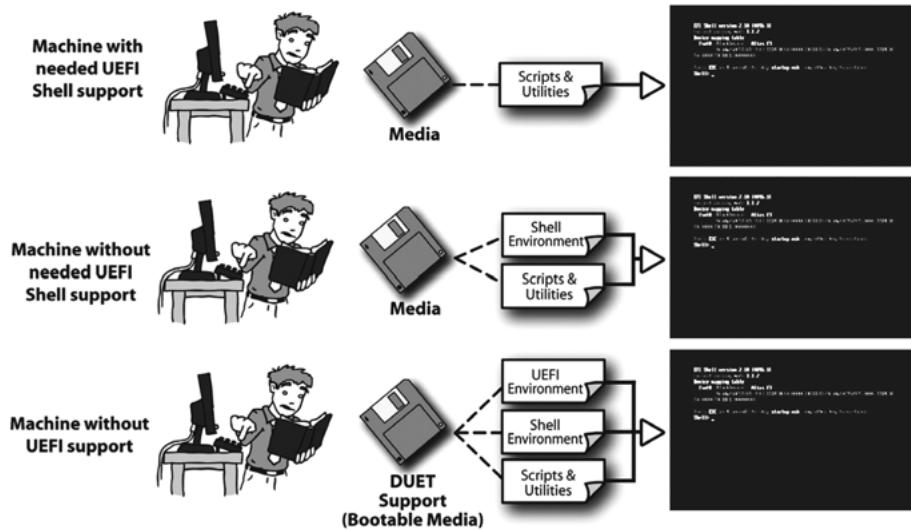


Figure 3.2: Different UEFI Shell usage models. One built within the platform, and the others provided by a bootable target.

Programmatic Shell Environment

Interfaces that are callable from binary programs are what form the UEFI Shell services. These services are what provide simplified access to various shell features and also simplify the interactions that shell clients would have with the underlying UEFI infrastructure. Figure 3.3 provides a high-level view of what the interactions would be between the UEFI infrastructure, shell interfaces, and shell clients.

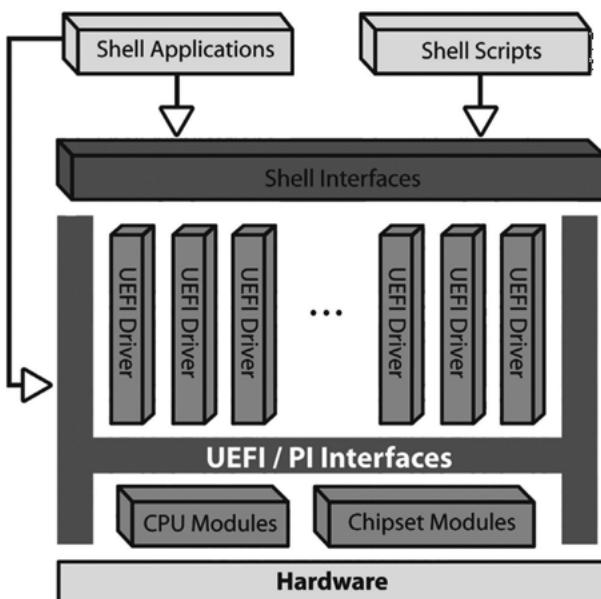


Figure 3.3: The architectural view of the UEFI Shell and the underlying platform infrastructure.

Even though Appendix B has an exhaustive enumeration of the UEFI Shell script commands, and Appendix C has an exhaustive enumeration of the UEFI Shell environment interfaces, this chapter covers the basic programmatic capabilities, their relationship with the underlying infrastructure, and how they are practically used.

Using UEFI Shell Commands

Two classes of operations occur within the UEFI Shell environment. One class of operations runs a script file that uses built-in shell commands (such as DIR and COPY). The other class of operations are binary programs that when launched can use a variety of underlying services.

An example of this interaction would be when a script executes a DIR shell command. When doing this, the following steps occur:

- DIR command in a script file is interpreted by the Shell Interpreter.
- Shell Interpreter then calls a Shell Protocol function such as `OpenRoot()`.
- The Shell Protocol would then call a UEFI service such as the UEFI Simple File System Protocol's `OpenVolume()` routine.
- The UEFI Simple File System Protocol would then call other routines, which would ultimately interact directly with the hardware and return the requested information.

Figure 3.4 shows how a script that uses a UEFI Shell command will in turn interact with both the UEFI Shell interfaces and UEFI BIOS interfaces to achieve what is requested. It also shows that shell applications would also interact with the underlying UEFI Shell and UEFI BIOS interfaces.

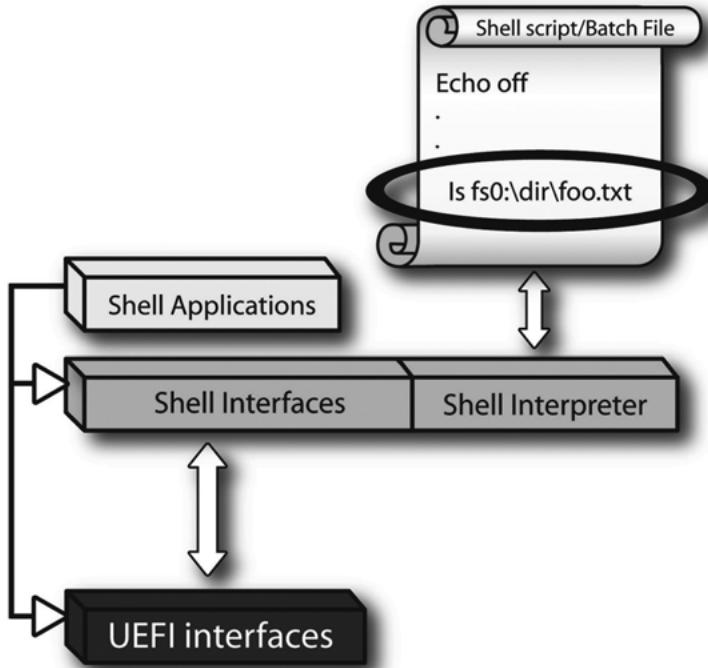


Figure 3.4: UEFI Shell Interpreter processing a script file

Localization Support

One of the inherent capabilities that were introduced into UEFI 2.1 was the ability to easily construct applications that can seamlessly support multiple languages. It should be noted that the primary difference between what someone might call a standard UEFI driver/application and a UEFI Shell application is that the latter has knowledge of the programmatic components of the UEFI Shell infrastructure. That being said, UEFI Shell applications can leverage the underlying localization support in the same manner as any other BIOS component, such as UEFI Drivers and Option ROMs.

Interactive Shell Environment

The concept of having a shell that is interactive is almost always assumed. The stereotypical scenario is the command prompt where a user might type a command and the results are printed to the screen (either locally or through a remote connection). With the advent of the *shellsupport* environment variable and the concept that a shell might have varying levels of support, it should not always be assumed that a shell will be interactive. In fact, it might be very common, based on the type of shell shipped with a given platform, that a script would launch and the shell environment would be closed as soon as the script was terminated (whether through a user-initiated event or the script completing).

In the interactive shell environment, the usage model for the UEFI Shell is similar to what would traditionally be thought of with most shells. Some of the basics that will be discussed are the launching of external binary applications, launching UEFI Shell scripts, and how the various UEFI Shell commands would ultimately resolve into programmatic interaction with the underlying UEFI Shell infrastructure as well as potentially interacting with the underlying UEFI firmware and hardware itself.

Scripting

Depending on the reader's background, three common terms might be used to represent this definition, "a list of commands that can be executed without requiring user interaction." These terms are *scripts*, *macros*, or *batch files*, and to simplify things, this book will try to settle on the term *scripts* when referring to the aforementioned definition.

The UEFI Shell environment is responsible for parsing the script file and interpreting the contents sufficiently to understand what type of action it is being requested to proxy. Some of the basic operations that this environment would need to accomplish would be:

- Execute UEFI Shell commands
- Chaining of UEFI Shell scripts
- Launch UEFI Shell applications
- Launch UEFI applications/drivers

Basic Overview of Commands and How They Interact with Shell Environment

Figure 3.5 illustrates the various components that the UEFI Shell interpreter would interact with when processing a script file. In this illustration, we see an example of a script-based command being parsed by the interpreter itself. Ultimately the interpreter (depending on the command being parsed) would potentially end up calling

some underlying UEFI firmware interfaces. In an example where a UEFI Shell application was launched, it in turn may end up calling programmatic UEFI Shell interfaces, which would then potentially interact with some underlying UEFI firmware interfaces.

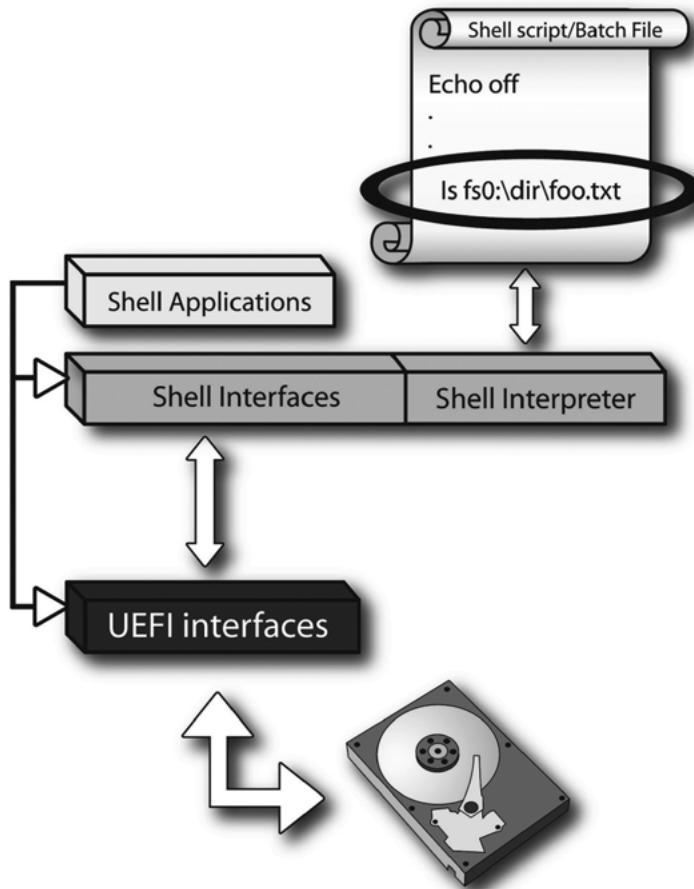


Figure 3.5: UEFI script interacting with the shell environment

1. Shell Script Interpreter parses each line of the script.
2. LS command is recognized and passed to Shell's LS handler.
3. The programmatic handler of the LS command reads the command-line parameters that were passed to it.

4. Using the UEFI Shell infrastructure, the parameters are associated with a particular set of UEFI firmware interfaces and the UEFI Shell calls these firmware interfaces.
5. The UEFI compatible firmware processes the request and in turn communicates with the underlying hardware that was ultimately referenced by the script.
6. This data request is fulfilled and eventually returned back to the UEFI Shell interpreter and the results are processed by the LS handler.
7. The script/user is then made aware of the results of the command having been processed.

Chaining of Script Commands

It is common practice for script files to execute shell commands, which for purposes of the script, are considered part of the shell environment. However, it is also common practice to launch commands or other scripts in shell environments. The UEFI Shell environment is no exception. The concept of one script launching another is often termed *chaining*, and as long as the target script is accessible, a script can choose to launch any other script it has been programmed to launch.

However, there are some definite distinctions between launching a text-based script and launching a binary program. Most of these distinctions have to do with how arguments are passed and what is or isn't accessible to a particular target program. Luckily enough, for most scripts, these distinctions are completely invisible and immaterial. Since many users who are creating binary programs will launch these programs with the UEFI Shell, they may want to understand how some of these interactions would work (for example, getting command-line arguments). The following section talks a bit more about the launching of binary programs and covers some of these underlying interactions.

Program Launch

It should be understood that the UEFI Shell is running within the scope of a UEFI-based firmware environment. This means that the shell itself is a UEFI-based component that complies with the descriptions that are laid out in the UEFI specification. That being said, binary programs launched by the shell will also be UEFI compatible. Since we are introducing the topic of launching UEFI programs, it should be noted that three distinct types of programs that would typically be launched by the UEFI Shell:

- *UEFI Driver* – This is a UEFI-compliant binary program that would follow the UEFI specification driver model. Upon launch, this program may remain in memory and install protocols or services that also remain resident in the system.
- *UEFI Application* – This is a UEFI-compliant binary program. Upon exit, this application will be unloaded from memory.

- *Shell Application* – This is a UEFI-compliant binary program. This program has the same primary characteristics as a typical UEFI application with the addition of having knowledge of how to interact with the underlying UEFI Shell environment.

Even though all of these programs are compliant with the UEFI specification, several characteristics may be unique to UEFI Shell applications.

Argument Passing and Return Codes

When launching a shell application, there is an assumption that parameters would be able to be passed to the application in some fashion. Unlike many conventions where an argument count and array of argument values are directly passed to an application, in a UEFI environment the standard entry point does not consist directly of this kind of data.

When an application is launched in UEFI, sufficient data is passed to the application for it to gain access to the essential components of the UEFI environment. Figure 3.6 shows this standard entry point and how the data contained within this will also provide access to other essential material in the UEFI environment.

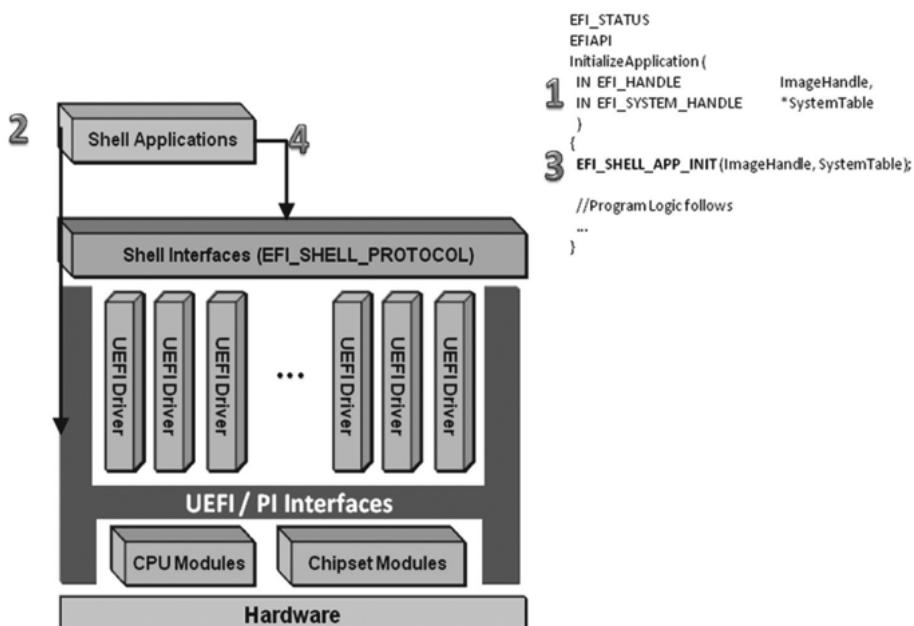


Figure 3.6: Anatomy of an application launch

1. This item illustrates what the standard entry point for any UEFI-compatible binary application or driver looks like. This is the fundamental starting point for all UEFI compatible programs which exposes the underlying UEFI firmware services.
2. During the initialization of a UEFI program, the standard entry point would be used to access the standard runtime and boot services that the UEFI-compatible firmware provides.
3. In most shell-aware applications, there would be either a library or macro which would be used to provide access to the underlying shell protocol interfaces. This library/macro isn't required by the UEFI shell specification, but would commonly be found in many of the available shell-aware programs.
4. In shell-aware applications, the availability of the functions defined in the `EFI_SHELL_PROTOCOL` can be leveraged.

Figure 3.7 shows three main components:

- *Standard Entry Point* – This is the fundamental starting point for all UEFI-compatible programs that exposes the underlying UEFI firmware services.
 - *Image Handle* – When an application is loaded into memory for execution, this value will be the unique identifier for the application.
 - *System Table* – The table that contains a variety of required data and also provides a means to acquire access to the runtime and boot services that UEFI provides.
- *Runtime and Boot Services* – These are the callable interfaces that can be used to interact with the UEFI environment. These interfaces encompass several general classifications:
 - *Task Priority Services*
 - *Memory Services*
 - *Event/Timer Services*
 - *Protocol Services*
 - *Image Services*
 - *Time Services*
 - *Variable Services*
 - *Miscellaneous Services*
- *Shell Parameters Protocol* – This is the protocol that is used in a shell environment to describe all of the command-line parameter data as well as standard handles for output, input, and error. An instance of this protocol is installed on the Image Handle of the application.

```
//  
// Standard Entry description for UEFI app or driver  
//  
EFI_STATUS  
InitializeApp (  
    IN EFI_HANDLE           ImageHandle,  
    IN EFI_SYSTEM_TABLE     *SystemTable  
)  
  
//  
// Excerpt of the above-referenced UEFI System Table  
//  
typedef struct _EFI_SYSTEM_TABLE {  
    EFI_TABLE_HEADER         Hdr;  
    .  
    .  
    .  
    //  
    // Gain access to UEFI Runtime services  
    //  
    EFI_RUNTIME_SERVICES     *RuntimeServices;  
  
    //  
    // Gain access to UEFI Boot services  
    //  
    EFI_BOOT_SERVICES        *BootServices;  
    .  
    .  
} EFI_SYSTEM_TABLE;  
  
//  
// This protocol is installed on the application's  
// ImageHandle.  
//  
typedef struct _EFI_SHELL_PARAMETERS_PROTOCOL {  
    CHAR16                  **Argv;   // Array of arguments  
    UINTN                   Argc;     // Argument Count  
    EFI_FILE_HANDLE          StdIn;    // Standard Input  
    EFI_FILE_HANDLE          StdOut;   // Standard Output  
    EFI_FILE_HANDLE          StdErr;   // Standard Error  
} EFI_SHELL_PARAMETERS_PROTOCOL;
```

Figure 3.7: Standard UEFI Entry point

Figure 3.8 is an example of how one might leverage the standard entry point data to acquire information like the passed in command-line parameters:

```
//  
// Standard Entry description for UEFI app or driver  
//  
EFI_STATUS  
InitializeApp (  
    IN EFI_HANDLE           ImageHandle,  
    IN EFI_SYSTEM_TABLE     *SystemTable  
)  
{  
    EFI_STATUS  Status;  
    EFI_SHELL_PARAMETERS_PROTOCOL ShellParameters;  
    CHAR16   *FilePathName;  
  
    //  
    // Search for the Shell Parameters Protocol which was  
    // installed on the application's ImageHandle.  
    //  
    Status = SystemTable->BootServices->HandleProtocol (   
        ImageHandle,  
        &ShellParametersProtocolGuid,  
        &ShellParameters  
    );  
  
    //  
    // The first parameter is the executable file path  
    // name. Subsequent parameters reflect the processed  
    // command-line parameters  
    //  
    FilePathName = &ShellParameters->Argv[0];  
  
    .  
    .  
    .  
    //  
    // When exiting an application, there will be a status  
    // returned. The UEFI Shell environment will reflect  
    // this status in the LastError environment variable so  
    // that scripts can see what the last error was  
    // produced by a given application or shell command.  
    //  
    return Status;  
}
```

Figure 3.8: Retrieving command-line data

Since many underlying functions such as acquisition of command-line data are normally abstracted by library services, the example in Figure 3.8 is good for showing how some of these fundamental pieces of data are interrelated. This is especially true since the ability to acquire the entry point for protocol services is key to creating a UEFI-aware application; and knowing how to acquire such data solely from the commonly passed-in entry point data will help not only for UEFI Shell programming but for any UEFI programming.

When programs are launched, they will have their returned status codes analyzed by the UEFI Shell environment and have an internal copy of the `LastError` environment variable updated with this result. Figure 3.8 shows that when a program exits (thus returning a status), the UEFI Shell environment will automatically be updated so that when a script checks `%LastError%`, it will automatically be reflected with this returned status.

File-System Abstractions

In a traditional operating system where scripting is prevalent, a simple abstraction for file systems are usually available (such as, “C:” or “D:”). The UEFI Shell environment is no different than these traditional operating systems. In fact, it goes one step further in that it provides clear abstractions for LBA-based (sector-based) accesses through a block I/O interface, and it provides abstractions for file-system based accesses through the disk I/O interface. Figure 3.9 introduces two common UEFI protocols that have to do with abstracting storage devices.

In Figure 3.9, references to BLK and FS are used to note either a block or file-system interface. These interfaces are constructed during the UEFI Shell environment’s initialization. These text-based references are used as simply text notations for various aspects of storage devices. When the UEFI Shell (and especially the MAP command) analyzes the UEFI environment, it searches for instances of `EFI_BLOCK_IO_PROTOCOL`. This is a UEFI protocol that provides an LBA-based abstraction for a storage device. Upon discovery, it will tag each discovered instance with a unique name. These interfaces are a logical abstraction, which means that they abstract a range of physical sectors on some media, and are not necessarily providing access to the entire media.

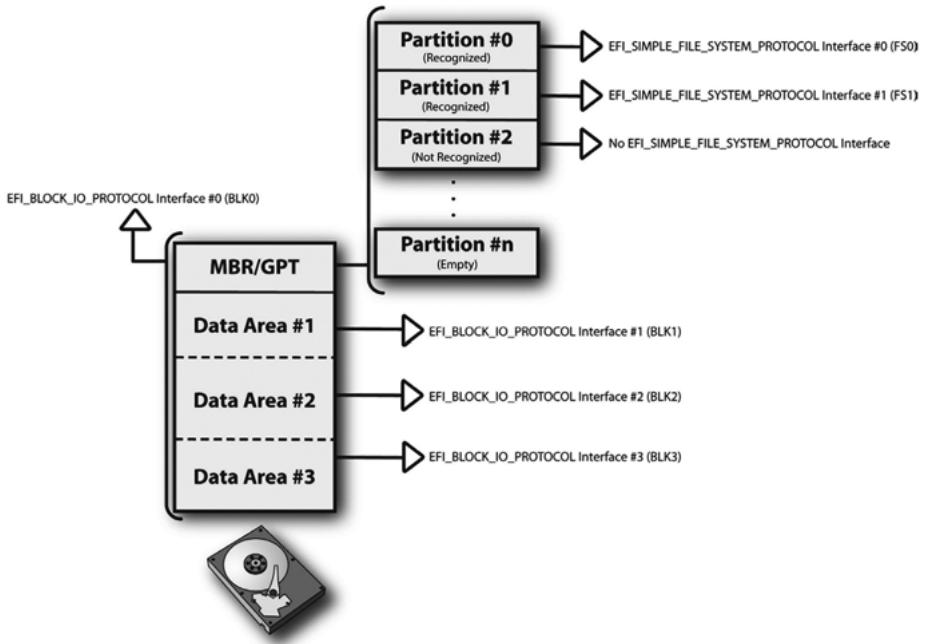


Figure 3.9: File System Abstractions in the UEFI Shell

In the example of BLK0, this is an abstraction used to designate the entire disk. That simply means that when a command references BLK0, the first sector is the real first sector of that disk, while the last sector is the real last sector of the disk. This would contrast with the usage model of a partition's BLK instance such as BLK1.

In the example, BLK1 is associated with the data range for a particular partition entry. That simply means that when a command references BLK1, the first sector is the first data sector of that partition (not the disk), while the last sector is the last sector of the partition.

In addition to the discovery of block devices during the UEFI Shell initialization, the UEFI Shell will analyze the UEFI environment looking for instances of `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL`. This is a UEFI protocol that provides abstractions for recognized file-systems. Upon discovery, it will tag each discovered instance with a unique name. It should be noted that this protocol will not be established if the formatting of the media or partition is not recognized. For instance, if a media is formatted as a FAT32 file system, a UEFI system will layer an `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` instance.

Shell Script Resolves into a UEFI Firmware Action

When the UEFI Shell is executing a script, a lot of data needs to be interpreted. Figure 3.10 shows a very common example where a statement such as

```
COPY FS0:\Source.txt FS0:\Destination.txt
```

is interpreted by the UEFI Shell environment. The UEFI Shell initiates several steps during this interpretation:

1. Determine to what command the UEFI Shell needs to pass this data. In this case, the COPY command is recognized and used.
2. Prior to launching the COPY command, install the `EFI_SHELL_PARAMETERS_PROTOCOL` on the target command's image handle. (Recall that this is used for understanding the command-line parameters)
3. Launch the COPY command.

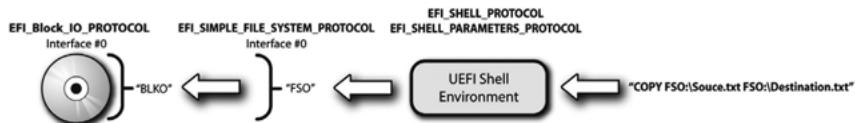


Figure 3.10: From script to hardware interaction

The target command will then be responsible for several actions to complete the requested process.

1. The called command (for example, COPY) will retrieve the command-line parameters to determine what it is being asked to do.
2. Since the COPY command is a shell-enabled command, it will use the appropriate shell interface commands to accomplish its action. For instance, to read/write a particular file (for example, `FS0:\Source.txt`) it will likely use the `EFI_SHELL_PROTOCOL.OpenByFileName()` function to obtain a file handle.
3. Once a file handle is obtained, the same protocol has worker functions to read or write to that file (`ReadFile`/`WriteFile`).
4. Various other miscellaneous activities would occur.

The UEFI Shell environment is ultimately responsible to handling the underlying functions associated with `EFI_SHELL_PROTOCOL`. When calls are made to the functions in this protocol, several actions end up taking place.

1. When a command (such as, COPY) calls the `OpenByFileName()` function to obtain a file handle, one of the key things is to determine where the file physically

- resides. This is determined by interpreting the passed-in data (for example, FS0:\Source.txt)
2. The UEFI Shell environment will have an internal mapping of the file systems that have been recognized (for example, FS0) and it can in turn call the protocol EFI_SIMPLE_FILE_SYSTEM_PROTOCOL associated with that text-based shortcut.
 3. When calling the aforementioned simple file-system protocol, it would pass in the path and file name (such as \Source.txt) and see if the file can be discovered.
 4. If found, the UEFI Shell would have an assigned handle for this opened file. This handle would later be used by subsequent calls to the UEFI Shell and easily associated with the physical file. When asked to read or write to such a handle, the appropriate UEFI interfaces can then be called to complete the request.

Chapter 4

Why We Need an Execution Environment before the OS

In every phenomenon the beginning remains always the most notable moment.

—Thomas Carlyle

As mentioned in Chapter 3, quite a number of software components on the platform execute prior to the operating system taking control. The state of the machine prior to the operating system runtime is generally referred to as the *pre-OS* state and can be broken down into many details. Why is the pre-OS state such a rich environment, what are all of the software components on the machine, and what do they do? These are some of the questions this chapter answers. This chapter provides a review of the states of a platform and the activities that occur therein. This review provides a foundation for successive chapters and their deeper treatment of the items discussed.

Evolution of a Machine

To begin with, the machine restarts in a nascent state. At this point of the machine evolution, there is no memory of I/O devices available. There is just a flash read-only memory (ROM) device on the system board from which the initial code is fetched. This code contains various modules that configure the I/O devices, memory, and system fabric. Figure 4.1 shows an example platform and its flash ROM container.

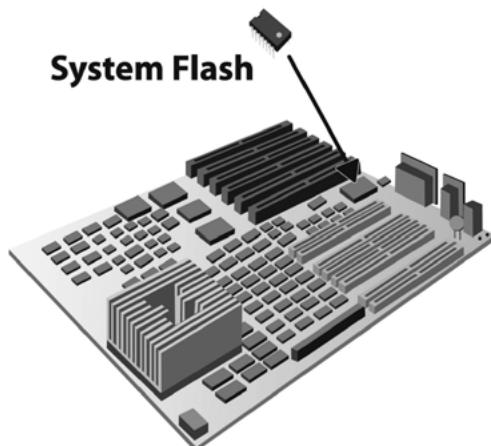


Figure 4.1: System Board with Flash

The flash ROM is manufactured as part of the system board. The executable modules within this device have *a priori* knowledge of the platform and its various components. The flash ROM contents are not intended to be third-party extensible; they are created and updated under the authority of the equipment manufacturer.

The Platform Initialization Flow

A series of executable modules and data files are organized in the flash; these elements abstract basic capabilities and the initialization functions listed above. Figure 4.2 shows the relationship between the types of elements that are board- and module-specific. The important distinction is that a series of modules initially execute in what is referred to as the Pre-EFI Initialization or PEI phase of execution.

The bulk of PEI executes in place (XIP) and uses some temporary memory stored on the platform as a call-stack and heap. This store can include but is not limited to a portion of the processor cache used in such a way that the accesses are not evicted. This use of the *cache as RAM* (CAR) allows for running PEI modules built from C code using standard compilers. But the limitations on this uncompressed XIP PEI code and paucity of CAR means that the minimum amount of activity needs to occur in PEI, namely “initialize useable main memory, discover the DXE core file, and invoke DXE.”

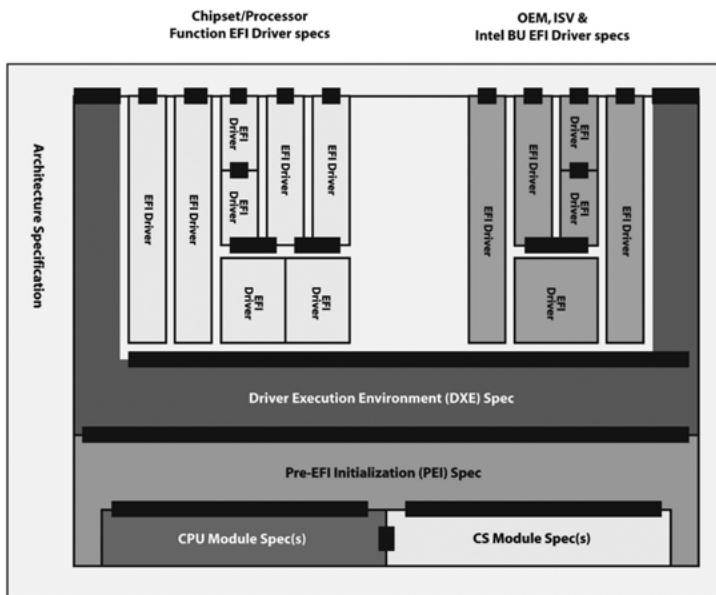


Figure 4.2: Layered Modules in Flash

PEI is the lowest level of elements shown in Figure 4.2 and runs first. Once PEI has enabled a sufficient amount of memory, however, the Driver Execution Environment (DXE) is invoked from the flash ROM. The DXE elements are richer in capability and do not suffer from the space/performance constraints of PEI since they have ample system memory. Figure 4.2 shows discrete, separable components. This separation is accomplished by having the modules communicate with the PEI core via the PEI core services, such as memory allocation and service registration/discovery. The executables themselves are referred to as PEI Modules (PEIMs) and can invoke the PEI core services, publish interfaces for another PEIM to leverage, namely a PEIM-to-PEIM interface (PPI), or bind to other PPIs. The PPIs are named by a Globally Unique Identifier (GUID) such that APIs and data associated with a PPI can evolve as new technology, buses, and capabilities evolve.

The PEI Services table is shown in Figure 4.3.

```

256     typedef struct _EFI_PEI_SERVICES {
257         EFI_TABLE_HEADER           Hdr;
258         //
259         // PPI Functions
260         //
261         EFI_PEI_INSTALL_PPI        InstallPpi;
262         EFI_PEI_REINSTALL_PPI      ReInstallPpi;
263         EFI_PEI_LOCATE_PPI         LocatePpi;
264         EFI_PEI_NOTIFY_PPI         NotifyPpi;
265         //
266         // Boot Mode Functions
267         //
268         EFI_PEI_GET_BOOT_MODE      GetBootMode;
269         EFI_PEI_SET_BOOT_MODE      SetBootMode;
270         //
271         // HOB Functions
272         //
273         EFI_PEI_GET_HOB_LIST        GetHobList;
274         EFI_PEI_CREATE_HOB          CreateHob;
275         //
276         // Firmware Volume Functions
277         //
278         EFI_PEI_FFS_FIND_NEXT_VOLUME2 FfsFindNextVolume;
279         EFI_PEI_FFS_FIND_NEXT_FILE2  FfsFindNextFile;
280         EFI_PEI_FFS_FIND_SECTION_DATA2 FfsFindSectionData;
281         //
282         // PEI Memory Functions
283         //
284         EFI_PEI_INSTALL_PEI_MEMORY   InstallPeiMemory;
285         EFI_PEI_ALLOCATE_PAGES       AllocatePages;
286         EFI_PEI_ALLOCATE_POOL        AllocatePool;
287         EFI_PEI_COPY_MEM             CopyMem;
288         EFI_PEI_SET_MEM              SetMem;
```

```

289      //  

290      // Status Code  

291      //  

292      EFI_PEI_REPORT_STATUS_CODE      PeiReportStatusCode;  

293      //  

294      // Reset  

295      //  

296      EFI_PEI_RESET_SYSTEM          PeiResetSystem;  

297      //  

298      // Pointer to PPI interface  

299      //  

300      EFI_PEI_CPU_IO_PPI           *CpuIo;  

301      EFI_PEI_PCI_CFG2_PPI         *PciCfg;  

302      EFI_PEI_FFS_FIND_BY_NAME     FfsFindFileName;  

303      EFI_PEI_FFS_GET_FILE_INFO    FfsGetFileInfo;  

304      EFI_PEI_FFS_GET_VOLUME_INFO  FfsGetVolumeInfo;  

305      EFI_PEI_REGISTER_FOR_SHADOW  RegisterForShadow;  

306 } EFI_PEI_SERVICES;  

307 //  

308 // PEI PPI Services  

309 //  

310 typedef  

311 EFI_STATUS  

312 (EFIAPI *EFI_PEI_INSTALL_PPI) (  

313     IN CONST EFI_PEI_SERVICES           **PeiServices,  

314     IN CONST EFI_PEI_PPI_DESCRIPTOR     *PpiList  

315 );  

316  

317  

318

```

Figure 4.3: PEI Services Table and Example Function Declaration of One of the PEI Services

In Figure 4.3, lines 257–307, show the full PEI Services table. Lines 310–317 show one function declaration from the PEI Services table, namely the EFI_PEI_INSTALL_PPI service. This latter API is used by a PEIM to publish an interface from its PEIM so that other PEIMs can discover and bind to the interface, respectively.

UEFI Transitions

Figure 4.4 shows the boot flow when PEI hands off into DXE. The early portion of the diagram shows the time evolution of the PEI, which then passes control into the DXE phase. The DXE phase invokes a series of drivers that orchestrate the possible testing of memory not covered by PEI, discover and allocate resources for I/O buses like PCI, initiate other buses like USB, and so on. Once the system fabric has been initialized and

basic platform capabilities are available, the DXE infrastructure provides a set of interfaces that conform to the UEFI specification. This is shown by the line in Figure 4.4 that reads *UEFI APIs*. At this point, DXE passes control to the Boot Device Selection (BDS) interface of the platform firmware and the boot manager capability of the UEFI specification takes control.

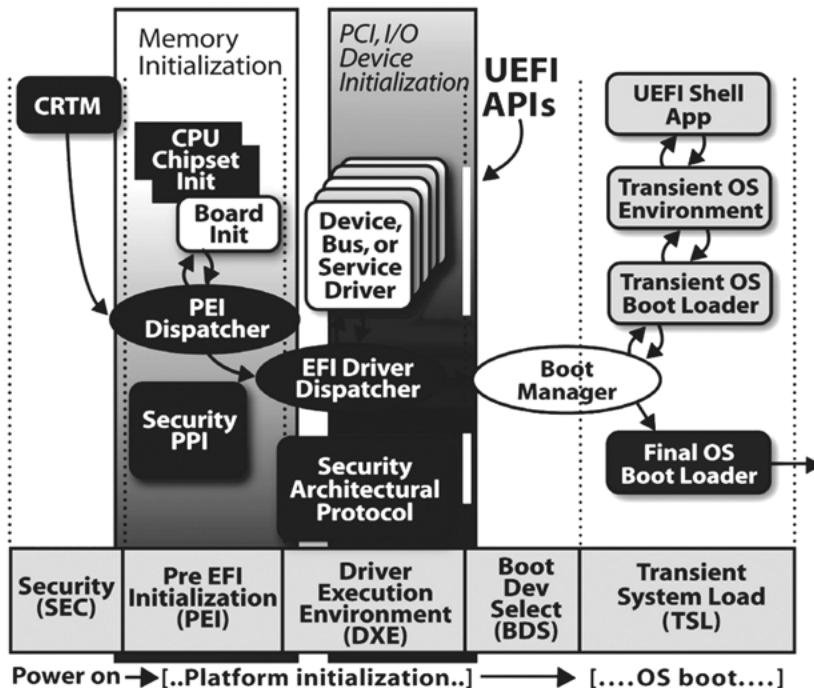


Figure 4.4: Time-Based View of Boot

All of the activity up to this point has been orchestrated by components that are under the authority of the system board or platform supplier (PS). The PEI and DXE components should be installed by the PS authority with suitable protections on the flash part and should not be extensible by third parties. In other words, all of the code running in PEI and DXE should come from the PS. Once BDS has been invoked and the UEFI APIs area is available, however, code from third parties, such as UEFI drivers in option ROMs and operating system loaders or applications from the UEFI system partition may be invoked.

Within this flash part are a series of components that successively initialize more of the platform state. The flash ROM with the boot UEFI code is only one portion of the platform, though. There are several other components on the system board,

whether soldered down or attached via cables. These include block devices, consoles, and networking devices.

The state of these platforms, especially the block devices that have things like the UEFI system partition with the operating system loader, can be installed at various points. The various states of the platform and its configuration will be briefly reviewed here but treated with more detail in subsequent chapters.

States of a Platform

Within this flash part are a series of components that successively initialize more of the platform state, as shown in Figure 4.5. State1 in shown in Figure 4.6 can include the raw system board with just the boot flash ROM and no attached peripherals.

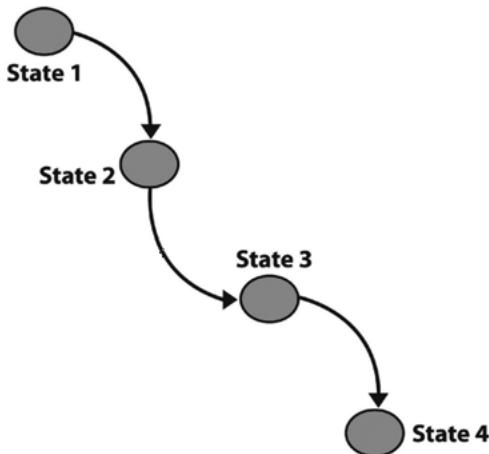


Figure 4.5: Various States of a Platform's Configuration and How They Evolve

In state1, the platform will not be configured with platform firmware or the operating system. We refer to this as *raw*, composed of just the CPU, chipset, RAM, and flash part. The latter will not be programmed with any of the binary code content, nor would other reprogrammable elements on the platform, such as EEPROM's or device flash components.

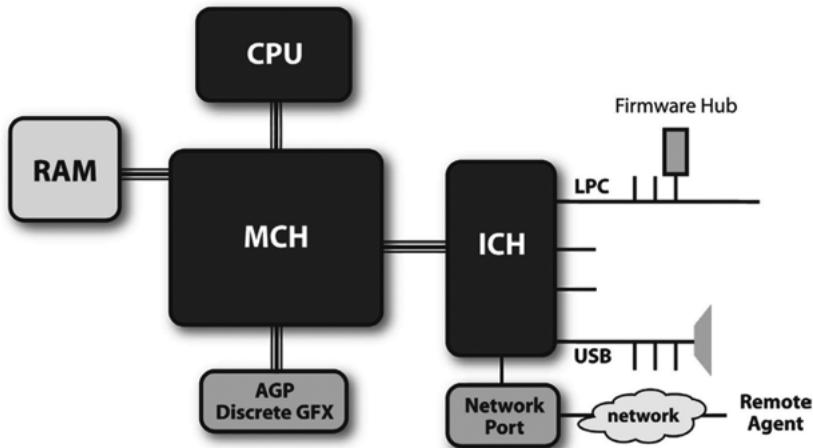


Figure 4.6: State1: Raw Platform

This raw system board is typically put into a box or integrated into a chassis by a vendor, during which time the peripherals may be added. We refer to this as state2, shown in Figure 4.7. At this point, there is typically no information on the added hard disk.

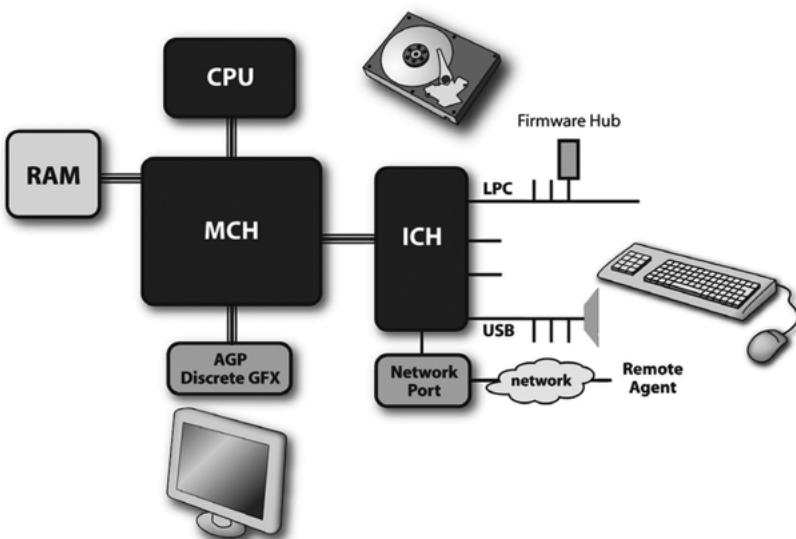


Figure 4.7: State2: System Board with Peripherals

State2 may evolve in the vendor factory (for a fully integrated system) or at a value-added reseller into state3. State3 entails installing the operating system on the machine. During state2 some vendor tests may be run in order to ensure the physical integrity of the system.

In addition, this type of deployment isn't limited to a single system board. A blade server in a rack can also be the target of the states of configuration described herein, as shown in Figure 4.8.

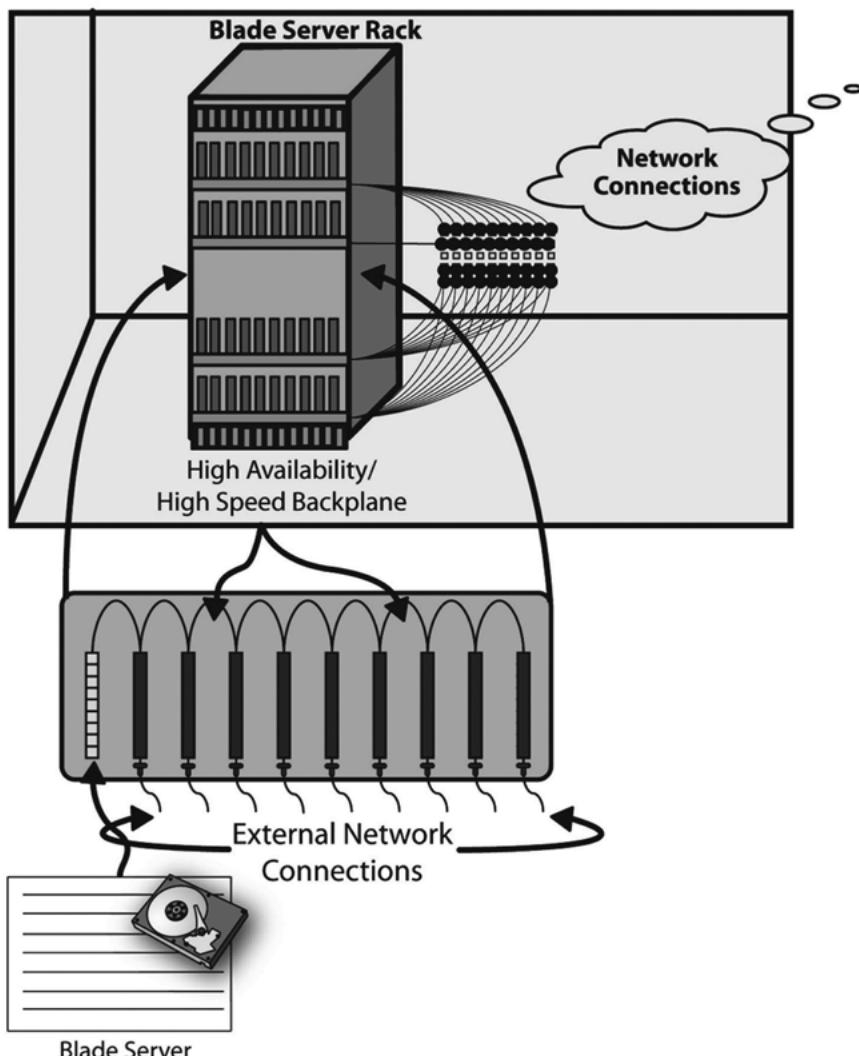


Figure 4.8: Blade server in a server rack

Readiness of UEFI

At this point UEFI and the PI firmware have discovered the hardware complex, but there is still no operating system. State3 entails “imaging” an operating system onto the hard disk via a locally attached device, such as a CD-ROM drive, or via a network boot, as shown in Figure 4.9. The act of imaging an OS is to write it directly to the logical block addresses of the storage media with a file system intermediary.

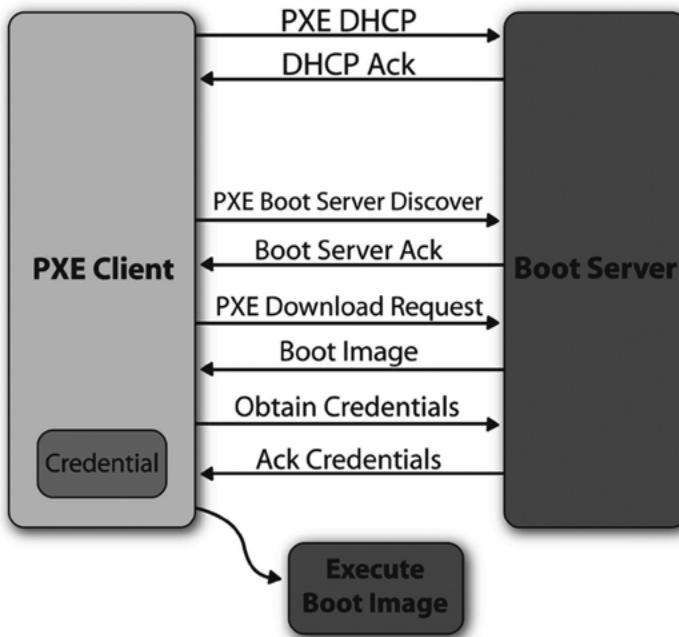


Figure 4.9: Network Boot

The network boot case is important for both manufacturing flow, when the final disk image is sent onto the newly built machine, and also for live deployment. In the latter case, a diskless client machine that boots an IT-authorized OS image each day, or a blade server that only has memory and CPU elements but requires loading the runtime OS image from the server area network (SAN), both entail the use of the platform networking capabilities.

The manufacturing case of network boot is valuable since the build-to-order machine may have the user select one from various operating system options, including Microsoft Windows or Linux. This final stage of manufacturing when the OS image is

transferred onto the disk via UEFI networking allows for a no-touch, remote configuration of the product.

As noted above, the UEFI interactions occur once the PEI and DXE have executed. The rich set of UEFI drivers and applications occur in the flow as shown in Figure 4.10.

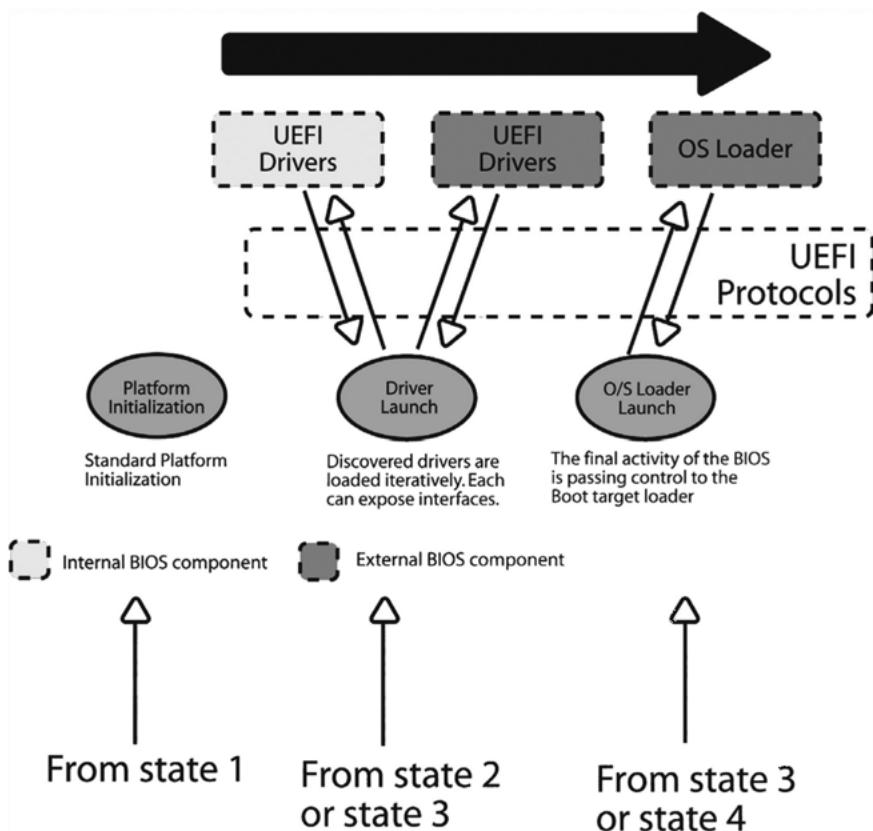


Figure 4.10: Flow of the System

The UEFI drivers can be loaded from the UEFI system partition or from a host-bus adapter (HBA), such as a PCI SCSI or NIC. The UEFI applications can include executables integrated into the flash ROM, on the UEFI system partition, or loaded across the network. These applications include diagnostics, operating system loaders, and the UEFI Shell.

The most important activity in state3 is to install the operating system on the disk. Recall from state2 that the disk was attached, but at this point it contained blank logical block addresses.

During state3, the platform manufacturer is still in control of the system. An operating system installer can run from the factory environment in order to deposit the OS loader, kernel, and support files on the disk, as shown in Figure 4.11.

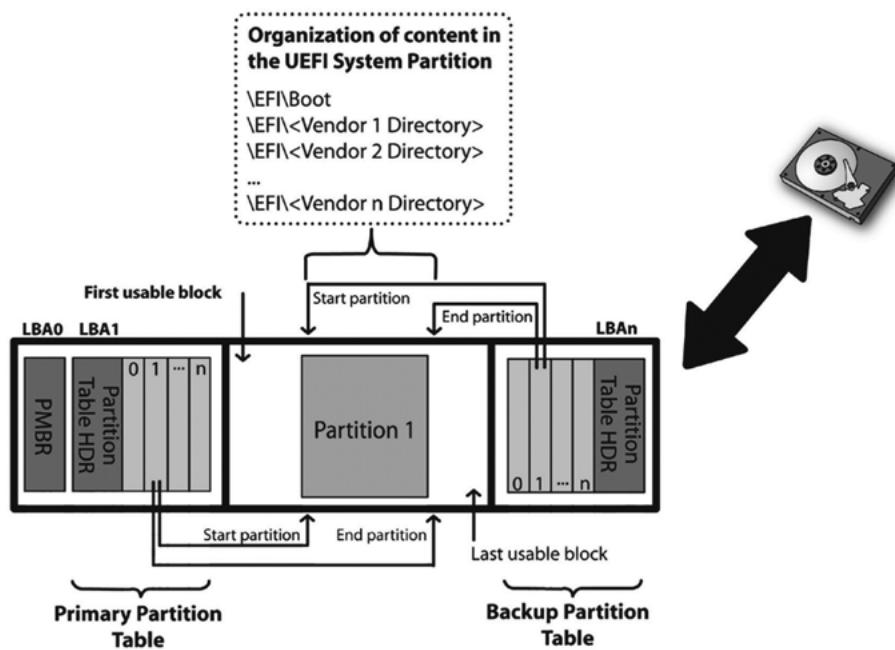


Figure 4.11: Disk with OS Installation

At this point, a complete system, such as a server or a laptop would ship to an end user. We refer to the end user as the owner of the platform. The owned platform, in our state diagram, now refers to state4. The owner can choose to run diagnostics, install additional operating systems, or reinstall the former OS as a repair or upgrade operation.

At state4, the ability to run the UEFI Shell, including diagnostics such as reading the SMBIOS tables or other asset information is still valuable. The UEFI Shell exposes all system memory and hardware resources. An end user may choose to boot into the UEFI Shell during the life of his or her platform in order to run some memory test or disk diagnostics. Such exhaustive testing cannot easily occur during the OS runtime since a failure in memory or I/O surfaces in unpredictable ways during an OS crash. Also, attempting to run such diagnostics *in situ* during OS runtime can lead to various

Heisenbugs (bugs where the observing agent and system under observation interact, named after Heisenberg's Uncertainty Principle, which noted that momentum and position of an electron could not be precisely measured at the same time because the act of observing the electron would perturb it).

Migration Using the UEFI Shell

Another important application of the UEFI Shell is to facilitate the migration of one machine to another. The scenario is as follows:

- A single machine is configured manually or via UEFI Shell script by IT per their requirements
 - This is referred to as the “golden” machine
 - The configuration of the golden machine is sent to a central network or rack repository, such as a Chassis Management Module (CMM)
- The golden machine is then attached to the network
 - Imagine the scenario below where the golden machine is a blade server in a rack
- The rack of blades is activated and a UEFI application runs on each of the unconfigured blades
 - Each blade talks to the well known network authority, such as CMM, to get the configuration
 - The UEFI Shell application on each blade applies the golden configuration
- The act of collecting the configuration on each of the blade’s configuration application actions will apply generic settings, such as OS boot targets, language codes, and other UEFI-defined specification options, but it will elide certain blade-specific options, such as the local blade MAC address
- Once the configuration of all the blades has occurred in the pre-OS state, the rack can be restarted and the successive blades booted to their locally attached storage or a network-based boot target via a PXE or iSCSI boot

This golden machine cloning or migration of the machine personality is something that must occur prior to the OS launch, and entails a state3 to state3 migration or state4 to state4 migration across the different computation units, as shown in Figure 4.12.

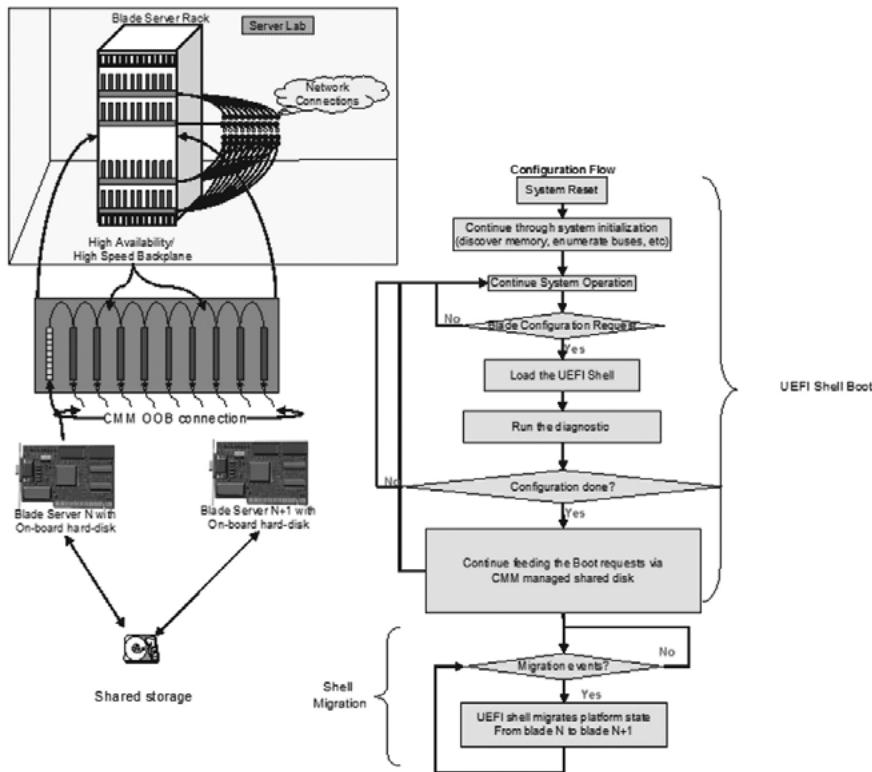


Figure 4.12: Migration of UEFI settings from one blade to another

Even though this example shows a migration across a series of blades in a rack, it would have been equally applicable across a class of enterprise client machines, handheld devices, or other appliances. The key common elements across these platforms are the UEFI-based firmware, the UEFI Shell, the ability to reach the common repository of settings, and the cloning application.

Going Forward

Subsequent chapters provide guidance on how UEFI-based technology such as the UEFI Shell can assist in these latter state evolutions of the platform. For example, configuration in state4 can entail enrollment of additional devices, such as “taking ownership” of Trusted Platform Modules (TPMs), the cloning of machines so that an enterprise owner of a large number of units can personalize them for particular business needs, or further storage options can be applied, such as configuration of a Redundant Array of Independent Disks (RAID) with additional options.

Chapter 5

Manufacturing

“Give me six hours to chop down a tree and I will spend the first four sharpening the axe.”

— Abraham Lincoln

The motherboard is finished. Screws hold it down within the shiny new case. Memory is inserted. Peripherals secured. Cables run from source to destination. Firmware image flashed. Enclosure sealed. Power applied.

Now what?

It’s a box. A very complicated box, yes. Yet still a box. There isn’t much there yet. The firmware is just sitting there on a flash chip and a CPU anxious to execute that firmware. Then the spark of life as power is applied and line after line of preprogrammed instructions carry the system forward until...until what?

The box doesn’t know. It is a *tabula rasa*—a clean slate—waiting to be given a purpose. While manufacturing starts with the correct placement and connection of electrical components, it finishes by ensuring that the assembled hardware is prepared for use.

Before that box can be useful, the box must be:

- *Provisioned*. The hard drive must contain a copy of the golden image—the latest validated operating system and pre-installed applications. See Chapter 6, “Bare Metal Provisioning,” for more information.
- *Configured*. The firmware must contain the golden configuration—the optimized settings for the specific hardware configuration and target operating system. See Chapter 7, “Configuration of Provisioned Material,” for more information.
- *Tested*. The hardware must quickly pass system and subsystem functionality tests before leaving the factory. See Chapter 8, “The Use of UEFI for Diagnostics,” for more information.

This is not a book about manufacturing, which is a vast subject in and of itself. But even the cursory analysis in this chapter shows how the UEFI Shell can play a valuable role in helping each box pass through these stages in the factory reliably and quickly. This chapter focuses on the vital role that manufacturing tools can play in making your product successful.

Throughput

Speed is essential. With 10 or 100 or 1,000 boxes the speed of testing, configuring, and provisioning each box is not a significant factor. But with 1,000,000?

Take a single hypothetical assembly line that runs for 24 hours a day where each box takes only 5 minutes to manufacture. One million boxes means 3,472 days or slightly over 9.5 years. Each additional second for a single box adds almost 12 days to the length of the production run. Consider that even with 100 such manufacturing lines, it is still over one month of non-stop, around-the-clock, no-down-time factory effort.

There are no more dreaded words than “line down.” For a firmware engineer, a line down issue means travel to the factory floor on the next available flight, no sleep, and hourly executive management status reports. Why? Because any delay means idle people and production equipment, delayed product launches, delayed delivery schedules, and delayed sales.

For many companies, the manufacturing environment is still Microsoft’s MS-DOS, or one of the various alternatives (PC-DOS, DR-DOS). Companies (and engineers!) are risk-averse: they don’t like to change what has been working. Considerable expertise has been developed over time. Special-purpose tools have been created for a company’s specific needs. Manufacturing-line servers that manage the whole process often have a fragile relationship with those tools. So, why break what works?

The trouble is that Microsoft hasn’t updated MS-DOS in over 20 years (version 6.22, 1994), and it and the related tool chains are moribund. Many times it is hard to even find them, and precious floppy disks are hoarded. Even more significant: it is hard to find engineers who understand the unique APIs and batch file syntax of DOS and the quirks of the 16-bit C compiler included with Visual Studio 1.52 (we are now at Visual Studio 14.0!). Python? Forget about it! More than 4GB (much less 640KB)? Unlikely!

Many alternatives have been proposed, from Windows PE to various versions of Linux, but the substantial hardware requirements for these environments makes it likely that the environment itself will be configured improperly.

The UEFI Shell provides a unique environment for testing the boxes at an early stage because:

- The UEFI Shell is powerful, including a full scripting language, logging capabilities, and access to the UEFI driver model. It is optimized for working without an attached storage device.
- The UEFI Shell is small. Some implementations require as little as 100KB of flash space. This means it can be burned into the flash device during the manufacturing process without requiring a larger flash device. UEFI Shell applications can be stored in the flash, on external storage devices or on a network server, accessed via TFTP, FTP or HTTP.
- The UEFI Shell has a fast boot time. This allows critical seconds to be shaved off of the testing time, since there is no wait for the hard drive to spin up or the network download to complete.
- The UEFI Shell has unrestricted hardware access—including access to all CPU cores and threads, all hardware registers, all system memory and all hardware interrupts.

- The UEFI Shell has few hardware dependencies. Basically, if memory, the timer, the timer interrupt, and a console device are working, the UEFI Shell is working. That means the UEFI Shell applications can test more.
- The UEFI Shell supports the latest hardware, including more than 4GB of memory, PCI Express, USB 3.x, and more. Since it runs directly on top of UEFI, it has access to all of the devices that UEFI does.

The UEFI Shell provides a complete, Posix-compliant C library and supports dozens of toolchains, including every major version of Microsoft's Visual Studio and GCC. This allows manufacturing tools to be migrated easily from the current manufacturing environment to the UEFI Shell.

Also, the UEFI Shell supports multiple file systems, including FAT12/16/32 and El Torito (used on optical media), and there are multiple sources for UEFI support of NTFS, EX4 and more.

Finally, the UEFI Shell shares many development principles with UEFI itself, which allows engineers who are an expert in one to also develop code for the other.

Manufacturing Test Tools

Testing plays a big part in the length of time that a box stays on the manufacturing line. The raw box may have defects. These defects may appear consistently or sporadically. The earlier in the manufacturing process these defects are detected, the less time is wasted on that box.

Some test failures only occur on some boxes. The testing phase also detects patterns among failing boxes, such as shared faulty components, shared malfunctioning assembly equipment, or shared suppliers. Then steps can be taken to correct the problems, possibly through something as simple as a modified configuration, a corrected process, or a firmly inserted cable.

Before the manufacturing line started, there was a golden system. Poked and prodded by technicians and firmware engineers, it passed every test thrown at it. The thousands or tens of thousands, or hundreds of thousands that follow must have the same level of quality. Will the other boxes maintain this level?

After the box has been put through circuit testing, making sure that there is continuity for the signals, the new box is put through tests to verify that it functions as expected. Since the UEFI Shell has access to the UEFI networking stack, a simple script downloads these tests from a network server on the factory floor, executes them, and then sends the log back to the network server. The network server quickly scans the log for errors and, if detected, flags the box and alerts the supervisor.

Some of the tests require more time, or require special environments, or are less likely to occur (but expensive if they do). Rather than hold up all of the manufacturing lines, a sample of the boxes from each of the lines is set aside. In the initial runs, all

units go through this more intensive testing. Later, during volume production, the sample size is reduced.

Some of these tests are *long-run* tests, consisting of repeating a single action over the course of thousands of reboots. For example, entering the standby or power-off state repeatedly to stress the hardware and the firmware. Some of these tests are *environment* tests, consisting of exposing the system for hours or days to high humidity, low humidity, high temperature, and low temperature. Some of these might be sub-system tests, such as memory, which are prone to failure because of slight variations in manufacturing lines, or component supplies.

The good news is that, if these failures can be isolated and diagnosed, many times a UEFI firmware-level fix can be implemented and then that UEFI firmware image can be fed back into the process. In addition, the update can be applied to those units that were diverted because of the issues found.

These benefits are not just reserved for the original equipment manufacturers. Other companies that provided integrated solutions, where they take a box and add specific peripherals, operating systems, components and applications can use the UEFI Shell to make sure that all of the systems they ship have the correct configuration by updating the firmware, checking the installed devices against a known-good list, configuring those devices, and provisioning the hard drive, all from the UEFI Shell.

Hardware Access with Manufacturing Tools

When writing tests during manufacturing, you need access to different levels of information or control. DOS gives you one level of control: INT 0x21 for the file system, and legacy BIOS gave you one more level of control: INT 0x13 for block I/O (or INT 0x10 for video). Of course, a library like the C library could abstract this, but it could not give you more control.

So your application looks something like Figure 5.1:

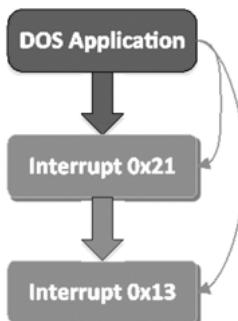


Figure 5.1: DOS Application Hardware Access

If your application needed more control, it must have intimate knowledge of the file systems, partitions, buses, and devices. With UEFI Shell applications, however, applications have access to the same file system level abstractions and block I/O abstractions that DOS offers. But they also have access to so much more, as shown in Figure 5.2:

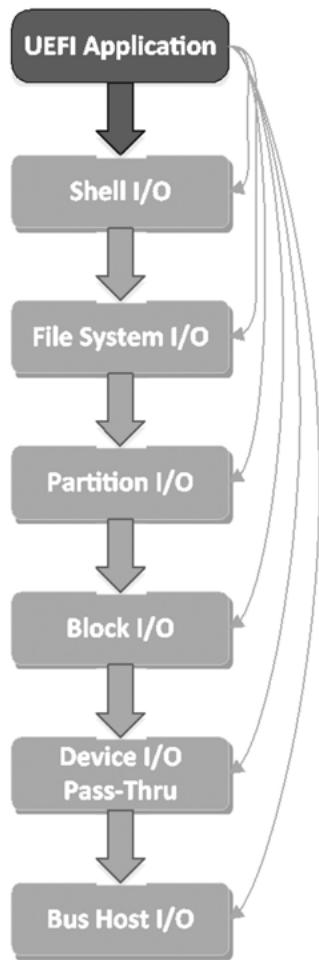


Figure 5.2: UEFI Application Hardware Access

Through the use of the UEFI programming model, it is possible to enumerate all of the devices of a specific type in the system. UEFI provides a standard data structure, the *device path*, which describes exactly how a device is attached to the box. The device path can be walked all the way from the host bus through the block device to the

file system and even to an individual file. This gives numerous options for a UEFI Shell application so that it can insert its test at exactly the right level of abstraction.

- *Shell I/O:* The UEFI Shell itself provides file system mappings (that is, FSO:) for each of the mounted devices in the system.
- *File System I/O:* UEFI abstracts access to the file systems installed in each partition of each device through the use of the Simple File System protocol. Using this information, it is possible to manipulate the one or more file systems installed on a particular device.
- *Partition I/O:* UEFI abstracts access to each partition on a device as a logical block I/O device using the Block I/O protocols. It also gives byte-oriented access to the device via the Disk I/O protocols. So a UEFI Shell application can easily read and write the contents of a storage device partition.
- *Block I/O:* UEFI abstracts access to each storage device via the physical Block I/O and Disk I/O protocols. So a UEFI Shell application can easily read and write to individual blocks on the disk.
- *Bus I/O:* UEFI abstracts direct access to the device on its native bus via the bus I/O or pass-through protocols. The bus I/O protocols allow sending and receiving data across an I/O bus such as USB, PCI, or serial. The pass-through protocols are similar, allowing sending and receiving storage-bus specific packets directly to storage devices (for example, SATA, IDE, SCSI). This frees up the application from having to know the details of specific device registers or how the bus has been configured.
- *Bus Host I/O:* UEFI abstracts direct access to the device's host controller beyond sending and receiving data to devices that are on the bus. Many of the bus controllers, such as PCI, USB, SCSI, and even (in the PI specifications) I2C and SPI have direct host controller protocols that allow even greater access for testing.

Using the right level of abstraction means that tests for a specific class of devices can be done without worrying about unnecessary details. This, in turn, means that the test will be more portable from one generation of boxes to the next generation of boxes. It also means it is more applicable, to other devices of the same type, but which are located on another similar bus. For example, many tests for IDE also work for SATA. Many tests for optical disks also work for SCSI.

Using the right abstraction level for access also means that the tests are less likely to destabilize the box (or subsystem in that box) under test. Since the box (or subsystem) is usually running many tests during the manufacturing process, it is important for them to be run one after the other to save time. This can only be done if the box (or subsystem, or device) is still in a working state after finishing the previous test. The worst case is that there must be either a physical intervention (that is, cable inserted or removed) with the box, or the box must be reset. While these are sometimes unavoidable, each one adds time to the manufacturing test. (Yes, `ResetSystem()` is disruptive and adds time.)

Using the right abstraction also allows third-party tests to be inserted into the manufacturing process. For example, each component vendor can provide tests of the condition of their individual component and even, in some cases, the UEFI-compatible driver that supports it. Since they use standard APIs provided by UEFI, they can be used over and over again, often with no more modification than changing a few command-line arguments. In addition, while they test their component, additional tests that use that component can be inserted in the same script.

Converting Manufacturing Tools

So, how do you convert tools to work in the UEFI Shell environment? Even though the UEFI Shell has many advantages, there is still work required to make the leap from the 16-bit, 1980's MS-DOS environment tools.

The clock is ticking. Every month that you defer converting over to a more modern tool environment simply makes the job harder. It is harder to find a system that can host the operating system and compiler necessary to build the tools. It is harder to find engineers in your company who still remember how the tools are supposed to work. It is harder to find documentation that describes the process.

Here are some key steps:

- UEFI. This may seem obvious, but converting the tools requires familiarity with the basic concepts behind UEFI and the UEFI Shell. The UEFI specifications are great reference material, but they aren't constructed to teach UEFI and the UEFI driver model. You are already taking a big step by reading this book. We would also recommend the authors' book *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*.
- UEFI Build Environment. Since UEFI Shell applications are a specialized type of PE/COFF format executable, there are multiple build environments for creating them, including open-source (EDK2, gnu-efi) and those provided by UEFI firmware vendors. All of these build environments provide sample C applications that can be used as a starting point. Most of them **do not support** standard C style make files, so there will be a learning curve. See Chapter 10, “UEFI Shell Programming,” for more information on the EDK2 build files.
- Tool Status. Make sure that you have access to the tool’s source code and the necessary tools to recreate the original tool.
- Review the code to see if the code flow is easy to understand.
- Identify the areas in the code where there is direct hardware access.
- Take note of direct calls to DOS and BIOS services (via INT 0x10 and INT 0x13) via some sort of library wrapper.
- Watch out for Software SMIs! Many older tools use an output to a hardware I/O port (such as 0xb2). These indicate a dependency between the tool and the underlying firmware code written in (System) Management Mode.

- Mark these to revisit later.
- Key Engineers. Find the key engineers in your company who have knowledge about these tools. Engineers are often, by nature, hoarders. Somewhere they may have design documents, however incomplete or out of date. Spend some time with them to understand the tools and then **write down what you find out**. This will save time later, both for you and whoever has to do this job again later.
- New Goals. Many times, once you decide to do the conversion, everyone presents their wish lists. These are the things that others in your company wished they could do, but didn't because the cost and difficulty of updating the old manufacturing tools was too high. This may include new tests, new logging formats, new manufacturing server infrastructure, or something else. So be prepared!
- Clean Up. One of the goals may simply be to streamline the code.

Just one final note on converting manufacturing tools: accept the new interface! Sometimes there is a temptation to create some sort of “interface layer” that makes UEFI look like DOS or Linux or some other favorite environment. Resist the temptation. Don’t try to make UEFI look like Linux or DOS.

Conclusion

The UEFI Shell has a lot to offer during the manufacturing process:

- Fast Boot
- Small Siz.
- Direct Hardware Access
- Network Connectivity (Including Network Boot)
- Support for the Latest Hardware
- Optimized Scripting
- Generation-to-Generation Stability

Now is the time to make the upgrade to UEFI Shell-based manufacturing tools.

Chapter 6

Bare Metal Provisioning

Winning starts with beginning.

—Robert H. Schuller

One of the most apt uses for the UEFI Shell is in the bare metal provisioning of a system. Recall from earlier chapters on the states of the machine that there is a point where the system board has been manufactured, peripherals have been attached and tested, but there is not necessarily an operating system.

The act of configuring the operating system or providing a new operating system is called *bare metal provisioning*. We refer to this action as *bare metal* because the only services exposed by the platform at this point are carried with the system board, namely its UEFI firmware. This differs from OS-hosted provisioning wherein a fully extant operating system with all of its capabilities is used to host the provisioning session. In the bare-metal case, the system firmware provides the I/O and console interfaces to the provisioning agent.

To illustrate the usage of the UEFI Shell for provisioning, a network-based deployment scenario will be reviewed. During this scenario, the various facets of the UEFI Shell and its utility will be demonstrated.

Provisioning with the UEFI Shell

In this scenario, we'll return to our system platform, which we will refer to as the “UEFI Client,” shown in Figure 6.1. It is a client only inasmuch as it will interact with a provisioning server. The “client” could itself be a mobile device, Mobile Internet Device (MID), desktop PC, or server. The client moniker only designates its role in the networking scenario.

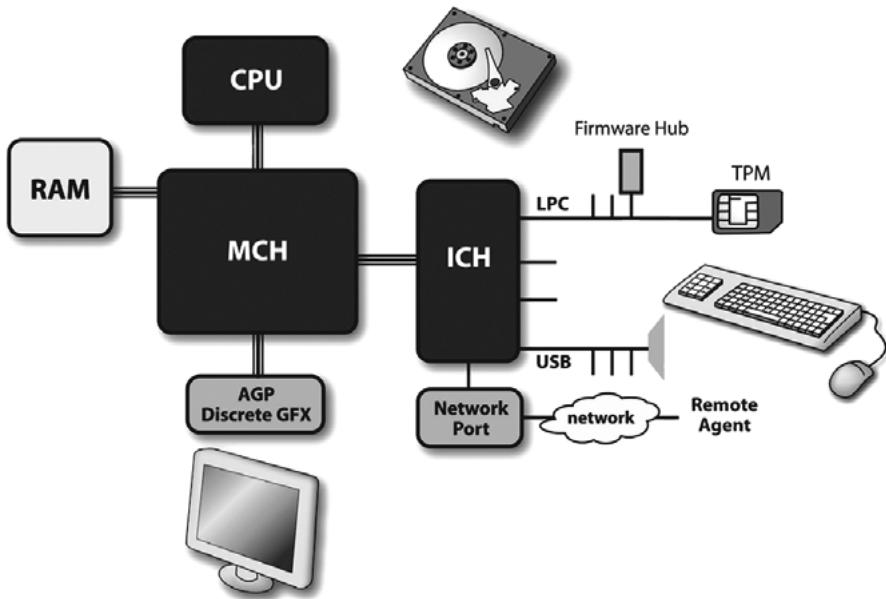


Figure 6.1: Client Platform

This client platform, depending upon its market segment, may or may not have a local disk, video, keyboard, or mouse. But what any platform will have for this scenario includes a central processing unit, chipset, memory, flash part with the UEFI firmware, and most importantly, a network connection with an appropriate network interface controller (NIC) and networking UEFI drivers.

UEFI Networking Stack

The UEFI drivers in this scenario include the UEFI2.2 networking stack. This networking stack supports ARP, UDP, TCP, the PXE network-boot application, optional iSCSI, and a variant for both IPv4 and IPv6, as shown in Figure 6.2.

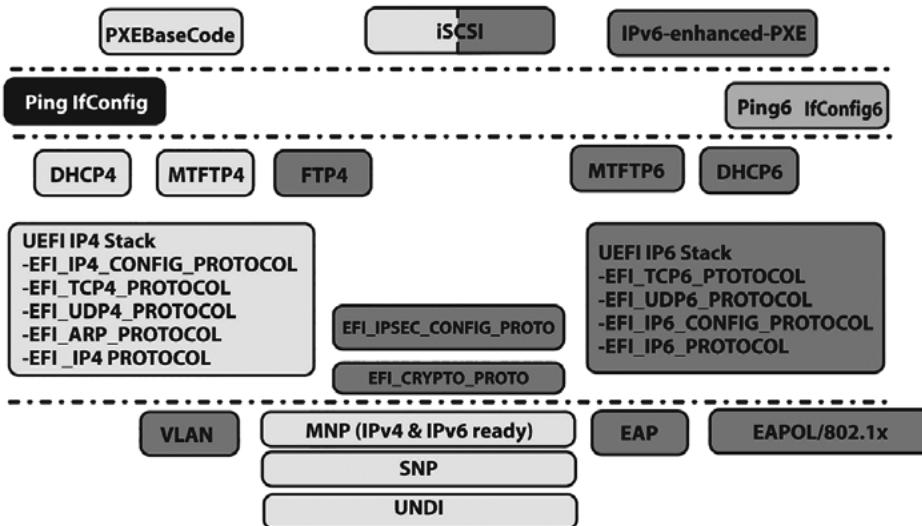


Figure 6.2: Networking Stack

The diagram in Figure 6.2, from the bottom up, shows the hardware elements of the stack, such as the Universal Network Device Interface (UNDI), which abstracts the NIC send and receive datagram capability. This raw UNDI interface is abstracted via the Simple Network Protocol (SNP). The SNP, in turn, is abstracted by the Managed Network Protocol (MNP). As opposed to the earlier EFI network technology wherein only one application could open the SNP exclusively, the MNP allows for many concurrent listeners, thus allowing for several services to concurrently operate. What this means is that an Extensible Authentication Protocol (EAP) handler looking for layer 2 messages can coexist with both the UEFI IP4 and UEFI IP6 stack applications such as PXE and iSCSI.

The multiple-consumer nature of the MNP and the UEFI2.6 network stack is important because many of the network-based provisioning scenarios are “blended.” The blending is born of the fact that the scenario may entail some network authentication action as a preamble in order to allow the client onto the provisioning network, then an iSCSI mount in order to have a file system, and finally, a PXE boot so that an installer or test application can be downloaded. The latter UEFI application, in turn, will access the file system on iSCSI while performing its own network-specific file operations.

Securing the Network

Because of the expanded natures of local area networks (LANs), wide-area networks (WANs), and networks of networks, such as the Internet, providing secure interaction among agents on the wire is imperative. One of these scenarios is shown in Figure 6.3, which explains a network download and the salient trust elements.

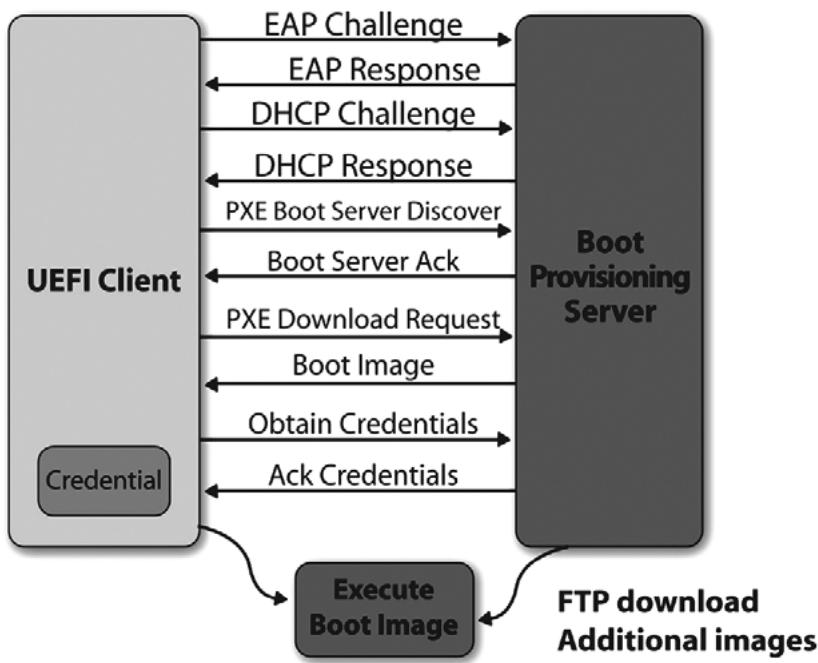


Figure 6.3: PXE Boot

The notable aspect of this download is the initial EAP transaction. Historically, PXE began with a DHCP discovery message in order to ascertain the location of the boot server. But corporate IT departments have some concerns with this former model. IT typically hosts the boot provisioning server, and the concept of a “bare-metal” machine joining their corporate network is a concern. With the advent of 802.1x-controlled ports and EAP, however, the client machine can be configured in such a way that it must satisfy a challenge-response prior to joining the network, as shown in Figure 6.4. This entails the client responding to a set of EAP layer 2 messages from the switch; the messages are in turn transmitted to the authentication server via the well-known Radius protocol.

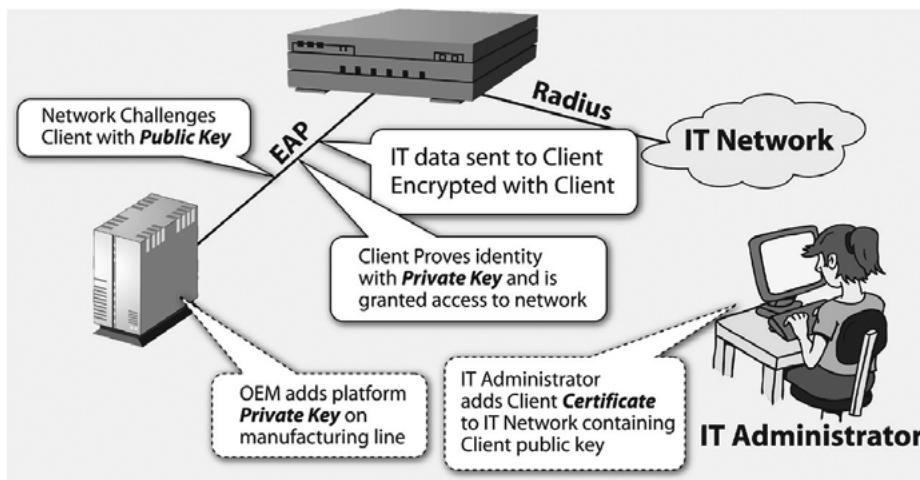


Figure 6.4: 802.1x/EAP Challenge/Response

Once the client machine has been authenticated via the EAP challenge/response, it is allowed onto the network. UEFI 2.2 supports both the 802.1x state machine in the firmware and the ability to register a plurality of different EAP handlers. These handlers can include pre-shared key (PSK) methods, such as CHAP, and can be extended to more sophisticated single or mutual authentication methods based on asymmetric cryptography like RSA, including but not limited to a Transport Layer Security (TLS) handshake.

As needs arise, other network perimeter EAP methods can be included in the UEFI clients, such as EAP Kerberos or the Trusted Computer Group's EAP Trusted Network Connect (TNC). The former is interesting as it will allow the UEFI Client to participate into an extant enterprise network topology, and the latter is useful since the pre-OS posture of the UEFI client, such as its code identity as described in the Trusted Platform Module's (TPM) Platform Configuration Registers (PCR), can be used to assess the posture of the UEFI client prior to letting it on the network.

EAP is at layer 2, so it works on IPv4 or IPv6 networks. After gaining access to the network, the UEFI client can attempt a DHCPv4 or DHCPv6-based network discover request. The ability to do either or both is enabled via UEFI and its dual stack. With the imminent exhaustion of the 32-bit Internet Protocol version 4 addresses, deploying UEFI 2.2 firmware with Internet Protocol version 6 support is key. IPv6 opens up the address space to 128-bits. And as UEFI firmware proliferates to compute platforms beyond PCs and servers (UEFI for appliances, non-standard platforms, and so on), the ability to support IPv6 networking in the pre-OS is imperative.

After the UEFI PXE application has negotiated with the boot server for a server name and IP address, it can commence the download of the boot image. In UEFI, the boot image isn't the small 16-bit file of some tens of kilobytes as in conventional BIOS, but is instead a fully-qualified Portable Executable Common Object File Format (PE/COFF) image. The UEFI executable is downloaded to the client machine for execution.

Now recall how we mentioned that IT may set up EAP for authenticating the client so that a rogue UEFI machine may not wander onto IT networks. A similar concern emerges with respect to the downloaded executable. The UEFI client wants to defend itself from any random bits on the network, especially given the distributed nature of today's topologies, rogue wireless access points, and other venues for Man-in-the-Middle (MITM) attacks on the wire to occur.

The credential listed in Figure 6.4 would be something like an x509v2 certificate with a public verification key. The UEFI firmware uses the public key to verify the digital signature of the boot image in order to ensure that it hasn't been modified by an unauthorized party during transit. UEFI2.2 introduced the use of Authenticode image signing so that the trust hierarchy can be flat or nested, allowing for various deployment options. In addition, the rich UEFI network stack allows for the firmware to check for certificate expiry for possible future revocation models (for example, if the private key associated with the public key in the certificate has been divulged).

A sample certificate is shown in Figure 6.5:

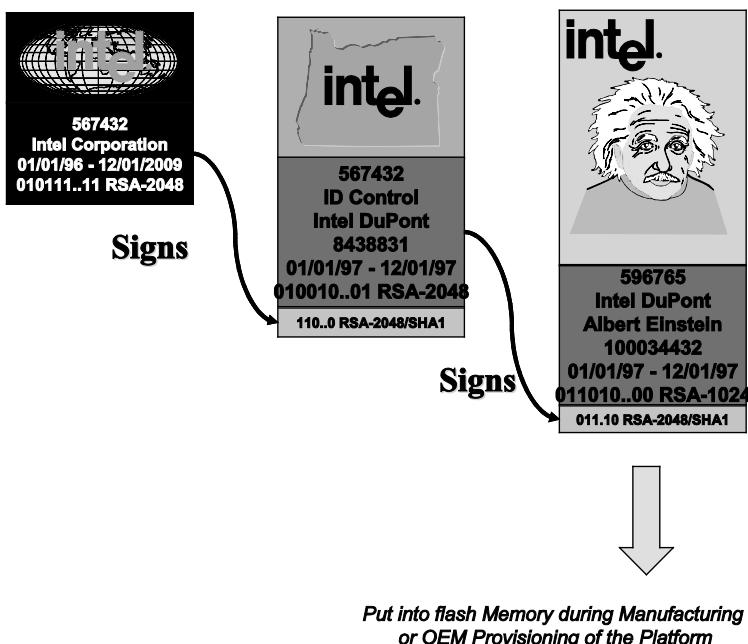


Figure 6.5: Example of an x509v2 Digital Certificate

Important fields include name information, the expiry date, and the signature. The signature in this example includes an SHA-1 digest of the data signed by a 2048-bit RSA asymmetric key. This example also includes a 2-level deep hierarchy where the authority in this case is Intel.

To bring all of these elements together, Figure 6.6 shows a flowchart of the end-to-end process of booting. It includes the actions of both the boot server and client machine.

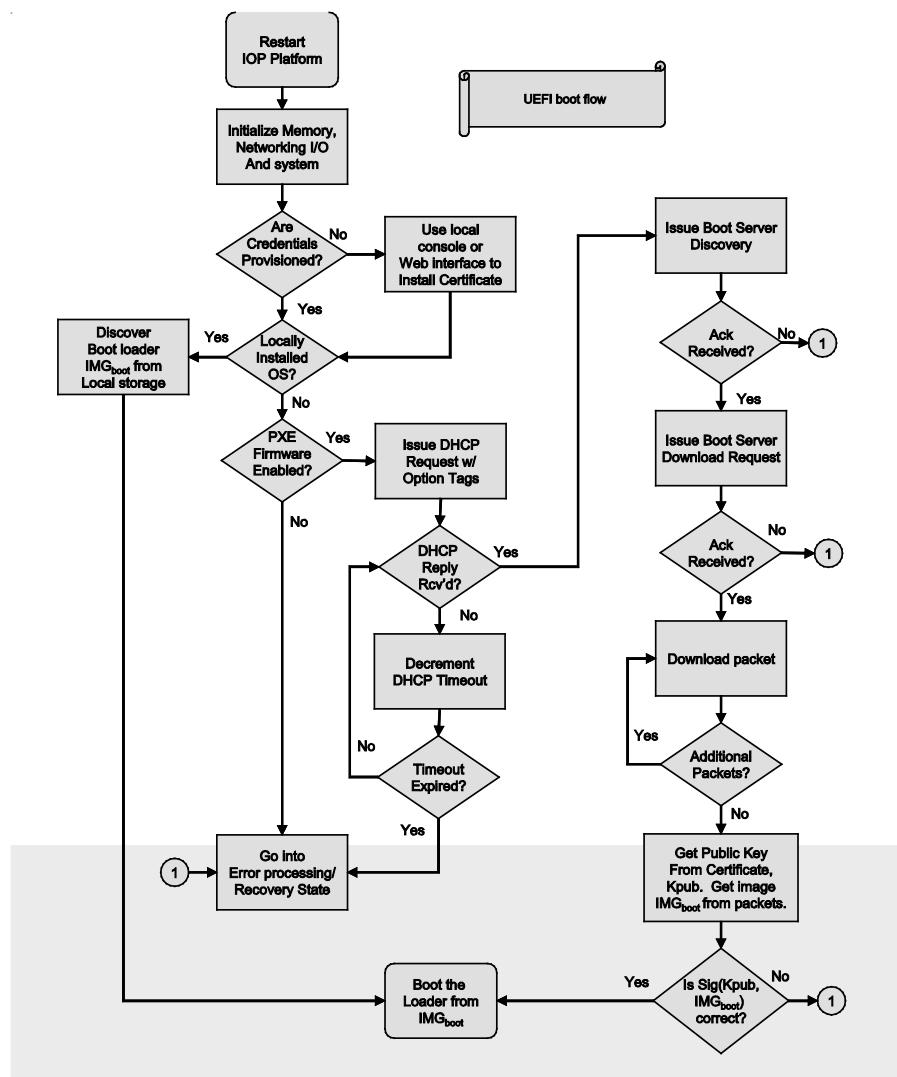


Figure 6.6: Overall Boot Flow

Figure 6.6 shows the overall network boot process. The first portion is the use of the Dynamic Host Configuration Protocol (DHCP) by the UEFI client in order to query the network for both an IP address and the availability of a boot server. The DHCP offer from the client and response from the server are important in that the client tells the boot server the “type” of client, such as x64 UEFI. The server, in turn, responds if it can support providing images to the machine.

One important image that can be downloaded to the platform is the UEFI Shell. Given that there are a plurality of activities that need to occur during provisioning, such as downloading additional files to image on the disk, activating certain devices like TPMs, the shell can be used to orchestrate their invocation and pass results.

Speeding Up the Network

One of the advantages of performing network downloads is the introduction of the File Transfer Protocol (FTP) into UEFI. One of the historical complaints about the network boot experience is the use of PXE and UDP. The user datagram protocol (UDP), upon which the trivial file transfer protocol (TFTP) is built, was originally chosen because of its simplicity, thus leading to smaller code implementations in the pre-OS. But TFTP’s disadvantages include the fact that it is connectionless (because of UDP), has small blocks, and requires several ACKs. This makes TFTP a very non-scalable protocol. FTP, on the other hand, is built upon the Transmission Control Protocol (TCP). TCP is a connection-oriented protocol that features much more robust download capabilities.

Going forward with PXE, the DHCP handshake will be extended to allow for today’s TFTP downloads, FTP, or even HTTP. The latter two can include the secure variants of FTP-S and HTTP-S, if necessary. The Hyper Text Transfer Protocol (HTTP) will offer the most flexible download going forward since it is routable across firewalls and HTTP is already supported in web servers. Of course, the Internet today runs on HTTP, so aligning the network boot paradigm with this technology ensures that the platform investment will carry forward.

Example of Putting It Together

Below is an extract of a File Transfer Protocol (FTP) utility that is built upon the UEFI TCP support to tie the notion of a connection-oriented protocol like FTP back to the shell.

This program leverages the UEFI, which uses the UEFI Transmission Control Protocol (TCP) to download a file. TCP is a connection-oriented protocol with guaranteed delivery with handshakes. This is in contrast to the Trivial File Transport Protocol

(TFTP), which is based upon Universal Datagram Protocol (UDP). UDP does not have any delivery guarantees.

```
256 #include "miniftp.h"
257 #include "utility.h"
258 #include "script.h"
259 #include "Log.h"
260 #include "Cli.h"
```

Lines 256–260

The header files contain basic support routines for the FTP utility.

```
261 MINIFTP_UI      Ui;
262 BOOLEAN         ToExit      = FALSE;
263 BOOLEAN         IsConnected = FALSE;
264 BOOLEAN         UseScript   = FALSE;
265 BOOLEAN         HasLogin    = FALSE;
266 BOOLEAN         NeedLog    = FALSE;
267 BOOLEAN         AppendLogFile = FALSE; // append log
268                      file if log file is exit
269 EFI_HANDLE      BackupImageHandle;
270 EFI_IP_ADDRESS  FtpServerIp;
271
272 CHAR16          ScriptFileName[256];
273 CHAR16          LogFileName[256];
```

Lines 261–273

These lines declare global variables used by the FTP utility.

```
274
275 VOID
276 PrintUsage (
277     VOID
278 );
279
280
281 EFI_STATUS
282 ParseArgs (
283     VOID
284 );
285
```

Lines 275–285

These lines provide forward declaration of service routines.

```
286 EFI_STATUS
287 EFI API
288 InitializeMiniFtp (
289     IN EFI_HANDLE           ImageHandle,
290     IN EFI_SYSTEM_TABLE     *SystemTable
291 );
```

```
292
293     EFI_GUID          MiniFtpGuid = EFI_MINIFTP_GUID;
294
295     //
296     // Name:
297     //     InitializeMiniFtp -- Entry point
298     // In:
299     //     ImageHandle
300     //     SystemTable
301     // Out:
302     //     EFI_SUCCESS
303     //
304     EFI_BOOTSHELL_CODE(EFI_APPLICATION_ENTRY_POINT(Initialize
305     MiniFtp))
306     EFI_STATUS
307     EFIAPI
308     InitializeMiniFtp (
309         IN EFI_HANDLE           ImageHandle,
310         IN EFI_SYSTEM_TABLE    *SystemTable
311     )
312     /*++
313     Routine Description:
314     Initial FTP shell application
315
316     Arguments:
317     ImageHandle -
318     SystemTable -
319     Returns:
320     --*/
321     {
322         EFI_STATUS           Status;
323         CLI_COMMAND_CONTEXT Context;
324         //
325         // We are now being installed as an internal command
326         driver, initialize
327         // as an nshell app and run
328         //
329         EFI_SHELL_APP_INIT (ImageHandle, SystemTable);
330
331         BackupImageHandle = ImageHandle;
332
333         Status           = ParseArgs ();
334         if (EFI_ERROR (Status)) {
335             PrintUsage ();
336             return Status;
337         }
338         Status = EfiMiniFtpLogInit (LogFileName);
339         if (EFI_ERROR (Status)) {
340             Print (L"Failed to initialize log file.\n\r");
341             goto done;
342         }
343         if (UseScript) {
344             Status = RunScript (ScriptFileName);
```

```

345         goto done;
346     }
347     //
348     //  EnablePageBreak(1, TRUE);
349     //
350     // Main loop
351     //
352     StrCpy (Context.Prompt, DEFAULT_PROMPT);
353     do {
354         GetCommandLine (&Context);
355         ToExit = ProcessCommandLine (&Context);
356         ResetCliContext (&Context);
357     } while (!ToExit);
358     done:
359     EfiMiniFtpLogFinal ();
360
361     if (!UseScript) {
362         //
363         //    Out->ClearScreen (Out);
364         //
365         Out->EnableCursor (Out, TRUE);
366     }
367
368     return Status;
369 }
```

Lines 286–369

These lines contain the code for the main body of the FTP application.

```

370
371     EFI_STATUS
372     ParseArgs (
373         VOID
374     )
375     /**+
376     Routine Description: Parse the input parameter. Miniftp
377     support parameter as
378     follows:
379     Usage           : Miniftp [-f <Script> ] [-l
380     <LogFile>] [-a]
381     <Script>        : The name of script to run.
382     <LogFile>       : The name of log file. ('-a' will
383                         be in effect if choose to write log file )
384     [-a]:          append information in log file, if
385                         the log file exists
386
387     Arguments:
388     Returns:
389         EFI_SUCCESS           -   success
390         EFI_INVALID_PARAMETER -   parameter is error or not
391         supported in MiniFTP
392     --*/
```

```

393     {
394         EFI_STATUS Status;
395         UINTN Index;
396         CHAR16 *Char;
397         BOOLEAN ServerIpConfigured;
398         BOOLEAN IsIPv4;
399
400         ServerIpConfigured = FALSE;
401         Status = EFI_SUCCESS;
402
403         for (Index = 1; Index < SI->Argc; Index++) {
404             if (SI->Argv[Index][0] == '-') {
405                 Char = SI->Argv[Index] + 1;
406                 switch (*Char) {
407                     case 'S':
408                     case 's':
409                         if ((Index == SI->Argc - 1) || (SI->Argv[Index +
410                             1][0] == '-')) {
411                             //
412                             // No value after "-s", Error!
413                             //
414                             return EFI_INVALID_PARAMETER;
415                         }
416
417                         IsIPv4 = FALSE;
418                         Status = StrToInetAddr (SI->Argv[Index + 1],
419                                     &FtpServerIp, &IsIPv4);
420                         if (EFI_ERROR (Status)) {
421                             goto Done;
422                         }
423
424                         ServerIpConfigured = TRUE;
425                         break;
426
427                     case 'f':
428                     case 'F':
429                         if (Index == SI->Argc - 1) {
430                             return EFI_INVALID_PARAMETER;
431                         }
432
433                         UseScript = TRUE;
434                         StrCpy (ScriptFileName, SI->Argv[Index + 1]);
435                         break;
436
437                     case 'l':
438                     case 'L':
439                         if (Index == SI->Argc - 1) {
440                             return EFI_INVALID_PARAMETER;
441                         }
442
443                         NeedLog = TRUE;
444                         StrCpy (LogFileName, SI->Argv[Index + 1]);
445                         break;

```

```

446
447     case 'a':
448     case 'A':
449         if (NeedLog == TRUE) {
450             AppendLogFile = TRUE;
451         }
452         break;
453     default:
454         Status = EFI_INVALID_PARAMETER;
455         goto Done;
456     }
457 }
458 Done:
459     return Status;
460 }
461 }
```

Lines 371–461

These lines contain the argument parsing code.

```

462
463     VOID
464     PrintUsage (
465         VOID
466         )
467     /*++
468
469     Routine Description:
470     --*/
471     {
472         Print (
473             L"Miniftp Client 0.01\n\r" L"Copyright (C) Intel Corp
474             2009. All rights reserved.\n\r" L"\n\r" L"Usage :
475             Miniftp [-f <Script>] [-l <LogFilename>] [-a]\n\r" L"\n"
476             L" <Script> : The name of script to run.\n\r" L"
477             <LogFilename> : The name of log file to create.\n\r"
478             L" [-a] : Append Log file if log file is
479             exist.\n\r"
480         );
481     }
482 }
```

Lines 463–482

These lines contain the help print routine.

The ability to create an FTP utility in UEFI is something that could not be done in BIOS. For one thing, the BIOS networking stack only exposed UDP in the base code, not a TCP interface. Also, BIOS does not have a consistent shell/command-line interface built into the ROM in the same fashion as the ability to integrate the UEFI Shell.

In addition to the use of FTP, the UEFI Shell can integrate the FTP utility as a built-in command, and the integrated FTP+UEFI Shell could be integrated into the ROM, could be put on the UEFI System partition, or could be PXE-booted itself, in order to allow for a rich set of scenarios.

Summary

This chapter has shown how emergent UEFI platform networking capabilities, combined with the UEFI Shell, allow for rich provisioning scenarios. These scenarios allow for flexibility without compromising scale or security of the solution.

Chapter 7

Configuration of Provisioned Material

Innovation distinguishes between a leader and a follower.

—Steve Jobs

Once material gets placed on a target system, one of the inevitable next steps would be the configuration of this material. Since much of the inherent configuration mechanisms that are available in the UEFI firmware environment are also supported in the UEFI Shell environment, this chapter's material will touch on configuration capabilities available in both environments.

The BIOS has never been known for having a great user interface. The ROM sizes were too limited and the video interfaces too unpredictable to support high-end graphical interfaces.

Much of the design that the UEFI configuration infrastructure covers has been heavily studied and analyzed by modern operating systems. The results of these efforts are what have turned into the configuration infrastructure that first was documented in the UEFI 2.1 specification.

Initialization Timeline

During platform initialization, several distinct steps occur. To simplify the timeline, the illustration in Figure 7.1 simply covers four general phases of operation for the system. These phases are intended to illustrate when configuration of the platform is possible. Note that configuration services are available very early in platform initialization.

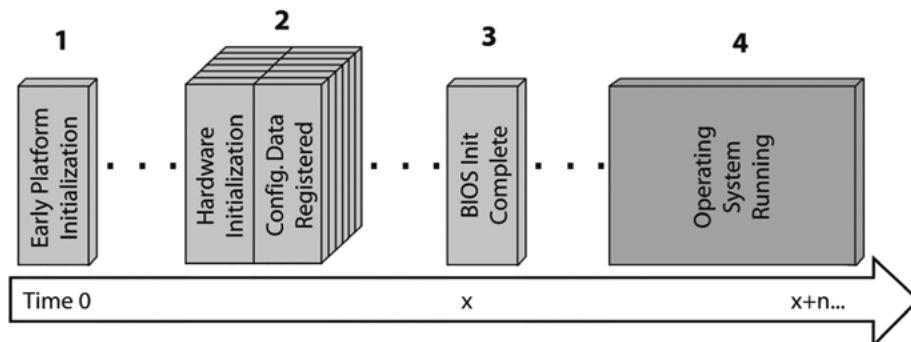


Figure 7.1: The Platform Initialization Timeline

Figure 7.1 gives a simplified view of the phase of operations for a platform:

- *Phase 1.* During early initialization, the UEFI-compatible BIOS initializes some of the underlying components of the platform (such as CPU, chipset, and memory). This also includes establishing the UEFI infrastructure so that the common UEFI components (such as the UEFI configuration infrastructure) are available to be used by later phases of operation.
- *Phase 2.* In the later phase, the BIOS starts to launch drivers (often located in add-in device option ROMs) that have configuration-related material associated with certain configurable devices. This configuration data is registered with the BIOS through something known as the Human Interface Infrastructure (HII) services. Between Phase 2 and 3 is usually when a local user would interact with the platform to configure it (picture a user at a BIOS setup menu).
- *Phase 3.* The BIOS initialization is complete and the BIOS proceeds to launch the boot target (which is usually an operating system for most platforms).
- *Phase 4.* The boot target is launched and running. For most platforms, this is the phase that the machine remains in for most of the time.

It should also be noted that a very common scenario for platform configuration is when a remote administrator interacts with a platform. This can be done with either the assistance of an operating system, as illustrated in Figure 7.2, or with a platform that contains an out-of-band management controller. Since the BIOS is often not interactive while the operating system is running, this poses some issues for updating BIOS-based configuration settings during the later phases of platform operation. Figure 7.2 shows a slightly modified example of the previous timeline, which now enables late configuration setting updates through a mechanism that applies the changes across a system reset, leveraging the underlying UEFI configuration infrastructure.

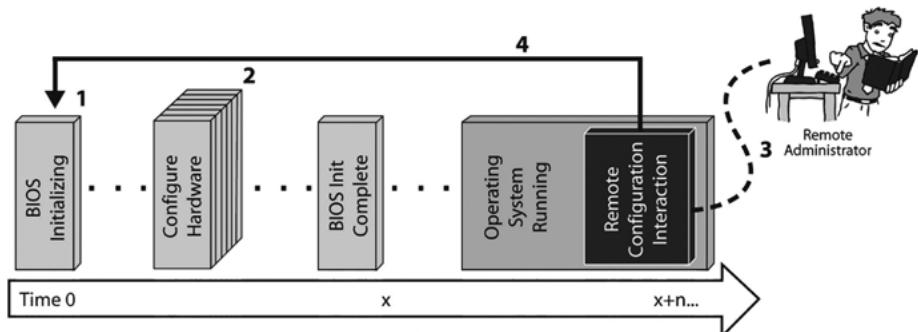


Figure 7.2: Timeline Illustrating How Late Requests for BIOS Setting Changes Can Be Accommodated

- *Step 1.* The UEFI-compliant BIOS initializes like it normally does.
- *Step 2.* The configuration step is usually based on some previously chosen settings that the user/administrator had previously applied to that device. These settings are stored in some nonvolatile location. During this step, the settings are typically read from the nonvolatile storage and the device is configured accordingly.
- *Step 3.* The platform is running the operating system in this illustration and has some interaction with a remote administrator. The remote administrator requests some BIOS setting changes to occur on the platform and some agent proxies this request by a couple of methods:
 - *Capsules.* The UEFI infrastructure supports an `UpdateCapsule()` service, which allows for an OS-present agent to call into the BIOS and communicate some configuration request, which will typically be acted upon across a platform reset. This is a very flexible method of enabling an across-reset update since it can potentially allow for updates of both on-motherboard devices as well as third-party devices (which often use their own proprietary local nonvolatile storage). Platform behavior changes typically do not occur until the platform has reset and the hardware has then been reconfigured based on the desired settings.
 - *EFI Variable.* The UEFI infrastructure provides abstractions to a platform nonvolatile storage service (that is, an EFI variable). This is primarily used by the motherboard devices and the setting requests can be directly established from the OS-present phase of operations. Platform behavior changes typically do not occur until the platform has reset and the hardware has then been reconfigured based on the desired settings.
- *Step 4.* Once the remote configuration update request has been received and acted upon, the platform typically resets so that the boot timeline is restarted. It should be noted, though, that during this subsequent boot, the items that normally occur in Step 2 would still occur, but typically based on the aforementioned configuration updates in the current configuration settings.

Configuration Infrastructure Overview

The modern UEFI configuration infrastructure that was first described in the UEFI 2.1 specification is known as the Human Interface Infrastructure (HII). HII includes the following set of services:

- *Database Services.* A series of UEFI protocols that are intended to be an in-memory repository of specialized databases. These database services are focused on differing types of information:

- *Database Repository*. This is the interface that drivers interact with to manipulate configuration-related contents. It is most often used to register data and update keyboard layout-related information.
- *String Repository*. This is the interface that drivers interact with to manipulate string-based data. It is most often used to extract strings associated with a given token value.
- *Font Repository*. The interface to which drivers may contribute font-related information for the system to use. Otherwise, it is primarily used by the underlying firmware to extract the built-in fonts to render text to the local monitor. Note that since not all platforms have inherent support for rendering fonts locally (think headless platforms), general purpose UI designs should not presume this capability.
- *Image Repository*. The interface to which drivers may contribute image-related information for the system to use. This is for purposes of referencing graphical items as a component of a user interface. Note that since not all platforms have inherent support for rendering images locally (think headless platforms), general purpose UI designs should not presume this capability.
- *Browser Services*. The interface that is provided by the platform's BIOS to interact with the built-in browser. This service's look and feel is implementation-specific, which allows for platform differentiation.
- *Configuration Routing Services*. The interface that manages the movement of configuration data from drivers to target configuration applications. It then serves as the single point to receive configuration information from configuration applications, routing the results to the appropriate drivers.
- *Configuration Access Services*. The interface that is exposed by a driver's configuration handler and is called by the configuration routing services. This service abstracts a driver's configuration settings and also provides a means by which the platform can call the driver to initiate driver-specific operations.

Using the Configuration Infrastructure

The overview introduced the components of the UEFI configuration infrastructure. This section discusses with a bit more detail how one goes about using aspects of this infrastructure. The following steps are initiated by a driver that is concerned with using the configuration infrastructure:

- *Initialize hardware*. The primary job of a device driver is typically to initialize the hardware that it owns. During this process of physically initializing the device, the driver is also responsible for establishing the proper configuration state information for that device. These typically include doing the following:

- *Installing required protocols.* Protocols are interfaces that will be used to communicate with the driver. One of the more pertinent protocols associated with configuration would be the Configuration Access protocol. This is used by the system BIOS and agents in the BIOS to interact with the driver. This is also the mechanism by which a driver can provide an abstraction to a proprietary nonvolatile storage that under normal circumstances would not be usable by anyone other than the driver itself. This is how configuration data can be exposed for add-in devices and others can send configuration update requests without needing direct knowledge of that device.
 - Creating an EFI device path on an EFI handle. A device path is a binary description of the device and typically how it is attached to the system. This provides a unique name for the managed device and will be used by the system to refer to the device later.
- *Register Configuration Content.* One of the latter parts of the driver initialization (once a device path has been established) is the registration of the configuration data with the underlying UEFI-compatible BIOS. The configuration data typically consists of sets of forms and strings that contain sufficient information for the platform to render pages for a user to interact with. It should also be noted that now that the configuration data is encapsulated in a binary format, what was previously an opaque meaningless set of data is now a well-known and exportable set of data that greatly expands the configurability of the device by both local and remote agents as well as BIOS and OS-present components.
 - *Respond to Configuration Event.* Once the initialization and registration functions have completed, the driver could potentially remain dormant until called upon. A driver would most often be called upon to act on a configuration event. A configuration event is an event that occurs when a BIOS component calls upon one of the interfaces that the driver exposed (such as the Configuration Access protocol) and sends the driver a directive. These directives typically would be something akin to “give me your current settings” or “adjust setting X’s value to a 5.”

Driver Model Interactions

The drivers that interact with the UEFI configuration infrastructure are often compliant with the UEFI driver model, as the examples shown in Figure 7.3 and Figure 7.4. Since driver model compliance is very common (and highly recommended) for device drivers, several examples are shown below that describe in detail how such a driver would most effectively leverage the configuration infrastructure.

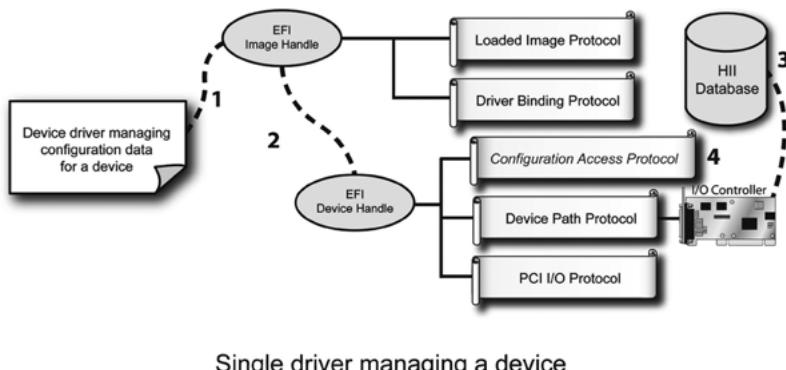


Figure 7.3: A Single Driver that Is Registering Its Configuration Data and Establishing Its Environment in a Recommended Fashion

- Step 1. During driver initialization, install services on the controller handle.
- Step 2. During driver initialization, discover the managed device. Create a device handle and then install various services on it.
- Step 3. During driver initialization, configuration data for the device is registered with the HII database (through the `NewPackageList()` API) using the device's device handle. A unique HII handle is created during the registration event.
- Step 4. During system operation, when a configuration event occurs, the system addresses (through the Configuration Access protocol) the configuration services associated with the device.

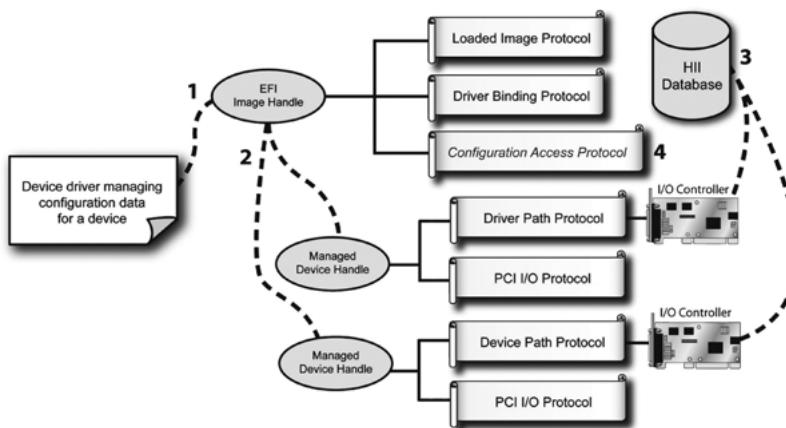


Figure 7.4: A Single Driver that Is Managing Multiple Devices, Registering Its Configuration Data, and Establishing Its Environment in a Recommended Fashion

- Step 1. During driver initialization, install services on the controller handle.
- Step 2. During driver initialization, discover the managed device(s). Create device handle(s) and then install various services on them.
- Step 3. During driver initialization, configuration data for each device is registered with the HII database (through the `NewPackageList()` API) using each device's device handle. A unique HII handle is created during the registration event.
- Step 4. During system operation, when a configuration event occurs, the system addresses (through the Configuration Access protocol) the configuration services associated with the driver. In this example, the configuration services will be required to disambiguate references to each of its managed devices by the passed-in HII handle.

Provisioning the Platform

Figure 7.5 is an illustration that builds on the previously introduced concepts and shows how the remote interaction would introduce the concept of bare-metal provisioning (putting content on a platform without the aid of a formal operating system). This kind of interaction could be used in the manufacturing environment to achieve the provisioning of the platform or in the after-market environment where one is remotely managing the platform and updating it.

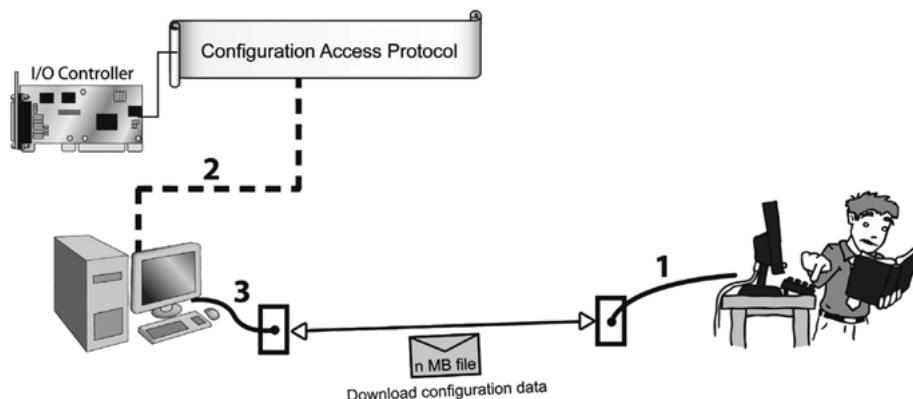


Figure 7.5: Remote Interaction Occurs with a Target System; the System in Turn Accesses the Configuration Abstractions Associated with a Device or Set of Devices

- *Step 1.* Remote administrator sends a query to a target workstation. This query could actually be a component of a broadcast by the administrator to all members of the network.
- *Step 2.* Request received and an agent (possibly a shell-based one) proxies the request to the appropriate device.
- *Step 3.* The agent responds based on interaction with the platform's underlying configuration infrastructure.

Configuring through the UEFI Shell

One of the usage models associated with the UEFI Shell is for the running of programs or scripts within it so that it can automatically execute a variety of tasks and leverage the power of the overall BIOS environment. Some of the built-in commands associated with certain support levels of the UEFI Shell provide both basic and advanced features. These features expose the ability to configure the system as well as query the system for some rather complicated sets of data through scripts. Having this capability in scripting certainly does not preclude the ability for anyone to leverage the full extent of the UEFI BIOS's capabilities through a binary program.

Basic Configuration

Some of the common commands that provide basic interaction with the configuration infrastructure include:

- *drvcfg.* This command invokes the platform's configuration infrastructure. This command is used primarily for the following purposes:
 - To invoke the system's browser through a script so that it displays a given device's setup pages.
 - To provide the ability to set a specific set of default behaviors for a given device.
 - To set a device's configuration from a user-defined group of settings contained in a file.
- *drvdiag.* This command invokes the Driver Diagnostics Protocol. This provides a script the ability to interact with the diagnostics services that a driver may expose in its list of services.

Command-line Usage for *drvcfg*

This section provides a set of examples of common usages for the *drvcfg* command, followed by the appropriate command-line syntax. The full command-line syntax is as follows:

```
drvcfg [-l XXX] [-c] [-f <Type>|-v|-s] [DriverHandle  
[DeviceHandle [ChildHandle]]] [-i filename] [-o filename]
```

To display the list of devices that are available for configuration:

```
Shell> drvcfg
```

To display the list of devices and child devices that are available for configuration:

```
Shell> drvcfg -c
```

To force defaults on all devices:

```
Shell> drvcfg -f 0
```

To force defaults on all devices that are managed by driver 0x17:

```
Shell> drvcfg -f 0 17
```

To force defaults on device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -f 0 17 28
```

To force defaults on all child devices of device 0x28 that are managed by driver 0x17:

```
Shell> drvcfg -f 0 17 28 -c
```

To force defaults on child device 0x30 of device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -f 0 17 28 30
```

To validate options on all devices:

```
Shell> drvcfg -v
```

To validate options on all devices that are managed by driver 0x17:

```
Shell> drvcfg -v 17
```

To validate options on device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -v 17 28
```

To validate options on all child devices of device 0x28 that are managed by driver 0x17:

```
Shell> drvcfg -v 17 28 -c
```

To validate options on child device 0x30 of device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -v 17 28 30
```

To set options on device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -s 17 28
```

To set options on child device 0x30 of device 0x28 that is managed by driver 0x17:

```
Shell> drvcfg -s 17 28 30
```

To set options on device 0x28 that is managed by driver 0x17 in English:

```
Shell> drvcfg -s 17 28 -l eng
```

To set options on device 0x28 that is managed by driver 0x17 in Spanish:

```
Shell> drvcfg -s 17 28 -l spa
```

Command-line Usage for drvdiag

This section provides a set of examples of common usages for the drvdiag command, followed by the appropriate command-line syntax. The full command-line syntax is as follows:

```
drvdiag [-c] [-l XXX] [-s|-e|-m] [DriverHandle  
[DeviceHandle [ChildHandle]]]
```

To display the list of devices that are available for diagnostics:

```
Shell> drvdiag
```

To display the list of devices and child devices that are available for diagnostics:

```
Shell> drvdiag -c
```

To run diagnostics in standard mode on all devices:

```
Shell> drvdiag -s
```

To run diagnostics in standard mode on all devices in English:

```
Shell> drvdiag -s -l eng
```

To run diagnostics in standard mode on all devices in Spanish:

```
Shell> drvdiag -s -l spa
```

To run diagnostics in standard mode on all devices and child devices:

```
Shell> drvdiag -s -c
```

To run diagnostics in extended mode on all devices:

```
Shell> drvdiag -e
```

To run diagnostics in manufacturing mode on all devices:

```
Shell> drvdiag -m
```

To run diagnostics in standard mode on all devices managed by driver 0x17:

```
Shell> drvdiag -s 17
```

To run diagnostics in standard mode on device 0x28 managed by driver 0x17:

```
Shell> drvdiag -s 17 28
```

To run diagnostics in standard mode on all child devices of device 0x28 managed by driver 0x17:

```
Shell> drvdiag -s 17 28 -c
```

To run diagnostics in standard mode on child device 0x30 of device 0x28 managed by driver 0x17:

```
Shell> drvdiag -s 17 28 30
```

Advanced Configuration Abilities

Some of the common commands that provide basic interaction with the configuration infrastructure include:

- *memmap*. This command displays the memory map associated with the UEFI environment.
- *dblk*. This command allows a script to interact with the underlying block (storage) device so that it can display the contents of one or more of its blocks/sectors.
- *dmem*. This command displays the contents of system memory or device memory. If an address is not specified, then the contents of the EFI system table are displayed. Otherwise, memory starting at a particular address is displayed. This is especially useful for displaying the contents of certain memory ranges like a device's PCI configuration space.
- *mm*. This command allows the user to display or modify the I/O register, memory contents, or PCI configuration space.

Command-line Usage for memmap

This section provides an example of the memmap command. The full command-line syntax is as follows:

```
memmap [-b] [-sfo]
```

```
fs0:\> memmap

Type      Start          End            # Pages       Attributes
available 0000000000750000-0000000001841FFF 00000000000010F2 0000000000000009
LoaderCode 00000000001842000-000000000018A3FFF 0000000000000062 0000000000000009
available 000000000018A4000-000000000018C1FFF 0000000000000001E 0000000000000009
LoaderData 000000000018C2000-000000000018CAFFF 00000000000000009 0000000000000009
BS_code    000000000018CB000-00000000001905FFF 0000000000000003B 0000000000000009
BS_data    00000000001906000-000000000019C9FFF 000000000000000C4 0000000000000009
...
RT_data   00000000001B2B000-0000000001B2BFFF 0000000000000001 8000000000000009
BS_data   00000000001B2C000-00000000001B4FFF 0000000000000024 0000000000000009
reserved  00000000001B50000-00000000001D4FFF 00000000000000200 0000000000000009

reserved :      512 Pages (2,097,152)
LoaderCode:     98 Pages (401,408)
LoaderData:     32 Pages (131,072)
BS_code :      335 Pages (1,372,160)
BS_data :      267 Pages (1,093,632)
RT_data :      19 Pages (77,824)
available : 4,369 Pages (17,895,424)
Total Memory: 20 MB (20,971,520) Bytes
```

Command-line Usage for dblk

This section provides a set of examples of common usages for the dblk command, followed by the appropriate command-line syntax. The full command-line syntax is as follows:

```
dblk device [lba] [blocks] [-b]
```

To display one block of blk0, beginning from block 0:

```
Shell>dblck blk0
```

To display one block of fs0, beginning from block 0x2:

```
Shell>dblck fs0 2
```

To display 0x5 blocks of fs0, beginning from block 0x12:

```
Shell>dblck fs0 12 5
```

To display 0x10 blocks of fs0, beginning from block 0x12:

```
Shell>dblck fs0 12 10
```

The attempt to display more than 0x10 blocks will display only 0x10 blocks:

```
Shell>dblck fs0 12 20
```

To display one block of blk2, beginning from the first block (blk0):

```
fs1:\tmpsl> dblk blk2 0 1

LBA 0000000000000000 Size 00000200 bytes BlkIo 3F0CEE78
00000000: EB 3C 90 4D 53 44 4F 53-35 2E 30 00 02 04 08 00 *.<.MSDOS5.0....*
00000010: 02 00 02 00 00 F8 CC 00-3F 00 FF 00 3F 00 00 00 *.....?...?...?...*
00000020: 8E 2F 03 00 80 01 29 2C-09 1B 60 4E 4F 20 4E 41 *./....),...NO NA*
00000030: 4D 45 20 20 20 46 41-54 31 36 20 20 20 33 C9 *ME FAT16 3.*
00000040: 8E D1 BC F0 7B 8E D9 B8-00 20 8E C0 FC BD 00 7C *.....*.....*.
00000050: 38 4E 24 7D 24 8B C1 99-E8 3C 01 72 1C 83 EB 3A *8NS.$....<.r...:*
00000060: 66 A1 1C 7C 26 66 3B 07-26 8A 57 FC 75 06 80 CA *f...&f;.&W.u...*
00000070: 02 88 56 02 80 C3 10 73-EB 33 C9 8A 46 10 98 F7 *..V...s.3..F...*
00000080: 66 16 03 46 1C 13 56 1E-03 46 0E 13 D1 8B 76 11 *f..F..V..F....v./*
00000090: 60 89 46 FC 89 56 FE B8-20 00 F7 E6 8B 5E 0B 03 *..F..V.. ....^..*
000000A0: C3 48 F7 F3 01 46 FC 11-4E FE 61 BF 00 00 E8 E6 *.H...F..N.a....*
000000B0: 00 72 39 26 38 2D 74 17-60 B1 0B BE A1 7D F3 A6 *.r9&8-t..*.....*
000000C0: 61 74 32 4E 74 09 83 C7-20 3B FB 72 E6 EB DC A0 *at2Nt... ;.r....*
000000D0: FB 7D B4 7D 8B F0 AC 98-40 74 0C 48 74 13 B4 0E *.....@t.Ht...*
000000E0: BB 07 00 CD 10 EB EF A0-FD 7D EB E6 A0 FC 7D EB *.....*.....*.
000000F0: E1 CD 16 CD 19 26 8B 55-1A 52 B0 01 BB 00 00 E8 *.....&U.R.....*
00000100: 3B 00 72 E8 5B 8A 56 24-BE 0B 7C 8B FC C7 46 F0 *;.r.[.V$.....F.*.
00000110: 3D 7D C7 46 F4 29 7D 8C-D9 89 4E F2 89 4E F6 C6 *=..F.)....N..N..*.
00000120: 06 96 7D CB EA 03 00 00-20 OF B6 C8 66 8B 46 F8 *.....*.....f.F.*.
00000130: 66 03 46 1C 66 8B DO 66-C1 EA 10 EB 5E 0F B6 C8 *f..F.f..f....^..*
00000140: 4A 4A 8A 46 0D 32 E4 F7-E2 03 46 FC 13 56 FE EB *JJ.F.2....F..V..*
00000150: 4A 52 50 06 53 6A 01 6A-10 91 8B 46 18 96 92 33 *JRP.Sj.j...F..3*
00000160: D2 F7 F6 91 F7 F6 42 87-CA F7 76 1A 8A F2 8A E8 *.....B....v....*
00000170: C0 CC 02 AA CC B8 01 02-80 7E 02 0E 75 04 BA 42 *.....*.....u.B*
00000180: 8B F4 8A 56 24 CD 13 61-61 72 0B 40 75 01 42 03 *...V$..aar.@u.B./*
00000190: 5E 0B 49 75 06 F8 C3 41-BB 00 00 60 66 6A 00 EB *^.Iu...A...`fj..*
000001A0: B0 4E 54 4C 44 52 20 20-20 20 20 20 0D 0A 52 65 *.NTLDR ..Re*
000001B0: 6D 6F 76 65 20 64 69 73-6B 73 20 6F 72 20 6F 74 *move disks or ot*
000001C0: 68 65 72 20 6D 65 64 69-61 2E FF 0D 0A 44 69 73 *her media....Dis*
000001D0: 6B 20 65 72 72 6F 72 FF-0D 0A 50 72 65 73 73 20 *k error...Press *
000001E0: 61 6E 79 20 6B 65 79 20-74 6F 20 72 65 73 74 61 *any key to resta*
000001F0: 72 74 0D 0A 00 00 00 00-00 00 00 AC CB D8 55 AA *rt.....U./*.
```

```
Fat 16 BPB  FatLabel: 'NO NAME'  SystemId: 'FAT16'  OemId: 'MSDOS5.0'
SectorSize 200  SectorsPerCluster 4 ReservedSectors 8 # Fats 2
Root Entries 200  Media F8  Sectors 32F8E  SectorsPerFat CC
SectorsPerTrack 3F Heads 255
```

Command-line Usage for dmem

This section provides example usages of the dmem command. The full command-line syntax is as follows:

```
dmem [-b] [address] [size] [-MMIO]
```

To display the EFI system table pointer entries:

```
fs0:\> dmem
```

```
Memory Address 000000003FF7D808 200 Bytes
3FF7D808: 49 42 49 20 53 59 53 54-02 00 01 00 78 00 00 00 *IBI SYST....x...*
3FF7D818: 5C 3E 6A FE 00 00 00 00-88 2E 1B 3F 00 00 00 00 *\>j.....?....*
3FF7D828: 26 00 0C 00 00 00 00 00-88 D3 1A 3F 00 00 00 00 *&.....?....*
3FF7D838: A8 CE 1A 3F 00 00 00 00-88 F2 1A 3F 00 00 00 00 *...?.....?....*
3FF7D848: 28 EE 1A 3F 00 00 00 00-08 DD 1A 3F 00 00 00 00 *(..?.....?....*
3FF7D858: A8 EB 1A 3F 00 00 00 00-18 C3 3F 3F 00 00 00 00 *...?.....* 
3FF7D868: 00 4B 3F 3F 00 00 00 00-06 00 00 00 00 00 00 00 *.K.....* 
3FF7D878: 08 DA F7 3F 00 00 00 00-70 74 61 6C 88 00 00 00 *...?....ptal....*
3FF7D888: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D898: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8A8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8B8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8C8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8D8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8E8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D8F8: 00 00 00 00 00 00 00 00-70 68 06 30 88 00 00 00 *.....ph.0....*
3FF7D908: 65 76 6E 74 00 00 00 00-02 02 00 60 00 00 00 00 *evnt.....`....*
3FF7D918: 18 6F 1A 3F 00 00 00 00-10 E0 3F 3F 00 00 00 00 *.o.?.....* 
3FF7D928: 10 00 00 00 00 00 00 00-40 C0 12 3F 00 00 00 00 *.....@..?....*
3FF7D938: 10 80 13 3F 00 00 00 00-00 00 00 00 00 00 00 00 *...?.....* 
3FF7D948: 00 00 00 00 00 00 00 00-40 7D 3F 3F 00 00 00 00 *.....@....* 
3FF7D958: 50 6F 1A 3F 00 00 00 00-00 00 00 00 00 00 00 00 *Po.?.....* 
3FF7D968: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D978: 00 00 00 00 00 00 00 00-70 74 61 6C 88 00 00 00 *.....ptal....*
3FF7D988: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D998: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9A8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9B8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9C8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9D8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9E8: 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 *.....* 
3FF7D9F8: 00 00 00 00 00 00 00 00-70 68 06 30 A0 00 00 00 *.....ph.0....*
```

```
Valid EFI Header at Address 000000003FF7D808
```

```
-----  
System: Table Structure size 00000078 revision 00010002  
ConIn (3FIAD388) ConOut (3F1AF288) StdErr (3FIADD08)  
Runtime Services 000000003F3FC318  
Boot Services 000000003F3F4B00  
SAL System Table 000000003FF22760  
ACPI Table 000000003FFD9FC0  
ACPI 2.0 Table 00000000000E2000  
MPS Table 000000003FFD0000  
SMBIOS Table 00000000000F0020
```

To display memory contents from 1af3088 with size of 16 bytes:

```
Shell> dmem 1af3088 16
Memory Address 000000001AF3088 16 Bytes
01AF3088: 49 42 49 20 53 59 53 54-00 00 02 00 18 00 00 00 *IBI SYST.....*
01AF3098: FF 9E D7 9B 00 00 .....*
```

To display memory mapped I/O contents from 1af3088 with size of 16 bytes:

```
Shell> dmem 1af3088 16 -MMIO
```

Command-line Usage for drvdig

This section provides example usages of the drvdig command. The full command-line syntax is as follows:

```
mm address [value] [-w 1|2|4|8] [-MEM | -MMIO | -IO | -PCI | -PCIE]
```

To display or modify memory:

```
Address 0x1b07288, default width=1 byte:
fs0:\> mm 1b07288
MEM 0x0000000001B07288 : 0x6D >
MEM 0x0000000001B07289 : 0x6D >
MEM 0x0000000001B0728A : 0x61 > 80
MEM 0x0000000001B0728B : 0x70 > q

fs0:\> mm 1b07288
MEM 0x0000000001B07288 : 0x6D >
MEM 0x0000000001B07289 : 0x6D >
MEM 0x0000000001B0728A : 0x80 > *Modified
MEM 0x0000000001B0728B : 0x70 > q
```

To modify memory: Address 0x1b07288, width = 2 bytes:

```
Shell> mm 1b07288 -w 2
MEM 0x0000000001B07288 : 0x6D6D >
MEM 0x0000000001B0728A : 0x7061 > 55aa
MEM 0x0000000001B0728C : 0x358C > q

Shell> mm 1b07288 -w 2
MEM 0x0000000001B07288 : 0x6D6D >
MEM 0x0000000001B0728A : 0x55AA > *Modified
MEM 0x0000000001B0728C : 0x358C > q
```

To display I/O space: Address 80h, width = 4 bytes:

```
Shell> mm 80 -w 4 -IO
IO 0x0000000000000080 : 0x000000FE >
IO 0x0000000000000084 : 0x00FF5E6D > q
```

To modify I/O space using non-interactive mode:

```
Shell> mm 80 52 -w 1 -IO
Shell> mm 80 -w 1 -IO
```

```
IO 0x00000000000000080 : 0x52 > FE      *Modified  
IO 0x00000000000000081 : 0xFF >  
IO 0x00000000000000082 : 0x00 >  
IO 0x00000000000000083 : 0x00 >  
IO 0x00000000000000084 : 0x6D >  
IO 0x00000000000000085 : 0x5E >  
IO 0x00000000000000086 : 0xFF >  
IO 0x00000000000000087 : 0x00 > q
```

To display PCI configuration space, ss=00, bb=00, dd=00, ff=00, rr=00:

```
Shell> mm 0000000000 -PCI  
PCI 0x0000000000000000 : 0x86 >  
PCI 0x0000000000000001 : 0x80 >  
PCI 0x0000000000000002 : 0x30 >  
PCI 0x0000000000000003 : 0x11 >  
PCI 0x0000000000000004 : 0x06 >  
PCI 0x0000000000000005 : 0x00 > q
```

To display PCIE configuration space, ss=00, bb=06, dd=00, ff=00, rrr=000:

```
Shell> mm 0006000000 -PCIE  
PCIE 0x000000060000000 : 0xAB >  
PCIE 0x000000060000001 : 0x11 >  
PCIE 0x000000060000002 : 0x61 >  
PCIE 0x000000060000003 : 0x43 >  
PCIE 0x000000060000004 : 0x00 > q
```


Chapter 8

The Use of UEFI for Diagnostics

To err is human, and to blame it on a computer is even more so.

—Robert Orben

This chapter describes some usages of the UEFI Shell for diagnostics. Although the PC ecosystem has rich examples of robust platform software and hardware components, occasionally things go awry. In those cases, the machine state needs to be diagnosed or assessed. To that end, the act of performing diagnostics is a key action for platform deployment and lifecycle maintenance.

Today, disk operating systems such as MS-DOS or PC DOS are still used by many platform manufacturers as a diagnostics environment because of the single-tasking nature of DOS, the large library of extant DOS utilities, the fact that DOS layers directly on PC/AT BIOS as its I/O stack, and the lack of memory protection in DOS. For the purposes of a modern OS, these features of DOS are difficult to use, but for diagnosing a machine or determining the root-cause of a failure, this close mapping to the hardware and controlled environment is appreciated. But DOS has various downsides for diagnostics on contemporary platform hardware, including a limited memory map, 16-bit operating mode, and difficulties in getting modern software ported to this environment.

This description of DOS is not intended to be pejorative. In fact, the existence of DOS coupled with PC/AT BIOS has been a contributing factor to the PC ecosystem success and customer-visible value of Moore’s Law and the associated platform.

The *Beyond DOS* aspect of the book title, though, describes how scenarios like DOS diagnostics now have an opportunity to move to UEFI. Int21h in DOS maps the appropriate UEFI service, for example. In addition, the full machine addressability of UEFI, richness of the UEFI and shell specifications, the ability to in fact access UEFI Platform Initialization (PI) interfaces if they’re available, and open software infrastructure like the EFI Development Kits at www.tianocore.org, are key enabling elements of this migration.

Types of Diagnostics

In the context of a UEFI system, many actors can contribute to the diagnostics role. We mentioned above the available, generic infrastructure that the UEFI Shell and main specifications at www.uefi.org provide, but within those specifications are some purpose-designed abstractions for diagnostics. One example would be the `EFI_DRIVER_DIAGNOSTICS_PROTOCOL`. The intent of a protocol such as this, like other UEFI interfaces, is to bind the API to the entity that can produce the domain-

specific behavior. What we mean by that is the UEFI driver that provides a capability, such as block abstraction from a disk driver, can also provide a diagnostics interface in cases of a failure of the underlying media or hardware.

So why is a device-specific abstraction valuable? This gives a platform manufacturer the opportunity to write a generic “disk diagnostics” capability into a shell application that can access the plurality of disk block instances via each driver’s `EFI_DRIVER_DIAGNOSTICS_PROTOCOL`. Without this per-driver API publication, such a “disk diagnostics” utility would have to contain vendor-specific information and code flows from the sundry disk controller vendors in the industry.

Regarding the usage of the `EFI_DRIVER_DIAGNOSTICS` protocol mentioned above, the UEFI Shell specification codifies usage via the `drvdiag` command.

Another type of diagnostic can be one that accesses the platform resources, such as the PCI bus. To that end, the UEFI Shell has the `mm` and `pci` commands to allow peeking (reading) and poking (writing) memory-mapped I/O, direct I/O, PCI configuration access, and PCI memory-mapped device access, respectively. Like other UEFI Shell commands, these hardware accesses can be done in an interactive mode or via scripting, with console and/or log file recording being possible, too.

The final discussion of a class of diagnostics entails the use of the UEFI Shell to ascertain information from the System Management BIOS tables. This example provides working reference code and is intended to tie together some of the earlier discussions around available software frameworks, infrastructure in both the UEFI main specification and UEFI Shell specifications, and a use-case that provides customer-visible value from using this technology.

The System Management BIOS (SMBIOS) tables are a set of data structures in memory that are referenced by the GUID in the UEFI system table, namely:

```
#define SMBIOS_TABLE_GUID \
{0xeb9d2d31, 0x2d88, 0x11d3, 0x9a, 0x16, 0x0, 0x90, 0x27, 0x3f, 0xc1, 0x4d}
```

The location of the SMBIOS table relative to other UEFI objects is shown in Figure 8.1. The important point to note is the location of the industry standard hand-off tables in the lower left-hand side of the diagram.

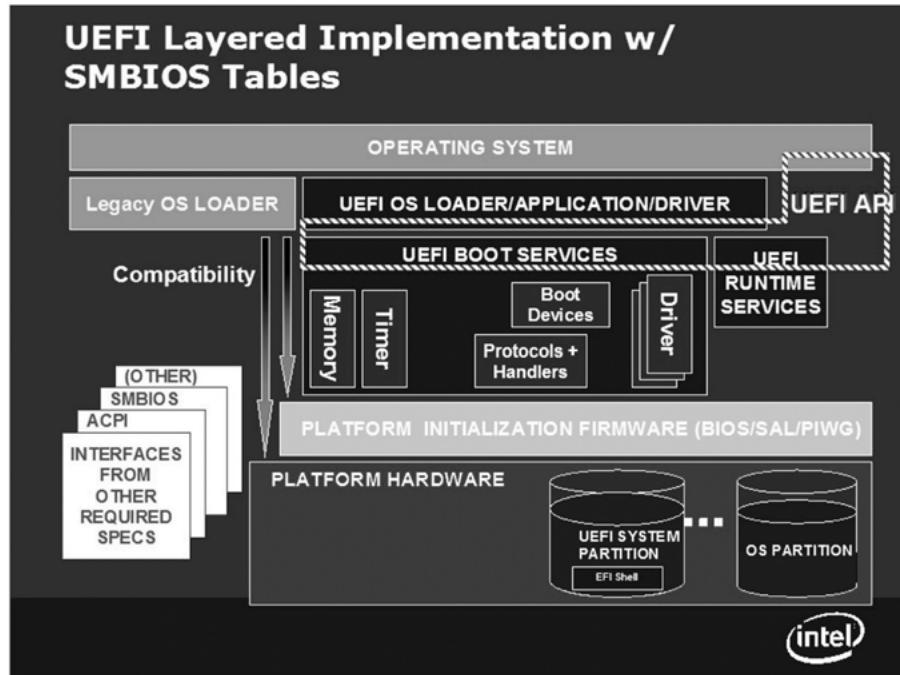


Figure 8.1: System Diagram with UEFI

This chapter describes a scenario wherein the system is not operational and different asset information is discovered using capabilities of the UEFI Shell. Before we describe the tool to ascertain the SMBIOS data, a little background information will be provided.

SMBIOS Table Organization

The purpose of this utility (named SMBIOSVIEW) is to get data from SMBIOS tables and translate the packed information into a human-readable form. As such, the SMBIOS structure table organization is the first issue to design with respect to this diagnostic UEFI Shell-based utility.

According to the SMBIOS specification, there are two access methods defined for the SMBIOS structures. The first method, defined in v2.0 of the SMBIOS specification, provides the SMBIOS structures through a plug-and-play function interface. A table-based method, defined in v2.1 of the SMBIOS specification, provides the SMBIOS structures as a packed list of data referenced by a table entry point.

A BIOS compliant with v2.1 of the SMBIOS specification can provide one or both methods. A BIOS compliant with v2.2 and later of this specification must provide the

table-based method and can optionally provide the plug-and-play function interface. EFI uses the second method.

In EFI, the SMBIOS core driver provides table-based information. SMBIOSVIEW gets the information and translates it to users. The table includes a table header, a structure table, and other data objects. See the SMBIOS specification at <http://www.dmtf.org/standards/smbios/> for more information on the table entries.

SMBIOS Structure Table Entry Point

The information of SMBIOS is organized as the SMBIOS structure, and the SMBIOS structure is accessed by the means of the SMBIOS structure table Entry Point Structure (EPS).

Table Organization Graph

The table organization graph shown in Figure 8.2 is used to make the SMBIOS table more understandable. The SMBIOS table includes a table header and a structure table.

The table header contains the general information of the table and the necessary information to access the structure table.

The structure table contains a series of structures. The type of the last structure is 127, which indicates End-of-table.

The EPS (Entry Point Structure) has information about the structure table:

- *Table-Address* points to the structure table starting address.
- *Table-Length* is the length of the structure table.
- *Num-of-Structures* is the number of structures in the structure table.

The first structure begins with the Table-Address. The second structure begins with the next byte at the end of the first one, and so on.

The type of the last structure is 127. The last structure is also indicated by the Num-of-Structures in the EPS.

The structures between the first and last are of random type. In other words, the structures are packed neither increasing type nor decreasing type, but random.

Each Structure has a common header. The header contains three fields:

- *Type* – type of this structure, following data is organized according to this type
- *Length* – number of bytes of format part, it does *not* include the text string length
- *Handle* – uniquely identifies the structure in the structure table

The format part follows the header. Text strings follow the format part. In text string parts, two bytes of 0x00 identify the end of structure. It also identifies the end of the structure table header.

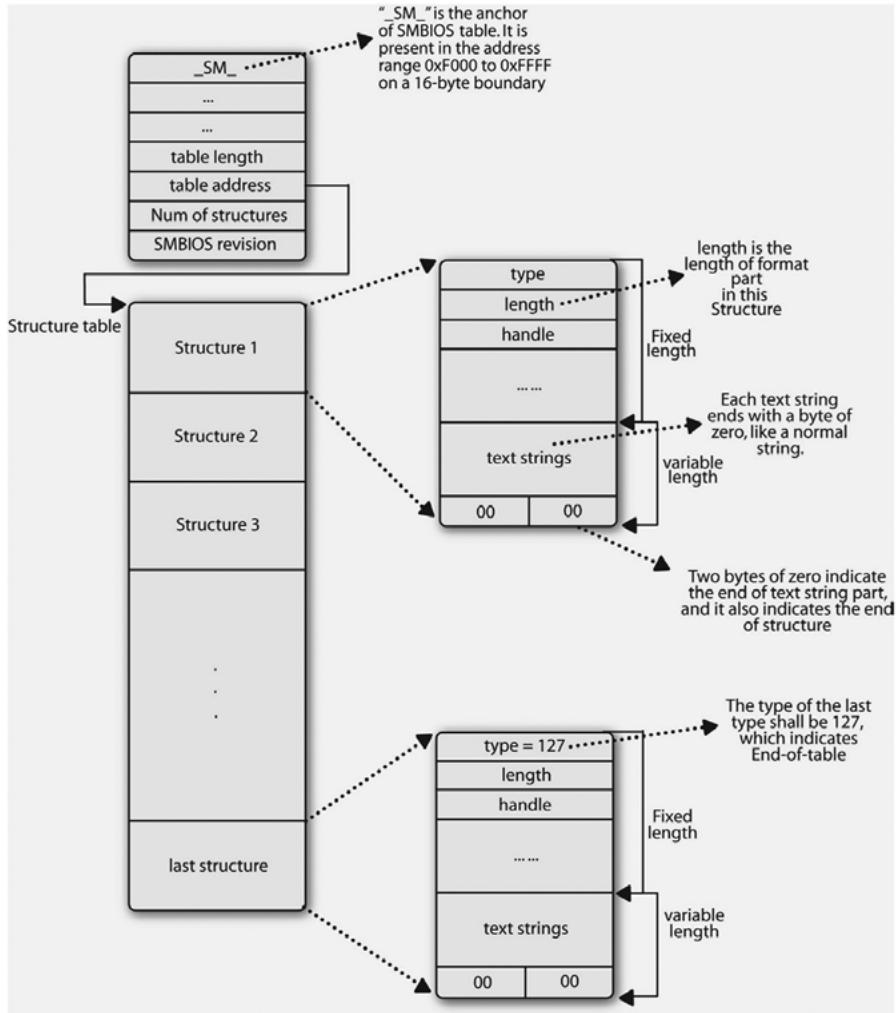


Figure 8.2: SMBIOS Table Organization

Structure Standards

Each SMBIOS structure has a formatted section and an optional unformatted section. The formatted section of each structure begins with a 4-byte header. Remaining data in the formatted section is determined by the structure type, as is the overall length of the formatted section.

Structure Evolution and Usage Guidelines

As the industry evolves, the structures defined in this specification will evolve. To ensure that the evolution occurs in a nondestructive fashion, the following guidelines must be followed:

1. If a new field is added to an existing structure, that field is added at the end of the formatted area of that structure and the structure's *Length* field is increased by the new field's size.
2. Any software that interprets a structure shall use the structure's *Length* field to determine the formatted area size for the structure rather than hard-coding or deriving the *Length* from a structure field.
3. Each structure shall be terminated by a double-null (0x0000), either directly following the formatted area (if no strings are present) or directly following the last string. This includes system- and OEM-specific structures and allows upper-level software to easily traverse the structure table. See below for structure-termination examples.
4. The unformatted section of the structure is used for passing variable data such as text strings, see 3.4.3 *Text Strings* of the SMBIOS specification for more information.
5. When an enumerated field's values are controlled by the DMTF, new values can be used as soon as they are defined by the DMTF without requiring an update to this specification.
6. Starting with v2.3, each SMBIOS structure type has a *minimum* length—enabling the addition of new, but optional, fields to SMBIOS structures. In no case shall a structure's length result in a field being less than fully populated. For example, a Voltage Probe structure with *Length* of 0x15 is invalid since the *Nominal Value* field would not be fully specified.
7. Software that interprets a structure field must verify that the structure's length is sufficient to encompass the optional field; if the length is insufficient, the optional field's value is *Unknown*. For example, if a Voltage Probe structure has a *Length* field of 0x14, the probe's *Nominal Value* is *Unknown*. A Voltage Probe structure with Length greater than 0x14 *always* includes a Nominal Value field.

Text Strings

Text strings associated with a given SMBIOS structure are returned in the *dmiStructBuffer*, appended directly after the formatted portion of the structure. This method of returning string information eliminates the need for application software to deal with pointers embedded in the SMBIOS structure. Each string is terminated with a null (0x00) *UINT8* and the set of strings is terminated with an additional null (0x00) *UINT8*. When the formatted portion of a SMBIOS structure references a string,

it does so by specifying a nonzero string number within the structure's string-set. For example, if a string field contains 0x02, it references the second string following the formatted portion of the SMBIOS structure. If a string field references no string, a null (0) is placed in that string field. If the formatted portion of the structure contains string-reference fields and all the string fields are set to 0 (no string references), the formatted section of the structure is followed by two null (0x00) BYTES. See *3.4.1 Structure Evolution and Usage Guidelines* on page 90 of the SMBIOS specification for a string-containing example.

Note: Each text string is limited to 64 significant characters due to system MIF limitations.

Required Structures and Data

Beginning with SMBIOS v2.3, compliant SMBIOS implementations include a base set of required structures and data within those structures. These structures include the BIOS information, system information, processor information, and several tables describing the system information.

Features

SMBIOSVIEW allows users to display SMBIOS structure information with different detail options. Its main goal is to provide a user-friendly interface of the SMBIOS structure. SMBIOSVIEW allows users to:

- Display structure table statistics information
- Display structure information with different levels:
 - SHOW_NONE – Don't interpret just dump the structure
 - SHOW_OUTLINE – Only display header information
 - SHOW_NORMAL – Display header information and element value
 - SHOW_DETAIL – Display header and element detail information (default)
- Display all structures' information of certain Type
 - Display structures' information of certain Handle
 - Display structures' information one by one or all at once
 - Controls (such as change display option) in application
 - Use a simple help guide (>SmbiosView -?).

User Interface Design

This section describes the details of the user interface (UI) to the SMBIOSVIEW shell command.

Design Guide

1. Command -line arguments determine what action the SMBIOS View tool should perform.
 2. In SMBIOSVIEW, change the options to determine how to display the respective SMBIOS table contents.
1. SmbiosView [-t type] | [-h handle] | [-s] | [-a]
- t - View structures of certain type
 - h - View structure of certain handle
 - s - View statistics of whole SMBIOS table
 - a - View all structures one at a time

2. Internal commands:

- :q - Quit SmbiosView
- :0 - SmbiosView display NONE info
- :1 - SmbiosView display OUTLINE info
- :2 - SmbiosView display NORMAL info
- :3 - SmbiosView display DETAIL info
- /? - Show help

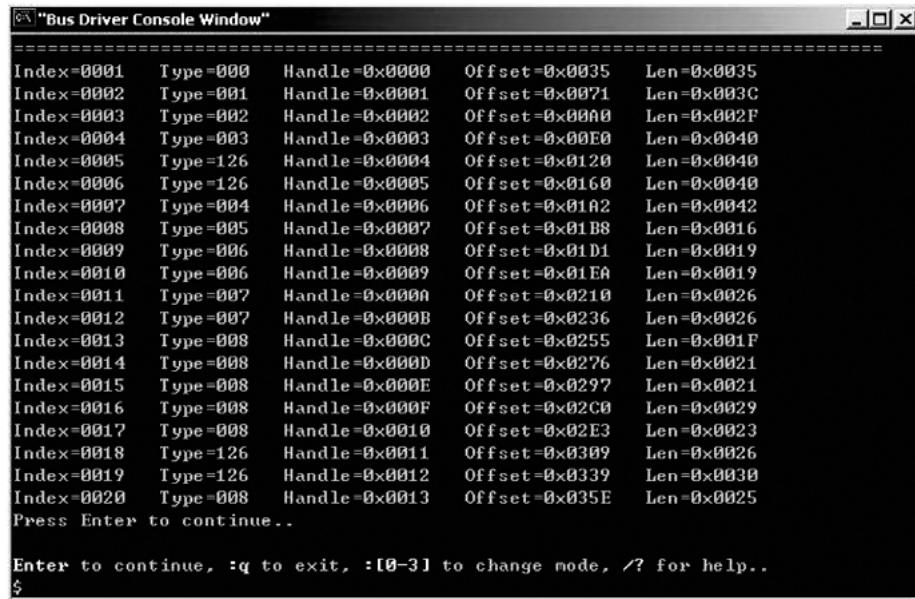
Note: Internal commands provide optional controls to users and they are gotten from users' input after a prompt '\$'. Users can also press Enter to skip internal commands. The following command allows for describing the various portions of the SMBIOS table. The various actions that can occur with respect to the table manipulation are encoded via various input command line parameters. These options include a description of the options via '-?.'

Usage

```
>SmbiosView -?           - Show help page
>SmbiosView             - Show structures as default
>SmbiosView -s          - Show statistics information, as shown in Figure 8.3
>SmbiosView -t 8         - Show all structures of type=8, as shown in Figure 8.4
>SmbiosView -h 25        - Show structure of handle=0x25
>SmbiosView -a > 1.log   - Show all structures and output to file of 1.log
```

Examples

```
fs0:\>SmbiosView -s
```



The screenshot shows a window titled "Bus Driver Console Window". The content of the window is a table of SMBIOS structure statistics. The columns are: Index, Type, Handle, Offset, and Len. The data is as follows:

Index	Type	Handle	Offset	Len
0001	000	0x0000	0x0035	0x0035
0002	001	0x0001	0x0071	0x003C
0003	002	0x0002	0x00A0	0x002F
0004	003	0x0003	0x00E0	0x0040
0005	126	0x0004	0x0120	0x0040
0006	126	0x0005	0x0160	0x0040
0007	004	0x0006	0x01A2	0x0042
0008	005	0x0007	0x01B8	0x0016
0009	006	0x0008	0x01D1	0x0019
0010	006	0x0009	0x01EA	0x0019
0011	007	0x000A	0x0210	0x0026
0012	007	0x000B	0x0236	0x0026
0013	008	0x000C	0x0255	0x001F
0014	008	0x000D	0x0276	0x0021
0015	008	0x000E	0x0297	0x0021
0016	008	0x000F	0x02C0	0x0029
0017	008	0x0010	0x02E3	0x0023
0018	126	0x0011	0x0309	0x0026
0019	126	0x0012	0x0339	0x0030
0020	008	0x0013	0x035E	0x0025

Press Enter to continue..

Enter to continue, :q to exit, :!0-3] to change mode, /? for help..

Figure 8.3: SmbiosView Statistics

```
fs0:\>SmbiosView -t 8
```

```

"Bus Driver Console Window"
fs0:\> smbiosview -t 8
SMBIOS Entry Point Structure:
Smbios BCD Revision: 0x23
Number of Structures: 45
Max Struct size: 85
Table Address: 0x16AA027
Table Length: 1566
Anchor String: _SM_
EPS Checksum: 0x29
Entry Point Length: 31
Major version: 2
Minor version: 3
Entry Point revision: 0x0
revision value:
Inter Anchor: _DMI_
Inter Checksum: 0xD6
=====
Query Structure, conditions are:
QueryType = 8
QueryHandle = Random
ShowType = SHOW_DETAIL

Enter to continue, :q to exit, :[0-3] to change mode, /? for help..
$
```

Figure 8.4: SmbiosView User Interface

Architecture Design

The SMBIOS utility components architecture is illustrated in Figure 8.5 (the arrows indicate the calling of another module), and can be described as follows:

- **Init Module.** Gets the SMBIOS table and initializes the environment of the SMBIOS utility.
- **Dispatch Module.** Gets and transacts the shell command parameters and user input.
- **User Input.** The user inputs the internal commands such as changing a display option or quitting the program.
- **SMBIOS Info Access Module.** Provides a set of APIs to access the SMBIOS table or structures.
- **Element Info Interpret Module.** Translates packed data to understandable text according to specification.
- **Data Unpack Module.** Translates packed data to understandable information.
- **Display Module.** Displays information as required options.

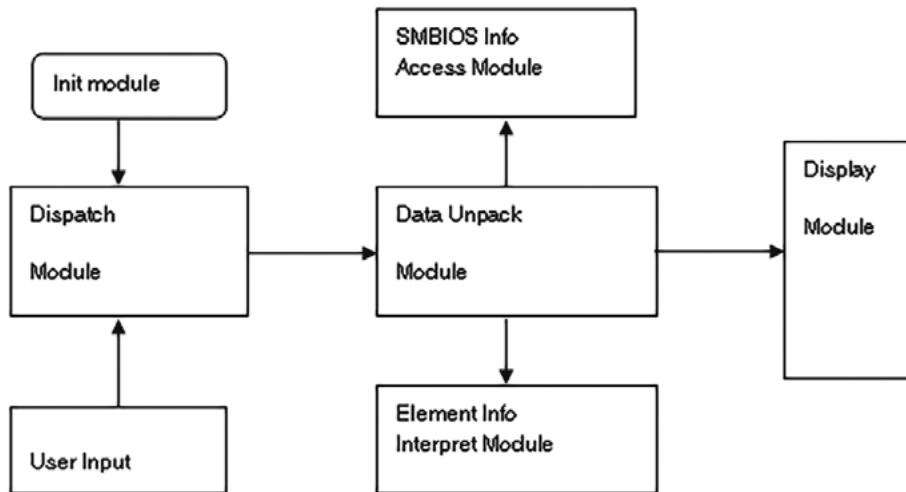


Figure 8.5: Smbios Utility Components Architecture

Data Structure

There are four key data structures in the editor implementation, as listed in Table 8.1.

Table 8.1: Key Data Structures

Data Structure Name	Header File Name
SMBIOS_STRUCTURE_TABLE	LibSmbios.h
SMBIOS_HEADER	LibSmbios.h
SMBIOS_STRUCTURE_POINTER	LibSmbios.h
STRUCTURE_STATISTICS	SmbiosView.h

SMBIOS_STRUCTURE_TABLE

The structure of the SMBIOS_STRUCTURE_TABLE is as follows:

```
#pragma pack(1)
typedef struct {
    UINT8    AnchorString[4];
    UINT8    EntryPointStructureChecksum;
    UINT8    EntryPointLength;
    UINT8    MajorVersion;
```

```
UINT8    MinorVersion;
UINT16   MaxStructureSize;
UINT8    EntryPointRevision;
UINT8    FormattedArea[5];
UINT8    IntermediateAnchorString[5];
UINT8    IntermediateChecksum;
UINT16   TableLength;
UINT32   TableAddress;
UINT16   NumberOfSmbiosStructures;
UINT8    SmbiosBcdRevision;
} SMBIOS_STRUCTURE_TABLE;
#pragma pack()
```

Descriptions

This structure is defined as the EPS (Entry Point Structure) of the SMBIOS table. Access to SMBIOS table information is by this structure. For detailed information, refer to Chapter 3.

Note: Because SMBIOS table uses a byte alignment data structure, this structure is also using #pragma pack(1) to configure structure data alignment of one byte. This is necessary because a general C declaration would be naturally aligned, but the present utility needs to map the data structure to an external specification.

Members

Table 8.2 lists the members of the SMBIOS_STRUCTURE_TABLE data structure.

These tables are going to be manipulated by the utility. The description below explains the specific entries and their meanings.

Table 8.2: Members of SMBIOS_STRUCTURE_TABLE

Member	Description
AnchorString	_SM_, specified as (5F 53 4D 5F)
EntryPointStructureChecksum	Checksum of the Entry Point Structure (EPS)
EntryPointLength	Length of the Entry Point Structure
MajorVersion	Identifies the major version of SMBIOS specification implemented in the table structures.
MinorVersion	Identifies the minor version of SMBIOS specification implemented in the table structures.
MaxStructureSize	Size of the largest SMBIOS structure, in bytes, encompasses the structure's formatted area and text strings.
EntryPointRevision	Identifies the EPS revision implemented in this structure and identifies the formatting of offsets 0Bh to 0Fh.
FormattedArea [5]	The value present in the Entry Point Revision field defines the interpretation to be placed upon these 5 bytes.
IntermediateAnchorString [5]	_DMI_, specified as five ASCII characters (5F 44 4D 49 5F).
IntermediateChecksum	Checksum of Intermediate Entry Point Structure (IEPS)
TableLength	Total length of SMBIOS Structure Table, pointed to by the Structure Table Address, in bytes.
TableAddress	Total length of SMBIOS Structure Table, pointed to by the Structure Table Address, in bytes.
NumberOfSmbiosStructures	Total number of structures present in the SMBIOS Structure Table.
SmbiosBcdRevision	Indicates compliance with a revision of SMBIOS specification.

SMBIOS_HEADER

The structure of SMBIOS_HEADER is as follows:

```
#pragma pack(1)
typedef struct {
    UINT8    Type;
    UINT8    Length;
    UINT16   Handle;
} SMBIOS_HEADER;
#pragma pack()
```

Members

Table 8.3 lists the members of SMBIOS_HEADER.

Table 8.3: Members of SMBIOS_HEADER

Member	Description
Type	Structure type
Length	Format part length of structure
Handle	Unique identifier of structure in structure table

SMBIOS_STRUCTURE_POINTER

The structure of SMBIOS_STRUCTURE_POINTER is as follows:

```
typedef union {
    SMBIOS_HEADER      *Hdr;
    SMBIOS_TYPE0       *Type0;
    SMBIOS_TYPE1       *Type1;
    SMBIOS_TYPE2       *Type2;
    SMBIOS_TYPE3       *Type3;
    SMBIOS_TYPE4       *Type4;
    SMBIOS_TYPE5       *Type5;
    SMBIOS_TYPE6       *Type6;
    SMBIOS_TYPE7       *Type7;
    SMBIOS_TYPE8       *Type8;
    SMBIOS_TYPE9       *Type9;
    SMBIOS_TYPE10      *Type10;
    SMBIOS_TYPE11      *Type11;
    SMBIOS_TYPE12      *Type12;
    SMBIOS_TYPE13      *Type13;
    SMBIOS_TYPE14      *Type14;
    SMBIOS_TYPE15      *Type15;
    SMBIOS_TYPE16      *Type16;
    SMBIOS_TYPE17      *Type17;
    SMBIOS_TYPE18      *Type18;
    SMBIOS_TYPE19      *Type19;
    SMBIOS_TYPE20      *Type20;
    SMBIOS_TYPE21      *Type21;
    SMBIOS_TYPE22      *Type22;
    SMBIOS_TYPE23      *Type23;
    SMBIOS_TYPE24      *Type24;
    SMBIOS_TYPE25      *Type25;
    SMBIOS_TYPE26      *Type26;
    SMBIOS_TYPE27      *Type27;
    SMBIOS_TYPE28      *Type28;
    SMBIOS_TYPE29      *Type29;
    SMBIOS_TYPE30      *Type30;
```

```

SMBIOS_TYPE31    *Type31;
SMBIOS_TYPE32    *Type32;
SMBIOS_TYPE33    *Type33;
SMBIOS_TYPE34    *Type34;
SMBIOS_TYPE35    *Type35;
SMBIOS_TYPE36    *Type36;
SMBIOS_TYPE37    *Type37;
SMBIOS_TYPE38    *Type38;
SMBIOS_TYPE39    *Type39;
SMBIOS_TYPE126   *Type126;
SMBIOS_TYPE127   *Type127;
UINT8            *Raw;
} SMBIOS_STRUCTURE_POINTER;

```

Descriptions

This structure is defined as a union. Each field in the union is a method of organizing the data of the structure, such as structure header, a structure type, or simply a byte raw array.

Members

Table 8.4 lists the members of SMBIOS_STRUCTURE_POINTER.

Table 8.4: Members of SMBIOS_STRUCTURE_POINTER

Members	Description
Hdr	Points to Structure header, common part of all structure types
Type (n)	Interprets the structure as certain structure type format
Raw	Interprets the Structure as simple bytes packet

STRUCTURE_STATISTICS

The structure of STRUCTURE_STATISTICS is as follows:

```

typedef struct {
    UINT16    Index;
    UINT8     Type;
    UINT16    Handle;
    UINT16    Addr;
    UINT16    Len;
} STRUCTURE_STATISTICS;

```

Members

Table 8.5 lists the members of STRUCTURE_STATISTICS.

Table 8.5: Members of STRUCTURE_STATISTICS

Member	Description
Index	Index in the SMBIOS structure table
Type	Structure type identified in the structure header
Handle	Structure handle unique to identified structure in the table
Addr	Structure offset from structure table start address
Len	Structure whole length including format part and text format

Source Code for the Utility

To bring the SMBIOS overview and design discussions together, the SMBIOS view command is described next. Various portions of this UEFI Shell utility are presented and decomposed in order to show how the theory of SMBIOS can be married to a particular UEFI practice.

```

1   #include "EfiShellLib.h"
2   #include "LlibSmbios.h"
3   #include "LibSmbiosView.h"
4   #include "smbiosview.h"
5   #include "smbios.h"
6

```

Lines 1–6

These lines contain the include files for the application.

```

7   STATIC UINT8           mInit      = 0;
8   STATIC SMBIOS_STRUCTURE_TABLE *mSmbiosTable = NULL;
9   STATIC SMBIOS_STRUCTURE_POINTER m_SmbiosStruct;
10  STATIC SMBIOS_STRUCTURE_POINTER *mSmbiosStruct =
11                           &m_SmbiosStruct;

```

Lines 7–11

These lines contain the module globals for the application.

```

12  EFI_STATUS
13  LibSmbiosInit (
14      VOID
15  )
16  /*++

```

```

17 Routine Description:
18     Init the SMBIOS VIEW API's environment.
19 Arguments:
20     None
21 Returns:
22     EFI_SUCCESS      - Successful to init the SMBIOS
23     VIEW Lib\
24     Others          - Cannot get SMBIOS Table
25 --*/
26 {
27     EFI_STATUS Status;
28
29     //
30     // Init only once
31     //
32     if (mInit == 1) {
33         return EFI_SUCCESS;
34     }
35     //
36     // Get SMBIOS table from System Configure table
37     //
38     Status = LibGetSystemConfigurationTable
39             (&gEfiSmbiosTableGuid, &mSmbiosTable);
40
41     if (mSmbiosTable == NULL) {
42         PrintToken (STRING_TOKEN
43                     (STR_SMBIOSVIEW_LIBSMBIOSVIEW_CANNOT_GET_TABLE),
44                     HiiHandle);
45
46         return EFI_NOT_FOUND;
47     }
48
49     if (EFI_ERROR (Status)) {
50         PrintToken (STRING_TOKEN
51                     (STR_SMBIOSVIEW_LIBSMBIOSVIEW_GET_TABLE_ERROR),
52                     HiiHandle, Status);
53         return Status;
54     }
55     //
56     // Init SMBIOS structure table address
57     //
58     mSmbiosStruct->Raw = (UINT8 *) (UINTN) (mSmbiosTable-
59                                     >TableAddress);
60
61     mInit           = 1;
62     return EFI_SUCCESS;
63 }
64

```

Lines 12–64

These lines contain an initialization routine for the application, including logic to discover the SMBIOS data in memory.

```

65      VOID
66      LibSmbiosGetEPS (
67          SMBIOS_STRUCTURE_TABLE **pEntryPointStructure
68      )
69  {
70      //
71      // return SMBIOS Table address
72      //
73      *pEntryPointStructure = mSmbiosTable;
74  }

```

Lines 65–74

These lines contain code to discover the entry point structure.

```

75      VOID
76      LibSmbiosGetStructHead (
77          SMBIOS_STRUCTURE_POINTER *pHead
78      )
79  {
80      //
81      // return SMBIOS structure table address
82      //
83      pHead = mSmbiosStruct;
84  }
85

```

Lines 75–85

These lines contain code to discover the head structure.

```

86      EFI_STATUS
87      LibGetSmbiosInfo (
88          OUT CHAR8    *dmiBIOSRevision,
89          OUT UINT16   *NumStructures,
90          OUT UINT16   *StructureSize,
91          OUT UINT32   *dmiStorageBase,
92          OUT UINT16   *dmiStorageSize
93      )
94  /* ++
95  Routine Description:
96      Get SMBIOS Information.
97
98  Arguments:
99      dmiBIOSRevision - Revision of the SMBIOS
100         Extensions.
101      NumStructures - Max. Number of Structures the
102         BIOS will return.
103      StructureSize - Size of largest SMBIOS Structure.
104      dmiStorageBase - 32-bit physical base address for
105         memory mapped SMBIOS data.
106      dmiStorageSize - Size of the memory-mapped SMBIOS
107         data.
108

```

```

109     Returns:
110         DMI_SUCCESS           - successful.
111         DMI_FUNCTION_NOT_SUPPORTED - Does not support SMBIOS
112 calling interface capability.
113     --*/
114 {
115     //
116     // If no SMIBOS table, unsupported.
117     //
118     if (mSmbiosTable == NULL) {
119         return DMI_FUNCTION_NOT_SUPPORTED;
120     }
121
122     *dmiBIOSRevision = mSmbiosTable->SmbiosBcdRevision;
123     *NumStructures = mSmbiosTable
124                     ->NumberOfSmbiosStructures;
125     *StructureSize = mSmbiosTable->MaxStructureSize;
126     *dmiStorageBase = mSmbiosTable->TableAddress;
127     *dmiStorageSize = mSmbiosTable->TableLength;
128
129     return DMI_SUCCESS;
130 }
```

Lines 86–130

These lines contain code to set various fields of the SMBIOS information.

```

131
132 EFI_STATUS
133 LibGetSmbiosStructure (
134     IN OUT UINT16    *Handle,
135     IN OUT UINT8    *Buffer,
136     OUT UINT16      *Length
137 )
138 /*++
139     Routine Description:
140     Get SMBIOS structure given the Handle, copy data to
141     the Buffer, Handle is changed to the next handle or
142     0xFFFF when the end is reached or the handle is not
143     found.
144
145     Arguments:
146         Handle: - 0xFFFF: get the first structure
147                  - Others: get a structure according to this
148                  value.
149         Buffer: - The pointer to the caller's memory
150                  buffer.
151         Length: - Length of return buffer in bytes.
152     Returns:
153         DMI_SUCCESS - Buffer contains the required
154                     structure data
155                     - Handle is updated with next structure
156                     handle or
157                     0xFFFF(end-of-list).
```

```

158
159         DMI_INVALID_HANDLE - Buffer not contain the
160                               requiring structure data
161     --*/
162 {
163     SMBIOS_STRUCTURE_POINTER Smbios;
164     SMBIOS_STRUCTURE_POINTER SmbiosEnd;
165     UINT8                  *Raw;
166
167     if (*Handle == INVALID_HANDLE) {
168         *Handle = mSmbiosStruct->Hdr->Handle;
169         return DMI_INVALID_HANDLE;
170     }
171
172     if (Buffer == NULL) {
173         PrintToken (STRING_TOKEN
174 (STR_SMBIOSVIEW_LIBSMBIOSVIEW_NO_BUFF_SPEC), HiiHandle);
175         return DMI_INVALID_HANDLE;
176     }
177     *Length      = 0;
178     Smbios.Hdr    = mSmbiosStruct->Hdr;
179     SmbiosEnd.Raw = Smbios.Raw + mSmbiosTable->TableLength;
180     while (Smbios.Raw < SmbiosEnd.Raw) {
181         if (Smbios.Hdr->Handle == *Handle) {
182             Raw = Smbios.Raw;
183             //
184             // Walk to next structure
185             //
186             LibGetSmbiosString (&Smbios, (UINT16) (-1));
187             //
188             //
189             // Length = Next structure head - this structure
190             // head
191             //
192             *Length = (UINT16) (Smbios.Raw - Raw);
193             CopyMem (Buffer, Raw, *Length);
194             //
195             // update with the next structure handle.
196             //
197             if (Smbios.Raw < SmbiosEnd.Raw) {
198                 *Handle = Smbios.Hdr->Handle;
199             } else {
200                 *Handle = INVALID_HANDLE;
201             }
202             return DMI_SUCCESS;
203         }
204         //
205         // Walk to next structure
206         //
207         LibGetSmbiosString (&Smbios, (UINT16) (-1));
208     }
209     *Handle = INVALID_HANDLE;

```

```
211     return DMI_INVALID_HANDLE;  
212 }
```

Lines 131–212

These lines contain code to discover specific SMBIOS structures.

Summary

This chapter has described how the UEFI Shell can be used for an important action in the platform development and deployment space, namely diagnostics. The discussion has included a discussion of using the UEFI Shell for SMBIOS. Specifically, an extant industry standard, such as SMBIOS, can be comprehended by UEFI and then assessed/managed via a UEFI Shell application. Again, this is just one instance where the UEFI Shell, with its rich application library, can assist in real-world platform deployment and management activities.

Chapter 9

UEFI Shell Scripting

I honestly have no strategy whatsoever. I'm waiting for that script to pop through the letterbox and completely surprise me.

—Ben Kingsley

UEFI Shell scripts are interpreted programs (usually with the extension .nsh) written in a text-based language supported directly by the UEFI Shell. Similar to many shell scripts (most notably the Windows command prompt), it also includes features unique to the pre-OS environment, such as standardized command output and redirection to and from environment variables.

The UEFI Shell searches for shell scripts first in the current directory and then in the directories specified by the path environment variable. Shell scripts are carriage-return delimited lists of shell commands that are executed (by default) from first to last. Shell scripts also support several additional commands that change the flow of control in a script or control the output:

- echo – Outputs text to the standard output device
- exit – Terminates the currently executing script
- for...endfor – Repeatedly executes a block of script commands
- goto – Continues execution with the specified label
- if...else...endif – Conditionally executes a block of script commands
- shift – Shifts positional command-line parameters

These are described in detail in Appendix B.

Using shell scripts, complex tasks can be performed simply. The following sections take on successively more difficult tasks using scripts and explain how they work, line by line:

- HelloWorld.nsh – The simplest script outputs “Hello, World” to the screen
- Echo1.nsh – Echoes 3 shell parameters to the screen
- Echo2.nsh – Echoes all shell parameters to the screen
- Echo3.nsh – Echoes all shell parameters to the screen with a count
- Concat1.nsh – Creates a new text file by joining together the contents of one or more user-specified text files
- Lsgrep.nsh – Asks the user for which file information (from the ls command) to output and then outputs just that information to the screen
- InstallCmd.nsh – Example script file that installs a new shell command and updates all of the necessary environment variables

After these examples, we will demonstrate how to create a UEFI boot option that invokes a shell script.

Hello, World!

The simplest script is shown in Figure 9.1.

```
1      @echo "Hello, World!"
```

Figure 9.1: HelloWorld.nsh

Line 1

The echo command copies the remainder of the line to standard output. The “@” prevents the script line itself from being displayed.

If you run this script, you will see:

```
Shell> HelloWorld
Hello, World!
```

Echo

This script simply echoes the first three arguments to the screen, as shown in Figure 9.2.

```
1      @echo First : %1
2      @echo Second: %2
3      @echo Third : %3
```

Figure 9.2: Echo1.nsh

Lines 1–3

This will print out the first three command-line parameters. The first command-line parameter is %1, the second is %2, all the way up to (theoretically) %9. What happens if there are more than 9? Well, that is the subject of our next script.

If you run this script using *abra*, *cadab*, and *ra* as your parameters, you will see:

```
Shell> Echo1 abra cadab ra
First : abra
Second: cadab
Third : ra
```

Normally, any whitespace character will separate one command-line parameter from the next. However, quotation marks can be used to break the rules. For example:

```
Shell> Echo1 "abra cadab" ra1 ra2
First : "abra cadab"
Second: ra1
Third : ra2
```

The quotation marks caused the space between abra cadab to be ignored. Also, the quotation marks were retained, rather than being discarded. Now try:

```
Shell> Echo1 ^"abra cadab^" ra
First : "abra
Second: cadab"
Third : ra
```

The “^” caret character forces the next character to be treated as a normal character.

Echo All Parameters

This script, shown in Figure 9.3, echoes all of the command-line parameters, no matter how many there are.

```
1      @echo -off
2      :start
3      if %1 == ""  then
4          goto Done
5      endif
6      echo Parameter: %1
7      shift
8      goto start
9      :Done
```

Figure 9.3: Echo2.nsh

Line 2

The word *start* is a label that can be referenced later in a *goto* command.

Line 3

Check to see whether or not the command-line parameter is empty. The only way a command-line parameter can be empty is if it is not present. So this is really a check to see whether there are any more command-line parameters. If it is the last parameter, then execution starts after the *endif*.

Line 6

If the parameter was present (see line 1), then display it.

Line 7

The `shift` command moves all of the command-line parameters over by one. So, `%2` is moved to `%1`, `%3` to `%2`, and so on. If the command line contains more than 9 parameters, the tenth will go to `%9`.

Line 8

The `goto` command causes execution to continue with the label specified after the command. In this case, execution continues after line 1.

If you execute this command, you might see:

```
Shell> Echo2
Shell> Echo2 abc
Parameter: abc
Shell> Echo2 1 2 3 4 5 6 7 8 9 20
Parameter: 1
Parameter: 2
Parameter: 3
Parameter: 4
Parameter: 5
Parameter: 6
Parameter: 7
Parameter: 8
Parameter: 9
Parameter: 20
```

Echo All Parameters (Improved Version)

The script shown in Figure 9.4 is an enhanced version of the previous script. Instead of just printing the static text “Parameter” before the text of each parameter, it actually prints “Parameter 1,” “Parameter 2,” and so on. Most of it is identical to the previous version, but it takes advantage of the `for...endfor` command’s ability to iterate through a series of integers.

```
1      #
2      # echo3.nsh
3      #
4
5      @echo -off
6      for %a run (1 100)
7          if "%1" == "" then
8              exit /b
9          endif
10         @echo Parameter %a: %1
11         shift
12     endfor
```

Figure 9.4: Echo3.nsh

Line 6

The `for...endfor` script command repeatedly executes a block of script commands until the list of possible index values has been exhausted. The block of script commands on lines 6–10 would execute 100 times.

Acting as an index variable, `%a` will be updated each time through the loop. There are 26 possible index variable names (`a` through `z`).

The keyword `run` indicates that the index variable will be initialized with the first value (1) during the first time through the loop and that it will be incremented by 1 each time through the loop. The loop will terminate when the index variable's value exceeds the second value (100). An optional third value indicates the amount to increment or decrement the index variable. If it is not present, then it is assumed to be 1 if the initial value is less than the ending value or -1 if the initial value is greater than the ending value.

Instead of `run`, the keyword `in` can be used to step through a list of space-delimited strings. This is demonstrated in the next example.

Line 7

This line checks to see if the next command-line parameter is blank. The command-line parameter can only be blank if there are no more command-line parameters present.

Line 8

The `exit` command, when used with the `/b` parameter, terminates the processing of the current script. The exit code may also be specified, but, if not present, 0 is assumed.

Line 10

As in the previous example, the parameter is output. But, in this example, `%a` is added. When the script is executed, the `%a` will be replaced with the actual contents of the index variable created in line 5.

If you execute this command, you might see:

```
Shell> Echo3
Shell> Echo3 abc
Parameter 1: abc
Shell> Echo3 1 2 3 4 5 6 7 8 9 20
Parameter 1: 1
Parameter 2: 2
Parameter 3: 3
Parameter 4: 4
Parameter 5: 5
Parameter 6: 6
Parameter 7: 7
Parameter 8: 8
```

```
Parameter 9: 9
Parameter 10: 20
```

Concatenate Text Files

This script, shown in Figure 9.5 creates a new text file that consists of the contents of zero or more other text files. The syntax is:

```
concat1 output-file [input-file1...]
```

Just to make things interesting, we will allow any of the input files to contain wildcard characters.

```
1      #
2      # concat1.nsh
3      #
4
5      @echo -off
6      if %1 == "" then
7          @echo Error: missing output file name
8          @exit /b 1
9      endif
10
11     set -v outfile %1
12
13     :nextparm
14     if not %2 == "" then
15         for %a in %2
16             type %a >> %outfile%
17         endfor
18         shift
19         goto nextparm
20     endif
```

Figure 9.5: Echo3.nsh

Lines 6–9

Check to see whether the output file name is specified. If not, an error message is generated and the script exits with the return code of “1”. If the /b were not specified, then the entire instance of the shell would be exited.

Line 11

Saves the output file name to a volatile environment variable. The name needs to be saved because, after the `shift` statement on line 18, it would be lost. If the `-v` were missing, the value of ‘outputfile’ would be saved across system reset or system power-cycle.

Lines 13–14, 19

The label ‘`nextparm`’, along with the `if` and the `goto` create a loop that exits only when there are no more command-line parameters after the first.

Lines 15–17

The `for...endfor` commands create a loop, with one iteration for each file name that matches the pattern specified by `%2`. Inside the loop, `%a` is initialized with the actual file name. The `type` command outputs the contents to standard output. The `>>` re-directs standard output so that it is appended to the specified file “`outputfile`.”

Line 18

This moves all the command-line parameters by one so that `%2` becomes the next input file name (if any).

List Only Selected “ls” Information

This script, shown in Figure 9.6, accepts the same syntax as “`ls`,” but it prompts the user to specify information about the files to display. Using `echo` and the `getkey` command from the previous chapter, it asks the user to select one of seven pieces of information about the file:

1. Full Name
2. Logical Size
3. Physical Size
4. Attributes
5. File Access Date
6. File Creation Date
7. File Modification Date

Then it pipes the *standard-format output* of the `ls` command to the `parse` command to extract the desired field. Many of the UEFI Shell commands have a special mode (`-sfo`) where the output is formatted in a well-defined, easy to parse format. Except for this special mode, the output of UEFI Shell commands is not standardized.

Standard format output consists of rows of information, with each row taking up one line. Each row is divided into columns by a comma. The first column contains an

identifier that describes the type of information that appears in the other columns. All columns except the first are quoted.

The `ls` command produces three different row types: `ShellCommand`, `VolumeInfo`, and `FileInfo`. The *ShellCommand* row type must be the first row produced by any shell command that produces standard-format output. The *VolumeInfo* row type describes such information as the volume name and how much free space is available. The *FileInfo* row type describes an individual file, with each column giving the information listed above.

The `parse` command can go through a file and extract any single column from standard-format output from a specified row type and display it.

```

1      #
2      # lsgrep.nsh
3      #
4
5      @echo 1) Full Name
6      @echo 2) Logical Size
7      @echo 3) Physical Size
8      @echo 4) Attributes
9      @echo 5) File Access Date
10     @echo 6) File Creation Date
11     @echo 7) File Modification Date
12     @echo
13     :wait
14     getkey "Select The File Data To Display: " _key
15
16     if %_key% lt 1 or %_key% gt 7 then
17         @echo %_key% is not between 1 and 7
18         goto wait
19     endif
20
21     math %_key% + 1 >v _key
22     @ls "%1" -sfo | parse FileInfo %_key%

```

Figure 9.6: Lsgrep.nsh

Lines 5–14

Display all of the possible fields on the screen for the user to select from and then wait for a key. The resulting key press is stored in the environment variable `_key`.

Lines 16–19

Check to see whether the user input is valid. If not, display an error message and go back to wait again for a key.

Line 21

Add one to the value entered by the user. The standard-format output for the ls command puts the full name in column 2, the logical size in column 3, and so on. The resulting value goes into the _key environment variable again.

Line 22

List all of the files, using the same command-line options passed to the script itself. Up to 9 command-line options can be passed without doing some additional work. Even if 9 weren't specified, the remaining options will be blanks. The output is redirected to a text file.

The parse command searches through the specified file for lines that begin with the tag “FileInfo,” extracts the column number specified by %_key%, and displays it.

Install Script

The script shown in Figure 9.7 acts as an installation script for the GetKey sample application from the previous chapter. It demonstrates how to:

1. Detect whether or not the UEFI Shell supports the features that are required to support the GetKey sample application.
2. Detect errors during installation and display error messages.
3. Install the executable and help files to the correct directory.
4. Update the path to point to the target directory.
5. Create a new UEFI Shell profile called “_shellbook”

The script has the following syntax:

`InstallCmd command-name target-directory`

where `command-name` is the name of the shell command (such as `GetKey`) and `target-directory` is the directory where the command should be installed.

```

1      #
2      # InstallCmd.nsh - Install a new UEFI Shell Command
3      #
4      # InstallCmd command-name target-directory
5      #
6      #
7      #
8      # Validate the UEFI Shell support level
9      #
10     #
11     if %shellsupport% ult 3 then
12         exit /b 2
13     endif
14     #
15     #

```

```
16      # Make sure that the command isn't already installed.  
17      #  
18      if exists(%2/%1.efi) then  
19          @echo %1.efi already exists at %2.  
20          exit /b 1  
21      endif  
22  
23      if available(%1.efi) then  
24          @echo %1.efi already exists in the path.  
25          exit /b 1  
26      endif  
27  
28      #  
29      # Create the target directory  
30      #  
31  
32      md %2  
33      if not exists(%2) then  
34          @echo Could not create target directory %2  
35          exit /b 1  
36      endif  
37  
38      #  
39      # Copy the executable and help file to the target directory  
40      #  
41  
42      cp -q %1.efi %2  
43      if not %lasterror == 0 then  
44          @echo Could not copy %1.efi to the target directory.  
45          exit /b 1  
46      endif  
47  
48      cp -q %1.man %2  
49      if not %lasterror == 0 then  
50          @echo Could not copy %1.man to the target directory.  
51          del %2/%1.efi  
52          exit /b 1  
53      endif  
54  
55      #  
56      # Create the profile _shellbook, if it doesn't exist  
57      #  
58  
59      if not profile(_shellbook) then  
60          set profiles "%profiles";_shellbook  
61      endif  
62  
63      #  
64      # If necessary, add the target directory to the path  
65      #  
66  
67      if not available(%1.efi)  
68          set path "%path%;%2"  
69      endif  
70
```

Figure 9.7: InstallCmd.nsh

Lines 11–13

These lines check the `%shellsupport%` environment variable to determine which UEFI Shell features are present. If the support level is too low, then the script exits with error code 2. No error message is displayed, since shell support levels less than 3 have no standard output support.

The UEFI Shell is very configurable and many of the features, APIs, and shell commands described in the UEFI Shell specification may or may not be present. However, the various standard support levels can be detected by a UEFI Shell script by examining `shellsupport`. The various valid values are described in Chapter 3 of the UEFI Shell specification. They are briefly summarized in Table 9.1.

Table 9.1: UEFI Shell Support Levels

Level	Name	Execute0/ Scripting/ startup.nsh	Interac- tive?	Commands
0	Minimal	No	No	None
1	Scripting	Yes	No	for, endfor, goto, if, else, endif, shift, exit
2	Basic	Yes	No	attrib, cd, cp, date ^{*1} , time*, del, load, ls, map, mkdir, mv, rm, reset, set, timezone*
3	Interactive	Yes	Yes	alias, date, echo, help, pause, time, touch, type, ver, cls, timezone

Level 0 (Minimal) isn't very interesting in the context of this chapter, since even scripting is not supported. Level 1 (Scripting) adds the basic scripting support described in this chapter, except for echo. Level 2 (Basic) adds basic file handling, date/time, and environment variable commands, which do not rely on standard input or standard output. Level 3 (Interactive) adds those commands that rely on standard input and/or standard output.

Lines 19–22

This section checks to see whether or not the specified shell command already exists in the target directory. The `exists` operator returns nonzero if the specified file does not exist in the specified directory (or in the current working directory, if no directory

^{*1} Some commands are listed for both support level 2 and 3, but the level 3 version adds output. For example, the date and time commands add the ability to show the current date and time.

is specified). If the file already exists, then the script exits with an error message and an error code of 1.

Lines 24–27

This section checks to see whether the specified shell command already exists in the current path. If the current shell command was installed and there was already a shell command in the path, it is confusing to the user, since they aren't sure which would be executed. The `available` operator returns nonzero if the specified file does not exist in the current directory or in any directory listed in the `path` environment variable. If the file already exists, then the script exits with an error message and an error code of 1.

Lines 33–37

This section creates the target directory using the `mc` (a built-in alias of `mkdir`) command. The `cp` command used in later sections of this script will fail if the target directory does not already exist, so it must be created first. The script does not check the error code to see whether there has been an error, but rather checks to see if the directory exists after trying to create it. Directories can be checked in the same way as files, using the `exists` operator.

Lines 43–47

This section copies the executable from the current directory to the target directory. It then checks to see whether there has been an error by examining the `%lasterror%` environment variable. Any nonzero value indicates one of the errors listed in Appendix C of the UEFI Shell Specification. If there has been an error, a message is displayed and the script is terminated with an error code.

Lines 49–54

This section copies the help file from the current directory to the target directory. The UEFI Shell automatically searches for help files in the same directory as the executable. If an error has occurred when copying, an error message is displayed and the script is terminated with an error code.

Lines 60–62

This section installs our custom UEFI Shell profile, with the name “`_shellbook`”. UEFI Shell profiles are collections of related UEFI Shell applications. Each UEFI Shell profile has a unique name. Standard profiles are described in Chapter 5 of the UEFI Shell Specification. Custom profile names must begin with an underscore (“`_`”) character.

The list of currently installed shell profiles is stored in the nonvolatile environment variable with the name `profiles`, separated by a semicolon (“`;`”) character. A profile name should not be added to `profiles` unless all of the commands are present, since other scripts may rely on the information.

The `profiles` operator is a Boolean operator that can be used to detect whether a specific profile is installed in the UEFI Shell. This section first checks to see whether the `_shellbook` profile exists and then, if not, adds it to the `profiles` environment variable using the `set` shell command.

The example given here assumes there is only one shell command in the profile. If there were more than one shell command in the profile, then there should be additional “`if available`” script commands to ensure that all other shell commands in the profile were present before adding the profile name.

Lines 68–70

This section makes sure that the target directory is in the path, which is a list of directories that the shell searches for executables and scripts. The path is stored in the `path` environment variable. Since the script has already copied the script executable to the target directory, the `available` operator should return nonzero if the target directory is already in the path. If it returns zero, then the path environment variable is updated with the target directory.

How to Make a Shell Script Appear as a Boot Option

UEFI Shell scripts and shell executables can be launched directly from the boot manager. The UEFI boot manager relies on the contents of several EFI variables to determine the boot order. The `bcfg` command can add new boot options easily. The UEFI Shell itself is a standard UEFI application that can take command-line arguments that are encoded as part of the boot option. First, let’s create a small script, shown in Figure 9.8, that just prints a message and then waits for a key to be pressed.

```

1      #
2      # HelloWorld2.nsh
3      #
4
5      @echo "Hello, World, Again!"
6      @pause

```

Figure 9.8: HelloWorld2.nsh

Now, let’s add the `HelloWorld2.nsh` script so that it will be launched as a boot option.

```
bcfg boot addp 1 shell.efi "Hello World Script" -opt
"helloWorld2.nsh"
```

This adds a new boot option that will show up as “Hello World Script.” When selected, it will launch the UEFI Shell, display “Hello, World, Again!,” wait for a keypress, and then exit back to the UEFI boot manager.

Chapter 10

UEFI Shell Programming

This chapter provides an overview of the techniques used when creating UEFI Shell applications. The UEFI Shell provides additional capabilities beyond those available to normal UEFI applications. It does this by using two additional protocols. First, the `EFI_SHELL_PARAMETERS_PROTOCOL` protocol is installed on the application’s image handle. This protocol provides access to the command-line parameters, as well as the handle of the standard input, standard output, and standard error logical devices. Second, the `EFI_SHELL_PROTOCOL` provides access to the file system, the environment variables and the device mappings.

If you go and look through the ShellPkg in EDK2 to find examples of shell application source code, you will be disappointed. The shell command source code is there, but they are not applications. Instead, all of the shell commands in the ShellPkg are built into the shell executable directly, using statically linked libraries. They can’t be delivered as standalone executables. There are a few examples in the AppPkg, but the best ones there are generally ports of other open source projects.

In this chapter, however, the full source code of five UEFI Shell applications is presented, along with detailed notes. These notes are designed to help understand what the applications are doing and how the shell resources are being used.

For information on setting up the build environment, see “Setting Up the Build Environment” later in this chapter.

A Simple UEFI Shell Application: HelloWorld

To create the simplest of UEFI Shell applications requires a:

- *New C (.c) source file.* This is the file processed by the compiler
- *New component information (.inf) file.* This describes how to build a single driver or application
- *Modified build description (.dsc) file.* This integrates the component into the overall build process

The Source File: HelloWorld.c

The simplest pure UEFI Shell application requires only a few lines, as show in Figure 10.1:

```

1  /*
2   | Basic "Hello World" application using only UEFI Shell services.
3   |
4   |
5  */
6  #include <Uefi.h>
7  #include <Library/UefiLib.h>
8
9
10 INTN
11 EFIAPI
12 ShellAppMain (
13     IN UINTN             Argc,
14     IN CHAR16           **Argv
15 )
16 {
17     Print(L"Hello World!\n");
18
19     return 0;
20 }
21

```

Figure 10.1: Simple UEFI Shell Application Source Code

Lines 6–9

The header file Uefi.h provides all of the fundamental UEFI type definitions. The library header files UefiLib describes wrappers around standard UEFI boot and runtime services.

Lines 10–15

This is the main function for a pure UEFI application. The UEFI Shell will pass in the number of arguments (Argc) and a pointer to an array of pointers to each of the arguments. The arguments are of type CHAR16 (not char or CHAR8), indicating that they are Unicode, not ASCII.

Line 17

`Print()` is a printf-like function that outputs the string to the standard console device.

Line 19

Returning `SHELL_SUCCESS (0)` indicates that there was no error. A non-zero value indicates an error.

The Component Information (.inf) File

The component information (.inf) file shown in Figure 10.2 describes the component and the information necessary to build a single application, driver, or library. This includes source files, build options, libraries, etc.

```

1 ## @file
2 # Hello World - written as a basic shell application.
3 #
4 #
5 ##
6
7 [Defines]
8   INF_VERSION           = 0x00010006
9   BASE_NAME              = HelloWorld
10  FILE_GUID               = 25214b0d-6bed-4ff1-8594-343962bb8208
11  MODULE_TYPE             = UEFI_APPLICATION
12  VERSION_STRING          = 0.1
13  ENTRY_POINT             = ShellCEEntryLib
14
15 [Sources]
16   HelloWorld.c
17
18 [Packages]
19   MdePkg/MdePkg.dec
20   ShellPkg/ShellPkg.dec
21
22 [LibraryClasses]
23   UefiLib
24   ShellCEEntryLib

```

Figure 10.2: Simple UEFI Shell Application Component Information File

Lines 7–13

The [defines] section lists various attributes of the component being built.

The INF_VERSION (line 8) attribute specifies the revision level of the file format. In this case, it is 1.6.

The BASE_NAME (line 9) attribute gives the name of the output UEFI shell application executable.

The FILE_GUID (line 10) specifies a GUID (globally unique identifier) associated with the UEFI shell application. This GUID must be unique within the entire build system described by the build description (.dsc) file.

The MODULE_TYPE (line 11) attribute specifies what type of component being built. In this case, it is a UEFI application. This is the same module type for all UEFI applications.

The `VERSION_STRING` (line 12) attribute is a simple version number that is built into the application's executable image.

The `ENTRY_POINT` (line 13) attribute specifies the name of the entry point for the application. The entry point for your application depends on the entry point library that the application is linked against. In this case, `ShellCEEntryLib` requires this entry point.

Lines 15–16

The `[Sources]` section lists all of the source files in this component. In this case, it is a single C file: `HelloWorld.c`.

Lines 18–20

The `[Packages]` section lists the relative path of the package declaration (.dec) file of all code packages on which this component depends. The path is relative to the `WORKSPACE` environment variable, which is the root of the build tree. More on this later. In this case, the `MdePkg` contains the declarations for all fundamental UEFI headers, libraries, and types. The `ShellPkg` contains the declarations for all UEFI Shell headers, libraries, and types. The declaration files advertise what the packages can provide to the build: GUIDs, protocols, include directories, libraries, etc.

Lines 22–24

The `[LibraryClasses]` section lists all of the library types on which this component is dependent. A library class defines a type of library. Each library class has a header file, but there may actually be several different implementations of that library, for different operating environments or even different behaviors. In this case, we are using two basic library classes for our application: `UefiLib`, which provides wrappers for basic UEFI services and `ShellCEEntryLib`, which provides the wrapper around this type of UEFI Shell application.

A Simple Standard Application: HelloWorld2

The next example shows how to create a standard C application. This UEFI Shell application does the same thing as the first example, except it uses the standard C library that is available with EDK2. This standard C library provides nearly all of the Posix-defined functions, from `strcpy` to `malloc` to `fopen`. More importantly, it also provides the standard C entry point.

The Source File: HelloWorld2.c

Figure 10.3 shows how it's done:

```

1  /**
2   * Basic "Hello World" application using the standard C library.
3   */
4
5
6  #include <stdio.h>
7
8  /**
9   * Basic "Hello World" C application entry point.
10 */
11
12
13  int
14  main (int argc, char **argv)
15 {
16     printf("Hello World!\n");
17
18     return 0;
19 }
```

Figure 10.3: Hello World Application Using Standard C Library

Line 6

The first noticeable difference is that the only `#include` used in this application is the `<stdio.h>` header file. So there is no reference to any of the UEFI functions. To use the UEFI functions, those header files must be included separately.

Line 13–14

The second noticeable difference is that the entry point function `main` looks just like the one for a standard C application (because it is!). While both this application and the previous example have the number of command-line parameters and pointers to each parameter's text, this version provides those parameters in ASCII instead of Unicode.

Line 16

The final noticeable difference is that this example application uses the standard C function `printf`. It is very similar to the `Print()` function provided by the UEFI library, but some of the format specifiers are slightly different.

The Component Information (.inf) File: HelloWorld2.inf

Figure 10.4 shows the component information (.inf) file for this application. There are a few crucial differences for a standard C UEFI Shell application.

```

1  ##
2  # Component Information File for "HelloWorld2"
3  ##
4
5  [Defines]
6      INF_VERSION          = 0x00010006
7      BASE_NAME             = HelloWorld2
8      FILE_GUID              = c3ef3c0b-22a4-40eb-be5d-9daaeb2ca146
9      MODULE_TYPE            = UEFI_APPLICATION
10     VERSION_STRING         = 0.1
11     ENTRY_POINT           = ShellCEEntryLib
12
13 [Sources]
14     HelloWorld2.c
15
16 [Packages]
17     StdLib/StdLib.dec
18     MdePkg/MdePkg.dec
19     ShellPkg/ShellPkg.dec
20
21 [LibraryClasses]
22     LibC
23     LibStdio

```

Figure 10.4: The HelloWorld2.inf Component Information File.

Line 17

The first major difference is a new package in the [Packages] section. The StdLib/StdLib.dec file provides all necessary attributes for the source code package containing the standard C library.

Lines 22–23

The second major difference is that the LibraryClasses only specify the LibC and LibStdio library classes. These are the static libraries that provide all of the UEFI Shell implementation details for the standard C library.

Read Keyboard Input in UEFI Shell Scripts: GetKey

Now let's try creating a UEFI Shell application that does something useful! GetKey optionally prints out a message, waits for a key to be pressed, converts that key into a human-readable text string and then outputs that text to the screen or to an environment variable. For example, “Ctrl+Alt+F1” or “j” or “Shift-PgUp”.

UEFI Shell scripts can use the output to ask for file names, IP addresses, or just wait for the user to be ready. In this sense, it an advanced version of the built-in pause UEFI Shell command. An example of how to use this application in a script is shown in Chapter 9, “UEFI Shell Scripting.”

The command has the syntax:

```
GetKey [prompt] [env-var-name]
GetKey -?
```

prompt is what will be displayed to the user and *env-var-name* is the name of the environment variable store the resulting key text in. If an error is returned, then an error occurred or the user pressed Ctrl-C.

The Source File: GetKey.c

This UEFI Shell application starts with the global declarations from GetKey.c showing in Figure 10.5.

```
1  /**
2   * Put text version of a keystroke into an environment variable.
3
4 */
5
6 #include <Uefi.h>
7 #include <Library/UefiLib.h>
8 #include <Library/UefiBootServicesTableLib.h>
9 #include <Library/ShellCEEntryLib.h>
10 #include <Protocol/EfiShell.h>
11 #include <Protocol/SimpleTextInEx.h>
12
13 BOOLEAN gHelpUsage;
14
15 CONST CHAR16 *gPrompt;
16 CONST CHAR16 *gEnvVar;
17
18 extern EFI_SHELL_PROTOCOL *gEfiShellProtocol;
```

Figure 10.5: Global Declarations from GetKey.c.

Lines 6–11

These include all of the global UEFI definitions (line 6), library definitions (7-10), and UEFI protocol definitions (10-11). Only a few of these are new:

UefiBootServicesTableLib.h defines a single global data variable gBS, which points to the UEFI Boot Services table.

EfiShell.h defines the Shell protocol (EFI_SHELL_PROTOCOL) that gives access to the UEFI Shell's resources, including environment variables, file services, and help text support.

SimpleTextEx.h defines the Simple Text Input protocols (EFI_SIMPLE_TEXT_INPUT_PROTOCOL and EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL) that provide the APIs for reading keystrokes from the input devices. The instance provided by the UEFI Shell extends this to take input from redirected files (<file-name>) and redirected environment variables (<env-var-name>).

Line 13

This global variable indicates whether the command-line specified display of help text (TRUE) or not (FALSE).

Line 15

This global variable indicates the prompt that will be displayed to the user before waiting for the input key. If this is NULL, then no prompt will be displayed.

Line 16

This global variable indicates the name of the UEFI Shell environment variable that the resulting converted key text will be stored in, or NULL if the key text will be displayed to standard output.

Line 18

This global variable points to the Shell protocol instance, which is used to access all of the Shell protocol services.

Now that the global variables have been declared, we can look at the GetKey entry point, shown in Figure 10.6.

```

179   INTN
180   EFIAPI
181   ShellAppMain (
182     IN UINTN          Argc,
183     IN CHAR16         **Argv
184   )
185 {
186   SHELL_STATUS      Ss;
187   EFI_KEY_DATA     Key;
188   CONST CHAR16     *KeyStr;
189
190   Ss = ParseCommandLine(Argc, Argv);
191   if (Ss != SHELL_SUCCESS || gHelpUsage) {
192     return Ss;
193   }
194
195   if (gPrompt != NULL) {
196     Print(gPrompt);
197   }
198
199   Ss = ReadKey(&Key);
200   if (Ss == SHELL_SUCCESS) {
201     KeyStr = ConvertKeyToText(&Key);
202     if (gEnvVar == NULL) {
203       Print(KeyStr);
204       Print(L"\n");
205     } else {
206       Ss = gEfiShellProtocol->SetEnv(gEnvVar, KeyStr, TRUE);
207     }
208   }
209   Print(L"\n");
210   return Ss;
211 }
```

Figure 10.6: Entry Point in GetKey.c.

Lines 179–184

This application uses the same entry point style as HelloWorld.

Lines 186

This local variable holds the return status for this UEFI application. Notice that it uses the type SHELL_STATUS. This is different from the type EFI_STATUS used in UEFI drivers. The main difference is that SHELL_STATUS values (such as SHELL_OUT_OF_RESOURCES) do not have the most significant bit set.

Line 187

This local variable holds the key that is read from the user. The type `EFI_KEY_DATA` is a standard UEFI data type, described in Chapter 11 of the UEFI Specification. It holds the keyboard scan code and the keyboard shift state. More on this later.

Line 188

This local variable is a string that holds the key text.

Line 190–192

The function `ParseCommandLine()` grabs all of the command-line arguments and puts them into the global variables `gHelpUsage`, `gPrompt` and `gEnvVar`. If there was an error or help was displayed, just exit.

Line 195–197

If a prompt was specified as the first command-line parameter, then print it.

Line 199–201

This section reads the key using `ReadKey()` and, if there was no error, converts it to the key text using the function `ConvertKeyToText()`. `ReadKey()` might generate an error if, for example, the user pressed Ctrl-C during script execution.

Line 202–207

If an environment variable name was specified as the second command-line parameter, then create it or change its value to the string returned from `ConvertKeyToText()`. It does this using the Shell protocol function `SetEnv()`, which takes the environment variable name, the new environment variable value, and a Boolean that specifies whether the environment variable should be volatile or non-volatile. Non-volatile environment variables persist across system reset, while volatile ones do not.

If no environment variable name was specified, then the key text is displayed via standard output.

Line 209–210

Exit from the application with the status.

The next section, shown in Figure 10.7, handles the simple command-line processing for this application. Walking through each command-line option, it examines them and sets global variables accordingly.

```

55  SHELL_STATUS
56  ParseCommandLine(
57      IN UINTN          Argc,
58      IN CONST CHAR16   **Argv
59  )
60  {
61      UINTN          Arg;
62
63      gPrompt = NULL;
64      gEnvVar = NULL;
65      gHelpUsage = FALSE;
66
67  for (Arg = 1; Arg < Argc; Arg++) {
68      if (Argv[Arg][0] == L'-') {
69          if (StrCmp(Argv[Arg], L"-?") == 0) {
70              gHelpUsage = TRUE;
71          } else {
72              Print(L"%s: unknown command-line argument.\n", Argv[Arg]);
73              return SHELL_INVALID_PARAMETER;
74          }
75      } else if (gPrompt == NULL) {
76          gPrompt = Argv[Arg];
77      } else if (gEnvVar == NULL) {
78          gEnvVar = Argv[Arg];
79      } else {
80          Print(L"%s: unexpected command-line argument.\n", Argv[Arg]);
81          return SHELL_INVALID_PARAMETER;
82      }
83  }
84
85  return SHELL_SUCCESS;
86 }

```

Figure 10.7: Parse Command Line in GetKey.c.

Lines 55–66

This function sets the global variables to default values. The value of NULL is used to indicate that no command-line option has been found for that global variable.

Line 67

This loop walks through all of the command-line options passed to this application. The UEFI Shell automatically divides the command-line into options using whitespace. Argv[0] is the actual command used when referring to this application.

Lines 68–74

There is only a single command-line flag. Flags are prefixed with the character “-”. If there is a request for help (-?) then set the flag. If there is none, but the command-line

option has a hyphen as its first character, then print an error message and return with an error.

Lines 75–76

If this wasn't a flag, and no prompt has been specified, then save the command-line option as the prompt.

Line 77–78

If this wasn't a flag and no prompt or environment variable name has been specified, then save the command-line option as the environment variable name.

Line 79–83

Otherwise, display an error message and return.

Line 85

Return successfully.

Now that all of the basics for the program have been completed, we can actually read a key, as shown in Figure 10.8.

```

20     SHELL_STATUS
21     ReadKey(OUT EFI_KEY_DATA *Key)
22 {
23     EFI_STATUS          Status;
24     UINTN              EventIndex;
25     EFI_EVENT           Events[2];
26     EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *Sti2;
27
28     Sti2 = NULL;
29     Status = gBS->HandleProtocol(gST->ConsoleInHandle, &gEfiSimpleTextInputExProtocolGuid, (VOID **)&Sti2);
30
31     if (!EFI_ERROR(Status)) {
32         Events[0] = Sti2->WaitForKeyEx;
33     } else {
34         Events[0] = gST->ConIn->WaitForKey;
35     }
36     Events[1] = gEfiShellProtocol->ExecutionBreak;
37
38     Status = gBS->WaitForEvent(2, Events, &EventIndex);
39     if (EventIndex == 1 || Status != EFI_SUCCESS) {
40         return SHELL_ABORTED;
41     }
42
43     if (Sti2 != NULL) {
44         Status = Sti2->ReadKeyStrokeEx(Sti2, Key);
45     } else {
46         Key->KeyState.KeyShiftState = 0;
47         Status = gST->ConIn->ReadKeyStroke(gST->ConIn, &Key->Key);
48     }
49
50     if (EFI_ERROR(Status)) {
51         return SHELL_DEVICE_ERROR;
52     }
53     return SHELL_SUCCESS;
54 }
```

Figure 10.8: Read a Key in GetKey.c.

Line 20–21

This function reads a key and places the resulting key information into `Key`. If everything goes according to plan, this function returns `SHELL_SUCCESS`. If someone presses Ctrl-C, it returns `SHELL_ABORTED`.

Lines 24–25

These local variables implement a simple array of event handles. UEFI defines events that can be signaled so that waiting applications can resume execution. The UEFI Shell signals an event when the user presses Ctrl-C. This event—`ExecutionBreak`—is one of the members of the Shell protocol. The keyboard driver has another event that it signals any time there is a key pressed. This event—`WaitForKey`—is a member of both the Simple Text and Simple Text Extended protocols. `EventIndex` indicates which of the two events was signaled: 0 = key press, 1 = Ctrl-C.

Lines 28–34

There are two different UEFI protocols for reading keys. The Simple Text protocol is the older one, and is required. The Simple Text Extended protocol is newer, and provides more information, but is optional.

The UEFI System Table (`gST`) provides the handle of the driver (`ConsoleInHandle`) that provides the console services and a pointer to the Simple Text Input protocol installed on that handle (`Con`). However, there is no pointer to the Simple Text Extended protocol in the UEFI System Table. As a result, this function tries to see an instance is installed on the console services handle using the UEFI boot service `HandleProtocol()`.

This code initializes the first element in the event array with the `WaitForKey` event handle. It prefers the event handle from the Simple Text Extended protocol.

Line 35

The second element in the event array is initialized with the `ExecutionBreak` event handle.

Lines 37–39

Then the UEFI boot service `WaitForEvent()` pauses the execution of the application and waits for one of the two events in the event handle array to be signaled. Upon return, `EventIndex` will hold the index of the actual event signaled.

If there was an error, or Ctrl-C was pressed, exit the function with the `SHELL_ABORTED` status code.

Line 42–47

If the Simple Text Input Extended protocol is supported, then read the key data using the `.ReadKeyStrokeEx()` member of that protocol. Otherwise, use the `.ReadKeyStroke()` member of the Simple Text Input protocol. In this case, the shift state is initialized to zero, since the older protocol doesn't support that field. Effectively, this means that the Alt and Shift modifier keys can't be detected in this case.

Lines 49–52

Return with the correct error code. This function can't just return back the status code it received from `.ReadKeyStroke()` or `.ReadKeyStrokeEx()`, because they return type `EFI_STATUS` while this function must return type `SHELL_STATUS`. Rather than try to decode and translate all possible return values, `GetKey` simply converts all errors to `SHELL_DEVICE_ERROR`.

Now we have arrived at the heart of the program, which parses the key data and produces the key text. This starts with the main function, `ConvertKeyToText()`, as shown in Figure 10.9

```

144 CONST CHAR16 *
145 ConvertKeyToText (IN EFI_KEY_DATA *Key)
146 {
147     CHAR16      KeyStr[2];
148     STATIC CHAR16  KeyChar[30];
149
150     KeyChar[0] = L'\0';
151
152     if ((Key->KeyState.KeyShiftState & (EFI_LEFT_CONTROL_PRESSED|EFI_RIGHT_CONTROL_PRESSED)) != 0 ||
153         (Key->Key.UnicodeChar != L'\0' && Key->Key.UnicodeChar < L' ')) {
154         StrCat(KeyChar, L"Ctrl+");
155     }
156     if ((Key->KeyState.KeyShiftState & (EFI_LEFT_ALT_PRESSED|EFI_RIGHT_ALT_PRESSED)) != 0) {
157         StrCat(KeyChar, L"Alt+");
158     }
159     if ((Key->KeyState.KeyShiftState & (EFI_LEFT_SHIFT_PRESSED|EFI_RIGHT_SHIFT_PRESSED)) != 0) {
160         StrCat(KeyChar, L"Shift+");
161     }
162
163     if (Key->Key.UnicodeChar == L'\0') {
164         StrCat(KeyChar, ConvertScanCodeText(Key->Key.ScanCode));
165     } else {
166         if (Key->Key.UnicodeChar < L' ') {
167             KeyStr[0] = Key->Key.UnicodeChar + L'@';
168         } else {
169             KeyStr[0] = Key->Key.UnicodeChar;
170         }
171         KeyStr[1] = L'\0';
172
173         StrCat(KeyChar, KeyStr);
174     }
175     return KeyChar;

```

Figure 10.9: Converting Keys to Key Text in `GetKey.c`.

Lines 144–145

This function takes the key data returned from `ReadKey()` as input and returns a null-terminated string that contains the key's description as human-readable text.

Lines 147–150

The key text is assembled in `KeyChar`. It is a local variable, but its storage class is static, which means that the variable doesn't disappear when the function returns. This is handy, because we want to return a pointer to this variable but don't want to clutter up the global namespace. The string is initialized to null-terminated so that subsequent calls to `StrCat ()` (below) will append correctly.

Lines 152–155

This section checks to see if the “Ctrl+” text should appear in the key text. This happens in two cases: when one of the control keys are pressed or when the key scan code returns an ASCII value less than 0x20, indicating a control value. The last case is necessary because the Simple Text Input protocol doesn't return the shift state, but does return control characters.

Note: Yes, there are separate indicators for left and right of all modifier keys, and it is possible to detect them separately.

Lines 156–158

This section detects if the “Alt+” text should appear in the key text. This happens when one of the alternate (Alt) keys are pressed. Notice that at this point, there may or may not be a Ctrl+ in the buffer, so we need to append our string.

Lines 159–161

This section detects if the “Shift+” text should appear in the key text. This happens when one of the shift keys are pressed.

Note: It is possible to get Shift-z (lower-case Z). For example, if Caps Lock is set, then pressing Shift-Z will return a lower-case ‘z’ instead of an upper-case ‘Z’. The shift modifier refers to the shift keys, not to the returned scan code.

Note: There are other modifier keys described in the UEFI specification, such as the Logo, Menu, and System Request (SysReq) keys, but I've left those for future expansion.

Lines 163–165

If this key is not a normal printable character, like Page-Up or Escape or one of the control characters, then the `UnicodeChar` field member has the value of 0. In this case, the key's value is found in the `ScanCode` member and appended to the end of the string.

Lines 166–174

Otherwise, the key represents some sort of printable character. The exception to this rule are control characters, which have a character value of 0x01 – 0x1f. In this case, we up-shift the value to 0x40-0x5F and append that character to the key text.

Line 175

Now the function returns the pointer to the assembled key text.

The final section, shown in Figures 10.10 and 10.11, describes `ConvertScanCodeText()`, which converts scan codes for non-printable characters to human readable text. The function uses a switch statement instead of a lookup array because the values are non-contiguous and this is easier to read and understand.

```

89 CONST CHAR16 *
90 ConvertScanCodeText(IN UINT16 ScanCode)
91 {
92     switch (ScanCode) {
93         case SCAN_NULL:           return L"NULL";
94         case SCAN_UP:            return L"Up";
95         case SCAN_DOWN:          return L"Down";
96         case SCAN_RIGHT:         return L"Right";
97         case SCAN_LEFT:          return L"Left";
98         case SCAN_HOME:          return L"Home";
99         case SCAN_END:           return L"End";
100        case SCAN_INSERT:         return L"Insert";
101        case SCAN_DELETE:         return L"Del";
102        case SCAN_PAGE_UP:        return L"PageUp";
103        case SCAN_PAGE_DOWN:      return L"PageDn";
104        case SCAN_F1:             return L"F1";
105        case SCAN_F2:             return L"F2";
106        case SCAN_F3:             return L"F3";
107        case SCAN_F4:             return L"F4";
108        case SCAN_F5:             return L"F5";
109        case SCAN_F6:             return L"F6";
110        case SCAN_F7:             return L"F7";
111        case SCAN_F8:             return L"F8";
112        case SCAN_F9:             return L"F9";
113        case SCAN_F10:            return L"F10";
114        case SCAN_ESC:            return L"Esc";
115        case SCAN_F11:            return L"F11";
116        case SCAN_F12:            return L"F12";
117        case SCAN_PAUSE:          return L"Pause";
118        case SCAN_F13:            return L"F13";
119        case SCAN_F14:            return L"F14";
120        case SCAN_F15:            return L"F15";
121        case SCAN_F16:            return L"F16";
122        case SCAN_F17:            return L"F17";
123        case SCAN_F18:            return L"F18";
124        case SCAN_F19:            return L"F19";
125        case SCAN_F20:            return L"F20";

```

Figure 10.10: `ConvertScanCodeText()` in `GetKey.c`, Part 1.

```

126     case SCAN_F21:           return L"F21";
127     case SCAN_F22:           return L"F22";
128     case SCAN_F23:           return L"F23";
129     case SCAN_F24:           return L"F24";
130     case SCAN_MUTE:          return L"Mute";
131     case SCAN_VOLUME_UP:    return L"VolUp";
132     case SCAN_VOLUME_DOWN:  return L"VolDn";
133     case SCAN_BRIGHTNESS_UP: return L" BrightUp";
134     case SCAN_BRIGHTNESS_DOWN: return L" BrightDn";
135     case SCAN_SUSPEND:       return L"Suspend";
136     case SCAN_HIBERNATE:     return L"Hibernate";
137     case SCAN_TOGGLE_DISPLAY: return L"ToggleDisplay";
138     case SCAN_RECOVERY:      return L"Recovery";
139     case SCAN_EJECT:         return L"Eject";
140   }
141   return L"Unknown";
142 }

```

Figure 10.11: ConvertScanCodeToText () in GetKey.c, Part 2.

Lines 89–90

This function takes a scan code on input and returns a null-terminated string on exit.

Lines 91–142

This switch statement simply translates from a value to a text string, or else “Unknown” if the value isn’t known.

The Component Information (.inf) File: GetKey.inf

Figure 10.12 shows the component information (.inf) file for this application. It is very similar to the first application.

```
1  ## @file
2  # Print a prompt, read a key, convert key to text, set environment vari-
3  #
4  #
5  ##
6
7  [Defines]
8      INF_VERSION          = 0x00010006
9      BASE_NAME             = GetKey
10     FILE_GUID              = 4816b91c-0c06-4318-8b5f-8219a0a3951f
11     MODULE_TYPE            = UEFI_APPLICATION
12     VERSION_STRING         = 0.1
13     ENTRY_POINT             = ShellCEEntryLib
14
15 [Sources]
16     GetKey.c
17
18 [Packages]
19     MdePkg/MdePkg.dec
20     ShellPkg/ShellPkg.dec
21
22 [Protocols]
23     gEfiSimpleTextInputExProtocolGuid
24
25 [LibraryClasses]
26     UefiLib
27     ShellCEEntryLib
28     ShellLib
--
```

Figure 10.12: Component Information (.inf) File for GetKey.

Lines 22–23

The only point of interest in this file is the declaration of the protocol `gEfiSimpleTextInputExProtocolGuid`'s GUID. The value for this GUID can be found in `MdePkg.dec`. By declaring it here, its value will be automatically generated into the component.

The Build Description (.dsc) File

The changes to the build description file are very similar to those for the previous applications.

```

1 [LibraryClasses]
2
3 UefiLib|MdePkg/Library/UefiLib/UefiLib.inf
4 ShellCEEntryLib|ShellPkg/Library/UefiShellCEEntryLib/UefiShellCEEntryLib.inf
5 ShellLib|ShellPkg/Library/UefiShellLib/UefiShellLib.inf
6
7 [Components]
8 ShellBookPkg/Applications/GetKey/GetKey.inf           # Get a key, return key as text

```

Figure 10.13: Extracted Portions of a Build Description (.dsc) File.

Line 1

The [LibraryClasses] section specifies a mapping between a library class (as we saw back in Lines 22-24 of the .inf file) and the component information file for a specific library. This allows the association between library class and actual library to be manipulated at the project level, not just at the individual component level. In this case, UefiLib (line 3) is associated with a standard component in the MdePkg while ShellCEEntryLib (line 4) is associated with a component in the ShellPkg. ShellLib provides the Shell protocol instance.

Line 7

The [Components] section specifies those components that will be built. Here we list the relative path of the application's .inf file, relative to WORKSPACE.

Calculate Math Expressions: Math

While the UEFI Shell provides a number of operators with the built-in “if” script operator, these are mostly limited to Boolean operators, such as greater than, or less than, or equal. But what if you need a more complicated expression, including addition or subtraction. For example, if you read a value from a memory-mapped I/O location and need to isolate a single bit in order to make a decision in a script file. Welcome to Math, an application that treats the command-line arguments as expressions and outputs the result to standard output. The expression uses standard C-style operator precedence, all terms are unsigned 64-bit integers.

For example:

Math 4 + 5

9

Or:

```
Math 9 shl 1  
18
```

Or:

```
set ABC 3  
Math %ABC% and 1  
1
```

The operators are (in order of lowest-to-highest priority):

and Bitwise AND

or Bitwise OR

==/eq Equal

!=/ne Not Equal

ge Greater Than or Equal

le Less Than or Equal

lt Less Than

gt Greater Than

shl Shift Left

shr Shift Right

+

Add

-

Subtract

*

Multiply

/

Divide

mod Modulus

not,! Logical Not

~ Bitwise Not

() Parentheses

The Source File: Math.c

This section, displayed in Figure 10.14, describes the global variables and necessary header files.

```
1  /*
2   Perform Command-Line Math Calculations
3  */
4
5  #include <Uefi.h>
6  #include <Library/UefiLib.h>
7
8  #include <Library/ShellLib.h>
9
10
11
12
13
14 // Forward References
15 BOOLEAN
16 ParseExpr(
17     IN UINTN          Argc,
18     IN CHAR16        **Argv,
19     OUT UINT64        *Result
20 );
```

Figure 10.14: Global Variables in Math.c

Lines 5–9

These include the standard UEFI definitions (`Uefi.h`) and the various library class header files. `UefiLib.h` describes standard wrapper functions. `ShellCEntryLib.h` handles the entry point. `ShellLib` wraps UEFI Shell services. `EfiShell.h` encapsulates the `EFI_SHELL_PROTOCOL` definition.

Line 12

This variable is used to record the command-line **parameters** related to showing help.

Lines 14–20

This is the forward declaration to the main expression parsing subroutine.

The next section describes the main entry point, `ShellAppMain`, shown in Figure 10.15.

```

339     INTN
340     EFIAPI
341     ShellAppMain (
342         IN UINTN             Argc,
343         IN CHAR16           **Argv
344     )
345     {
346         UINT64 Result;
347
348         if (!ParseExpr(Argc, Argv, &Result)) {
349             Print(L"error parsing expression.\n");
350             return SHELL_INVALID_PARAMETER;
351         }
352
353         Print(L"%ld\n", Result);
354         return SHELL_SUCCESS;
355     }

```

Figure 10.15: Entry Point for Math.c.

Lines 339–344

This UEFI Shell application uses the version of the UEFI Shell application entry point that provides the argument count (Argc) and pointers to the arguments (Argv).

Line 346

The calculated result will be placed into Result.

Lines 348–351

This kicks off the parsing by calling the main expression parsing function ParseExpr. If there is an error, then it will print out an error message and return with an error code.

Lines 353–354

Otherwise, print out the result and exit with success.

Now ParseExpr () is the top-level function for the expression parsing. The utility uses a standard recursive-descent parser, shown in Figure 10.16.

```

316    BOOLEAN
317    ParseExpr(
318        IN UINTN          Argc,
319        IN CHAR16         **Argv,
320        OUT UINT64        *Result
321    )
322    {
323        UINTN Arg;
324
325        Arg = 1;           // start with the first parameter.
326        if (!ParseExpr5(&Arg, Argc, Argv, Result)) {
327            return FALSE;
328        }
329        if (Arg != Argc) {
330            Print(L"illegal expression '%s'.\n", Argv[Arg]);
331            return FALSE;
332        }
333        return TRUE;
334    }

```

Figure 10.16: Top Level Parsing Function ParseExpr() in Math.c

Lines 316–321

This function takes information about the command line, such as the command-line argument count and pointers to the command-line arguments. It also takes a pointer to the 64-bit unsigned integer that will hold the result.

Lines 323–328

Parsing starts with the second command-line argument. The first command-line argument is the command-line of this application (that is, Math). So we pass this into the parsing function for the lowest priority parsing function (ParseExpr5). If an error is returned, then that error is returned back.

Lines 329–332

If the current argument returned from ParseExpr5() isn't past the last command-line argument, it means that the parsing functions didn't recognize something and parsing stopped. This happens when an unidentified operator identifier is found. So we'll return an error.

Lines 33

Everything went according to plan and the resulting value is in Result.

The next section shows a simple utility function that is used for checking whether an operator is present on the command-line, as shown in Figure 10.17 and, if so, move to the next argument.

```

23   BOOLEAN
24   ParseExprToken(
25     IN UINTN          *Arg,
26     IN UINTN          Argc,
27     IN CHAR16         **Argv,
28     IN CONST CHAR16  *Token
29   )
30   {
31   if (*Arg < Argc) {
32   if (StrCmp(Argv[*Arg], Token) == 0) {
33   (*Arg)++;
34   return TRUE;
35   }
36   }
37   return FALSE;
38 }
```

Figure 10.17: Parse operators on the command-line in Math.c.

Lines 23–29

This function takes the same command-line arguments as the other parsing functions, a pointer to the current argument index (Arg), the number of arguments on the command-line (Argc), and an array of pointers to the arguments (Argv). It also takes a pointer to a null-terminated string that specifies the token that is being checked for on the command line.

Line 31–36

If there are any more arguments on the command line (line 31) and the current command-line argument matches the passed-in token string, then increment the argument index and return TRUE. This uses the library function `StrCmp`, which is a UEFI-specific case-sensitive comparison function for null-terminated Unicode strings—very similar to `strcmp()` in the C standard library.

Line 37

If there were no more arguments or the current argument didn't match, then nothing happens and the function returns FALSE.

The next section describes `ParseExpr5()` – `ParseExpr1()`, shown in Figures 10.18–10.22. These all take the same basic form: try to parse something high in priority, then look for an operator and, if present, parse another value of equal or higher priority, then calculate the resulting operation value and return.

```

285     BOOLEAN
286     ParseExpr5(
287         IN UINTN *Arg,
288         IN UINTN Argc,
289         IN CHAR16 **Argv,
290         OUT UINT64 *Result
291     )
292     {
293         UINT64 Left;
294         UINT64 Right;
295
296         if (!ParseExpr4(Arg, Argc, Argv, &Left)) {
297             return FALSE;
298         }
299
300         if (ParseExprToken(Arg, Argc, Argv, L"and")) {
301             if (!ParseExpr5(Arg, Argc, Argv, &Right)) {
302                 return FALSE;
303             }
304
305             *Result = Left & Right;
306             return TRUE;
307         } else if (ParseExprToken(Arg, Argc, Argv, L"or")) {
308             if (!ParseExpr5(Arg, Argc, Argv, &Right)) {
309                 return FALSE;
310             }
311
312             *Result = Left | Right;
313             return TRUE;
314         }
315
316         *Result = Left;
317         return TRUE;
318     }

```

Figure 10.18: Parsing “bitwise and” and “bitwise or” operators in Math.c.

Lines 285–291

These parsing functions all take the same form: a pointer to the current argument index, the argument count, an array of pointers to all arguments, and a pointer to the returned result value. When the function successfully parses something, the current argument index is incremented.

Lines 293–294

These parsing functions declare the unsigned integer value that is left of the operator and right of the operator.

Lines 296–298

First, the function attempts to evaluate any higher priority operators to the left of the operators at this priority level and puts the result in Left. If anything was parsed, then Arg will be updated to point to the first argument after the last one parsed. For example, the expression $5 + 3 \text{ AND } 6$ will have a call stack.

- ParseExpr5, ParseExpr4, ParseExpr3, ParseExpr2, ParseExpr1, ParseExprTerm, Left = 5,
- Return back to ParseExpr3,
- parse the ‘+’, ParseExpr4, ParseExprTerm, Right = 3,
- Return back to ParseExpr3, Left = 8 ($5 + 3$),
- Return back to ParseExpr5,
- Parse the ‘AND’, ParseExpr4, ParseExpr3, ParseExpr2, ParseExpr1, ParseExprTerm, Right = 6,
- Return back to ParseExpr5, Left = 8 AND 6 (12), return with Result = 12.

It is a little tricky because of recursion, but forms the basis of almost all expression parsing algorithms, although others may have optimizations.

Line 300

Next, after getting a value from the higher priority operators, we check if there is an AND operator using the `ParseExprToken()` function. If this function returns TRUE, it indicates that the AND operator was present, and Arg will be updated. The C operator ‘&’ could not be used. See Lines 303ff for a detailed explanation.

Lines 301–306

Now we have to find the right-hand value for the AND operator. Notice that `ParseExpr5()` is called (same operator priority) rather than `ParseExpr4()` (higher operator priority). That is because C-style operator precedence says that operators of the same priority are processed left to right. After the result comes back, the bitwise AND is calculated and the result is stored in Result.

Line 307–314

This is the same as lines 301-306, except for the logical OR operator. The C operator ‘|’ could not be used because ‘|’ has a special meaning on the command-line. So OR was used instead. Even though ‘&’ could have been used for bitwise AND, AND was chosen instead for consistency.

Line 316–317

If Math gets to this point, it means that there was no operator of this priority level present. So it transfers the value from the higher priority operators from Left into Result, since no operator will be processed.

```

218     BOOLEAN
219     ParseExpr4(
220         IN UINTN *Arg,
221         IN UINTN Argc,
222         IN CHAR16 **Argv,
223         OUT UINT64 *Result
224     )
225     {
226         UINT64 Left;
227         UINT64 Right;
228
229         if (!ParseExpr3(Arg, Argc, Argv, &Left)) {
230             return FALSE;
231         }
232
233         if (ParseExprToken(Arg, Argc, Argv, L"==") ||
234             ParseExprToken(Arg, Argc, Argv, L"eq")) {
235             if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
236                 return FALSE;
237             }
238
239             *Result = Left == Right;
240             return TRUE;
241         } else if (ParseExprToken(Arg, Argc, Argv, L"!=") ||
242             ParseExprToken(Arg, Argc, Argv, L"ne")) {
243             if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
244                 return FALSE;
245             }
246
247             *Result = Left != Right;
248             return TRUE;
249         } else if (ParseExprToken(Arg, Argc, Argv, L"ge")) {
250             if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
251                 return FALSE;
252             }
253             *Result = Left >= Right;
254             return TRUE;

```

Figure 10.19a: Parse Equality Operators in Math.c.

Lines 229–231

First, parse the left side of the equality operators using `ParseExpr3()` and put the result in `Left`.

Lines 233–240

Handle the processing of the equality (`==` or `eq`) operator.

Lines 241–248

Handle the processing of the inequality (`!=` or `ne`) operator.

Lines 249–254

Handle the processing of the greater-than or equal (le) operator.

```

255     } else if (ParseExprToken(Arg, Argc, Argv, L"le")) {
256         if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
257             return FALSE;
258         }
259
260         *Result = Left <= Right;
261         return TRUE;
262     } else if (ParseExprToken(Arg, Argc, Argv, L"lt")) {
263         if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
264             return FALSE;
265         }
266
267         *Result = Left < Right;
268         return TRUE;
269     } else if (ParseExprToken(Arg, Argc, Argv, L"gt")) {
270         if (!ParseExpr4(Arg, Argc, Argv, &Right)) {
271             return FALSE;
272         }
273
274         *Result = Left > Right;
275         return TRUE;
276     }
277
278     *Result = Left;
279     return TRUE;
280 }
```

Figure 10.19b: Parse Equality Operators in Math.c.

Lines 255–261

Handle the processing of the less-than or equal (le) operator.

Lines 262–268

Handle the processing of the less-than (lt) operator.

Lines 269–276

Handle the processing of the greater-than (gt) operator.

Lines 277–278

Otherwise, if there was no comparison operator, return the higher priority operator results.

```

183     BOOLEAN
184     ParseExpr3(
185         IN UINTN *Arg,
186         IN UINTN Argc,
187         IN CHAR16 **Argv,
188         OUT UINT64 *Result
189     )
190     {
191         UINT64 Left;
192         UINT64 Right;
193
194         if (!ParseExpr2(Arg, Argc, Argv, &Left)) {
195             return FALSE;
196         }
197
198         if (ParseExprToken(Arg, Argc, Argv, L"shl")) {
199             if (!ParseExpr3(Arg, Argc, Argv, &Right)) {
200                 return FALSE;
201             }
202
203             *Result = LShiftU64(Left, (UINTN) Right);
204             return TRUE;
205         } else if (ParseExprToken(Arg, Argc, Argv, L"shr")) {
206             if (!ParseExpr3(Arg, Argc, Argv, &Right)) {
207                 return FALSE;
208             }
209
210             *Result = RShiftU64(Left, (UINTN) Right);
211             return TRUE;
212         }
213
214         *Result = Left;
215         return TRUE;
216     }

```

Figure 10.20: Parsing Shift Operators in Math.c.

Lines 194–196

Get the left side of the operators by calling `ParseExpr2()`.

Lines 198–212

Check for the shift-left operator (199–205) and shift-right operator (206–216) and, if present, perform the operators and place the result in Result. “shl” and “shr” were used instead of the more traditional << and >> operators found in C because >> has a special meaning (create and redirect to file) in the UEFI Shell.

Lines 214–215

If there were no shift operators, transfer Left to Result and return.

```
148     BOOLEAN
149     ParseExpr2(
150         IN UINTN *Arg,
151         IN UINTN Argc,
152         IN CHAR16 **Argv,
153         OUT UINT64 *Result
154     )
155     {
156         UINT64 Left;
157         UINT64 Right;
158
159         if (!ParseExpr1(Arg, Argc, Argv, &Left)) {
160             return FALSE;
161         }
162
163         if (ParseExprToken(Arg, Argc, Argv, L"+")) {
164             if (!ParseExpr2(Arg, Argc, Argv, &Right)) {
165                 return FALSE;
166             }
167
168             *Result = Left + Right;
169             return TRUE;
170         } else if (ParseExprToken(Arg, Argc, Argv, L"-")) {
171             if (!ParseExpr2(Arg, Argc, Argv, &Right)) {
172                 return FALSE;
173             }
174
175             *Result = Left - Right;
176             return TRUE;
177         }
178
179         *Result = Left;
180         return TRUE;
181     }
```

Figure 10.21: Parsing Addition and Subtraction Operators in Math.c.

Lines 148–181

Handle the parsing of the plus and minus operators and calculate the results.

```

106    BOOLEAN
107    ParseExpr1(
108        IN UINTN           *Arg,
109        IN UINTN           Argc,
110        IN CHAR16          **Argv,
111        OUT UINT64          *Result
112    )
113    {
114        UINT64 Left;
115        UINT64 Right;
116
117        if (!ParseExprUnary(Arg, Argc, Argv, &Left)) {
118            return FALSE;
119        }
120
121        if (ParseExprToken(Arg, Argc, Argv, L"*")) {
122            if (!ParseExpr1(Arg, Argc, Argv, &Right)) {
123                return FALSE;
124            }
125
126            *Result = MultU64x64(Left, Right);
127            return TRUE;
128        } else if (ParseExprToken(Arg, Argc, Argv, L"/")) {
129            if (!ParseExpr1(Arg, Argc, Argv, &Right) || (Right == 0)) {
130                return FALSE;
131            }
132
133            *Result = DivU64x64Remainder(Left, Right, NULL);
134            return TRUE;
135        } else if (ParseExprToken(Arg, Argc, Argv, L"mod")) {
136            if (!ParseExpr1(Arg, Argc, Argv, &Right) || (Right == 0)) {
137                return FALSE;
138            }
139
140            DivU64x64Remainder(Left, Right, Result);
141            return TRUE;
142        }
143
144        *Result = Left;
145        return TRUE;
146    }

```

Figure 10.22: Parsing Multiplication, Division and Modulo Operators in Math.c.

Lines 110–147

Using the same style, handle the multiplication (*), division (/), and modulo operators (mod). The C operator % could not be used because % has a special meaning in command-line processing, introducing environment variable names.

The next section, shown in Figure 10.23, describes `ParseExprUnary()`, which deals with unary operators, such as “not” (logical not) and “~” (bitwise not).

```

76     BOOLEAN
77     ParseExprUnary(
78         IN UINTN           *Arg,
79         IN UINTN           Argc,
80         IN CHAR16          **Argv,
81         OUT UINT64         *Result
82     )
83 {
84     UINT64 Right;
85
86     if (ParseExprToken(Arg, Argc, Argv, L"not") ||
87         ParseExprToken(Arg, Argc, Argv, L"!")) {
88         if (!ParseExprUnary(Arg, Argc, Argv, &Right)) {
89             return FALSE;
90         }
91
92         *Result = !Right;
93         return TRUE;
94     } else if (ParseExprToken(Arg, Argc, Argv, L"~")) {
95         if (!ParseExprUnary(Arg, Argc, Argv, &Right)) {
96             return FALSE;
97         }
98
99         *Result = ~Right;
100        return TRUE;
101    }
102
103    return ParseExprTerm(Arg, Argc, Argv, Result);
104}

```

Figure 10.23: Unary operators in Math.c.

Line 84

Unlike binary operators, Math is only concerned with what appears to the right of the operators. Math doesn't support any of the C programming languages post-fix operators (such as `++`), so that makes things simpler.

Lines 86–87

For unary operators, the operator is checked first. In this case, it is checking for the logical NOT operators (“`not`” or its alias “`!`”).

Lines 88–93

For these operators, these lines try to get the value to the right of the operator and, if successful, place the result in `Result`.

Lines 94–101

These do the same thing, but for the bitwise NOT operator (“`~`”).

Line 103

If neither of the unary operators were present, then just get a simple term by calling `ParseExprTerm()`.

Finally, for the last section, Math handles the parsing of unsigned integers and parentheses, shown in Figure 10.24.

```

48 ParseExprTerm(
49     IN UINTN          *Arg,
50     IN UINTN          Argc,
51     IN CHAR16         **Argv,
52     OUT UINT64        *Result
53 )
54 {
55     if (ParseExprToken(Arg, Argc, Argv, L"()")) {
56         if (!ParseExpr5(Arg, Argc, Argv, Result)) {
57             return FALSE;
58         }
59         if (ParseExprToken(Arg, Argc, Argv, L")")) {
60             return TRUE;
61         }
62     } else if (Argc > *Arg) {
63         if (!EFI_ERROR(ShellConvertStringToInt64(Argv[*Arg], Result, FALSE, FALSE))) {
64             (*Arg)++;
65             return TRUE;
66         }
67     }
68     if (*Arg < Argc) {
69         Print(L"unexpected expression term '%s'.\n", Argv[*Arg]);
70     } else {
71         Print(L"missing expression.\n");
72     }
73 }
74 }
```

Figure 10.24: Parsing unsigned integers and parentheses in Math.c.

Lines 55–61

First, check for the left parentheses operator. Parentheses are handled by simply calling the lowest priority operator parser (`ParseExpr5()`) and then looking for the right parentheses.

Lines 62–67

If there are any more command-line arguments, then the only legal possibility is that it is an unsigned integer. The function `ShellConvertStringToInt64()` is a standard library function that converts a null-terminated string to an integer. If the number is prefixed by a “0x” then it is treated as hexadecimal. Otherwise, it is treated as decimal unless the third parameter is set to TRUE (which we don’t).

Lines 68–72

At this point, there are no legal options left, so print an error.

The Component Information (.inf) File: Math.inf

```

1  ##
2  # Perform Command-Line Math Calculations
3  #
4  ##
5
6  [Defines]
7      INF_VERSION          = 0x00010006
8      BASE_NAME            = Math
9      FILE_GUID             = e5787a0d-521f-4afe-93a7-9515f4a60501
10     MODULE_TYPE           = UEFI_APPLICATION
11     VERSION_STRING        = 0.1
12     ENTRY_POINT           = ShellCEEntryLib
13
14 [Sources]
15     Math.c
16
17 [Packages]
18     StdLib/StdLib.dec
19     MdePkg/MdePkg.dec
20     ShellPkg/ShellPkg.dec
21
22 [LibraryClasses]
23     UefiLib
24     ShellCEEntryLib
25     ShellLib

```

Figure 10.25: Component Information (.inf) File for Math.

Lines 1–25

This is a fairly simple component information file, where only `BASE_NAME` and `FILE_GUID` have been updated from the one found in the `HelloWorld` UEFI Shell application.

Convert ASCII to Unicode and Back: UniCodeDecode

The `UniCodeDecode` application converts to and from ASCII and Unicode UCS-2 encoded files. It supports explicit control by the user as to the file formats and also employs various forms of auto-detection, using either the Unicode-defined file marker or some heuristics.

The application has the syntax:

```

ucd -?
ucd input-file output-file [-a2u|-u2a]

```

This application demonstrates several key shell capabilities:

- *File Handling.* This application shows how to create, open, and write files.
- *Command-Line Handling.* This application shows more advanced command-line handling, including defaulting and error checking.
- *Help Text.* This application shows how to use the built-in help text support.
- *Unicode.* This application delves into some of the Unicode handling aspects of UEFI applications, including file markers. The UEFI specification uses Unicode's UCS-2 (little-endian) extensively. UCS-2 differs from UTF-16 in that it does not handle surrogate pairs. For more information, see www.unicode.org.

The Source File: UniCodeDecode.c

The entry point source code, shown in Figures 10.26 and 10.27, performs the basic initialization and then walks through the four basic steps: parsing the command-line, opening the input and output files, converting the file, and then closing the files.

```

48     BOOLEAN gHelpUsage = FALSE;
49     BOOLEAN gAscii2Ucs = FALSE;
50     BOOLEAN gUcs2Ascii = FALSE;
51
52     CONST CHAR16 *InputFileName = NULL;
53     CONST CHAR16 *OutputFileName = NULL;
54
55     SHELL_FILE_HANDLE InputFileHandle = NULL;
56     SHELL_FILE_HANDLE OutputFileHandle = NULL;
57
58     extern EFI_SHELL_PROTOCOL *gEfiShellProtocol;
```

Figure 10.26: Command-line related global variables in UnicodeDecode.c.

Lines 48–53

These Booleans record the presence of specific command-line options, including the –a2u, –?, –u2a, and pointers to the input and output file names.

Lines 55–56

These file handles are used during the actual processing of the files.

Lines 58

The global pointer to the Shell protocol. This pointer is initialized by the Shell library.

```

298     INTN
299     EFIAPI
300     ShellAppMain (
301         IN UINTN             Argc,
302         IN CHAR16           **Argv
303     )
304     {
305         SHELL_STATUS         Status;
306
307         Status = ParseCommandLine(Argc, Argv);
308         if (Status != SHELL_SUCCESS) {
309             return Status;
310         }
311         if (gHelpUsage) {
312             ShowHelp();
313             return SHELL_SUCCESS;
314         }
315
316         Status = OpenFiles();
317         if (Status != SHELL_SUCCESS) {
318             return Status;
319         }
320
321         if (gAscii2Ucs) {
322             ConvertAsciiToUcs2();
323         } else if (gUcs2Ascii) {
324             ConvertUcs2ToAscii();
325         }
326
327         Status = CloseFiles();
328         if (Status != SHELL_SUCCESS) {
329             return Status;
330         }
331
332         return SHELL_SUCCESS;
333     }

```

Figure 10.27: Entry point in UnicodeDecode.c.

Lines 307–310

Parse the command-line options. If there was an error, exit. If help was requested, show it.

Lines 316–319

Open the input and the output files, and optionally detect the file type.

Lines 321–325

If the input file is an ASCII file, then convert it to Unicode. If the input file is a Unicode file, then convert it to ASCII.

Lines 327-333

Close the files and exit.

```

60     SHELL_STATUS
61     ParseCommandLine(
62         IN UINTN          Argc,
63         IN CHAR16        **Argv
64     )
65 {
66     UINT32 n;
67
68     for (n = 1; n < Argc; n++) {
69         if (Argv[n][0] == L'-') {
70             if (StrCmp(Argv[n], L"-?") == 0) {
71                 gHelpUsage = TRUE;
72             } else if (StrCmp(Argv[n], L"-a2u") == 0) {
73                 gAscii2Ucs = TRUE;
74             } else if (StrCmp(Argv[n], L"-u2a") == 0) {
75                 gUcs2Ascii = TRUE;
76             } else {
77                 Print(L"%s: unknown command-line argument.\n", Argv[n]);
78                 return SHELL_INVALID_PARAMETER;
79             }
80         } else if (InputFileName == NULL) {
81             InputFileName = Argv[n];
82         } else if (OutputFileName == NULL) {
83             OutputFileName = Argv[n];
84         } else {
85             Print(L"%s: unexpected command-line argument.\n", Argv[n]);
86             return SHELL_INVALID_PARAMETER;
87         }
88     }
89
90     if (gHelpUsage) {
91         return SHELL_SUCCESS;
92     }
93     if (InputFileName == NULL) {
94         Print(L"missing input file name.\n");
95         return SHELL_INVALID_PARAMETER;
96     }
97     if (OutputFileName == NULL) {
98         Print(L"missing output file name.\n");
99         return SHELL_INVALID_PARAMETER;
100    }
101    return SHELL_SUCCESS;
102 }

```

Figure 10.28: Command-line parsing in UnicodeDecode.c.

Lines 68–88

Walk through each of the command-line options. If the option starts with a ‘-’ character, then check for -?, -a2u, or -u2c. If not, generate an error. If the option does not

start with a ‘-’ character, then assign that option to the input file name or output file name, if they haven’t already been specified.

Lines 90–101

If help was requested, then ignore everything else. Otherwise, verify that both an input and output file names have been specified.

The next section opens the input file, checks for the file type, and then opens the output file, shown in Figures 10.29 and 10.30.

```

104     SHELL_STATUS
105     OpenFiles(VOID)
106     {
107         EFI_STATUS Status;
108
109         Status = gEfiShellProtocol->OpenFileByName(
110             InputFileName,
111             &InputFileHandle,
112             EFI_FILE_MODE_READ
113         );
114         if (Status != EFI_SUCCESS) {
115             Print(L"unable to open input file %s\n", InputFileName);
116             return SHELL_INVALID_PARAMETER;
117         }
118
119         if (!gAscii2Ucs && !gUcs2Ascii) {
120             CHAR16 ch;
121             UINTN sch = sizeof(ch);
122
123             Status = gEfiShellProtocol->ReadFile(InputFileHandle, &sch, &ch);
124             if (Status != EFI_SUCCESS) {
125                 Print(L"unable to read from input file\n");
126                 gEfiShellProtocol->CloseFile(InputFileHandle);
127                 return SHELL_INVALID_PARAMETER;
128             }
129             gEfiShellProtocol->SetFilePosition(InputFileHandle, 0);
130
131             if (ch == 0xff || ch == 0xfe) {
132                 gUcs2Ascii = TRUE;
133             } else if ((ch & 0xff00) == 0 || (ch & 0x0ff) == 0) {
134                 gUcs2Ascii = TRUE;
135             } else {
136                 gAscii2Ucs = TRUE;
137             }
138         }
    
```

Figure 10.29: Open the input files in UnicodeDecode.c.

Lines 109–111

Opens the input file for reading and saves the handle for later usage. If the file can’t be opened, display an error.

Lines 119–138

If the user didn't explicitly specify whether to do ASCII-to-Unicode or Unicode-to-ASCII conversion, try to auto-detect by reading the first two bytes of the input file. If it is the Unicode byte-order mark, convert from Unicode to ASCII. If either byte is a zero, then convert from Unicode to ASCII. Otherwise, assume it must be ASCII-to-Unicode. After reading the two bytes, reset the file position to the start of the file.

```

140     Status = gEfiShellProtocol->OpenFileByName(
141                                         OutputFileName,
142                                         &OutputFileHandle,
143                                         EFI_FILE_MODE_CREATE|EFI_FILE_MODE_WRITE
144                                         );
145     if (Status != EFI_SUCCESS) {
146         Print(L"unable to open output file %s\n", OutputFileName);
147         gEfiShellProtocol->CloseFile(InputFileHandle);
148         return SHELL_INVALID_PARAMETER;
149     }
150
151     return SHELL_SUCCESS;
152 }
153

```

Figure 10.30: Create the output file in UnicodeDecode.c.

Lines 140–151

Creates the output file and opens it for writing, saving the handle for later usage. If the file can't be created, close the input file and display an error. If `EFI_FILE_MODE_CREATE` is not specified, then the file must already exist.

These functions, shown in Figures 10.31 through 10.34, use a very simple algorithm, reading one character at a time from the input file, converting it, and then writing the converted character to the output file. The `GetFileSize()` function returns the file's entire size while `ReadFile()` and `WriteFile()` write `n` characters from/to the specified file.

In general, the translation between the two character sets is straightforward, since the first 256 characters in the UCS-2 (and UTF-16) Unicode character sets match those in the ISO-Latin-1 ASCII character set. However, there are several places where there is more than one Unicode character value that can be translated to a single ASCII character. In particular, the small and wide forms of letters and punctuation.

```

172 VOID
173 ConvertAsciiToUcs2(VOID)
174 {
175     EFI_STATUS Status;
176     UINT64 FileSize;
177     CHAR8 ch;
178     CHAR16 uch;
179     UINTN ReadSize;
180     UINTN WriteSize;
181
182     Status = gEfiShellProtocol->GetFileSize(InputFileHandle, &FileSize);
183     if (Status != EFI_SUCCESS || FileSize == 0) {
184         Print(L"unable to get input file size.\n");
185         return;
186     }
187
188     do {
189         ReadSize = sizeof(ch);
190
191         Status = gEfiShellProtocol->ReadFile(InputFileHandle, &ReadSize, &ch);
192         if (Status != EFI_SUCCESS) {
193             Print(L"trouble reading input file.\n");
194             return;
195         }
196
197         uch = ch;
198         WriteSize = sizeof(uch);
199
200         Status = gEfiShellProtocol->WriteFile(OutputFileHandle, &WriteSize, &uch);
201         if (Status != EFI_SUCCESS) {
202             Print(L"trouble writing output file.\n");
203             return;
204         }
205
206         FileSize -= ReadSize;
207     } while (FileSize > 0);
208 }

```

Figure 10.31: Convert ASCII to UCS-2 in UnicodeDecode.c.

Lines 182–186

Get the file size using `GetFileSize()`. If the file size can't be read or the file size is zero, exit with an error message.

Lines 189–195

Read a single ASCII character.

Lines 197–204

Write a single UCS-2 character.

Lines 206–207

Continue until we have converted all characters in the input file.

```

172     VOID
173     ConvertAsciiToUcs2(VOID)
174     {
175         EFI_STATUS Status;
176         UINT64 FileSize;
177         CHAR8 ch;
178         CHAR16 uch;
179         UINTN ReadSize;
180         UINTN WriteSize;
181
182         Status = gEfiShellProtocol->GetFileSize(InputFileHandle, &FileSize);
183         if (Status != EFI_SUCCESS || FileSize == 0) {
184             Print(L"unable to get input file size.\n");
185             return;
186         }
187
188         do {
189             ReadSize = sizeof(ch);
190
191             Status = gEfiShellProtocol->ReadFile(InputFileHandle, &ReadSize, &ch);
192             if (Status != EFI_SUCCESS) {
193                 Print(L"trouble reading input file.\n");
194                 return;
195             }
196
197             uch = ch;
198             WriteSize = sizeof(uch);
199
200             Status = gEfiShellProtocol->WriteFile(OutputFileHandle, &WriteSize, &uch);
201             if (Status != EFI_SUCCESS) {
202                 Print(L"trouble writing output file.\n");
203                 return;
204             }
205
206             FileSize -= ReadSize;
207         } while (FileSize > 0);
208     }

```

Figure 10.32: Convert UCS-2 to ASCII in UnicodeDecode.c.

Lines 268–272

Get the file size using `GetFileSize()`. If the file size can't be read or the file size is zero, then exit with an error message.

Lines 275–281

Read a single character using `ReadFile()`.

Lines 283–284

Up until this point, this function has matched the flow of `ConvertAsciiToUnicode()`. But here, the function calls `ConvertUcs2ToAsciiChar()` to handle the cases where the UCS-2 encoded character is outside the range of ASCII characters. If this function returns a null character ('\0'), then don't write it.

Lines 287–292

Write a single character using `WriteFile()`.

Lines 294–295

If there are no more files left to write, then exit.

```

210     CHAR8
211     ConvertUcs2ToAsciiChar(IN CHAR16 uch)
212     {
213         switch(uch) {
214             case UNICODE_FULLWIDTH_CENT_SIGN:    return 0xa2; break;
215             case UNICODE_FULLWIDTH_POUND_SIGN:  return 0xa3; break;
216             case UNICODE_FULLWIDTH_NOT_SIGN:   return 0xac; break;
217             case UNICODE_FULLWIDTH_MACRON:    return 0xaf; break;
218             case UNICODE_FULLWIDTH_BROKEN_BAR: return 0xa6; break;
219             case UNICODE_FULLWIDTH_YEN_SIGN:   return 0xa5; break;
220
221             case UNICODE_SMALL_COMMA:          return ','; break;
222             case UNICODE_SMALL_FULL_STOP:     return '.'; break;
223             case UNICODE_SMALL_SEMICOLON:     return ';'; break;
224             case UNICODE_SMALL_COLON:         return ':'; break;
225             case UNICODE_SMALL_QUESTION_MARK: return '?'; break;
226             case UNICODE_SMALL_EXCLAMATION_MARK: return '!'; break;
227             case UNICODE_SMALL_LEFT_PARENTHESIS: return '('; break;
228             case UNICODE_SMALL_RIGHT_PARENTHESIS: return ')'; break;
229             case UNICODE_SMALL_LEFT_CURLY_BRACKET: return '['; break;
230             case UNICODE_SMALL_RIGHT_CURLY_BRACKET: return ']'; break;
231             case UNICODE_SMALL_NUMBER_SIGN:    return '#'; break;
232             case UNICODE_SMALL_AMPERSAND:      return '&'; break;
233             case UNICODE_SMALL_ASTERISK:       return '*'; break;
234             case UNICODE_SMALL_PLUS_SIGN:     return '+'; break;
235             case UNICODE_SMALL_HYPHEN_MINUS:  return '-'; break;
236             case UNICODE_SMALL_LESS_THAN_SIGN: return '<'; break;
237             case UNICODE_SMALL_GREATER_THAN_SIGN: return '>'; break;
238             case UNICODE_SMALL_EQUALS_SIGN:   return '='; break;
239             case UNICODE_SMALL_REVERSE_SOLIDUS: return '\\'; break;
240             case UNICODE_SMALL_DOLLAR_SIGN:   return '$'; break;
241             case UNICODE_SMALL_PERCENT_SIGN:  return '%'; break;
242             case UNICODE_SMALL_COMMERCIAL_AT: return '@'; break;
243
244             case UNICODE_ZERO_WIDTH_NON_BREAKING_SPACE: return '\0';
245             case UNICODE_BYTE_ORDER_CHAR:      return '\0';

```

Figure 10.33: Converting extended Unicode characters to ASCII in `UnicodeDecode.c`.

Lines 213–245

There are several cases where the Unicode specification describes “small” versions of punctuation characters. In a few other cases, there are non-printing characters, such as the byte-order marks or the non-breaking space.

```
247     default:
248         if (uch > 0xff00 && uch < 0xff5f) {
249             return (CHAR8)((uch & 0xff) + ' ');
250         } else if (uch < 0x100) {
251             return (CHAR8)(uch & 0xff);
252         }
253         break;
254     }
255     return '?';
256 }
```

Figure 10.34: Converting wide UCS-2 characters to ASCII in UnicodeDecode.c.

Lines 247–252

If the UCS-2 character is a wide version of the ASCII character, or an ASCII character, then return the ASCII equivalent, between 0x00 – 0xFF.

Lines 255

Otherwise, return the ‘?’ character, when replacing a Unicode character that does not have an ASCII equivalent.

The Component Information (.inf) File

The component information (.inf) file shown in Figure 10.35 describes the component and the information necessary to build a single application, driver, or library. This includes source files, build options, libraries, and so on.

```
1  ## @file
2  #  UnicodeDecode - convert files to/from ASCII & Unicode.
3  #
4  #
5  ##
6
7  [Defines]
8      INF_VERSION          = 0x00010006
9      BASE_NAME             = ucd
10     FILE_GUID              = 91c83945-314c-468c-af0f-6f0902908d35
11     MODULE_TYPE            = UEFI_APPLICATION
12     VERSION_STRING         = 0.1
13     ENTRY_POINT             = ShellCEEntryLib
14
15 [Sources]
16     UnicodeDecode.c
17
18 [Protocols]
19     gEfiShellProtocolGuid
20
21 [Packages]
22     MdePkg/MdePkg.dec
23     ShellPkg/ShellPkg.dec
24
25 [LibraryClasses]
26     UefiLib
27     ShellCEEntryLib
28     ShellLib
```

Figure 10.35: UnicodeDecode Component Information File.

Lines 1–28

This is a fairly simple component information file, where only `BASE_NAME` and `FILE_GUID` have been updated from the one found in the `HelloWorld` UEFI Shell application.

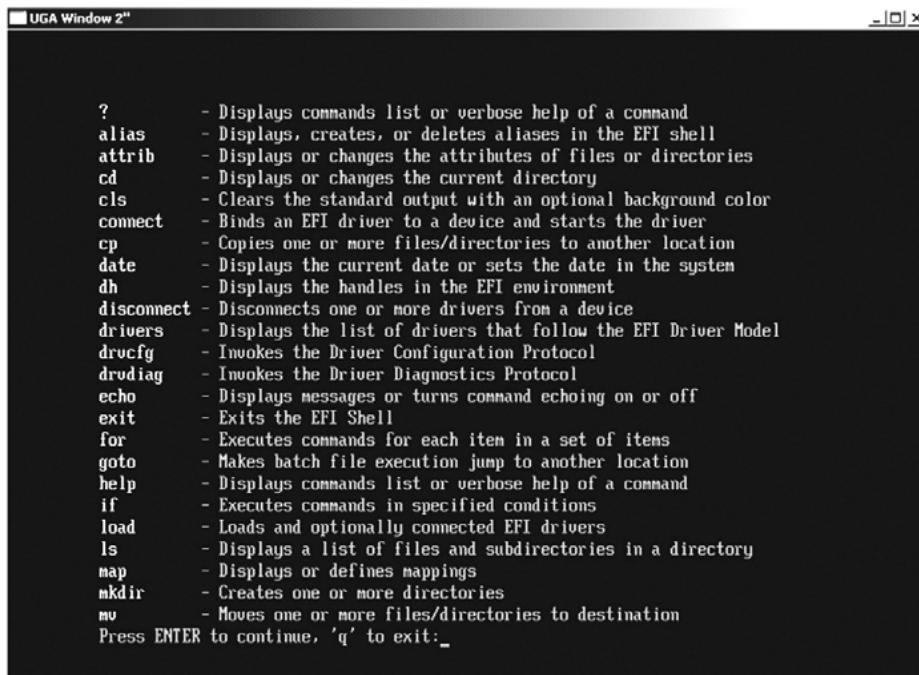
Chapter 11

Managing UEFI Drivers Using the Shell

He has half the deed done who has made a beginning.

—Horace

Several UEFI Shell commands can be used to help debug UEFI drivers. These UEFI Shell commands are already documented in the *UEFI 2.0 Shell Specification*, so the full capabilities of the UEFI Shell commands are not discussed here. There is also a built-in UEFI Shell command called `help` that provides a detailed description of an UEFI Shell command. Figure 11.1 shows the results of issuing the “`help -b`” command.



The screenshot shows a terminal window titled "UGA Window 2". The window contains a list of UEFI Shell commands and their descriptions. The commands listed are:

- ? - Displays commands list or verbose help of a command
- alias - Displays, creates, or deletes aliases in the EFI shell
- attrib - Displays or changes the attributes of files or directories
- cd - Displays or changes the current directory
- cls - Clears the standard output with an optional background color
- connect - Binds an EFI driver to a device and starts the driver
- cp - Copies one or more files/directories to another location
- date - Displays the current date or sets the date in the system
- dh - Displays the handles in the EFI environment
- disconnect - Disconnects one or more drivers from a device
- drivers - Displays the list of drivers that follow the EFI Driver Model
- drvcfg - Invokes the Driver Configuration Protocol
- drudiaq - Invokes the Driver Diagnostics Protocol
- echo - Displays messages or turns command echoing on or off
- exit - Exits the EFI Shell
- for - Executes commands for each item in a set of items
- goto - Makes batch file execution jump to another location
- help - Displays commands list or verbose help of a command
- if - Executes commands in specified conditions
- load - Loads and optionally connects EFI drivers
- ls - Displays a list of files and subdirectories in a directory
- map - Displays or defines mappings
- mkdir - Creates one or more directories
- mv - Moves one or more files/directories to destination

At the bottom of the list, it says "Press ENTER to continue, 'q' to exit:_".

Figure 11.1: The `help -b` Command.

Testing Specific Protocols

Table 11.1 lists UEFI Shell commands that can be used to test and debug UEFI drivers along with the protocol and/or service exercised.

Table 11.1: UEFI Shell Commands

Command	Protocol Tested	Service Tested
Load -nc		DriverEntryPoint Supported()
Load	Driver Binding	DriverEntryPoint
	Driver Binding	Supported()
	Driver Binding	Start()
Unload	Loaded Image	Unload()
Connect	Driver Binding	Supported()
	Driver Binding	Start()
Disconnect	Driver Binding	Stop()
Reconnect	Driver Binding	Supported()
	Driver Binding	Start()
	Driver Binding	Stop()
Drivers	Component Name	GetDriverName()
Devices	Component Name	GetDriverName()
	Component Name	GetControllerName()
DevTree	Component Name	GetControllerName()
Dh -d	Component Name	GetDriverName()
	Component Name	GetControllerName()
DrvCfg -s	Driver Configuration	SetOptions()
DrvCfg -f	Driver Configuration	ForceDefaults()
DrvCfg -v	Driver Configuration	OptionsValid()
DrvDiag	Driver Diagnostics	RunDiagnostics()

Other tests can be performed from within the UEFI Shell, as listed in Table 11.2. These are not testing a specific protocol, but are testing for other coding practices.

Table 11.2: Other Shell Testing Procedures

Shell Command sequence	What it tests
Shell> Memmap Shell> Dh	Tests for incorrectly matched up DriverEntryPoint and Unload() functions.
Shell> Load DriverName.efi Shell> Memmap Shell> Dh	This will catch memory allocation that is not unallocated, protocols that are installed and not uninstalled, and so on.
Shell> Unload DriverHandle Shell> Memmap Shell> Dh	
Shell> Memmap Shell> Connect DeviceHandle DriverHandle Shell> Memmap	Tests for incorrectly matched up DriverBinding Start() and Stop() functions. This will catch memory allocation that is not unallocated.
Shell> Disconnect DeviceHandle DriverHandle Shell> Memmap Shell> Reconnect DeviceHandle Shell> Memmap	
Shell> dh Shell> Connect DeviceHandle DriverHandle Shell> dh	Tests for incorrectly matched up DriverBinding Start() and Stop() functions. This will catch protocols that are installed and not uninstalled.
Shell> Disconnect DeviceHandle DriverHandle Shell> dh Shell> Reconnect DeviceHandle Shell> dh	
Shell> OpenInfo DeviceHandle Shell> Connect DeviceHandle DriverHandle Shell> OpenInfo DeviceHandle Shell> Disconnect DeviceHandle DriverHandle	Tests for incorrectly matched up DriverBinding Start() and Stop() functions. This will catch protocols that are opened and not closed.
Shell> OpenInfo DeviceHandle Shell> Reconnect DeviceHandle Shell> OpenInfo DeviceHandle	

Loading and Unloading UEFI Drivers

Two UEFI Shell commands are available to load and start UEFI drivers, Load and LoadPciRom. The UEFI Shell command that can be used to unload a UEFI driver if it is unloadable is Unload.

Load

The Load command loads a UEFI driver from a file. UEFI driver files typically have an extension of `.efi`. This command has one important option, the `-nc` (“No Connect”) option, for UEFI driver developers. When the Load command is used without the `-nc` option, the loaded driver is automatically connected to any devices in the system that it is able to manage. This means that the UEFI driver’s entry point is executed and then the EFI Boot Service `ConnectController()` is called. If the UEFI driver produces the Driver Binding Protocol in the driver’s entry point, then the `ConnectController()` call exercises the `Supported()` and `Start()` services of Driver Binding Protocol that was produced.

If the `-nc` option is used with the Load command, then this automatic connect operation is not performed. Instead, only the UEFI driver’s entry point is executed. When the `-nc` option is used, the UEFI Shell command Connect can be used to connect the UEFI driver to any devices in the system that it is able to manage. The Load command can also take wild cards, so multiple UEFI drivers can be loaded at the same time.

The following are some examples of the Load command.

Example 1 loads and does not connect the UEFI driver image `EfiDriver.efi`. This example exercises only the UEFI driver’s entry point:

```
fs0:> Load -nc EfiDriver.efi
```

Example 2 loads and connects the UEFI driver image called `EfiDriver.efi`. This example exercises the UEFI driver’s entry point and the `Supported()` and `Start()` functions of the Driver Binding Protocol:

```
fs0:> Load EfiDriver.efi
```

Example 3 loads and connects all the UEFI drivers with an `.efi` extension from `fs0:`, exercising the UEFI driver entry points and their `Supported()` and `Start()` functions of the Driver Binding Protocol:

```
fs0:> Load *.efi
```

LoadPciRom

The LoadPciRom command simulates the load of a PCI option ROM by the PCI bus driver. This command parses a ROM image that was produced with the `EfiRom` build utility. Details on the `EfiRom` build utility can be found at www.tianocore.org. The LoadPciRom command finds all the UEFI drivers in the ROM image and attempts to load and start all the UEFI drivers. This command helps test the ROM image before it

is burned into a PCI adapter's ROM. No automatic connects are performed by this command, so only the UEFI driver's entry point is exercised by this command. The UEFI Shell command Connect must be used for the loaded UEFI drivers to start managing devices. The example below loads and calls the entry point of all the UEFI drivers in the ROM file called `MyAdapter.ROM`:

```
fs0:> LoadPciRom MyAdapter.ROM
```

Unload

The Unload command unloads a UEFI driver if it is unloadable. This command takes a single argument that is the image handle number of the UEFI driver to unload. The Dh -p Image command and the Drivers command can be used to search for the image handle of the driver to unload. Once the image handle number is known, an unload operation can be attempted. The Unload command may fail for one of the following two reasons:

1. The UEFI driver may not be unloadable, because UEFI drivers are not required to be unloadable.
2. The UEFI driver may be unloadable, but it may not be able to be unloaded right now.

Some UEFI drivers may need to be disconnected before they are unloaded. They can be disconnected with the Disconnect command. The following example unloads the UEFI driver on handle 27. If the UEFI driver on handle 27 is unloadable, it will have registered an `Unload()` function in its Loaded Image Protocol. This command exercises the UEFI driver's `Unload()` function.

```
Shell> Unload 27
```

Connecting UEFI Drivers

Three UEFI Shell commands can be used to test the connecting of UEFI drivers to devices: Connect, Disconnect, and Reconnect. These commands have many options. A few are described in the following sections.

Connect

The Connect command can be used to connect all UEFI drivers to all devices in the system or connect UEFI drivers to a single device. The following are some examples of the Connect command.

Example 1 connects all drivers to all devices:

```
fs0:> Connect -r
```

Example 2 connects all drivers to the device that is abstracted by handle 23:

```
fs0:> Connect 23
```

Example 3 connects the UEFI driver on handle 27 to the device that is abstracted by handle 23:

```
fs0:> Connect 23 27
```

Disconnect

The Disconnect command stops UEFI drivers from managing a device. The following are some examples of the Disconnect command.

Example 1 disconnects all drivers from all devices. However, the use of this command is not recommended, because it also disconnects all the console devices:

```
fs0:> Disconnect -r
```

Example 2 disconnects all the UEFI drivers from the device represented by handle 23:

```
fs0:> Disconnect 23
```

Example 3 disconnects the UEFI driver on handle 27 from the device represented by handle 23:

```
fs0:> Disconnect 23 27
```

Example 4 destroys the child represented by handle 32. The UEFI driver on handle 27 produced the child when it started managing the device on handle 23:

```
fs0:> Disconnect 23 27 32
```

Reconnect

The Reconnect command is the equivalent of executing the Disconnect and Connect commands back to back. The Reconnect command is the best command for testing the Driver Binding Protocol of UEFI drivers. This command tests the `Supported()`, `Start()`, and `Stop()` services of the Driver Binding Protocol. The `Reconnect -r` command tests the Driver Binding Protocol for every UEFI driver that follows the UEFI Driver Model. Use this command before an UEFI driver is loaded to

verify that the current set of drivers pass the Reconnect -r test, and then load the new UEFI driver and rerun the Reconnect -r test. A UEFI driver is not complete until it passes this interoperability test with the UEFI core and the full set of UEFI drivers. The following are some examples of the Reconnect command.

Example 1 reconnects all the UEFI drivers to the device handle 23:

```
fs0:> Reconnect 23
```

Example 2 reconnects the UEFI driver on handle 27 to the device on handle 23:

```
fs0:> Reconnect 23 27
```

Example 3 reconnects all the UEFI drivers in the system:

```
fs0:> Reconnect -r
```

Driver and Device Information

Five UEFI Shell commands can be used to dump information about the UEFI drivers that follow the UEFI Driver Model. Each of these commands shows information from a slightly different perspective.

Drivers

The Drivers command lists all the UEFI drivers that follow the UEFI Driver Model. It uses the GetDriverName () service of the Component Name protocol to retrieve the human-readable name of each UEFI driver if it is available. It also shows the file path from which the UEFI driver was loaded. As UEFI drivers are loaded with the Load command, they will appear in the list of drivers produced by the Drivers command. The Drivers command can also show the name of the UEFI driver in different languages. The following are some examples of the Drivers command.

Example 1 shows the Drivers command being used to list the UEFI drivers in the default language:

```
fs0:> Drivers
```

Example 2 shows the driver names in Spanish:

```
fs0:> Drivers -lspa
```

Devices

The Devices command lists all the devices that are being managed or produced by UEFI drivers that follow the UEFI Driver Model. This command uses the `GetControllerName()` service of the Component Name protocol to retrieve the human-readable name of each device that is being managed or produced by UEFI drivers. If a human-readable name is not available, then the EFI device path is used. The following are some examples of the Devices command.

Example 1 shows the Devices command being used to list the UEFI drivers in the default language:

```
fs0:> Devices
```

Example 2 shows the device names in Spanish:

```
fs0:> Devices -lspa
```

DevTree

Similar to the Devices command, the DevTree command lists all the devices being managed by UEFI drivers that follow the UEFI Driver Model. This command uses the `GetControllerName()` service of the Component Name Protocol to retrieve the human-readable name of each device that is being managed or produced by UEFI drivers. If the human-readable name is not available, then the EFI device path is used. This command also shows the parent/child relationships between all of the devices visually by displaying them in a tree structure. The lower a device is in the tree of devices, the more the device name is indented. The following are some examples of the DevTree command.

Example 1 displays the device tree with the device names in the default language:

```
fs0:> DevTree
```

Example 2 displays the device tree with the device names in Spanish:

```
fs0:> DevTree -lspa
```

Example 3 displays the device tree with the device names shown as EFI device paths:

```
fs0:> DevTree -d
```

Dh -d

The Dh -d command provides a more detailed view of a single driver or a single device than the Drivers, Devices, and DevTree commands. If a driver binding handle is used with the Dh -d command, then a detailed description of that UEFI driver is provided along with the devices that the driver is managing and the child devices that the driver has produced. If a device handle is used with the Dh -d command, then a detailed description of that device is provided along with the drivers that are managing that device, that device's parent controllers, and the device's child controllers. If the Dh -d command is used without any parameters, then detailed information on all of the drivers and devices is displayed. The following are some examples of the Dh -d command.

Example 1 displays the details on the UEFI driver on handle 27:

```
fs0:> Dh -d 27
```

Example 2 displays the details for the device on handle 23:

```
fs0:> Dh -d 23
```

Example 3 shows details on all the drivers and devices in the system:

```
fs0:> Dh -d
```

OpenInfo

The OpenInfo command provides detailed information about a device handle that is being managed by one or more UEFI drivers that follow the UEFI Driver Model. The OpenInfo command displays each protocol interface installed on the device handle and the list of agents that have opened that protocol interface with the `OpenProtocol()` Boot Service. This command can be used in conjunction with the Connect, Disconnect, and Reconnect commands to verify that an UEFI driver is opening and closing protocol interfaces correctly. The following example shows the OpenInfo command being used to display the list of protocol interfaces on device handle 23 along with the list of agents that have opened those protocol interfaces:

```
fs0:> OpenInfo 23
```

Testing the Driver Configuration and Driver Diagnostics Protocols

The UEFI Shell provides a command that can be used to test the Driver Configuration Protocol, DrvCfg, and one that can be used to test the Driver Diagnostics Protocol, DrvDiag.

DrvCfg

The DrvCfg command provides the services that are required to test the Driver Configuration Protocol implementation of a UEFI driver. This command can show all the devices that are being managed by UEFI drivers that support the Driver Configuration Protocol. The Devices and Drivers commands also show the drivers that support the Driver Configuration Protocol and the devices that those drivers are managing or have produced. Once a device has been chosen, the DrvCfg command can be used to invoke the SetOptions(), ForceDefaults(), or OptionsValid() services of the Driver Configuration Protocol. The following are examples of the DrvCfg command.

Example 1 displays all the devices that are being managed by UEFI drivers that support the Driver Configuration Protocol:

```
fs0:> DrvCfg
```

Example 2 forces defaults on all the devices in the system:

```
fs0:> DrvCfg -f
```

Example 3 validates the options on all the devices in the system:

```
fs0:> DrvCfg -v
```

Example 4 invokes the SetOptions() service of the Driver Configuration Protocol for the driver on handle 27 and the device on handle 23:

```
fs0:> DrvCfg -s 23 27
```

DrvDiag

The DrvDiag command provides the ability to test all the services of the Driver Diagnostics Protocol that are produced by a UEFI driver. This command shows the devices that are being managed by UEFI drivers that support the Driver Diagnostics Protocol. The Devices and Drivers commands also show the drivers that support the Driver Diagnostics Protocol and the devices that those drivers are managing or have produced. Once a device has been chosen, the DrvDiag command can be used to invoke the

RunDiagnostics () service of the Driver Diagnostics Protocol. The following are some examples of the DrvDiag command.

Example 1 displays all the devices that are being managed by UEFI drivers that support the Driver Diagnostics Protocol:

```
fs0:> DrvDiag
```

Example 2 invokes the RunDiagnostics () service of the Driver Diagnostics Protocol in standard mode for the driver on handle 27 and the device on handle 23:

```
fs0:> DrvDiag -s 23 27
```

Example 3 invokes the RunDiagnostics () service of the Driver Diagnostics Protocol in manufacturing mode for the driver on handle 27 and the device on handle 23:

```
fs0:> DrvDiag -m 23 27
```

Debugging Code Statements

Every module has a debug (check) build and a clean build. The debug build includes code for debug that will not be included in normal clean production builds. A debug build is enabled when the identifier `EFI_DEBUG` exists. A clean build is defined as when the `EFI_DEBUG` identifier does not exist.

The following debug macros can be used to insert debug code into a checked build. This debug code can greatly reduce the amount of time it takes to root cause a bug. These macros are enabled only in a debug build, so they do not take up any executable space in the production build. Table 11.3 describes the debug macros that are available.

Table 11.3: Available Debug Macros

Debug Macro	Description
ASSERT (Expression)	For check builds, if <code>Expression</code> evaluates to <code>FALSE</code> , a diagnostic message is printed and the program is aborted. Aborting a program is usually done via the <code>EFI_BREAKPOINT ()</code> macro. For clean builds, <code>Expression</code> does not exist in the program and no action is taken. Code that is required for normal program execution should never be placed inside an <code>ASSERT</code> macro, because the code will not exist in a production build.
ASSERT_EFI_ERROR (Status)	For check builds, an assert is generated if <code>Status</code> is an error. This macro is equivalent to <code>ASSERT (!EFI_ERROR (Status))</code> but is easier to read.
DEBUG ((ErrorLevel, String, ...))	For check builds, <code>String</code> and its associated arguments will be printed if the <code>ErrorLevel</code> of the macro is active. See Table 11.4 for a definition of the <code>ErrorLevel</code> values.
DEBUG_CODE (Code)	For check builds, <code>Code</code> is included in the build. <code>DEBUG_CODE (</code> is on its own line and indented like normal code. All the debug code follows on subsequent lines and is indented an extra level. The <code>)</code> is on the line following all the code and indented at the same level as <code>DEBUG_CODE (</code> .
EFI_BREAKPOINT ()	On a check build, inserts a break point into the code.
DEBUG_SET_MEM (Address, Length)	For a check build, initializes the memory starting at <code>Address</code> or <code>Length</code> bytes with the value <code>BAD_POINTER</code> . This initialization is done to enable debug of code that uses memory buffers that are not initialized.
CR (Record, TYPE, Field, Signature)	The containing record macro returns a pointer to <code>TYPE</code> when given the structure's <code>Field</code> name and a pointer to it (<code>Record</code>). The <code>CR</code> macro returns the <code>TYPE</code> pointer for check and production builds. For a check build, an <code>ASSERT ()</code> is generated if the <code>Signature</code> field of <code>TYPE</code> is not equal to the <code>Signature</code> in the <code>CR ()</code> macro.

The `ErrorLevel` parameter referenced in the `DEBUG ()` macro allows a UEFI driver to assign a different error level to each debug message. Critical errors should always be sent to the standard error device. However, informational messages that are used

only to debug a driver should be sent to the standard error device only if the user wants to see those specific types of messages. The UEFI Shell supports the `Err` command that allows the user to set the error level. The UEFI Boot Maintenance Manager allows the user to enable and select a standard error device. It is recommended that a serial port be used as a standard error device during debug so the messages can be logged to a file with a terminal emulator. Table 11.4 contains the list of error levels that are supported in the UEFI Shell. Other levels are usable, but not defined for a specific area.

Note: `DEBUG ((ErrorLevel, String, ...))` is not universally supported. Some EFI-compliant systems may not print out the message.

Table 11.4: Error Levels

Mnemonic	Value	Description
<code>EFI_D_INIT</code>	0x00000001	Initialization messages
<code>EFI_D_WARN</code>	0x00000002	Warning messages
<code>EFI_D_LOAD</code>	0x00000004	Load events
<code>EFI_D_FS</code>	0x00000008	EFI file system messages
<code>EFI_D_POOL</code>	0x00000010	EFI pool allocation and free messages
<code>EFI_D_PAGE</code>	0x00000020	EFI page allocation and free messages
<code>EFI_D_INFO</code>	0x00000040	Informational messages
<code>EFI_D_VARIABLE</code>	0x00000100	EFI variable service messages
<code>EFI_D_BM</code>	0x00000400	UEFI boot manager messages
<code>EFI_D_BLKIO</code>	0x00001000	EFI Block I/O Protocol messages
<code>EFI_D_NET</code>	0x00004000	EFI Simple Network Protocol, PXE base code, BIS messages
<code>EFI_D_UNDI</code>	0x00010000	UNDI driver messages
<code>EFI_D_LOADFILE</code>	0x00020000	Load File Protocol messages
<code>EFI_D_EVENT</code>	0x00080000	EFI Event Services messages
<code>EFI_D_ERROR</code>	0x80000000	Critical error messages

POST Codes

If an UEFI driver is being developed that cannot make use of the `DEBUG ()` and `ASSERT ()` macros, then a different mechanism must be used to help in the debugging process. Under these conditions, it is usually sufficient to send a small amount of output to a device to indicate what portions of an UEFI driver have executed and where error conditions have been detected. A few possibilities are presented below, but

many others are possible depending on the devices that may be available on a specific platform. It is important to note that these mechanisms are useful during driver development and debug, but they should never be present in production versions of UEFI drivers because these types of devices are not present on all platforms.

The first possibility we will describe here is to use a POST card.

Post Card Debug

A POST card is an add-in card adapter that displays the hex value of an 8-bit I/O write cycle to address 0x80 (and sometimes 0x81). If a UEFI driver can depend on the PCI Root Bridge I/O Protocol being present, then the driver can use the services of the PCI Root Bridge I/O Protocol to send an 8-bit I/O write cycle to address 0x80. A driver can also use the services of the PCI I/O Protocol to write to address 0x80, as long as the pass-through BAR value is used. Figure 11.2 shows how the PCI Root Bridge I/O and PCI I/O Protocols can be used to send a value to a POST card.

```

EFI_STATUS Status;
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *PciRootBridgeIo;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 Value;

Value = 0xAA;
Status = PciRootBridgeIo->Io.Write (
    PciRootBridgeIo,
    EfiPciWidthUint8,
    0x80,
    1,
    &Value
);

Value = 0xAA;
Status = PciIo->Io.Write (
    PciIo,
    EfiPciIoWidthUint8,
    EFI_PCI_IO_PASS_THROUGH_BAR,
    0x80,
    1,
    &Value
);

```

Figure 11.2: POST Code Examples

Text-Mode VGA Frame Buffer

The next possibility is a text-mode VGA frame buffer. If a system initializes the text-mode VGA display by default before the UEFI driver executes, then the UEFI driver can make use of the PCI Root Bridge I/O or PCI I/O Protocols to write text characters to the text-mode VGA display directly. Figure 11.3 shows how the PCI Root Bridge I/O and PCI I/O Protocols can be used to send the text message “ABCD” to the text-mode VGA frame buffer. Some systems do not have a VGA controller, so this solution will not work on all systems.

```

EFI_STATUS
EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL *PciRootBridgeIo;
EFI_PCI_IO_PROTOCOL *PciIo;
UINT8 *Value;

Value = {'A', 0x0f, 'B', 0x0f, 'C', 0x0f, 'D', 0x0f};

Status = PciRootBridgeIo->Mem.Write (
    PciRootBridgeIo,
    EfiPciWidthUint8,
    0xB8000,
    8,
    Value
);

Status = PciIo->Mem.Write (
    PciIo,
    EfiPciIoWidthUint8,
    EFI_PCI_IO_PASS_THROUGH_BAR,
    0xB8000,
    8,
    Value
);

```

Figure 11.3: VGA Display Examples

Other Options

Another option that can be used if the UEFI driver being developed cannot make use of the `DEBUG()` and `ASSERT()` macros is to use some type of byte-stream-based device. This device could include a UART or an SMBus, for example. Like the POST

card, the idea is to use the services of the PCI Root Bridge I/O or PCI I/O Protocols to initialize and send characters to the byte-stream device.

Many EFI-compliant implementations allow for the use of a COM cable to send debug information to another system. This allows the developer or tester to see debug code statements and other output from a separate system.

Appendix A

Security Considerations

We will bankrupt ourselves in the vain search for absolute security.

—Dwight D. Eisenhower

This appendix describes some options for hardening the integrity of the UEFI Shell.

UEFI Shell Binary Integrity

Recall that the UEFI Shell can be stored in the platform ROM, on disk, or across the network. For the latter two scenarios, the integrity of the UEFI Shell may be a concern in that a possibly hostile agent in the operating system may corrupt the UEFI system partition or a man-in-the-middle (MITM) attack may occur during the network download of the UEFI Shell.

Signing of the UEFI Shell is one option to handle this case of ensuring integrity of code introduced into the platform, especially from a mutable disk or across the network.

Overview

The UEFI specification provides a standard format for executables. These executables may be located on unsecured media (such as a hard drive or unprotected flash device) or may be delivered via an unsecured transport layer (such as a network) or originate from a unsecured port (such as an Express Card device or USB device). In each of these cases, the system provider may decide to authenticate either the origin of the executable or its integrity (that is, that it has not been tampered with).

The UEFI specification describes a means of generating a digital signature for a UEFI executable, embedding that digital signature within the UEFI executable, and automatically verifying that the digital signature is from the authorized source. It is to allay concerns of the BIOS vendors regarding the wide availability of the firmware image construction tools and documentation. The firmware tools and the encoding can be made public, as would the security scheme used in the UEFI specification. The privacy would only involve the private key used by the OEM in his factory to sign the image.

One of the main goals for pre-operating-system security is to ensure various integrity goals. These goals include protecting the platform firmware from possibly errant or malicious third-party content. One way to meet this goal is to ensure that binary executable content, such as a shell not built into the platform firmware itself, comes from a well-known source. Cryptographic signatures applied to the binary with

the platform firmware perform the verification action prior to performing the LoadImage/StartImage action on the executable.

The platform construction and integrity precepts used to evaluate the system can be derived from a more formal, commercial integrity model, such as Clark-Wilson. More information can be found in Clark, David D. and Wilson, David R. “*A Comparison of Commercial and Military Computer Security Policies*” in *Proceedings of the 1987 IEEE Symposium on Research in Security and Privacy (SP'87)*, May 1987, Oakland, CA, IEEE Press, pp. 184–193.

Signed Executable Overview

Figure A.1 shows the format of an original sample UEFI executable. The UEFI executable format is compatible with *Microsoft Portable Executable (PE) and Common Object File Format (COFF) Specification, Version 8.0*.

An executable primarily consists of file header, section table, and section data. See Chapters 3, 4, 5, and 6 in the PE and COFF Specification for more detailed executable structure description.

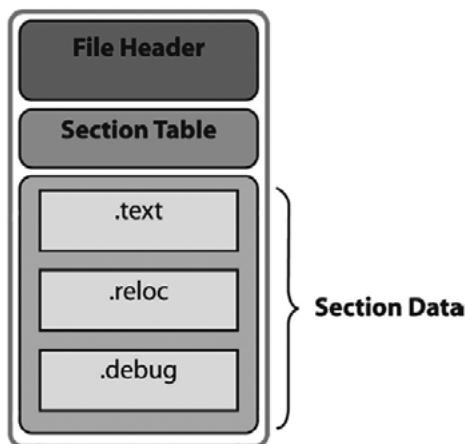


Figure A. 1: Sample UEFI Executable Format.

A signed UEFI executable encapsulates a certificate, which is used during the executable load process to detect unauthorized tampering or ensure its origination. The certificate may contain a digital signature used for validating the driver. Figure A.2 illustrates how a certificate is embedded in the PE/COFF file.

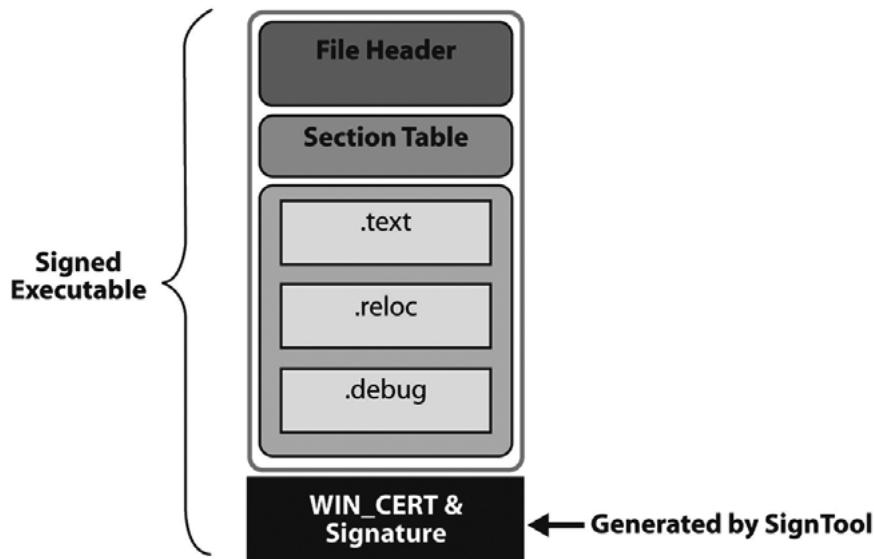


Figure A.2: Signed UEFI Executable.

The signature of the executable is generated by a sign tool, which appends the `WIN_CERT` data and signature information of the whole executable to the end of the original file.

Digital Signature

As a rule, digital signatures require two pieces: the *data* (often referred to as the *message*) and a *public/private key pair*. In order to create a digital signature, the message is processed by a hashing algorithm to create a hash value. The hash value is, in turn, encrypted using a signature algorithm and the private key to create the digital signature. Figure A.3 illustrates the process to create a digital signature.

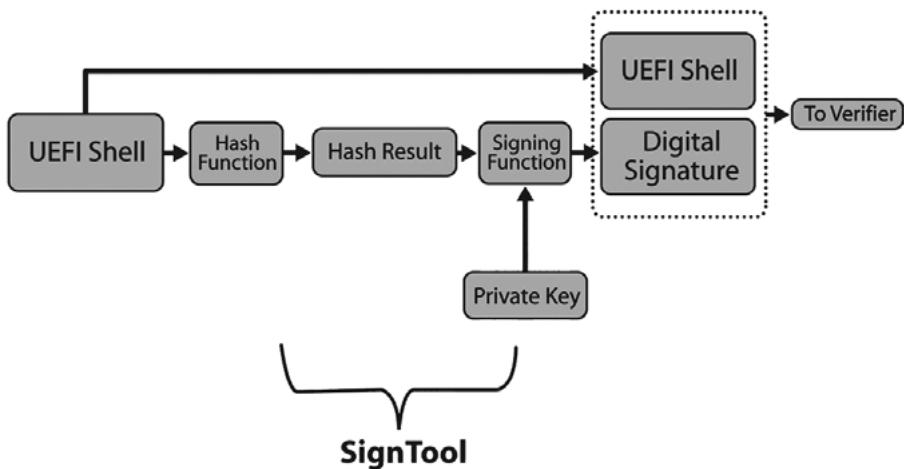


Figure A.3: Creating a digital signature.

In order to verify a signature, two pieces of data are required: the *original message* and the *public key*. First, the hash must be calculated exactly as it was calculated when the signature was created. Then the digital signature is decoded using the public key and the result is compared against the computed hash. If the two are identical, then you can be sure that message data is the one originally signed and it has not been tampered with. Figure A.4 illustrates the process of verifying a digital signature.

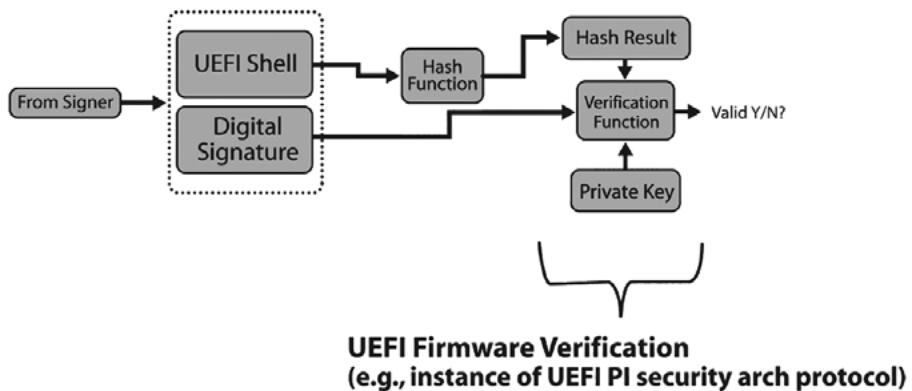


Figure A.4: Verifying a digital signature.

Therefore, the signature process dealing with the UEFI executable can be described briefly as follows:

```
Hash = SHA-256 (Executable File contents)
Signature = RSA_SIGN (Private_Key, Hash)
```

Signed Executable Processing

Signed executable processing is accomplished by an OS-present application (SignTool.exe) that calculates the signature of a UEFI executable file (SHA-256 and PKCS1V15 RSA_Sign) and then embeds that signature into the executable. The following subsections define the responsibilities at each stop along the signed executable processing path.

Signed Executable Generation Application (SignTool)

The SignTool application is an OS-present application that is responsible for locating the executable and output a signed executable. SignTool creates an Authenticode-formatted signature block within the PE/COFF image.

To calculate the executable digital signature that loaded into memory, SignTool must:

1. For PE/COFF headers and sections, align them on the appropriate section alignment for the architecture, as stipulated in the *SectionAlignment* field of header. Pad inter-section regions (that is, in case file alignment and section alignment are not the same) with zeroes. Some sections, like debug and security may be ignorable.
2. Skip the following fields in the calculation:
 - CheckSum
 - *Certificate Data Directory* (the fifth entry in data directory)

As SignTool's input, the whole well-formatted executable is hashed and signed by SignTool. The final signature is appended to the original executable file to generate a signed executable file. Figure A.5 illustrates the layout of a signed executable.

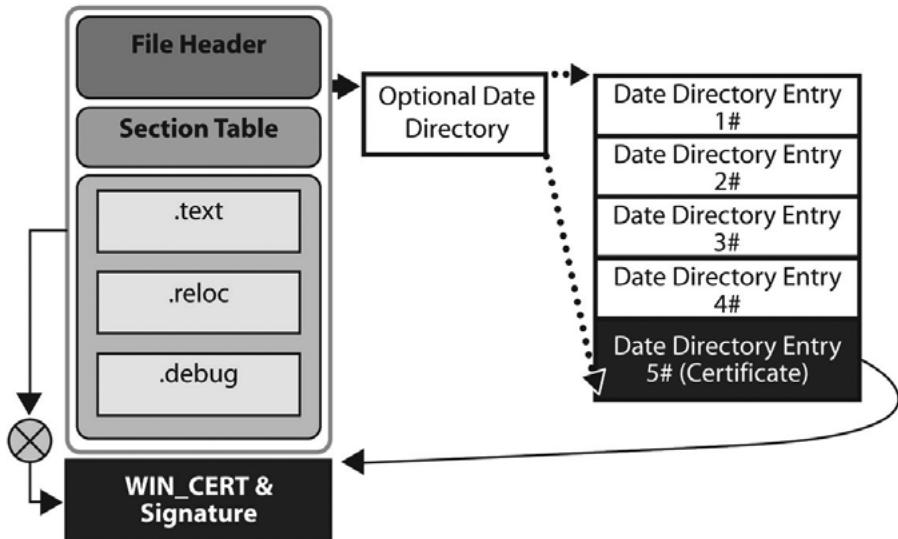


Figure A.5: Layout of signed executable.

UEFI Load Image

During the processing of UEFI images, the UEFI driver verification is processed in the UEFI implementation, such as a UEFI PI core. One of the UEFI core's responsibilities is to load UEFI executables into physical memory; it may utilize an instance of the Security Architectural Protocol (SAP) to verify their signatures in order to ensure they are not tampered with and that they come from an authorized source.

SignTool

SignTool is a command-line application that digitally signs a UEFI executable (.efi file) and generates a signed executable (.signed). This section describes the command-line arguments, design, and their expected effect on the behavior of the utility.

Build Environment

SignTool is available from Microsoft.

Example usage

An example usage of the tool is as follows:

```
SignTool sign /f EntityPrivateKey.pfx  
ProduceApplication.efd
```

Where /f designates the local file with the private key and the final argument is the UEFI executable to be signed.

For the Linux community, there are similar tools, such as Fedora's pesign and Jeremy Kerr's sbsign based upon OpenSSL.

An example of the latter tool in action includes:

```
sbsign -key KEK.key -cert KEK.crt -output  
ProductApplication-signed.efd ProductApplication.efd
```

where the key and certificate files can be generated by

```
openssl req -new -x509 -newkey rsa:2048 -keyout KEK.key -  
out KEK.crt -days <length> -subj "/CN=<my common name>/"
```

and

```
openssl x509 -in KEK.crt -out KEK.cer -outform DER
```

respectively.

Appendix B

Command Reference

This appendix enumerates the commands that are part of the UEFI Shell standard and provide insight into what commands are associated with a given shell support level and profile.

Command Profiles and Support Levels

The UEFI Shell has associated with it two concepts which assist the user or program in determining what capabilities a given shell environment has. Since each shell implementation allows for a vast amount of differentiation, the concepts of shell support levels and shell profiles were introduced.

- *Shell Support Levels.* The level associated with a given shell environment is discovered by analyzing the `shellsupport` environment variable. This variable reflects the current support level provided by the currently running shell environment. Table B.1 shows a series of commands that correspond to a particular shell level value.
 - 0 – No commands built in. Only the APIs are available.
 - 1 – Basic commands required to support scripting, such as if, exit, etc.
 - 2 – Basic file system commands that do not require console output, such as `mkdir`, `cd`, etc.
 - 3 – Other basic commands that require console output.
- *Shell Profiles.* Shell command profiles are groups of shell commands that are identified by a profile name. The profile(s) supported by a given shell environment is derived by analyzing the `profiles` environment variable. Profile names that begin with the underscore character (`_`) are reserved for individual implementation.

Command List

Table B.1 provides a list of commands in the UEFI Shell standard. Note that some commands are listed as 2/3. In this case the level 2 version allows the setting to be changed, while the level 3 version allows the setting to be displayed and changed.

Table B.1: Commands in the UEFI Shell Specification

Command	Description	Required at Shell Level or Profile
Alias	Displays, creates, or deletes aliases in the UEFI Shell environment.	3
Attrib	Displays or changes the attributes of files or directories.	2
Bcfg	Manipulates boot order and driver order.	Debug1, Install1
Cd	Displays or changes the current directory.	2
Cls	Clears the standard output and optionally changes the background color.	3
Comp	Compares the contents of two files on a byte-for-byte basis.	Debug1
Connect	Binds a driver to a specific device and starts the driver.	Driver1
Cp	Copies one or more source files or directories to a destination.	Driver1
Date	Displays and sets the current date for the system.	2/2
Dblk	Displays the contents of one or more blocks from a block device.	Debug1
Del	Deletes one or more files or directories.	2
Devices	Displays the list of devices managed by UEFI drivers.	Driver1
Devtree	Displays the tree of devices compliant with the UEFI Driver Model.	Driver1
Dh	Displays the device handles in the UEFI environment.	Driver1
Dir	Lists directory contents or file information.	2
Disconnect	Disconnects one or more drivers from the specified devices.	Driver1
Dmem	Displays the contents of system or device memory.	Debug1
Dmpstore	Manages all UEFI variables.	Debug1
Drivers	Displays a list of information for drivers that follow the UEFI Driver Model.	Driver1
Drvcfg	Configures the driver using the UEFI Driver Configuration protocol.	Driver1
Drvdiag	Performs device diagnostics using the UEFI Driver Diagnostics protocol.	Driver1
Echo	Controls whether or not script commands are displayed as they are read from the script file and prints the given message to the console.	3
Edit	Full-screen editor for ASCII or UCS-2 (Unicode) files.	Debug1
eficompress	Compresses a file using the UEFI Compression algorithm.	Debug1
efidecompress	Decompresses a file using the UEFI Compression algorithm.	Debug1

Command	Description	Required at Shell Level or Profile
else	Conditionally executes commands if a previous if command condition was false. Only valid in scripts.	1
endfor	Ends a loop started with a for command. Only valid in scripts.	1
endif	Ends a conditional block started with a previous if command. Only valid in scripts.	1
exit	Exits the UEFI Shell environment or the current UEFI Shell script.	1
for	Starts iterating over a block of script commands. The block is terminated by a matching endfor command. Only valid in scripts.	1
getmtc	Returns the current UEFI monotonic count.	3
goto	Moves execution within the script to a specific label. Only valid in scripts.	1
guid	Displays all GUIDs registered with the UEFI Shell.	Debug1
help	Displays a list of commands that are built into the UEFI Shell and their usage.	3
hexedit	Full-screen hex editor for files, block devices, or memory.	Debug1
if	Starts a block of script commands that are executed only if the condition specified evaluates to TRUE. Only valid in scripts.	1
ifconfig	Displays or modifies the current TCP/IP configuration for IPv4.	Network1
ifconfig6	Display or modify the current TCP/IP configuration for IPv6.	Network2
load	Loads a UEFI driver into memory.	2
loadpciom	Loads a UEFI driver from a binary file that is formatted as a PCI Option ROM.	Debug1
ls	Lists a directory's contents or file information.	2
map	Defines a mapping between a user-defined identifier and a device handle.	2
mem	Displays the contents of system or device memory.	Debug1
memmap	Displays the memory map maintained by the UEFI environment.	Debug1
mkdir	Creates one or more new directories.	2
mm	Displays or modifies memory, memory-mapped I/O, I/O, PCI configuration space, or PCI Express configuration space.	Debug1
mode	Displays or changes the console output device's mode.	Debug1
mv	Moves one or more files to a destination within a file system.	2
openinfo	Displays the protocols and agents associated with a UEFI driver handle.	Driver1

Command	Description	Required at Shell Level or Profile
parse	Parses the data returned from a UEFI Shell command as standard formatted output. See “Standardizing Command Output” (below) for more information.	2
pause	Pauses script execution and waits for a keypress.	3
pci	Displays a list of PCI devices in the system and their PCI configuration space.	Debug1
ping	Checks the response of a network IPv4 address.	Network1
ping6	Checks the response of a network IPv6 address.	Network2
reconnect	Reconnects drivers to a specific device.	Driver1
reset	Rests the system.	2
rm	Deletes one or more files or directories.	2
seremode	Sets the serial port attributes.	Debug1
set	Displays or sets a UEFI Shell environment variable.	2
setszie	Changes the size of a file.	Debug1
setvar	Displays, deletes, or updates the value of a UEFI variable.	Debug1
shift	Shifts the contents of the script positional parameters by one. Only valid in scripts.	1
smbiosview	Display the system’s SMBIOS information.	Debug1
time	Displays or sets the current system time.	2/3
timezone	Displays or sets the system time zone information.	2/3
touch	Updates the time and date of a file to the current time and date.	3
type	Sends the contents of a file to the console.	3
unload	Unloads a driver image that has already been loaded.	Driver1
ver	Displays the version information of the UEFI environment.	3
vol	Changes/displays the volume label.	3

Standardizing Command Output

One of the goals of the UEFI specifications is to clearly describe the required behavior while leaving room for implementation-specific differentiation. Yet there was also a strong desire to standardize the output of the shell commands so that tools and scripts could operate consistently on the output of these commands.

The standard format output is available with many of the UEFI Shell commands by specifying the `-sfo` command-line option. When this option is provided, all output is produced to standard output in rows. Each row starts with an identifier (the “Table Name”) that gives the type of data in the row, followed by zero or more columns. Each

column is separated by a comma and contains a quoted string. Each column's meaning depends on the Table Name and the position of the column (1st, 2nd, 3rd, etc.). If the Table Name starts with an underscore (_), it indicates that the contents of the row are implementation-specific.

The `parse` shell command was designed to allow shell scripts to easily parse the contents of standard format output, selecting specific table names, row instances and column instances. For more information, see the `parse` UEFI Shell command description, below.

For example, for the standard format output for the “`ls`” command:

```
VolumeInfo,"MikesVolume","136314880","False","1024","512"
FileInfo,"fs0:/efi/boot/timsfile.txt","1250","a"
FileInfo,"fs0:/efi/boot/BOOTx64.EFI","1250","arsh"
```

Command Details

This section gives a brief overview of each command and its parameters. It is intended as a quick reference for the purpose of understanding the basic command capabilities. For full details on each of the commands, refer to the latest edition of the UEFI Shell specification.

alias

This command displays, creates, or deletes aliases in the UEFI Shell environment. An alias provides a new name for an existing UEFI Shell command or UEFI Shell application. Once the alias is created, it can be used to run the command or launch the UEFI Shell application.

The UEFI Shell specification defines several required aliases: `dir` for `ls`, `del` for `rm`, `copy` for `cp`, `mem` for `dmem`/and `md` for `mkdir`. These aliases cannot be deleted or changed. In addition, the EDK2 implementation adds `ren` and `move` for `mv`, `mount` for `map`/and `cat` for `type`.

```
alias [-d|-v] [alias-name] [command-name]
alias-name – Alias name.
```

command-name – Original command’s name or original UEFI Shell application’s name and directory.

`-d` – Delete an alias. command-name must not be present.

-v – Make the alias volatile. After exiting the UEFI Shell environment, the alias definition will be lost.

attrib

This command displays and sets the attributes of files or directories. If no file or directory is specified, then all files in the current working directory are displayed. If no change in attribute is specified, then all attributes will be displayed.

There are four attributes supported in the UEFI Simple File System: archive (a), system (s), hidden (h) and read-only (r). Each attribute may be enabled or disabled for a file. In addition, if a file is a directory, the ‘d’ attribute will be displayed, but cannot be modified.

`attrib [+a|-a] [+r|-r] [+s|-s] [+h|-h] [file...|directory...]`

`+a|-a` – Set or clear the archive attribute.

`+r|-r` – Set or clear the read-only attribute.

`+s|-s` – Set or clear the system attribute.

`+h|-h` – Set or clear the hidden attribute.

`file...` – File name. Wild cards are permitted.

`directory...` Directory name. Wild cards are permitted.

bcfg

This command manages the boot and driver options stored in UEFI variables, as described in the UEFI specification.

The `dump` option displays the `Boot####` or `Driver####` variables. The `add`, `addp/and` `addh` options add a new `Boot####` or `Driver##` variable. The `mod`, `modp/modf`, and `modh` options modify an existing `Boot####` or `Driver####` variable. The `rm` option deletes a `Boot####` or `Driver####` variable.

The `add`, `addp`, `addh`, `rm/and` `mv` options may also update the `BootOrder` or `DriverOrder` variables as appropriate.

The `-opt` option either updates the optional data in a `Boot####` or `Driver####` variable or updates the hot key data associated with a `Boot####` variable in the corresponding `Key####` variable.

```
bcfg driver|boot "dump" [-v]
bcfg driver|boot "add" option file "description"
bcfg driver|boot "addp" option file "description"
bcfg driver|boot "addh" option handle "description"
bcfg driver|boot mod option "description"
bcfg driver|boot modf option file
bcfg driver|boot modp option file
bcfg driver|boot modh option handle
bcfg rm option
bcfg mv option1 option2
bcfg -opt option filename
bcfg -opt option "data"
bcfg -opt key-data keys*
```

`dump` – Command that specifies that the option list specified by driver or boot will be displayed. If `-v` is added, extra information will be displayed, including the optional data.

`add option file "description"` – Add an option. The option is the option number to add, in hexadecimal. The file name is the name of the UEFI driver or application and will be added to the file system path. The quoted string is the user-readable description of the option.

`addh option handle "description"` – Add an option. The option is the option number to add, in hexadecimal. The handle is the driver or device handle (as reported by the `dh` command), in hexadecimal. The device path for the option is retrieved from the Loaded Image protocol(s) associated with the handle. The quoted string is the user-readable description of the option.

`addp index file "description"` – Add an option. The option is the option number to add, in hexadecimal. The file is the name of the UEFI driver or application. Only the portion of the device path starting with the hard drive partition is placed in the option. The quoted string is the user-readable description of the option.

`mod` – Modify description of an existing option. The option is the option number to modify, in hexadecimal. The quoted string is the user-readable description of the option.

`modf` – Modify device path stored in an existing option, using a file name. The option is the option number to modify, in hexadecimal. The file is the new file name of the UEFI application/driver to store in the option.

`modp` – Modify device path stored in an existing option, using a file name. The option is the option number to modify, in hexadecimal. The file is the new file name of the UEFI application/driver, but only the portion of the device path starting with the hard drive partition is stored in the option.

`modh` – Modify device path stored in an existing option, using a device handle. The option is the option number to modify, in hexadecimal. The handle is the device handle number (as listed by the `dh` command), in hexadecimal, and the device path of this handle is stored in the option.

`rm index` – Remove an option. The option is the option number to remove, in hexadecimal. The entry in the boot or driver list (BootOrder/DriverOrder) will also be removed.

`mv option1 option2` – Move an option. The first option (`option1`) is the option number to move. The second option (`option2`) is the new option number.

`-opt` – Display or modify the optional data associated with a boot option. It may be followed either by a file name containing that data, a hexadecimal stream containing the bytes of that data. This can also be the hot key data associated with the option. This data is encoded as flags (EFI_KEY_DATA from the UEFI specification) and then 1 to 4 scan-code/Unicode pairs.

cd

This command changes the current working directory that is used by the UEFI Shell environment. If no path is specified, then the current working directory for the current file system is displayed. If a file system mapping is specified, then the current working directory is changed for that device. Otherwise, the current working directory is changed for the current device.

Some pre-defined directory names are intended to have special meaning. These are:

- . Refers to the current directory.
- .. Refers to the parent directory of the current directory.
- \ Refers to the root of the current file system.

The current working directory is maintained in the environment variable `cwd`.

`cd [path]`

`path` – The relative or absolute directory path.

cls

This command clears the console and, optionally, clears it to a specific color. If the color is not specified, then the background color does not change.

`cls [background-color [foreground-color] | [-sfo]]`

`background-color` – New background color. Valid values are 0=Black, 1=Blue, 2=Green, 3=Cyan, 4=Red, 5=Magenta, 6=Yellow, 7=Light gray.

`foreground-color` – New foreground color. Valid values are 0=Black, 1=Blue, 2=Green, 3=Cyan, 4=Red, 5=Magenta, 6=Yellow, 7=Light gray.

`-sfo` – Console output is not cleared, but the information about the console is displayed in standard-format output. For more information on the format, see the UEFI Shell specification.

comp

This command compares the contents of two files in binary mode to determine if they are equal or not equal. It displays up to 10 differences between the two files. For each difference, up to 32 bytes from the location where the difference starts is dumped. It will exit immediately if the lengths of the compared files are different.

`comp [-b] file1 file2 [-n count] [-s size]`

`file1` – Specifies the first file name. Directory names or wildcards are not permitted.

file2 – Specifies the second file name. Directory names or wildcards are not permitted.

-b – Display one screen at a time

-n count – Unsigned integer that specifies the maximum number of differences to display or else the keyword all that specifies that all differences should be displayed. The default is 10.

-s size – Unsigned integer that specifies the number of bytes to display after finding a difference. The default is 4.

connect

This command binds a driver to a specific device and starts the driver. If the **-r** flag is used, then the connection is done recursively until no further connections between devices and drivers are made. If the **-c** flag is used, then the connect command binds the proper drivers to the console devices that are described in the UEFI variables.

If no parameters are specified, then the command will attempt to bind all proper devices to all devices without recursion. Each connection status will be displayed.

If only a single handle is specified and the handle has the Driver Binding protocol installed on it, it is assumed to be a driver handle. Otherwise, it is assumed to be a device handle.

```
connect [[device-handle] [driver-handle]] [-c] [-r]
```

-r – Recursively scan all handles and check to see if any loaded or embedded driver can match the specified device. If so, the driver will be bound to the device. Additionally, if more device handles are created during the binding, these handles will also be checked to see if a matching driver can bind to these devices as well. The process is repeated until no more drivers are able to be bound to any more devices. However, without this option, the newly created device handles will not be further bound to any drivers.

-c – Connect console devices as described in the UEFI variables.

device-handle – Device handle is a hexadecimal number (see dh) that specifies the handle of the device to be connected.

driver-handle – Driver handle is a hexadecimal number (see dh) that specifies the handle of the driver to be connected.

cp/copy

This command copies one or more source files or directories to a destination. If the source is a directory, the **-r** flag must be specified. If **-r** is specified, then the contents of the source directory will be recursively copied to the destination (which means that all subdirectories will be copied). If a destination is not specified, then the current working directory is assumed to be the destination.

If the target file (not directory) already exists, then the user will be prompted to confirm replacing the file, unless the **-q** option was specified.

If there are multiple source files, then the destination must be a directory.

When copying to another directory, that directory must exist.

```
cp [-r] [-q] src [src...] [dst]
```

src – Source file or directory name. Wildcards are permitted.

dst – Destination file or directory name. Wildcards are not permitted. If not specified, then the current working directory is assumed to be the destination. If there is more than one directory specified, then the last is always assumed to be the destination.

-r – Copy recursively.

-q – Quiet copy (no prompts). When executing a script, the default is quiet. Otherwise the default is to prompt.

date

This command displays and/or sets the current system date. If no parameters are used, it shows the current date. If a valid month, day, and year are provided, then the system's date will be updated.

```
date [mm/dd/[yy]yy] -sfo
```

mm – Month of the date to be set (1-12).

dd – Day of the date to be set (1-31).

yy/yyyy – Year of the date to be set. If only two digits, then 9x = 199x, otherwise 20xx.

-sfo – Standard format output. See the UEFI Shell specification for details.

dblk

This command displays the contents of one or more blocks from a block device. The logical block address (LBA) and block count should be specified as a hexadecimal value. If the LBA is not specified, block 0 is assumed. If the block count is not specified, then 1 block is assumed. The maximum number of blocks that can be displayed at one time is 16 (0x10).

If the block has the format of a Master Boot Record (MBR), then the partition information will be displayed. If the block has the format of a FAT partition, then some FAT parameters will be displayed.

`dblk device [lba] [blocks] [-b]`

device – The mapping name of the block device.

lba – Index of the first block to be displayed, specified as a hexadecimal number. The default is 0.

blocks – Number of blocks to be displayed, specified as a hexadecimal number. The default is 0. If larger than 10 (16 decimal), then only 16 will be displayed.

-b – Display only one screen at a time.

del

Internal alias of the `rm` command.

devices

This command prints a list of devices that are being managed by drivers that follow the UEFI Driver Model, as described in the UEFI specification.

`devices [-b] [-l lang] [-sfo]`

-b – Display one screen at a time.

-l lang – Dumps information using the language associated with the specified language code, such as `en-US` (for English). These language codes are described in

Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

-sfo – Display information as standard format output. See the UEFI Shell specification for more details.

devtree

This command prints a tree of devices that are being managed by drivers that follow the UEFI Driver Model. By default, the devices are printed using device names that are retrieved from the UEFI Component Name protocol. If the option **-d** is specified, then the device paths are printed instead.

`devtree [-b] [-d] [-l lang] [device-handle]`

device-handle – Display device tree information for devices attached to the specified device. The handle is the same as reported by the **dh** command.

-b – Display one screen at a time.

-d – Display device information using device paths.

-l lang – Dumps information using the language associated with the specified language code, such as en-US (for English). These language codes are described in Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

dh

This command displays the device handles in the UEFI environment for those devices managed according to the UEFI Driver Model. If the **dh** command is used with a specific handle, the details of all protocols that are associated with the device handle are displayed. Otherwise, the **-p** option can be used to list the device handles that contain a specific protocol. If neither **-p** nor handle is specified, then all device handles are displayed.

If **decode** is specified, then only decode information is displayed. With no additional parameters, this command will display all possible identifiers and their associated GUID in alphabetical order. If **-p** is also used, then only decode information for the specified protocol identifier is dumped. Decode information includes the full GUID and the string representation that can be used instead.

```
dh [-l lang] [handle | -p protocol-id] [-d] [-v] [-sfo]
dh decode -p protocol-id
```

handle – Specifies a device handle associated with the device about which information should be displayed. The handle is a hexadecimal number. If not present, then information about all devices will be displayed.

-p – Dumps all handles that support the specified protocols.

-d – Dumps UEFI Driver Model-related information.

-l lang – Dumps information using the language associated with the specified language code, such as en-US (for English). These language codes are described in Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

-v, -verbose – Dumps additional information about the handles.

-sfo – Displays information as standard format output, as described in the UEFI Shell specification.

decode – Display decoded information about a protocol specified by either a GUID or a protocol identifier.

dir/ls

Internal alias for the ls command.

disconnect

This command disconnects one or more drivers from the specified devices. If the **-r** option is specified, all drivers are disconnected from all devices in the system. If the **-nc** option is used along with the **-r** option, the console devices are not reconnected.

```
disconnect device-handle [driver-handle [child-handle]]
```

device-handle – The device handle, specified as a hexadecimal number. These numbers are the same as those displayed by the dh command.

driver-handle – The driver handle, specified as a hexadecimal number. These numbers are the same as those displayed by the `dh` command.

child-handle – The child handle of a device, specified as a hexadecimal number. If not specified, then all child handles of the device-handle will be disconnected.

-r – Disconnect all drivers from all devices.

-nc – Do not reconnect console devices.

dmem

This command displays the contents of system memory, I/O, PCI configuration space, or device memory. The address and size are specified as hexadecimal numbers.

If the address is not specified, then the contents of the UEFI System Table are displayed. Otherwise, the memory starting at the specified address is displayed. If the size is not specified, then the default is 512 bytes.

If `-MMIO`, `-IO`, `-PCI` and `-PCIE` are not specified, then system memory is displayed. If `-MMIO` is specified, then memory-mapped I/O is displayed by using the PCI Root Bridge I/O protocol. If `-IO` is specified, then I/O access are used. If `-PCI` is specified, then PCI configuration space accesses are used. If `-PCIE` is specified, then PCI express configurations space accesses are used.

```
dmem [-b] [address [size]] [-MMIO|-IO|-PCI|-PCIE]
```

address – The starting address, in hexadecimal. The default is the address of the UEFI System Table. For PCI and PCI Express, the format is `ssssbbddffrrr`, where `ssss` is the segment number, `bb` is the bus number, `dd` is the device number, `ff` is the function number and `rrr` is the register number.

size – The number of bytes to display, in hexadecimal. The default is 512.

-b – Display one screen at a time.

-MMIO – Forces address cycles to the PCI bus.

-IO – I/O access.

-PCI – PCI configuration space access.

-PCIE – PCI Express configuration space access.

dmpstore

This command displays, deletes, saves, or loads the UEFI variable contents.

```
dmpstore [-b] [-all | <var-name [-guid var-guid]>][-sfo]
```

```
dmpstore -d [-all | <var-name [-guid var-guid]>]
```

```
dmpstore -s file [-all | <var-name [-guid var-guid]>]
```

```
dmpstore -l file [-all | <var-name [-guid var-guid]>]
```

-all – Indicates that all variables should be displayed, loaded, saved, or deleted.

var-name – Specifies the name of the UEFI variable. If a variable name is specified, then only this variable will be displayed, deleted, loaded, or saved.

-guid var-guid – Specified the GUID of the UEFI variable. If not specified, then the Global Variable GUID is assumed (as defined in the UEFI specification).

-d – Delete the variable(s) specified.

-l file – Load the variable contents from the specified file.

-s file – Save the variable contents to the specified file.

-sfo – Display as standard-format output, as described in the UEFI Shell specification.

drivers

This command displays a list of information about drivers that follow the UEFI Driver Model. This includes the handle of the UEFI driver, the version of the UEFI driver, the driver type, the number of devices that the driver is managing, the number of child devices this driver has created, the type of driver (bus, device, or other) the name of the driver and the driver's file path (if any).

```
drivers [-l lang] [-sfo]
```

-l lang – Dumps information using the language associated with the specified language code, such as en-US (for English). These language codes are described in Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

-sfo – Displays driver information as standard-format output. See the UEFI Shell specification for more information.

drvcfg

This command invokes the platform's configuration infrastructure to allow the device to be configured.

```
drvcfg [-l lang] [-c] [-f type] [-v|-s] [driver-handle [device-handle [child-handle]]] [-i file] [-o file]
```

-f type – The type of defaults to set on the controller, specified as an unsigned integer. 0 = standard, 1 = manufacturing, 2 = safe mode. Other values are driver-specific.

driver-handle – The handle of the driver to configure, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

device-handle – The handle of the device to configure, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

child-handle – The handle of the device that is a child of the device specified by device-handle to configure, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

-c – Configure all child devices.

-l lang – Dumps information using the language associated with the specified language code, such as en-US (for English). These language codes are described in Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

-f – Force defaults.

-v – Validate options.

-s – Set options.

-i file – Receive configuration options from the specified file.

-o file – Export settings of the specified driver to the specified file.

drvdiag

This command invokes the Driver Diagnostic protocol. If no handles are specified, then all drivers that support this protocol will be listed.

```
drvdiag [-c] [-l lang] [-s|-e|-m] [driver-handle [device-handle [child-handle]]]
```

driver-handle – The handle of the driver to diagnose, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

device-handle – The handle of the device to diagnose, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

child-handle – The handle of the device that is a child of the device specified by device-handle to diagnose, specified as a hexadecimal number. These numbers correspond to the numbers output by the dh command.

-c – Diagnose all child devices.

-s – Run diagnostics in standard mode.

-e – Run diagnostics in extended mode.

-m – Run diagnostics in manufacturing mode.

-l lang – Dumps information using the language associated with the specified language code, such as en-US (for English). These language codes are described in Appendix M of the UEFI Specification. If none is specified, then the current platform language will be used.

echo

The first form of this command controls whether or not script commands are displayed as they are read from the script file. If no argument is given, the current “on” or “off” status is displayed. The second form prints the given message to the display.

```
echo [-on|-off]
```

```
echo [message]
```

message – Message to display

-on – Enables display when reading commands from script files

-off – Disables display when reading commands from script files

edit

This command allows a file to be edited using a full screen editor. The editor supports both UCS-2 and ASCII file types. If no file is specified, then an empty file will be edited.

edit [file]

file – Name of file to be edited. If none is specified, then an empty file is created with a default file name.

eficompress

This command is used to compress a file using EFI Compression Algorithm and write the compressed form out to a new file.

eficompress infile outfile

infile – Filename for uncompressed input file

outfile – Filename for compressed output file

efidecompress

This command is used to decompress a file using the EFI Decompression Algorithm and write the decompressed form out to a new file.

efidecompress infile outfile

infile – Filename for compressed input file

outfile – Filename for decompressed output file

exit

This command exits the UEFI Shell or, if /b is specified, the current script.

```
exit [/b] [exit-code]
```

/b – Indicates that only the current UEFI Shell script should be terminated. Ignored if not used within a script.

exit-code – If exiting a UEFI Shell script, the value placed into the environment variable `lasterror`. If exiting an instance of the UEFI Shell, the value returned to the caller. If not specified, then 0 is returned.

for

The `for` command executes one or more commands for each item in a set of items. The set may be text strings or file names or a mixture of both, separated by spaces (if not in quotation marks). If the length of an element in the set is between 0 and 256, and if the string contains wildcards, the string is treated as a file name containing wildcards, and will be expanded before `command` is executed.

If after expansion no such files are found, the literal string itself is kept. The `indexvar` variable is any alphabet character from “a” to “z” or “A” to “Z”, and they are case-sensitive. It should not be a digit (0–9) because `%digit` will be interpreted as a positional argument on the command line that launches the script. The namespace for index variables is separate from that for environment variables, so if `indexvar` has the same name as an existing environment variable, the environment variable will remain unchanged by the `for` loop.

Each command is executed once for each item in the set, with any occurrence of `%indexvar` in the command replaced with the current item. In the second format of `for ... endfor` statement, `indexvar` will be assigned a value from `start` to `end` with an interval of `step`. `start` and `end` can be any integer whose length is less than 7 digits excluding sign, and it can also be applied to `step` with one exception of zero. `step` is optional, if `step` is not specified it will be automatically determined by following rule, if `start ≤ end` then `step = 1`, otherwise `step = -1`. `start`, `end`/and `step` are divided by space.

This command may only be used in scripts.

This command does not change the value of the environment variable `lasterror`.

```
for %indexvar in set
    command [arguments]
    [command [arguments] ]
    ...
endfor
```

```

for %indexvar run (start end [step])
    command [arguments]
    [command [arguments] ]
    ...
endfor

```

getmtc

This command displays the current monotonic counter value. The lower 32 bits increment every time this command is executed. Every time the system is reset, the upper 32 bits are incremented and the lower 32 bits are reset to 0.

```
getmtc
```

goto

This command directs script file execution to the line in the script file after the given label. The command is not supported from the interactive shell. A label is a line beginning with a colon (:). It can either appear after the `goto` command, or before the `goto` command. The search for label is done forward in the script file, from the current file position. If the end of the file is reached, the search resumes at the top of the file and continues until label is found or the starting point is reached. If label is not found, the script process terminates and an error message is displayed. If a label is encountered but no `goto` command is executed, the label lines are ignored.

This command is only valid in script files.

```
goto label
```

help

The `help` command displays information about one or more shell commands. If no other options are specified, each command is displayed along with a brief description of its function.

If no other options are specified, then each command is displayed, along with a brief description of its function. If `-verbose` is specified, then all help information for the specified command is displayed. If `-section` is specified, only the help section specified will be displayed. If `-usage` is specified, then the usage information will be displayed.

```
help [cmd | pattern | special] [-usage] [-verbose] [-section sectionname] [-b]
```

cmd – Command to display help about.

pattern – Pattern that describes the commands to be displayed.

special – Displays a list of the special characters used in the shell command line.

-usage – Display the usage information for the command. The same as specifying **-section:NAME** and **-section:SYNOPSIS**

-section sectionname – Display the specified section of the help information.

hexedit

This command allows a file, block device, or memory region to be edited. The region being edited is displayed as hexadecimal bytes, and the contents can be modified and saved.

```
hexedit [[-f] filename | [-d diskname offset size] | [-m address size]]
```

-f – Name of file to edit.

-d – Disk block to edit. The **diskname** specifies the name of disk to edit (for example **fs0**). The **offset** specifies the starting block number (beginning from 0). The **size** specifies the number of blocks to be edited.

-m – Memory region to edit. The **address** specifies the starting 32-bit memory address (beginning from 0). The **size** specifies the size of the memory region to be edited in bytes.

if

The **if** command executes one or more commands before the **else** or **endif** commands, if the specified condition is true; otherwise commands between **else** (if present) and **endif** are executed.

In the first usage of `if`, the `exist` condition is true when the file specified by `filename` exists. The `filename` argument may include device and path information. Also wildcard expansion is supported by this form. If more than one file matches the wildcard pattern, the condition evaluates to TRUE.

In the second usage, the `string1 == string2` condition is true if the two strings are identical. Here the comparison can be case-sensitive or case-insensitive; it depends on the optional switch `/i`. If `/i` is specified, it compares strings in the case-insensitive manner; otherwise, it compares strings in the case-sensitive manner.

In the third usage, general purpose comparison is supported using expressions optionally separated by `and` or `or`. Since `<` and `>` are used for redirection, the expressions use common two character (FORTRAN) abbreviations for the operators (augmented with unsigned equivalents).

By default, comparisons are done numerically if the strings on both sides of the operator are numbers and in case-sensitive character sort order otherwise. Spaces separate the operators from operands. The `/s` and `/i` switches apply to the entire line and must appear at the start of the line (just after the `if` itself). When performing comparisons, the Unicode Byte Ordering Character is ignored at the beginning of any argument.

Conditional expressions are evaluated strictly from left to right. Complex conditionals requiring precedence may be implemented as nested `if` commands.

The expressions in the third usage have the following syntax:

```
conditional-expression ::= expression | expression and expression
expression | expression or expression
expression ::= expr | not expr
expr ::= item binop item | boolfunc(string)
item ::= mapfunc(string) | string
mapfunc ::= eferror | pierror | oemerror
boolfunc ::= isint | exists | available | profile
binop ::= gt | lt | eq | ne | ge | le | == | ugt | ult |
uge | ule
```

For the comparisons, the operators are defined as:

`gt` – greater than

`ugt` – unsigned Greater than

`lt` – less than

`ult` – unsigned Less than

`ge` – greater than or equal

`uge` – unsigned greater than or equal

`le` – less than or equal

`ule` – unsigned less than or equal

`ne` – not equal

`eq` – equals (semantically equivalent to `==`)

`==` – equals (semantically equivalent to `eq`)

The error mapping functions referenced in *item* are used to convert integers into UEFI, PI, or OEM error codes, as defined by Appendix D of the UEFI specification. The following are the defined functions:

`UefiError` – Sets top nibble of parameter to 0100 binary (0x8)

`PiError` – Sets top nibble of parameter to 1010 binary (0xA)

`OemError` – Sets top nibble of parameter to 1100 binary (0xC)

For example, to check for write protect (UEFI error #8):

```
if %lasterror% == UefiError(8) then
```

The Boolean functions are defined as:

`isint()` – Evaluates to true if the parameter string that follows is a number (as defined below) and false otherwise.

`exists()` – Evaluates to true if the file specified by string exists in the current working directory or false if not.

`available()` – Evaluates to true if the file specified by string is in the current working directory or current path.

`profile()` – Determines whether the parameter string matches one of the profile names in the `profiles` environment variable.

```
if [not] exist filename then
    command [arguments]
```

```
[command [arguments]]  
...  
[else  
command [arguments]  
[command [arguments]]  
...  
]  
endif  
  
if [/i] [not] string1 == string2 then  
command [arguments]  
[command [arguments]]  
...  
[else  
command [arguments]  
[command [arguments]]  
...  
]  
endif  
  
if [/i][/s] conditional-expression then  
command [arguments]  
[command [arguments]]  
...  
[else  
command [arguments]  
[command [arguments]]  
...  
]  
endif
```

conditional-expression – Conditional expression, as described in the section “Expressions” below.

/s – Forces string comparisons.

/i – Forces case-insensitive string comparisons.

ifconfig

This command is used to modify the default IP address for the UEFI IPv4 network stack.

```
ifconfig -r [name]
ifconfig -l [name]
ifconfig -s name dhcp
ifconfig -s name static ip4 subnet-mask gateway-mask
ifconfig -s dns ip4 [,ip4...]
```

name – Adapter name, such as eth0

-l name – Lists the configuration for all or the specified interface.

-s name static ip4 subnet-mask gateway-mask – Use static IP4 address configuration for all interfaces or the specified interface.

ip4 – IP4 address in four integer values (each between 0-255). i.e., 192.168.0.1

subnet-mask – Subnet mask in four integer values (each between 0-255), i.e., 255.255.255.0

gateway-mask – Default gateway in four integer values (each between 0-255), such as 192.168.0.1

-s name dhcp – Use DHCPv4 to request the IPv4 address configuration dynamically for all interfaces or the specified interface.

-s name dns ip4[,ip4...] – Configure DNS server addresses for the specified interface. IPs can be combined, separated by a space.

ifconfig6

This command is used to display or modify IPv6 configuration for network interface.

```
ifconfig6 -r [name]
```

```
ifconfig6 -l [name]
ifconfig6 -s name dad number
ifconfig6 -s name auto
ifconfig6 -s man id mac
ifconfig6 -s man host ip6 gw ip6
ifconfig6 -s man dns ip6
```

-r name – Reconfigure all interfaces or the specified interface, and set automatic policy. If the specified interface is already set to automatic, then refresh the IPv6 configuration.

-l name – List the configuration of the specified interface or all interfaces.

-s name dad number – Set dad transmits count of the specified interface.

-s name auto – Set automatic policy of the specified interface.

-s name man id mac – Set alternative interface id of the specified interface. Must under manual policy.

-s name man host ip6 gw ip6 – Set static host IP and gateway address of the specified interface. Must under manual policy.

-s name man dns ip6 – Set DNS server IP addresses of the specified interface. Must under manual policy.

load

This command loads a driver into memory. It can load multiple files at one time, and the file name supports wildcards. If the **-nc** flag is not specified, this command will try to connect the driver to a proper device; it may also cause already loaded drivers to be connected to their corresponding devices.

```
load [-nc] file [file...]
```

-nc – Load the driver, but do not connect the driver.

file – File that contains the image of a UEFI driver. Wildcards are permitted.

loadpciorom

This command is used to load PCI option ROM images into memory for execution. The file can contain legacy images and multiple PE32 images, in which case all PE32 images will be loaded.

```
loadpciorom [-nc] romfile [romfile...]
```

-nc – Load the ROM image but do not connect the driver.

romfile – PCI option ROM image file (wildcards are permitted).

ls

This command lists directory contents or file information. If no file name or directory name is specified, then the current working directory is assumed.

If no attribute is specified using the **-ar/-ah/-as/-aa/-ad** options, then all non-system and non-hidden files will be displayed.

```
ls [-r] [-ar] [-ah] [-as] [-aa] [-ad] [-sfo] [file]
```

-r – Displays directory contents recursively (including subdirectories).

-ar – Display only read-only files.

-ah – Display only hidden files.

-as – Display only system files.

-aa – Display only files that are marked for archival.

-ad – Display only directories.

-sfo – Display information as standard format output. See the UEFI Shell specification for more details.

file – Name of the file or directory. Wildcards are permitted.

map

This command creates a mapping between a user-defined name and a device. The most common use of this command is to create the mapped name for devices that support a file system protocol. Once these mappings are created, the names can be used with all the file manipulation commands.

The UEFI Shell creates default mappings for all devices that support a file system.

This command can be used to create additional mappings, or it can be used to delete an existing mapping with the `-d` option. If the command is used without any options, then all of the current mappings will be displayed.

The `-r` option resets all of the default mappings for file systems known to the UEFI environment. This option is useful if the configuration has changed since the last boot.

The `-u` option adds mappings for newly installed devices and removes mappings for uninstalled devices but will not change the mappings of existing devices. Any user-defined mappings are also preserved. A mapping history will be saved so that the original mapping name is used for a device with a specific device path if that mapping name was used for that device path last time. The current directory is also preserved if the current device is not changed.

The mapping consist of digits and characters. Other characters are illegal.

This command supports wildcards. You can use the wildcards to delete or show the mapping. However, when you assign the mapping, wildcards are forbidden.

```
map -d sname
map -r sname
map -v [sname]
map -f sname
map -u sname
map -t type,[,type...]sname
map sname [handle | mapname]
```

sname – Mapping name.

handle – The device handle, as displayed by the `dh` command, in hexadecimal.

mapping – The device's mapped name. Use this parameter to assign a new mapping to a device. The mapping must end with a “`:`” (colon).

-t – Shows the device mappings, filtered according to the device type. The supported types are `flop` (floppy), `hd` (hard disk), and `cd` (CD-ROM). Types can be combined by putting a comma between two types. Spaces are not allowed between types.

-d – Deletes a mapping.

-r – Resets to default mappings.

-v – Lists verbose information about all mappings.

-c – Shows the consistent mapping.

-f – Shows the normal mapping.

-u – This option will add mappings for newly-installed devices and remove mappings for uninstalled devices but will not change the mappings of existing devices. The user-defined mappings are also preserved.

-sfo – Display mappings as standard format output. See the UEFI Shell Specification for details.

md

An internal alias for `mkdir`.

mem

An internal alias for `dmem`.

memmap

This command displays the memory map that is maintained by the UEFI environment. The UEFI environment keeps track all the physical memory in the system and how it is currently being used. The UEFI Specification defines a set of Memory Type Descriptors. Please see the UEFI Specification for a description of how each of these memory types is used.

`memmap [-b] [-sfo]`

-b – Display one screen at a time.

-sfo – Display output in standard format output. See the UEFI Shell Specification for the latest details.

mkdir

This command creates one or more new directories. If the directory specified by dir includes nested directories, then parent directories will be created before child directories. If the directory already exists, then the command will exit with an error.

`mkdir dir [dir...]`

dir – Name of directory or directories to be created. Wildcards are not permitted.

mm

This command allows the user to display or modify I/O register, memory contents, or PCI configuration space.

`mm address [value] [-w 1|2|4|8] [-MEM | -MMIO | -IO | -PMEM | -PCI | -PCIE] [-n]`

address – Starting address.

value – The value to write. If not specified, then the current value will be displayed.

-MEM – Memory Address type.

-IO – I/O Address type

-PCI – PCI Configuration Space. The address will have the format `0x000000ssbbddffrr`, where ss = Segment, bb = Bus, dd = Device, ff = Function, and rr = Register. This is the same format used in the PCI command.

-PCIE – PCI Express Configuration Space. The address will have the format `0x0000000ssbbddffrrr`, where ss = Segment, bb = Bus, dd = Device, ff = Function, and rrr = Register.

-w – Access Width, in bytes. 1 = 1 byte, 2 = 2 bytes, 4 = 4 bytes, 8 = 8 bytes. If not specified, then 1 is assumed.

-n – Non-interactive mode.

mode

This command is used to change the display mode for the console output device. When this command is used without any parameters, it shows the list of modes that the standard output device currently supports.

```
mode [col row]
```

row – Number of rows.

col – Number of columns.

mv

This command moves one or more files to a destination within a file system .If the destination is an existing directory, then the sources are moved into that directory. Otherwise, the sources are moved into the directory as if the directory had been renamed. If the destination is not specified, then the current working directory is assumed.

Attempting to move a read-only file or directory generates an error. Moving a directory that contains read-only files is allowed. It is not allowed to move a directory into itself or its subdirectories.

```
mv src [src...] [dst]
```

src – Source file/directory name (wildcards are permitted)

dst – Destination file/directory name (wildcards are permitted). If not specified, then the current working directory is assumed to be the destination. If there is more than one argument on the command line, the last one will always be considered the destination.

openinfo

This command is used to display the open protocols on a given handle.

```
openinfo handle [-b]
```

handle – Display open protocol information for a specified handle.

-b – Display one screen at a time.

parse

This command enables the parsing of data from a file that contains data output from a command having used the `-sfo` parameter. Since the standard formatted output has a well-known means of parsing, this command is intended to be used as a simplified means of having scripts consume such constructed output files and use this retrieved data in logic of the scripts being written for the UEFI Shell.

```
parse filename tablename column [-i <instance>] [-s <instance>]
```

`filename` – Source file name.

`tablename` – The name of the table being parsed.

`column` – The one-based column index to use to determine which value from a particular record to parse.

`-i instance` – Start parsing with the *n*th instance of specified `tablename`, after the specified instance of `ShellCommand`. If not present, then all instances will be returned.

`-s <Instance>` – Start parsing with the *n*th instance of the `ShellCommand` table. If not present, then 1 is assumed.

pause

The pause command prints a message to the display and then suspends script file execution and waits for keyboard input. Pressing any key resumes execution, except for q or Q. If q or Q is pressed, script processing terminates; otherwise execution continues with the next line after the pause command.

```
pause [-q]
```

`-q` – Hide the pause message.

pci

This command will display all the PCI devices found in the system. It can also display the configuration space of a PCI device according to specified bus (`bus`), device (`dev`), and function (`func`) addresses. If the function address is not specified, it will

default to 0. The **-i** option is used to display verbose information for the specified PCI device. The PCI configuration space for the device will be dumped with a detailed interpretation. If no parameters are specified, all PCI devices will be listed.

```
pci [bus dev [func] [-s seg] [-i] [-ec id]]
```

bus – Bus number (0-255).

dev – Device number (0-31).

func – Function number (0-7).

-s – Optional segment number (0-65,535).

-i – Display detailed interpretation of the PCI configuration space.

-ec – Display detailed interpretation of the specified PCIe extended capability, specified as a hexadecimal number.

ping

This command is used to ping a target machine with UEFI IPv4 network stack.

```
ping [-n count] [-l size] [-s ip4] target-ip
```

-n – Number of echo request datagram to be sent.

-l – Size of data buffer in echo request datagram.

-s – Specifies the source adapter as the IP address specified.

target-ip – IPv4 address of the target machine.

ping6

This command is used to ping a target machine with UEFI IPv6 network stack.

```
ping6 [-l size] [-n count] [-s ip6] target-ip
```

-l size – Send buffer size in bytes (default=16, min=16, max=32768).

-n count – Send request count (default=10, min=1, max=10000).

`-s ip6` – Source IPv6 address.

`target-ip` – Target IPv6 address.

reconnect

This command reconnects drivers to the specific device. It first disconnects the specified driver from the specified device and then connects the driver to the device recursively. If the `-r` option is used, all drivers are reconnected to all devices. Any drivers that are bound to any devices will be disconnected first and then connected recursively. See the `connect` and `disconnect` commands for more details.

`reconnect device-handle [driver-handle [child-handle]]`

`reconnect -r`

`device-handle` – Device handle (a hexadecimal number).

`driver-handle` – Driver handle (a hexadecimal number). If not specified, all drivers on the specified device will be reconnected.

`child-handle` – Child handle of device (a hexadecimal number). If not specified, then all child handles of the specified device will be reconnected.

`-r` – Reconnect drivers to all devices.

reset

This command resets the system. The default is to perform a cold reset. If the `reset` string is specified, then it is passed into the `ResetSystem()` function, so the system can know the reason for the system reset.

If `-fwui` is specified and the system firmware supports it, on the next boot, the boot process will enter the firmware's user interface. If the system firmware does not support booting to the firmware's user interface, this command returns an error.

`reset [-w [string]] [-fwui]`

`reset [-s [string]] [-fwui]`

`reset [-c [string]] [-fwui]`

`-s` – Performs a shutdown.

-w – Performs a warm boot.

-c – Performs a cold boot.

-fwui – Mark the *OsIndications* variable to indicate a request to reboot into the firmware’s user interface.

rm

This command deletes one or more files or directories. If the target is a directory, it deletes the directory, including all files and all subdirectories. It is not allowed to redirect to a file whose parent directory (or the file itself) is being deleted.

Removing a read-only file will result in an error. Removing a directory that contains a read-only file will result in an error. If an error occurs, this command will exit immediately and later files/directories will not be removed.

rm [-q] file/directory [file/directory...]

file – File name. Wildcards are permitted.

directory – Directory name. Wildcards are permitted.

-q – Quiet mode. In this mode, there are no prompts for a confirmation.

sermode

This command displays or sets baud rate, parity attribute, data bits, and stop bits of serial ports. If no attributes are specified, then the current settings are displayed. If no handle is specified, then all serial ports are displayed.

sermode [handle [baudrate parity databits stopbits]]

handle – Device handle for a serial port in hexadecimal. The **dh** command can be used to retrieve the correct handle.

baudrate – Baud rate for specified serial port. The following values are supported: 50, 75, 110, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600 (default), 19200, 38400, 57600, 115200, 230400, and 460800. All other values will be converted to the next highest setting.

parity – Parity bit settings for specified serial port. Any one of the following settings can be used:

- d – Default parity
- n – No parity
- e – Even parity
- o – Odd parity
- m – Mark parity
- s – Space parity

databits – Data bits for the specified serial port. The following settings are supported: 4, 7, 8 (default). All other settings are invalid.

stopbits – Stop bits for the specified serial port. The following settings are supported:

- 0 (0 stop bits – default setting)
- 1 (1 stop bit)
- 2 (2 stop bits)
- 15 (1.5 stop bits)

set

This command is used to create, delete, or change UEFI Shell environment variables. The **set** command will set the environment variable that is specified by **sname** to **value**. This command can be used to create a new environment variable or to modify an existing environment variable. If the **set** command is used without any parameters, then all the environment variables are displayed.

This command does not change the value of the environment variable *lasterror*.

```
set [-v] [sname [value]]  
set [-d <sname>]
```

-d – Deletes the environment variable.

-v – Volatile variable.

sname – Environment variable name.

value – Environment variable value.

setsize

This command adjusts the size of a particular target file. When adjusting the size of a file, it should be noted that this command automatically truncates or extends the size of a file based on the passed-in parameters. If the file does not exist, it will be created.

```
setsize size file [file...]
```

file – The file or files that will have their size adjusted.

size – The desired size of the file once it is adjusted. Setting the size smaller than the actual data contained in this file truncates this data.

setvar

This command creates, deletes, or changes the UEFI variable specified by **name** and **guid**. If **=** is specified, but **data** is not, the variable is deleted, if it exists. If **=** is not specified, then the current variable contents are displayed. If **=data** is specified, then the variable's value is changed to the value specified by **data**.

```
setvar variable-name [-guid guid] [-bs] [-rs] [-nv] [=data]
```

variable-name – Specifies the name of the UEFI variable to modify or display.

-guid – Specifies the GUID of the UEFI variable to modify or display. If not present, then the GUID **EFI_GLOBAL_VARIABLE** is assumed, as defined in the UEFI specification.

-bs – Indicates that the variable is a boot variable. Should only be present for new variables, otherwise it is ignored.

-rt – Indicates that the variable is a runtime variable. Should only be present for new variables, otherwise it is ignored.

-nv – Indicates that the variable is nonvolatile. If not present, then the variable is assumed to be volatile. Should only be present for new variables, otherwise it is ignored.

=data – New data for the variable. If there is nothing after the “**=**” then the variable is deleted. If **=** is not present, then the current value of the variable is dumped as hexadecimal bytes. The data may consist of zero or more of the following:

`xx [xx]` : Hexadecimal bytes

`"ascii-string"` or `S"ascii-string"`: ASCII string with no null-terminator

`L"UCS2-string"`: UCS-2 encoded string with no null-terminator

`-P"device-path"` or `--device`: Device path text format, as specified by EFI Device Path Display Format Overview section of the UEFI 2.1 specification.

shift

The `shift` command shifts the contents of a UEFI Shell script's positional parameters so that %1 is discarded, %2 is copied to %1, %3 is copied to %2, %4 is copied to %3, and so on. This allows UEFI Shell scripts to process script parameters from left to right. The `shift` command is available only in UEFI Shell scripts.

`Shift`

smbiosview

This command displays the SMBIOS information. Users can display the information of SMBIOS structures specified by type or handle.

`smbiosview [-t SmbiosType] | [-h SmbiosHandle] | [-s] | [-a]`

`-t` – Display all structures of `SmbiosType`. The following values are supported:

- 0 – BIOS Information
- 1 – System Information
- 3 – System Enclosure
- 4 – Processor Information
- 5 – Memory Controller Information
- 6 – Memory Module Information
- 7 – Cache Information
- 8 – Port Connector Information
- 9 – System Slots
- 10 – On Board Devices Information
- 15 – System Event Log
- 16 – Physical Memory Array
- 17 – Memory Device
- 18 – 32-bit Memory Error Information

- 19 – Memory Array Mapped Address
- 20 – Memory Device Mapped Address
- 21 – Built-in Pointing Device
- 22 – Portable Battery
- 34 – Management Device
- 37 – Memory Channel
- 38 – IPMI Device Information
- 39 – System Power Supply

-h – Display the structure of SmbiosHandle, the unique 16-bit value assigned to each SMBIOS structure. SmbiosHandle can be specified in either decimal or hexadecimal format. Use the 0x prefix for hexadecimal values.

-s – Display statistics table.

-a – Display all information.

stall

This command is used to establish a timed stall of operations during a script.

stall time

time – The number of microseconds for the processor to stall.

time

This command displays or sets the current time for the system. If no parameters are used, it shows the current time. If valid hours, minutes, and seconds are provided, then the system's time is updated.

time [hh:mm[:ss]] [-tz tz] [-d dl]

hh – New hour (0–23) (required).

mm – New minute (0–59) (required).

ss – New second (0–59). If not specified, then zero is used.

-tz – Time zone adjustment, measured in minutes offset from GMT. Valid values can be between -1440 and 1440 or 2047. If not present or set to 2047, time is interpreted as local time.

-d – Indicates that time is not affected by daylight saving time (0), time is affected by daylight saving time but time has not been adjusted (1), or time is affected by daylight saving time and has been adjusted (3). All other values are invalid. If no value follows **-d**, then the current daylight saving time is displayed.

time

This command displays and sets the current time zone for the system. If no parameters are used, it shows the current time zone. If a valid `hh:mm` parameter is provided, then the system's time zone information is updated.

```
timezone [-s hh:mm | -l] [-b] [-f]
```

-s – Set time zone associated with `hh:mm` offset from GMT.

-l – Display list of all time zones.

-b – Display one screen at a time.

-f – Display full information for specified time zone.

touch

This command updates the time and date on the file that is specified by the `file` parameter to the current time and date.

```
touch [-r] file [file ...]
```

file – The name or pattern of the file or directory. There can be multiple files on the command line.

-r – Recurse into subdirectories.

type

This command sends the contents of a file to the standard output device. If no options are used, then the command attempts to automatically detect the file type. If it fails, then UCS-2 is presumed.

```
type [-a|-u] file [file...]
```

file – Name of the file to display.

unload

This command unloads a driver image that was already loaded and that supports the unloading option in the `EFI_LOADED_IMAGE_PROTOCOL` protocol.

```
unload [-n] [-v|-verbose] handle
```

-n – Skips all prompts during unloading, so that it can be used in a script file.

-v, -verbose – Dump verbose status information before the image is unloaded.

handle – Handle of driver to unload, always taken as hexadecimal number.

ver

This command displays the version information for this EFI Firmware or the version information for the UEFI Shell itself. The information is retrieved through the UEFI System Table or the Shell image.

```
ver [-s|-terse]
```

-s – Displays only the UEFI Shell version.

-terse – Abbreviated version display.

vol

This command displays the volume information for the file system specified by `fs`. If `fs` is not specified, the current file system is used. If `-n` is specified, then the volume label for `fs` will be set to `VolumeLabel`. The maximum length for `VolumeLabel` is 11 characters.

```
vol [fs]
vol [fs] -n volume-label
vol [fs] -d
```

fs – The name of the file system.

volume-label – The name of the file system. The following characters cannot be used:

% ^ * + = [] | : ; " < > ? /

No spaces are allowed in the volume label.

-d – Empty volume label.

Appendix C

Programming Reference

This appendix gives guidance on the programming environment associated with the UEFI Shell. The UEFI Shell provides programmatic interfaces that are not part of the main UEFI specification. The data in this reference should provide some insight into the programmatic interactions that are possible. This appendix is intended to be a useful summary of the UEFI Shell programming environment. However, if more details are required, refer to the UEFI Shell Specification.

Script-based Programming

Even though Appendix B is really intended as the enumeration of all of the shell commands that can be executed by a script, a few aspects of the scripting environment require additional explanation beyond the descriptions of the commands themselves. These topics include:

- Parameter passing
- Redirection and piping
- Return codes
- Environment variables

Parameter Passing

Positional parameters are the first ten arguments (%0–%9) passed from the command line into a UEFI Shell script. The first parameter after the UEFI Shell script name becomes %1, the second %2, the third %3, and so on. The argument %0 is the full path name of the script itself.

When executing the UEFI Shell script, the %n is replaced by the corresponding argument on the command line that invoked the script. If a positional parameter is referenced in the UEFI Shell script but that parameter was not present, then an empty string is substituted.

When passing parameters to commands, the associated commands might at times accept parameters with wildcards in them. The most common use of such wildcards is in reference to file names. The asterisk (*) and question mark (?) are often used for filename expansion. The asterisk would be used in a case where one is searching for 0 or more characters in a filename. For example, the reference to a filename of “*.*” would be one searching for any valid file names with any valid extension. This typically means “give me everything.” The question mark is intended to be used when looking to match exactly one character in a given filename. An example

of this would be “?I?.TXT” where items that might match this sequence would be JIM.TXT and KIM.TXT.

Redirection and Piping

Depending on the background of the reader, mixing the terms redirection and piping may not seem natural. For purposes of this section, let us explicitly define the terms:

- *Redirection* – The ability to redirect the output of an application or command to a file or an environment variable. This also includes the ability to use the content of a file or an environment variable as the standard input to an application or command.
- *Command Piping* – The ability to channel the output of an application or command and feed the data to the standard input of another program.

Redirection

When using output redirection, there are several options available for both the source of the data as well as the output of the data. The command syntax for output redirection is:

- *Command [options] > Target* – Redirect the output of a command to a target. This will create a new *Target* and will overwrite any pre-existing item of the same name.
- *Command [options] >> Target* – Append the output of a command to the *Target* location.

It should be noted that the aforementioned *Target* location can be either a traditional file on some non-volatile media or it can be a volatile environment variable. The latter is introduced to support the operation of scripting logic even in a read-only type of environment. Table C.1 summarizes the redirection character sequences.

Table C.1: Output Redirection Support

	Unicode	ASCII	Unicode Variable	ASCII Variable
Standard Output	>	>a	>v	>av
Standard Error	2>	2>a	2>v	2>av
Standard Output - Append	>>	>>a	>>v	>>av
Standard Error - Append	2>>	2>>a	2>>v	2>>av

When using input redirection, the content of a file or environment variable is read and used as the standard input to an application or shell command. The command syntax for input redirection is:

- *Command [options] < Source* – Use the *Source* as the standard input for the *Command*.

Table C.2 summarizes the input redirection character sequences.

Table C.2: Input Redirection Support

	Unicode	ASCII	Unicode Variable	ASCII Variable
Sequence	<	<a	<v	<av

Command Piping

By using the pipe (|) character, a data channel is formed that takes the standard Unicode output of a file and feeds the data as standard input to another program. The format for this support is as follows:

- *Command [options] | Command*

This capability is found in most modern shells and since many common utilities presume the use of pipe operations, this enables maximal environment compatibility for those who port their favorite utilities to this environment. Table C.3 summarizes command piping support.

Table C.3: Command Piping Support

Character Sequence	Description
	Pipe output of a command to another program in UCS-2 format.
a	Pipe output of a command to another program in ASCII format.

Return Codes

During the execution of most shell commands, a return status is given when that command completes execution. In the UEFI Shell specification, there is a `SHELL_STATUS` set of return codes used by shell commands.

The `lasterror` shell variable allows scripts to test the results of the most recently executed command using the `if` command. This variable is maintained by the shell,

is read-only, and cannot be modified by command `set`. An example of a script testing to see if a command succeeded would be:

```
PROGRAM.EFI
if %lasterror% == 0 then goto success
echo PROGRAM.EFI had an error
:success
echo PROGRAM.EFI succeeded
```

Environment Variables

Environment variables are variables that can hold user-specified contents and can be employed on the command line or in scripts. Each environment variable has a case-sensitive name (a C-style identifier) and a string value. Environment variables can be either volatile (they will lose their value on reset or power-off) or non-volatile (they will maintain their value across reset or power-off).

Environment variables can be used on the command line by using `%variable-name%` where `variable-name` is the environment variable's name. Variable substitution is not recursive. Environment variables can also be retrieved by a UEFI Shell command by using the `GetEnv()` function.

Environment variables can be displayed or changed using the `set` shell command. They can also be changed by a UEFI Shell command using the `SetEnv()` function. Table C.4 lists the environment variables that have special meaning to the UEFI Shell. Each variable is defined to be Volatile (V) or Non-volatile (NV) as well as having Read-Only (RO) or Read-Write (RW) attributes associated with it:

Table C.4: Table C.4 Shell Environment Variables with Special Meaning

Variable	V/NV RO/RW	Description
Cwd	V/RO	The current working directory, including the current working file system.
Lasterror	V/RO	Last returned error from a UEFI Shell command or batch script
path	V/RW	The UEFI Shell has a default volatile environment variable path, which contains the default path that UEFI Shell will search if necessary. When the user wants to launch a UEFI application, UEFI Shell will first try to search the current directory if it exists, and then search the path list sequentially. If the application is found in one of the

Variable	V/NV RO/RW	Description
Profiles	NV/RO	paths, it will stop searching and execute that application. If the application is not found in all the paths, UEFI Shell will report the application is not found.
Shellsupport	V/RO	Reflects the current support level enabled by the currently running shell environment. The contents of the variable will reflect the text-based numeric version in the form that looks like: 3 This variable is produced by the shell itself and is intended as read-only, any attempt to modify the contents will be ignored.
uefishellversion	V/RO	Reflects the revision of the UEFI Shell specification that the shell supports. The contents are formatted as text: 2.00
Uefiversion	V/RO	Reflects the revision of the UEFI specification which the underlying firmware supports. The contents will look like this: 2.10

Non-Script-based Programming

While the users of shell environments may often focus on the shell commands and scripts, a wide variety of programmatic interfaces are available in the shell environment. In most cases, this kind of infrastructure comes into play when a user wants to create a shell extension (such as a new shell command). One should realize that the UEFI Shell environment is provided by a UEFI application that complies with the UEFI specification. This means that the return codes and underlying system services are at least partially composed of UEFI service calls and conventions. There are, however, two main protocols (programmatic services) that are introduced by the UEFI Shell environment:

- Shell Protocol
- Shell Parameters Protocol

Shell Protocol

Table C.5 summarizes the functions of the `EFI_SHELL_PROTOCOL` whose purpose is to provide shell services to UEFI applications. This protocol is the workhorse of the UEFI Shell environment and is used to provide abstractions to services that facilitate the interaction with the underlying UEFI services as well as shell features.

Table C.5: Shell Protocol Functions

Function	Description
<code>BatchIsActive</code>	This function tells whether any script files are currently being processed.
<code>CloseFile</code>	This function closes a specified file handle. All “dirty” cached file data is flushed to the device, and the file is closed. In all cases the handle is closed.
<code>CreateFile</code>	This function creates an empty new file or directory with the specified attributes and returns the new file’s handle.
<code>DeleteFile</code>	This function closes and deletes a file. In all cases the file handle is closed.
<code>DeleteFileByName</code>	This function deletes the file specified by the file handle.
<code>DisablePageBreak</code>	This function disables the page break output mode.
<code>EnablePageBreak</code>	This function enables the page break output mode.
<code>Execute</code>	This function creates a nested instance of the shell and executes the specified command with the specified environment.
<code>FindFiles</code>	This function searches for all files and directories that match the specified file pattern. The file pattern can contain wild-card characters.
<code>FindFilesInDir</code>	This function returns all files in a specified directory.
<code>FlushFile</code>	This function flushes all modified data associated with a file to a device.
<code>FreeFileList</code>	This function cleans up the file list and any related data structures. It has no impact on the files themselves.
<code>GetCurDir</code>	This function returns the current directory on a device.
<code>GetDeviceName</code>	This function gets the user-readable name of the device specified by the device handle.
<code>GetDevicePathFromFilePath</code>	This function gets the device path associated with a mapping.

Function	Description
GetDevicePathFromMap	This function converts a file system style name to a device path, by replacing any mapping references to the associated device path.
GetEnv	This function returns the current value of the specified environment variable.
GetFileInfo	This function allocates a buffer to store the file's information. It's the caller's responsibility to free the buffer.
GetFilePathFromDevicePath	This function converts a device path to a file system path by replacing part, or all, of the device path with the file-system mapping.
GetFilePosition	This function returns the current file position for the file handle.
GetFileSize	This function returns the size of the specified file.
GetHelpText	This function returns the help information for the specified command.
GetMapFromDevicePath	This function returns the mapping which corresponds to a particular device path.
GetPageBreak	User can use this function to determine current page break mode.
IsRootShell	This function informs the user whether the active shell is the root shell.
OpenFileByName	This function opens the specified file and returns a file handle.
OpenFileList	This function opens the files that match the path pattern specified.
OpenRoot	This function opens the root directory of a device and returns a file handle to it.
OpenRootByHandle	This function returns the root directory of a file system on a particular handle.
ReadFile	This function reads the requested number of bytes from the file at the file's current position and returns them in a buffer.
RemoveDupInFileList	This function deletes the duplicate files in the given file list.
SetAlias	This function adds or removes the alias for a specific shell command.
SetCurDir	This function changes the current directory on a device.
SetEnv	This function changes the current value of the specified environment variable.

Function	Description
SetFileInfo	This function sets the file information of an opened file handle.
SetFilePosition	This function sets the current read/write file position for the handle to the position supplied.
SetMap	This function creates, updates, or deletes a mapping between a device and a device path.
WriteFile	This function writes the specified number of bytes to the file at the current file position. The current file position is also advanced by the actual number of bytes written.
ExecutionBreak	Event signaled by the UEFI Shell when the user presses Ctrl-C to indicate that the current UEFI Shell command execution should be interrupted.
MajorVersion	The major version of the shell environment.
MinorVersion	The minor version of the shell environment.

Shell Parameters Protocol

Table C.6 summarizes the functions of the `EFI_SHELL_PARAMETERS_PROTOCOL` whose purpose is to handle the shell application’s arguments. This protocol handles state information associated with the command line as well as the current input, output, and error consoles.

Table C.6: Shell Parameters Protocol Functions

Parameter	Description
Argv	Points to an Argc-element array of pointers to null-terminated strings containing the command-line parameters. The first entry in the array is always the full file path of the executable. Any quotation marks that were used to preserve whitespace have been removed.
Argc	The number of elements in the Argv array.
StdIn	The file handle for the standard input for this executable. This may be different from the ConInHandle in the EFI SYSTEM TABLE.
StdOut	The file handle for the standard output for this executable. This may be different from the ConOutHandle in the EFI SYSTEM TABLE.
StdErr	The file handle for the standard error output for this executable. This may be different from the StdErrHandle in the EFI SYSTEM TABLE.

Appendix D

UEFI Shell Library

This appendix provides an annotated reference for the standard macros, functions, and data structures in the UEFI Shell Library. This library is designed to provide a broad range of services for using the UEFI and UEFI Shell APIs, in addition to common functions related to strings, files, and so on.

Functions

This section describes all of the functions in the UEFI Shell Developer Kit Shell library.

File I/O Functions

The UEFI Shell Library provides a set of functions that operate on file I/O. Table D.1 lists the file I/O support functions that are described in the following sections. For more information about `EFI_FILE_INFO` and `EFI_FILE` please refer to the UEFI specification.

Table D.1: File I/O Functions

Function Name	Function Description
<code>ShellCloseFile</code>	Closes the file handle.
<code>ShellCloseFileMetaArg</code>	Closes the files that were previously opened using <code>ShellOpenFileMetaArg</code> .
<code>ShellCreateDirectory</code>	Creates a directory by the directory name.
<code>ShellDeleteFile</code>	Closes and deletes the file handle.
<code>ShellDeleteFileByName</code>	Deletes a file by name.
<code>ShellFileExists</code>	Determines if a given file name exists.
<code>ShellFileHandleReadLine</code>	Reads a single line of text from a handle into an existing buffer, excluding the terminating <code>\n</code> character, and return whether it was ASCII or Unicode.
<code>ShellFileHandleReturnLine</code>	Reads a single line of text from a handle into an allocated buffer, excluding the terminating <code>\n</code> character and return whether it was ASCII or Unicode.

Function Name	Function Description
ShellFindFilePath	Find a file by searching the current working directory and then the path environment variable.
ShellFindFilePathEx	Find a file by searching the current working directory and then the path environment variable, using zero or more alternate file extensions.
ShellFindFirstFile	Gets the first file in a directory.
ShellFindNextFile	Gets the next file in a directory.
ShellFlushFile	Flushes data back to the file handle.
ShellGetFileInfo	Gets the file information from an open file handle and stores it in a buffer allocated from pool.
ShellGetFileSize	Gets the size of a file.
ShellGetFilePosition	Gets a file's current position.
ShellIsDirectory	Returns whether the specified file path is a directory.
ShellIsFile	Returns whether the specified file path represents a file.
ShellIsFileInPath	Returns whether the specified file is in the current working directory or the path.
ShellOpenFile	Opens a file.
ShellOpenFileByName	Opens a file specified by a file name.
ShellOpenFileByDevicePath	Opens a file specified by a device path.
ShellOpenFileMetaArg	Open files based on a file name that may contain wildcards.
ShellSetFileInfo	Sets the file information to an open file handle.
ShellSetFilePosition	Sets a file's current position.
ShellReadFile	Reads data from the file.
ShellWriteFile	Writes data to the file.

Miscellaneous Functions

The UEFI Shell Library also provides a number of additional functions that abstract the UEFI Shell protocol.

Table D.2: Miscellaneous Functions

Function Name	Function Description
ShellExecute	Parse and execute a command line.
ShellGetCurrentDir	Return the current working directory for the currently selected file system or the specified file system.
ShellGetExecutionBreakFlag	Returns whether Ctrl-C has been pressed by the user.
ShellGetEnvironmentVariable	Returns the value of the specified environment variable.
ShellInitialize	Initializes the shell library. Normally only used by the shell itself.
ShellSetEnvironmentVariable	Changes the value of the specified environment variable.
ShellSetPageBreakMode	Sets (enables or disables) the page break mode.

Command Line Parsing

The command-line parsing functions find and validate command-line options.

Table D.3: Command-Line Parsing Functions

Function Name	Function Description
ShellCommandLineCheckDuplicate	Detect if a command-line argument occurred more than once on the command-line.
ShellCommandLineFreeVarList	Cleans up the command-line arguments returned from the command-line parsing functions.
ShellCommandLineGetCount	Return the number of command-line parameters that were passed, excluding flags.
ShellCommandLineGetFlag	Checks the parsed command-line options for a specific flag.
ShellCommandLineGetValue	Return the value associated with a specific flag.
ShellCommandLineGetRawValue	Return the raw value associated with the specific flag.

Function Name	Function Description
ShellCommandLineParseEx	Checks the command-line arguments passed in against the list of valid ones. Adds more flexibility for numeric arguments.
ShellCommandLineParse	Checks the command-line arguments passed in against the list of valid ones.

Text I/O

The text I/O functions display text on the console as well as taking certain types of user input.

Table D.4: Text I/O Functions

Function Name	Function Description
ShellPrintEx	Print a printf-style string at the specified row and column using a Unicode format string.
ShellPrintHelp	Prints the specified section of the help file content for a specific command.
ShellPrintHiiEx	Print a printf-style string at the specified row and column using a string from the HII database.
ShellPromptForResponse	Prompt the user and return the resulting answer.
ShellPromptForResponseHii	Prompt the user and return the resulting answer, using a string from the HII database.

String Functions

Table D.5 lists the support functions for handling strings. These are in addition to those already found in the UEFI libraries.

Table D.5: String Functions

Function Name	Function Description
ShellCopySearchAndReplace	Replace each instance of one string with another.
ShellConvertStringToInt64	Convert a string to a 64-bit unsigned integer or else return an error.
ShellHexStrToIntn	Converts a string to an unsigned integer. Supports hexadecimal only.
ShellIsDecimalDigitCharacter	Returns whether the character is 0-9.
ShellIsHexaDecimalDigitCharacter	Returns whether the character is 0-9, a-f or A-F or not.
ShellIsHexOrDecimalNumber	Returns whether an entire string is a valid number.
ShellStrToIntn	Converts the string to an unsigned integer. Supports decimal and hexadeciml (starting with 0x)
StrnCatGrow	Safely append with automatic string resizing given the length of the destination and the desired number of characters from the source.

ShellCloseFile()

This function closes a specified file handle. All “dirty” cached file data is flushed to the device and the file is closed. In all cases the handle is closed.

Prototype

```
EFI_STATUS
EFIAPI
ShellCloseFile (
    IN SHELL_FILE_HANDLE *FileHandle
);
```

Parameters

FileHandle – Pointer to the file handle to be closed.

Status Codes Returned

`EFI_SUCCESS` – The file was closed successfully.

`EFI_INVALID_PARAMETER` – The file handle was invalid.

ShellCloseFileMetaArg()

This function closes all of the files and frees the list of files returned from `ShellOpenFileMetaArg()`.

Prototype

```
EFI_STATUS
EFIAPI
ShellCloseFileMetaArg (
    IN OUT EFI_SHELL_FILE_INFO **ListHead
);
```

Parameters

`ListHead` – The pointer to the head of the list of files to free.

Status Codes Returned

`EFI_SUCCESS` – The operation was successful.

`EFI_INVALID_PARAMETER` – The list or entry in the list was invalid.

ShellCommandLineCheckDuplicate()

Determine if a parameter is duplicated and, if so, return its value.

Prototype

```
EFI_STATUS
EFIAPI
ShellCommandLineCheckDuplicate (
    IN CONST LIST_ENTRY *CheckPackage,
    OUT CHAR16           **Param
);
```

Parameters

`CheckPackage` – The list of command-line arguments to free.

`Param` – Pointer to a pointer to the returned parameter value, if a duplicate was found.

Status Codes Returned

`EFI_SUCCESS` – No parameters were duplicated.

`EFI_DEVICE_ERROR` – A duplicate was found and `Param` points to the pointer to the value.

ShellCommandLineFreeVarList()

Free the list of command-line arguments found by `ShellCommandLineParseEx()`.

Prototype

```
VOID
EFIAPI
ShellCommandLineFreeVarList (
    IN LIST_ENTRY *CheckPackage
);
```

Parameters

`CheckPackage` – The list of command-line arguments to free.

Status Codes Returned

None

ShellCommandLineGetCount()

Returns the number of arguments on the command-line, excluding any flags.

Prototype

```
UINTN
EFIAPI
ShellCommandLineGetCount (
```

```
IN CONST LIST_ENTRY *CheckPackage
);
```

Parameters

CheckPackage – List of parsed command-line arguments.

Return Values

Unsigned integer that indicates the number of arguments, or -1 if no parsing has occurred.

ShellCommandLineGetFlag()

Check for the presence of a flag parameter. Flag parameters are in the form “-<key>” or “/<key>” but do not have a value after the flag.

Prototype

```
BOOLEAN
EFIAPI
ShellCommandLineGetFlag (
    IN CONST LIST_ENTRY * CONST CheckPackage,
    IN CONST CHAR16      * CONST KeyString
);
```

Parameters

CheckPackage – List of parsed command-line arguments.

KeyString – Pointer to null-terminated string that specifies the key.

Return Values

TRUE – The flag is present on the command line.

FALSE – The flag is not present on the command line.

ShellCommandLineGetValue()

Returns the value associated with the specified key on the command line, if any. Value parameters are in the form “-<key> value” or “/<key> value”

Prototype

```
CONST CHAR16*
EFIAPI
ShellCommandLineGetValue (
    IN CONST LIST_ENTRY * CheckPackage,
    IN CHAR16           * KeyString
) ;
```

Parameters

CheckPackage – List of parsed command-line arguments.

KeyString – Pointer to null-terminated string that specifies the key.

Return Values

The pointer to a null-terminated string that contains the value associated with the key, or else NULL if the key is not present.

ShellCommandLineGetRawValue()

Returns the raw value at a specific position in the list of parsed command-line arguments. This is different from `ShellCommandLineGetFlag()` or `ShellCommandLineGetValue()` because, instead of specifying the key, this specifies the exact position.

Prototype

```
CONST CHAR16*
EFIAPI
ShellCommandLineGetRawValue (
    IN CONST LIST_ENTRY * CONST CheckPackage,
    IN UINTN            Position
) ;
```

Parameters

CheckPackage – List of parsed command-line arguments.

Position – Unsigned integer that specifies the index of the argument to retrieve from the list.

Return Values

The pointer to the command-line argument or else NULL if no such argument exists.

ShellCommandLineParseEx()

Check the command-line arguments against the list of valid ones and return the results in the list.

Note: The library header file also provides the macro “`ShellCommandLineParse`” which is equivalent to this function, but with the `AlwaysAllowNumbers` parameter set to FALSE.

Prototype

```
EFI_STATUS
EFIAPI
ShellCommandLineParseEx (
    IN CONST SHELL_PARAM_ITEM *CheckList,
    OUT LIST_ENTRY           **CheckPackage,
    OUT CHAR16                **ProblemParam OPTIONAL,
    IN BOOLEAN                 AutoPageBreak,
    IN BOOLEAN                 AlwaysAllowNumbers
);
```

Parameters

`CheckList` – The pointer to list of parameters to check. See Shell Parameters in “Data Structures.”

`CheckPackage` – The returned list of checked values.

`ProblemParam` – Optional pointer to pointer to string that returns the parameter that caused failure.

`AutoPageBreak` – Will automatically set page break to enabled.

`AlwaysAllowNumbers` – Will never fail for number based flags.

Status Codes Returned

`EFI_SUCCESS` – The operation completed successfully.

`EFI_OUT_OF_RESOURCES` – A memory allocation failed.

`EFI_INVALID_PARAMETER` – A parameter was invalid.

`EFI_VOLUME_CORRUPTED` – The command line was corrupt.

EFI_DEVICE_ERROR – The commands contained 2 opposing arguments. One of the command line arguments was returned in `ProblemParam` if provided.

EFI_NOT_FOUND – An argument required a value that was missing. The invalid command line argument was returned in `ProblemParam` if provided.

ShellCopySearchAndReplace()

This function finds zero or more instances of a string in another string and replaces it. Upon successful return the `NewString` is a copy of `SourceString` with each instance of `FindTarget` replaced with `ReplaceWith`.

Prototype

```
EFI_STATUS
EFIAPI
ShellCopySearchAndReplace(
    IN CHAR16 CONST  *SourceString,
    IN OUT CHAR16     *NewString,
    IN UINTN          NewSize,
    IN CONST CHAR16   *FindTarget,
    IN CONST CHAR16   *ReplaceWith,
    IN CONST BOOLEAN   SkipPreCarrot,
    IN CONST BOOLEAN   ParameterReplacing
);
```

Parameters

`SourceString` – The string with source buffer.

`NewString` – The string with resultant buffer.

`NewSize` – The size in bytes of `NewString`.

`FindTarget` – The string to look for.

`ReplaceWith` – The string to replace `FindTarget` with.

`SkipPreCarrot` – If TRUE will skip a `FindTarget` that has a '^' immediately before it.

`ParameterReplacing` – If TRUE will add "" around items with spaces.

Status Codes Returned

EFI_SUCCESS – The string was successfully copied with replacement.

EFI_INVALID_PARAMETER – SourceString was NULL, or NewString was NULL, or FindTarget was NULL, or ReplaceWith was NULL, or FindTarget had length < 1, or SourceString had length < 1.

EFI_BUFFER_TOO_SMALL – NewSize was less than the minimum size to hold the new string (truncation occurred).

ShellConvertStringToInt64()

This function verifies and converts a string to its numerical 64 bit representation. For hexadecimal, it must be preceded with a 0x, 0X, or ForceHex must be set to TRUE.

Prototype

```
EFI_STATUS
EFI API
ShellConvertStringToInt64(
    IN CONST CHAR16    *String,
    OUT UINT64         *Value,
    IN CONST BOOLEAN   ForceHex,
    IN CONST BOOLEAN   StopAtSpace
);
```

Parameters

String – The string to evaluate.

Value – Upon a successful return, the value of the conversion.

ForceHex – Boolean that specifies whether the String will be assumed to be all hexadeciml characters (TRUE) or the radix will be auto-detected (FALSE).

StopAtSpace – Boolean that specifies whether to halt upon finding a space (TRUE) or process the entire String (FALSE).

Status Codes Returned

EFI_SUCCESS – The conversion was successful.

EFI_INVALID_PARAMETER – String contained an invalid character.

`EFI_NOT_FOUND` – String was a number, but Value was NULL.

ShellCreateDirectory()

This function creates a directory of the directory names. If return is `EFI_SUCCESS`, the `FileHandle` is the directory's handle; otherwise, the `FileHandle` is NULL. If the file already exists, this function opens the existing directory.

Prototype

```
EFI_STATUS  
EFIAPI  
ShellCreateDirectory(  
    IN  CHAR16           *DirName,  
    OUT SHELL_FILE_HANDLE *FileHandle  
) ;
```

Parameters

`DirName` – A pointer to the directory name.

`FileHandle` – A pointer to the opened directory handle.

Status Codes Returned

`EFI_SUCCESS` – The file was opened.

`EFI_INVALID_PARAMETER` – One of the parameters has an invalid value.

`EFI_UNSUPPORTED` – The file path could not be opened.

`EFI_NOT_FOUND` – The specified file could not be found on the device or the file system could not be found on the device.

`EFI_NO_MEDIA` – The device has no media.

`EFI_MEDIA_CHANGED` – The device has a different medium in it or the medium is no longer supported.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – The file or medium is write protected.

`EFI_ACCESS_DENIED` – The file was read-only.

`EFI_OUT_OF_RESOURCES` – Not enough resources were available to open the file.

`EFI_VOLUME_FULL` – The volume is full.

ShellDeleteFile()

This function closes and deletes a file. In all cases, the file handle is closed. If the file cannot be deleted, the warning code `EFI_WARN_DELETE_FAILURE` is returned, but the handle is still closed.

Prototype

```
EFI_STATUS  
EFIAPI  
ShellDeleteFile (   
    IN SHELL_FILE_HANDLE *FileHandle  
);
```

Parameters

`FileHandle` – Pointer to the file handle to delete.

Status Codes Returned

`EFI_SUCCESS` – The file was closed and deleted, and the handle was closed.

`EFI_WARN_DELETE_FAILURE` – The handle was closed but the file was not deleted.

`EFI_INVALID_PARAMETER` – The file handle is invalid.

ShellDeleteFileByName()

This function deletes the file specified by the file name.

Prototype

```
EFI_STATUS  
EFIAPI  
ShellDeleteFileByName (  
    IN CONST CHAR16 *FileName  
) ;
```

Parameters

FileName – Pointer to a null-terminated string that specifies the file name to delete.

Status Codes Returned

EFI_SUCCESS – The file was deleted successfully.

EFI_WARN_DELETE_FAILURE – The handle was closed, but the file was not deleted.

EFI_INVALID_PARAMETER – One of the parameters has an invalid value.

EFI_NOT_FOUND – The specified file could not be found on the device or the file system could not be found on the device.

EFI_NO_MEDIA – The device has no medium.

EFI_MEDIA_CHANGED – The device has a different medium in it or the medium is no longer supported.

EFI_DEVICE_ERROR – The device reported an error.

EFI_VOLUME_CORRUPTED – The file system structures are corrupted.

EFI_WRITE_PROTECTED – The file or medium is write-protected.

EFI_ACCESS_DENIED – The file was opened read-only.

EFI_OUT_OF_RESOURCES – Not enough resources were available to open the file.

ShellExecute()

This function causes the shell to parse and execute the command line. It creates a nested instance of the shell and executes the specified command (CommandLine)

with the specified environment (`Environment`). Upon return, the status code returned by the specified command is placed in `StatusCode`.

If `Environment` is NULL, then the current environment is used and all changes made by the commands executed will be reflected in the current environment. If the `Environment` is non-NUL, then the changes made will be discarded.

The `CommandLine` is executed from the current working directory on the current device.

Prototype

```
EFI_STATUS
EFIAPI
ShellExecute (
    IN EFI_HANDLE   ImageHandle,
    IN CHAR16       *CmdLine,
    IN BOOLEAN      Output
    IN CHAR16       **EnvironmentVariables OPTIONAL,
    OUT EFI_STATUS  *Status
);
```

Parameters

`ImageHandle` – A handle of an image that is initializing the library.

`CmdLine` – Command line.

`Output` – Boolean that specifies whether the output will be displayed (TRUE) or not displayed (FALSE).

`EnvironmentVariables` – Optional pointer to an array of environment variables in the form “`x=y`”. If NULL, then the current set of environment variables are used.

`Status` – On return, points to the status returned by the executed command.

Status Codes Returned

`EFI_SUCCESS` – The command executed successfully. `Status` holds the returned status.

`EFI_INVALID_PARAMETER` – The parameters are invalid.

`EFI_OUT_OF_RESOURCES` – Out of resources.

`EFI_UNSUPPORTED` – The operation is not allowed.

ShellFileExists()

This function determines if a file exists.

Prototype

```
EFI_STATUS
EFIAPI
ShellFileExists(
    IN CONST CHAR16 *Name
) ;
```

Parameters

Name – Pointer to a null-terminated string that specifies the path of the file to test.

Status Codes Returned

EFI_SUCCESS – The file specified by Name exists.

EFI_NOT_FOUND – The file specified by Name does not exist.

ShellFileHandleReturnLine()

This function reads a single line from a file handle. The \n is not included in the returned buffer. The returned buffer must be freed by the caller.

If the file position upon start is 0, then *Ascii* is updated based on the presence or absence of Unicode byte order marks. This value should not be changed for all operations with the same file.

Note: Lines returned by this function are **always** in Unicode UCS-2, even if the original file was in ASCII.

Prototype

```
CHAR16*
EFIAPI
ShellFileHandleReturnLine(
    IN SHELL_FILE_HANDLE Handle,
    IN OUT BOOLEAN      *Ascii
) ;
```

Parameters

`Handle` – File handle to read from.

`Ascii` – On entry, specifies whether the file is ASCII (TRUE) or Unicode UCS2 (FALSE). On exit, if the file position was at 0, this indicates whether the Unicode byte order marks were found at the beginning of the file (FALSE) or not (TRUE).

Return Values

Pointer to a null-terminated string, or NULL if there are no more lines.

ShellFileHandleReadLine()

This function reads a single line from a file handle into a provided buffer. The \n is not included in the provided buffer.

If the file position upon start is 0, then `Ascii` is updated based on the presence or absence of Unicode byte order marks. This value should not be changed for all operations with the same file.

Note: Lines returned by this function are **always** in Unicode UCS-2, even if the original file was in ASCII.

Prototype

```
EFI_STATUS
EFIAPI
ShellFileHandleReadLine (
    IN SHELL_FILE_HANDLE    Handle,
    IN OUT CHAR16           *Buffer,
    IN OUT UINTN            *Size,
    IN BOOLEAN              Truncate,
    IN OUT BOOLEAN          *Ascii
);
```

Parameters

`Handle` – File handle to read from.

`Buffer` – The pointer to buffer to read into. If this function returns `EFI_SUCCESS`, then on exit, `Buffer` will contain a Unicode UCS-2 string, even if the file being read is ASCII.

`Size` – On entry, pointer to an unsigned integer that specifies number of bytes in `Buffer`. On exit, unchanged unless `Buffer` is too small to contain the next line of the file. In that case `Size` is set to the number of bytes needed to hold the next line of the file (as a UCS2 string, even if it is an ASCII file).

`Truncate` – Boolean that specifies that, if the buffer is too small and this TRUE, the line will be truncated. If the buffer is too small and `Truncate` is FALSE, then no read will occur. If the buffer is large enough, this has no effect.

`Ascii` – On entry, specifies whether the file is ASCII (TRUE) or Unicode UCS2 (FALSE). On exit, if the file position was at 0, this indicates whether the Unicode byte order marks were found at the beginning of the file (FALSE) or not (TRUE).

Status Codes Returned

`EFI_SUCCESS` – The operation was successful. The line is stored in `Buffer`.

`EFI_END_OF_FILE` – There are no more lines in the file.

`EFI_INVALID_PARAMETER` – `Handle` was NULL or `Size` was NULL.

`EFI_BUFFER_TOO_SMALL` – `Size` was not large enough to store the line. `Size` was updated to the minimum space required.

ShellFindFilePath()

This function finds a file by searching the current working directory and then the directories specified by the `path` environment variable.

Prototype

```
CHAR16 *
EFIAPI
ShellFindFilePath (
    IN CONST CHAR16 *FileName
);
```

Parameters

`FileName` – Pointer to a null-terminated string that specifies the name of the file to search for.

Return Values

The pointer to the complete path of the file found, or `NULL` if the file was not found. The caller is responsible for freeing this string.

ShellFindFilePathEx()

This function finds a file by searching the current working directory and then the directories specified by the `path` environment variable, optionally trying alternate file extensions in the file name. If the file name specified is not found it will try again for each file extension in `FileExtension` in the order provided and return the first one successful. If `FileExtension` is `NULL`, then the behavior is identical to `ShellFindFilePath`.

Prototype

```
CHAR16 *
EFIAPI
ShellFindFilePathEx (
    IN CONST CHAR16 *FileName,
    IN CONST CHAR16 *FileExtension
) ;
```

Parameters

`FileName` – Pointer to a null-terminated string that specifies the file name to search for.

`FileExtension` – Pointer to a null-terminated string that specifies the list of zero or more possible file extensions, delimited by a semicolon.

Return Values

Pointer to null-terminated string that indicates the complete path of the file found, or `NULL` if none was found. The caller is responsible for freeing this string.

ShellFindFirstFile()

This function opens a directory and gets the first file's information in the directory. Caller can use `ShellFindNextFile()` to get other files.

Prototype

```
EFI_STATUS
EFIAPI
ShellFindFirstFile(
    IN SHELL_FILE_HANDLE DirHandle,
    OUT EFI_FILE_INFO    **Buffer
) ;
```

Parameters

`DirHandle` – The handle of the directory to search in.

`Buffer` – The pointer to the buffer containing the first file's returned info. The structure `EFI_FILE_INFO` is defined in the UEFI Specification.

Status Codes Returned

`EFI_SUCCESS` – Found the first file.

`EFI_NOT_FOUND` – Cannot find the directory.

`EFI_NO_MEDIA` – The device has no media.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

ShellFindNextFile()

This function retrieves the next entries from a directory. To use this function, the caller must first call the `ShellFindFirstFile()` function to get the first directory entry. Subsequent directory entries are retrieved by using the `ShellFindNextFile()` function. This function can be called several times to get each entry from the directory. If the call of `ShellFindNextFile()` retrieved the last directory entry, the next call of this function will set `*NoFile` to TRUE and free the buffer.

Prototype

```
EFI_STATUS
EFIAPI
ShellFindNextFile(
    IN SHELL_FILE_HANDLE DirHandle,
    IN OUT EFI_FILE_INFO *Buffer,
```

```
OUT BOOLEAN          *NoFile
) ;
```

Parameters

`DirHandle` – The file handle of the directory to search in.

`Buffer` – On entry, specifies the file information for the previous file found. On exit, contains the next file's information.

`NoFile` – On return, indicates whether any more files exist.

Status Codes Returned

`EFI_SUCCESS` – Found the next file.

`EFI_NO_MEDIA` – The device has no media.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

ShellFlushFile()

This function flushes all modified data associated with a file to a device.

Prototype

```
EFI_STATUS
EFIAPI
ShellFlushFile (
    IN SHELL_FILE_HANDLE FileHandle
) ;
```

Parameters

`FileHandle` – The file handle on which to flush data.

Status Codes Returned

`EFI_SUCCESS` – The data was flushed.

`EFI_NO_MEDIA` – The device has no media.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – The file or medium is write-protected.

`EFI_ACCESS_DENIED` – The file was opened for read-only access.

`EFI_VOLUME_FULL` – The volume is full.

SHELL_FREE_NON_NULL()

This macro frees the block of memory specified by a variable if it is non-NULL, and then sets the variable to NULL.

Prototype

```
#define SHELL_FREE_NON_NULL(Pointer) \
    do { \
        if ((Pointer) != NULL) { \
            FreePool((Pointer)); \
            (Pointer) = NULL; \
        } \
    } while(FALSE)
```

Parameters

`Pointer` – L-value that contains the pointer to the block of memory to be freed.

Return Values

None

ShellGetCurrentDir()

This function returns the current working directory for the specified file system. If the `DeviceName` is NULL, it returns the current device's current directory. If the `DeviceName` is not NULL, it returns the current directory name for the specified device.

Note: The current directory string should exclude the tailing backslash character.

Prototype

```
CONST CHAR16*
EFIAPI
ShellGetCurrentDir (
    IN CHAR16 * CONST DeviceName OPTIONAL
);
```

Parameters

DeviceName – Optional pointer to a null-terminated string that specifies the device to return the current directory for. If not specified, then the current working directory is returned.

Return Values

Pointer to a null-terminated string that contains the current directory for the specified device, or NULL if the device doesn't exist or doesn't have a current directory.

ShellGetEnvironmentVariable()

This function returns the value of the specified environment variable.

Prototype

```
CHAR16 *
EFIAPI
ShellGetEnvironmentVariable (
    IN CONST CHAR16 *EnvKey
);
```

Parameters

EnvKey – A pointer to the environment variable name.

Return Values

Pointer to the null-terminated string that contains the environment variable value, or NULL if the environment variable does not exist.

ShellGetExecutionBreakFlag()

This function retrieves the status of the break execution flag. This function is useful to check whether the application is being asked to halt by the UEFI Shell when, for example, the user presses Ctrl-C.

Prototype

```
BOOLEAN
EFI API
ShellGetExecutionBreakFlag (
    VOID
);
```

Parameters

None

Return Values

Boolean that indicates whether a break has been requested (TRUE) or not (FALSE).

ShellGetFileInfo()

This function gets the file information from an open file handle and stores it in an allocated buffer. The caller is responsible for freeing this buffer.

Prototype

```
EFI_FILE_INFO *
EFI API
ShellGetFileInfo (
    IN SHELL_FILE_HANDLE FileHandle
);
```

Parameters

FileHandle – The file handle associated with the file about which information is being requested.

Return Values

A pointer to the returned file information, or NULL if there was an error. The structure **EFI_FILE_INFO** is defined in the UEFI specification.

ShellGetPosition()

This function returns the current file position for the file handle. For directories, the current file position has no meaning outside of the file system driver and as such the operation is not supported.

An error is returned if *FileHandle* is a directory.

Prototype

```
EFI_STATUS
EFIAPI
ShellGetPosition (
    IN SHELL_FILE_HANDLE FileHandle,
    OUT UINT64           *Position
);
```

Parameters

FileHandle – The file handle on which to get the current position.

Position – Byte position from the start of the file.

Status Codes Returned

`EFI_SUCCESS` – Data was accessed.

`EFI_INVALID_PARAMETER` – File handle is invalid or *Position* is NULL.

`EFI_UNSUPPORTED` – The request is not valid on open directories.

ShellGetFileSize()

This function returns a file's size

Prototype

```
EFI_STATUS
EFIAPI
ShellGetFileSize (
    IN SHELL_FILE_HANDLE FileHandle,
    OUT UINT64           *Size
);
```

Parameters

`FileHandle` – The file handle associated with the file.

`Size` – Pointer to the returned size of this file.

Status Codes Returned

`EFI_SUCCESS` – The operation completed successfully. `Size` was updated with the file's size.

`EFI_DEVICE_ERROR` – Cannot access the file.

ShellHexStrToIntn()

This function return the number converted from a hexadecimal representation of a number.

Note: This function cannot be used when `(UINTN)(-1)`, `(0xFFFFFFFF)` may be a valid result. Use `ShellConvertStringToInt64` instead.

Prototype

```
UINTN
EFIAPI
ShellHexStrToIntn(
    IN CONST CHAR16 *String
);
```

Parameters

`String` – Pointer to a null-terminated string that specifies the string representation of a number.

Return Values

The unsigned integer result of the conversion, or `(UINTN)(-1)` if there was an error.

ShellInitialize()

This function causes the shell library to initialize itself. If the shell library is already initialized it will de-initialize all the current protocol pointers and re-populate them again.

When the library is used with `PcdShellLibAutoInitialize` set to true this function will return `EFI_SUCCESS` and perform no actions.

Prototype

```
EFI_STATUS
EFIAPI
ShellInitialize (
    VOID
);
```

Parameters

None

Status Codes Returned

`EFI_SUCCESS` – Initialization completed successfully.

ShellIsDecimalDigitCharacter()

This function checks whether a Unicode character is a decimal character. The valid characters are L'0' to L'9'.

Prototype

```
BOOLEAN
EFIAPI
ShellIsDecimalDigitCharacter (
    IN CHAR16 Char
);
```

Parameters

Char – The character to test.

Return Values

Boolean that indicates whether the character is a valid decimal character (TRUE) or not (FALSE).

ShellIsDirectory()

This function returns whether the specified file name represents a directory.

Prototype

```
EFI_STATUS  
EFIAPI  
ShellIsDirectory(  
    IN CONST CHAR16 *DirName  
) ;
```

Parameters

DirName – Pointer to a null-terminated string that specifies the path to test.

Status Codes Returned

EFI_SUCCESS – The path represents a directory.

EFI_NOT_FOUND – The path does not represent a directory.

ShellIsFile()

This function returns whether the file exists in the current working directory.

Prototype

```
EFI_STATUS  
EFIAPI  
ShellIsFile(  
    IN CONST CHAR16 *Name  
) ;
```

Parameters

Name – Pointer to a null-terminated string that specifies the file name to test.

Status Codes Returned

EFI_SUCCESS – The path represents a file.

EFI_NOT_FOUND – The path does not represent a file.

ShellIsFileInPath()

This function returns whether the specified file exists in the current working directory or any of the directories specified by the `path` environment variable.

Prototype

```
EFI_STATUS
EFIAPI
ShellIsFileInPath(
    IN CONST CHAR16 *Name
);
```

Parameters

`Name` – Pointer to a null-terminated string that specifies the file name to test.

Status Codes Returned

`EFI_SUCCESS` – The path represents a file.

`EFI_NOT_FOUND` – The path does not represent a file.

ShellIsHexaDecimalDigitCharacter()

Returns whether a Unicode character is a hexadecimal character. This function checks if a Unicode character is a numeric or hexadecimal character. The valid hexadeciml characters are L'0' to L'9', L'a' to L'f', or L'A' to L'F'.

Prototype

```
BOOLEAN
EFIAPI
ShellIsHexaDecimalDigitCharacter (
    IN CHAR16 Char
);
```

Parameters

`Char` – The character to test.

Return Values

Boolean that indicates whether the character is a valid hexadeciml character (`TRUE`) or not (`FALSE`).

ShellIsHexOrDecimalNumber()

This function checks whether an entire string is a valid number. For hexadecimal numbers, the number must be preceded with a 0x, 0X, or else ForceHex is set to TRUE.

Prototype

```
BOOLEAN
EFI API
ShellIsHexOrDecimalNumber (
    IN CONST CHAR16    *String,
    IN CONST BOOLEAN   ForceHex,
    IN CONST BOOLEAN   StopAtSpace
);
```

Parameters

String – Pointer to a null-terminated string to evaluate.

ForceHex – Boolean that specifies whether the entire string should be evaluated as a hexadecimal representation (TRUE) or a possible decimal or hexadecimal string (FALSE).

StopAtSpace – Boolean that specifies whether to halt upon finding a space (TRUE) or keep going (FALSE).

Return Values

Boolean that indicates whether the string is valid (TRUE) or not (FALSE).

ShellOpenFileByDevicePath()

This function opens a file or a directory by device path with the specified mode. If this function returns EFI_SUCCESS, the *FileHandle* is the opened file's handle; otherwise, the *FileHandle* is NULL. The *Attributes* is valid only for EFI_FILE_MODE_CREATE.

Prototype

```
EFI_STATUS
EFI API
ShellOpenFileByDevicePath (
    IN OUT EFI_DEVICE_PATH_PROTOCOL **FilePath,
```

```

OUT EFI_HANDLE           *DeviceHandle,
OUT SHELL_FILE_HANDLE   *FileHandle,
IN  UINT64                OpenMode,
IN  UINT64                Attributes
) ;

```

Parameters

`FilePath` – On entry, a pointer to a device path. On exit, a pointer to the remainder of the device path.

`DeviceHandle` – On exit, a pointer to the opened device handle.

`FileHandle` – On exit, a pointer to the opened file handle.

`OpenMode` – File open mode. See `EFI_FILE_MODE_x` in the UEFI specification.

`Attributes` – The file's *File attributes*.

Status Codes Returned

`EFI_SUCCESS` – The file was opened.

`EFI_INVALID_PARAMETER` – One of the parameters has an invalid value.

`EFI_UNSUPPORTED` – Could not open the file path.

`EFI_NOT_FOUND` – The specified file could not be found on the device or the file system could not be found on the device.

`EFI_NO_MEDIA` – The device has no medium.

`EFI_MEDIA_CHANGED` – The device has a different medium in it or the medium is no longer supported.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – The file or medium is write-protected.

`EFI_ACCESS_DENIED` – The file was opened read-only.

`EFI_OUT_OF_RESOURCES` – Not enough resources were available to open the file.

`EFI_VOLUME_FULL` – The volume is full.

ShellOpenFileByName()

This function opens a file or a directory by file name, with the specified mode.

If this function returns `EFI_SUCCESS`, the *FileHandle* is the opened file's handle; otherwise, the *FileHandle* is NULL. The *Attributes* is valid only for `EFI_FILE_MODE_CREATE`.

Prototype

```
EFI_STATUS
EFIAPI
ShellOpenFileByName (
    IN     CHAR16           *FileName,
    OUT    SHELL_FILE_HANDLE *FileHandle,
    IN     UINT64            OpenMode,
    IN     UINT64            Attributes
);
```

Parameters

`FileName` – A pointer to the file name.

`FileHandle` – A pointer to the opened file handle.

`OpenMode` – File open mode.

`Attributes` – The file's *File* attributes.

Status Codes Returned

`EFI_SUCCESS` – The file was opened.

`EFI_INVALID_PARAMETER` – One of the parameters has an invalid value.

`EFI_UNSUPPORTED` – Could not open the file path.

`EFI_NOT_FOUND` – The specified file could not be found on the device or the file system could not be found on the device.

`EFI_NO_MEDIA` – The device has no medium.

`EFI_MEDIA_CHANGED` – The device has a different medium in it or the medium is no longer supported.

`EFI_DEVICE_ERROR` – The device reported an error or cannot get the file path according to the *FileName*.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – An attempt was made to create a file, or to open a file for writing, when the medium is write-protected.

`EFI_ACCESS_DENIED` – The service denied access to the file.

`EFI_OUT_OF_RESOURCES` – Not enough resources were available to open the file.

`EFI_VOLUME_FULL` – The volume is full.

ShellOpenFileMetaArg()

This function opens the files that match the path specified. It uses the *Arg* pointer to open all the matching files. Each matched file has a `SHELL_FILE_ARG` structure to record the file information. These structures are placed on the list *ListHead*. Users can get the `SHELL_FILE_ARG` structures from *ListHead* to access each file. This function supports wildcards.

Prototype

```
EFI_STATUS
EFIAPI
ShellOpenFileMetaArg (
    IN     CHAR16      *Arg,
    IN     UINT64       OpenMode,
    IN OUT EFI_LIST_ENTRY *ListHead
);
```

Parameters

Arg – Pointer a null-terminated string that specifies the path.

`OpenMode` – Bitmask that specifies the mode of the newly opened file. See `EFI_FILE_MODE_x` in the UEFI specification.

`ListHead` – A list of files that match the specified path.

Status Codes Returned

`EFI_SUCCESS` – The file list was successfully created.

ShellPrintEx()

Print at a specific location on the screen. This function will move the cursor to a given screen location and print the specified string. If -1 is specified for either the Row or Col the current screen location for BOTH will be used.

In addition to the standard %-based flags as supported by UefiLib's `Print()`, this supports the following additional flags:

- %N – Set output attribute to normal
- %H – Set output attribute to highlight
- %E – Set output attribute to error
- %B – Set output attribute to blue color
- %V – Set output attribute to green color

Note: The background color is controlled by the shell command `cls`.

Prototype

```
EFI_STATUS
EFIAPI
ShellPrintEx(
    IN INT32                  Col OPTIONAL,
    IN INT32                  Row OPTIONAL,
    IN CONST CHAR16           *Format,
    ...
);
```

Parameters

`Col` – The column at which to print.

`Row` – The row at which to print.

`Format` – The format string.

. . . – The variable argument list.

Status Codes Returned

`EFI_SUCCESS` – The printing was successful.

`EFI_DEVICE_ERROR` – The console device reported an error.

ShellPrintHelp()

This function prints help file/man page content for a UEFI Shell command.

Prototype

```
EFI_STATUS  
EFI API  
ShellPrintHelp (   
    IN CONST CHAR16 *CommandToGetHelpOn,  
    IN CONST CHAR16 *SectionToGetHelpOn,  
    IN BOOLEAN      PrintCommandText  
);
```

Parameters

`CommandToGetHelpOn` – Pointer to a null-terminated string that specifies the command name of the help file to be printed.

`SectionToGetHelpOn` – Pointer to the null-terminated string that specifies the section(s).

`PrintCommandText` – Boolean that specifies whether the command should be printed and then the help content (TRUE) or just the help content (FALSE).

Status Codes Returned

`EFI_SUCCESS` – The operation was successful.

`EFI_DEVICE_ERROR` – The help data format was incorrect.

`EFI_NOT_FOUND` – The help data could not be found.

ShellPrintHiiEx()

Print a string from the HII database at a specific location on the screen. This function will move the cursor to a given screen location and print the specified string. If -1 is specified for either the Row or Col the current screen location for BOTH will be used.

In addition to the standard %-based flags as supported by UefiLib's Print(), this supports the following additional flags:

- %N – Set output attribute to normal
- %H – Set output attribute to highlight
- %E – Set output attribute to error
- %B – Set output attribute to blue color
- %V – Set output attribute to green color

Note: The background color is controlled by the shell command cls.

Prototype

```
EFI_STATUS
EFIAPI
ShellPrintHiiEx(
    IN INT32                  Col OPTIONAL,
    IN INT32                  Row OPTIONAL,
    IN CONST CHAR8             *Language OPTIONAL,
    IN CONST EFI_STRING_ID     HiiFormatStringId,
    IN CONST EFI_HANDLE         HiiFormatHandle,
    ...
);
```

Parameters

Col – The column at which to print.

Row – The row at which to print.

Language – Pointer to a null-terminated ASCII string that specifies the language to retrieve. If NULL, then the current platform language is used.

HiiFormatStringId – The string identifier associated with the string to use to format the output.

HiiFormatHandle – The HII handle of the package list that contains the string specified by HiiFormatStringId.

. . . – The variable argument list.

ShellPromptForResponse()

This function prompts the user and returns the resulting answer to the caller. This function will display the requested question on the shell prompt and then wait for an appropriate answer to be input from the console.

If the Type parameter has a value of ShellPromptResponseTypeYesNo, ShellPromptResponseTypeQuitContinue or ShellPromptResponseTypeYesNoAllCancel, then Response points to the response enumerated value on exit. If the Type is ShellPromptResponseTypeFreeform then Response points to a CHAR16 * value on exit. In either case *Response must be caller freed if Response was not NULL;

Prototype

```
EFI_STATUS
EFI API
ShellPromptForResponse (
    IN SHELL_PROMPT_REQUEST_TYPE Type,
    IN CHAR16                  * Prompt OPTIONAL,
    IN OUT VOID                 * * Response OPTIONAL
);
```

Parameters

Type – Enumerated value that specifies what type of question is asked. This is used to filter the input to prevent invalid answers to question. Valid values are:

ShellPromptResponseTypeYesNo – Yes or No

ShellPromptResponseTypeYesNoCancel – Yes, No, or Cancel

ShellPromptResponseTypeFreeform – Arbitrary user-typed string.

ShellPromptResponseTypeQuitContinue – Quit or Continue

ShellPromptResponseTypeYesNoAllCancel – Yes, No, All, or Cancel

ShellPromptResponseTypeEnterContinue – Enter

ShellPromptResponseTypeAnyKeyContinue – Any Key

Prompt – Pointer to a null-terminated string that specifies the prompt used to request input.

Response – On exit, the pointer to the response, which will be populated upon return. For FreeForm type response this is actually a pointer to a CHAR16 pointer. For all others, it is one of the following enumerated values:

ShellPromptResponseYes – Yes

ShellPromptResponseNo – No

ShellPromptResponseCancel – Cancel

ShellPromptResponseQuit – Quit

ShellPromptResponseContinue – Continue

ShellPromptResponseAll – All

Status Codes Returned

EFI_SUCCESS – The operation was successful.

EFI_UNSUPPORTED – The operation is not supported as requested.

EFI_INVALID_PARAMETER – A parameter was invalid.

ShellPromptForResponseHii0

This function prompts the user using a string from the HII database and return the resulting answer to the caller. This function will display the requested question on the shell prompt and then wait for an appropriate answer to be input from the console.

If the Type parameter has a value of ShellPromptResponseTypeYesNo, ShellPromptResponseTypeQuitContinue or ShellPromptResponseTypeYesNoAllCancel, then Response points to the response enumerated value on exit. If the Type is ShellPromptResponseTypeFreeform then Response points to a CHAR16 * value on exit. In either case *Response must be caller freed if Response was not NULL;

Prototype

```
EFI_STATUS
EFIAPI
ShellPromptForResponseHii (
    IN SHELL_PROMPT_REQUEST_TYPE Type,
    IN CONST EFI_STRING_ID           HiiPromptStringId,
    IN CONST EFI_HANDLE              HiiPromptHandle,
    IN OUT VOID                     **Response
);
```

Parameters

Type – Enumerated value that specifies what type of question is asked. This is used to filter the input to prevent invalid answers to the question. Valid values are:

`ShellPromptResponseTypeYesNo` – Yes or No

`ShellPromptResponseTypeYesNoCancel` – Yes, No, or Cancel

`ShellPromptResponseTypeFreeform` – Arbitrary user-typed string.

`ShellPromptResponseTypeQuitContinue` – Quit or Continue

`ShellPromptResponseTypeYesNoAllCancel` – Yes, No, All, or Cancel

`ShellPromptResponseTypeEnterContinue` – Enter

`ShellPromptResponseTypeAnyKeyContinue` – Any Key

`HiiPromptStringId` – The string identifier associated with the string to use to format the output.

`HiiPromptHandle` – The HII handle of the package list that contains the string specified by `HiiPromptStringId`.

`Response` – On exit, the pointer to the response, which will be populated upon return. For `FreeForm` type response this is actually a pointer to a CHAR16 pointer. For all others, it is one of the following enumerated values:

`ShellPromptResponseYes` – Yes

`ShellPromptResponseNo` – No

`ShellPromptResponseCancel` – Cancel

`ShellPromptResponseQuit` – Quit

`ShellPromptResponseContinue` – Continue

`ShellPromptResponseAll` – All

Status Codes Returned

`EFI_SUCCESS` – The operation was successful.

`EFI_UNSUPPORTED` – The operation is not supported as requested.

`EFI_INVALID_PARAMETER` – A parameter was invalid.

ShellReadFile()

This function reads data from the file.

If `FileHandle` is not a directory, the function reads the requested number of bytes from the file at the file's current position and returns them in `Buffer`. If the read goes beyond the end of the file, the read length is truncated to the end of the file. The file's current position is increased by the number of bytes returned.

If `FileHandle` is a directory, the function reads the directory entry at the file's current position and returns the entry in `Buffer`. If the `Buffer` is not large enough to hold the current directory entry, then `EFI_BUFFER_TOO_SMALL` is returned and the current file position is not updated. `BufferSize` is set to be the size of the buffer needed to read the entry. On success, the current position is updated to the next directory entry. If there are no more directory entries, the read returns a zero-length buffer. `EFI_FILE_INFO` is the structure returned as the directory entry.

Prototype

```
EFI_STATUS
EFIAPI
ShellReadFile (
    IN      SHELL_FILE_HANDLE FileHandle,
    IN OUT  UINTN           *ReadSize,
    OUT     VOID             *Buffer
) ;
```

Parameters

`FileHandle` – The opened file handle for reading.

`ReadSize` – On input, the size of `Buffer`. On output, the amount of data in `Buffer`. In both cases, the size is measured in bytes.

`Buffer` – The buffer in which data is read.

Status Codes Returned

`EFI_SUCCESS` – Data was read.

`EFI_NO_MEDIA` – The device has no media.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_BUFFER_TOO_SMALL` – Buffer is too small. `ReadSize` contains required size.

ShellSetFileInfo()

This function sets the file information to an opened file handle.

Prototype

```
EFI_STATUS
EFIAPI
ShellSetFileInfo (
    IN SHELL_FILE_HANDLE  FileHandle,
    IN EFI_FILE_INFO      *FileInfo
);
```

Parameters

`FileHandle` – A file handle.

`FileInfo` – Pointer to the updated file information. `EFI_FILE_INFO` is defined in the UEFI specification.

Status Codes Returned

`EFI_SUCCESS` – The information was set.

`EFI_INVALID_PARAMETER` – A parameter was out of range or invalid.

`EFI_UNSUPPORTED` – The file handle does not support file information.

`EFI_NO_MEDIA` – The device has no medium.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – The file or medium is write-protected.

`EFI_ACCESS_DENIED` – The file was opened with read-only access.

`EFI_VOLUME_FULL` – The volume is full.

ShellSetFilePosition()

This function sets the current file position for the handle to the position supplied. With the exception of seeking to position `0xFFFFFFFFFFFFFFF`, only absolute positioning is supported, and seeking past the end of the file is allowed (a subsequent write would grow the file). Seeking to position `0xFFFFFFFFFFFFFFF` causes the current position to be set to the end of the file.

If `FileHandle` is a directory, the only position that may be set is zero.

This has the effect of starting the read process of the directory entries over.

Prototype

```
EFI_STATUS
EFIAPI
ShellSetFilePosition (
    IN SHELL_FILE_HANDLE FileHandle,
    IN UINT64             Position
);
```

Parameters

`FileHandle` – The file handle on which the requested position will be set.

`Position` – Byte position from the start of the file.

Status Codes Returned

`EFI_SUCCESS` – Data was written.

`EFI_INVALID_PARAMETER` – One of the parameters had an invalid value.

`EFI_UNSUPPORTED` – The seek request for nonzero is not valid on open directories.

ShellSetEnvironmentVariable()

This function changes the current value of the specified environment variable. If the environment variable exists and the Value is an empty string, then the environment variable is deleted. If the environment variable exists and the Value is not an empty string, then the value of the environment variable is changed. If the environment variable does not exist and the Value is an empty string, there is no action. If the environment variable does not exist and the Value is a non-empty string, then the environment variable is created and assigned the specified value.

Prototype

```
EFI_STATUS
EFI API
ShellSetEnvironmentVariable (
    IN CONST CHAR16                  *EnvKey,
    IN CONST CHAR16                  *EnvVal,
    IN BOOLEAN                         Volatile
);
```

Parameters

`EnvKey` – Pointer to the null-terminated string that specifies the name of the environment variable.

`EnvVal` – Pointer to the null-terminated string that specifies the value of the environment variable

`Volatile` – Boolean that specifies whether the variable is non-volatile (FALSE) or volatile (TRUE).

Status Codes Returned

`EFI_SUCCESS` – The operation completed successfully.

ShellSetPageBreakMode()

This function sets (enabled or disabled) the page break mode. When page break mode is enabled the screen will stop scrolling and wait for operator input before scrolling a subsequent screen.

Prototype

```
VOID
EFIAPI
ShellSetPageBreakMode (
    IN BOOLEAN CurrentState
);
```

Parameters

CurrentState – Boolean that specifies page break is enabled (TRUE) or disabled (FALSE).

Return Values

None

ShellStrToIntn()

This function returns the number converted from the string.

Note: This function cannot be used when `(UINTN)(-1)`, `(0xFFFFFFFF)` may be a valid result. Use `ShellConvertStringToInt64` instead.

Prototype

```
UINTN
EFIAPI
ShellStrToIntn (
    IN CONST CHAR16 *String
);
```

Parameters

`String` – Pointer to a null-terminated string that represents a number.

Return Values

Unsigned integer that indicates the result of the conversion or `(UINTN)(-1)` if the conversion failed.

ShellWriteFile()

This function writes the specified number of bytes to the file at the current file position. The current file position is advanced the actual number of bytes written, which is returned in `BufferSize`.

Partial writes only occur when a data error has occurred during the write attempt (such as “volume space full”). The file is automatically grown to hold the data if required. Direct writes to opened directories are not supported.

Prototype

```
EFI_STATUS
ShellWriteFile(
    IN      SHELL_FILE_HANDLE  FileHandle,
    IN OUT  UINTN             *BufferSize,
    OUT     VOID              *Buffer
);
```

Parameters

`FileHandle` – The opened file handle for writing.

`BufferSize` – On input, pointer to the number of bytes in `Buffer`. On output, the number of bytes written. In both cases, the size is measured in bytes.

`Buffer` – Pointer to the buffer in which data is written.

Status Codes Returned

`EFI_SUCCESS` – Data was written.

`EFI_UNSUPPORTED` – Writes to an open directory are not supported.

`EFI_NO_MEDIA` – The device has no media.

`EFI_DEVICE_ERROR` – The device reported an error.

`EFI_VOLUME_CORRUPTED` – The file system structures are corrupted.

`EFI_WRITE_PROTECTED` – The device is write-protected.

`EFI_ACCESS_DENIED` – The file was open for read-only access.

`EFI_VOLUME_FULL` – The volume is full.

StrnCatGrow()

This function safely appends with automatic string resizing given length of Destination and desired length of copy from Source.

Append the first D characters of Source to the end of Destination, where D is the lesser of Count and the `StrLen()` of Source. If appending those D characters will fit within Destination (whose Size is given as `CurrentSize`) and still leave room for a NULL terminator, then those characters are appended, starting at the original terminating NULL of Destination, and a new terminating NULL is appended.

If appending D characters onto Destination will result in an overflow of the size given in `CurrentSize` the string will be grown such that the copy can be performed and `CurrentSize` will be updated to the new size.

If Source is NULL, there is nothing to append, so return the current buffer in Destination.

Prototype

```
CHAR16*
EFIAPI
StrnCatGrow (
    IN OUT CHAR16      **Destination,
    IN OUT UINTN       *CurrentSize,
    IN CONST CHAR16   *Source,
    IN UINTN           Count
);
```

Parameters

`Destination` – On entry, the string on which to append. On exit, the updated string.

`CurrentSize` – On entry, the number of bytes in Destination. On exit, possibly the new size (still in bytes). If NULL, then allocate whatever is needed.

Source – The string to append.

Count – The maximum number of characters to append. If 0, then all are appended.

Return Values

The Destination, after appending the Source.

Data Structures

The following are data structures used in the UEFI Shell Library.

Format Strings

Format strings are normal strings with placeholders for runtime values. The placeholders are prefixed with the % (percent) character. The characters that follow the % character determine the type and format of the data to be displayed and have the following format:

%*w.l*F

Where *w* is the width, *l* is the field width and *F* is the format. The width *w* can be any one of the following:

0 – Pad with zeroes

- (Hyphen) – Left justify (default is right justify)

, – Add commas to the field

* – Width provided on the stack

The field width *l*, if present, indicates that the value is 64-bits. The format *F* can be one of the following:

a – ASCII string

s – Unicode string

X – fixed 8-byte value in hexadecimal

x – hexadecimal value

d – value as decimal

c – Unicode character

t – EFI time structure

g – Pointer to GUID

r – EFI status code (result code)

% – Print a %

Shell Parameters

The command-line parsing functions in the Shell library return the parameters in a linked list that contains `SHELL_PARAM_ITEM` structures.

Prototype

```
typedef struct {
    CHAR16             *Name;
    SHELL_PARAM_TYPE   Type;
} SHELL_PARAM_ITEM;
```

Members

Name – Pointer to null-terminated string that specifies the parameter.

Type – Enumerated value that specifies the type of the parameter. Valid values are:

- TypeFlag (0) – A flag that is present or not present.
- TypeValue (1) – A flag that has some data following it with a space (i.e. “-a 1”).
- TypePosition (2) – Some data that did not follow a parameter, such as a file name.
- TypeStart (3) – A flag that has additional flags appended to the end. For example, -ad, -afd, or -adf.
- TypeDoubleValue (4) – A flag that has two values following it.
- TypeMaxValue (5) – A flag followed by all command-line parameters up to the next flag.
- TypeTimeValue – A flag that has a time value following it.

Index

16-bit 48, 53, 85

A

Ability 13, 43, 45, 59, 67, 76, 234
– endfor command's 110
ABORTED status code 133
Absolute directory path 197
Abstractions 5, 6, 29, 30, 71, 73, 86
– block I/O 51
– file system level 51
– right level of 52
– underlying program-matic 4
Access networking devices 5
Accessor functions 10
Addh 194, 195
Addh options 194, 195
Addp 194, 195
Address 79, 81, 82, 176, 178, 203, 219
– starting 88, 203, 219
Administrator remote 70, 71, 76
Agents 58, 73, 76, 191
– list of 173
Alias 17, 117, 118, 152, 190, 193, 239
– internal 200, 202, 218
Alt 134, 135
AlwaysAllowNumbers 250
AlwaysAllowNumbers parameter set 250
Answers appropriate 278, 279
APIs 5, 6, 11, 12, 74, 75, 85
Appending 287, 288
AppendLogFile 63, 67
Application 25, 26, 124, 125, 155, 234, 237
– command-line 186
– shell-aware 26
– single 123, 163
Application library 105
Ar 216
Arg 144, 146, 274
ARG structures 274
Argc 27, 66, 122, 142, 144, 240
Argument count 25, 27, 142, 145
Argument index current 144, 145
Arguments 122, 144, 208, 209, 213, 233,
 249
– current 143, 144
– invalid command line 251

– shell application's 240
Argv 27, 28, 66, 131, 142, 144
ASCII 159, 161, 163, 241, 257, 258, 259
ASCII characters 97, 161, 163
– single 159, 160
ASCII file 156, 259
ASCII file types 207
ASCII string 227, 288
– null-terminated 277
ASCII value 135
Ascii-string 227
ASCII-to-Unicode 158, 159
Asterisk 233
Attrib 17, 117, 190, 194
Attributes 113, 114, 123, 124, 194, 272, 273
Attributes of files 190, 194
Automatic policy 215
Automatic string resizing 245, 287
AutoPageBreak 250
Av 234, 235

B

B6, 80
Background color 190, 197, 275, 277
BackupImageHandle 63, 64
BAR 178, 179
Bare Metal Provisioning 55, 56, 58, 60, 62,
 64, 66
Basic file system commands 189
Basic Input Output System 1
Basic Overview of Commands 22
Bcfg 190, 194, 195
BDS (Boot Device Selection), 37
Beginning 33, 55, 80, 210, 211, 258, 259
Binary description 73
Binary executable content 181
Binary programs 19, 20, 24, 25, 76
BIOS 1, 15, 67, 69, 70, 71, 73
– underlying UEFI-compatible 73
BIOS and OS-present components 73
BIOS compliant 87
BIOS Information 91, 227
BIOS initialization 70
BIOS Setting Changes 70, 71
BIOS-based configuration settings 70
Bitwise 140, 146, 151, 152

Blade server 40, 41, 44
 Blades 44, 45
 Blk0, 30, 80
 BLK1, 30
 Block count 200
 Block devices 30, 38, 51, 190, 191, 200, 210
 Block I/O 50, 52
 Block number starting 210
 Blocks 29, 52, 80, 190, 191, 200, 210
 Board Devices Information 227
 Boolean 63, 250, 256, 258, 270, 271, 284
 Boolean functions 212
 Boolean operators 119, 139
 Boolfunc 211
 Boot 10, 35, 37, 41, 194, 195, 196
 – standard UEFI 122
 Boot Device Selection (BDS), 37
 Boot image 60
 Boot Mode Functions 35
 Boot option 119, 196
 Boot options new 119, 120
 Boot server 58, 60, 61, 62
 Boot services 10, 11, 26, 81, 173
 – querying UEFI 9
 Boot services application 10
 Boot target 1, 5, 44, 70
 Bootable device/media 18
 BootOrder/DriverOrder 196
 BootServices 11, 27, 28
 BOOTSHELL 64
 Boxes 39, 47, 48, 49, 50, 51, 52
 Break 48, 66, 67, 108, 265
 BREAKPOINT 176
 BRIDGE 178, 179
 Buffer 103, 258, 259, 261, 281, 282, 286
 – existing 241
 – returned 257
 BufferSize 281, 286
 Build 53, 121, 123, 124, 163, 175, 176
 Build description 123, 138, 139
 Build environment 53, 121, 186
 Bus 52, 204, 219, 221, 222
 Bus I/O 52
 Bus number 203, 222
 Buses 35, 36, 51
 Bytes 82, 88, 97, 159, 219, 286, 287
 – requested number of 239, 281

C
 Cadab 108, 109
 Caller 208, 257, 260, 261, 265, 278, 279
 Capabilities 10, 13, 35, 72, 76, 86, 87
 – basic command 193
 – functional 4
 – key shell 155
 Cases 15, 16, 66, 67, 162, 181, 238
 Cd 17, 80, 117, 189, 190, 196, 197
 CD-ROM drive 41
 Certificate 60, 182
 Chaining of Script Commands 24
 Chaining of UEFI Shell scripts 22
 Chaired 3
 Char 66, 122, 268, 270
 CHAR16, 27, 28, 63, 256, 264, 278, 287
 CHAR16 pointer 279, 280
 Characters 136, 157, 159, 233, 245, 287,
 288
 – non-printable 136
 – single 161, 162
 Chassis Management Module (CMM), 44
 CheckList 250
 CheckPackage 246, 247, 248, 249, 250
 Child 170, 203, 205, 206, 223
 Child devices 77, 78, 79, 173, 204, 205, 206
 Child devices of device 77, 78
 Class 20, 45, 52, 86
 Cli 4, 63, 64
 Client 15, 16, 55, 57, 58, 60, 62
 Client machine 58, 59, 60, 61
 CloseFile 238
 Cls 17, 117, 190, 197
 Cmd 210
 CmdLine 256
 CMM (Chassis Management Module), 44
 Code 33, 34, 37, 53, 79, 102, 176
 – specified language 200, 201, 202, 204,
 205, 206
 COFF (Common Object File Format), 182
 Col 220, 275, 277
 Color 197
 – blue 275, 277
 – green 275, 277
 Columns 113, 114, 115, 192, 193, 220, 244
 – parse filename tablename 221
 Comma 113, 193, 217, 288
 Command 173, 208, 209, 210, 213, 234, 235
 – additional 107

- appropriate shell interface 31
- associated 233
- basic 189
- batch 7
- bcfg 119
- best 170
- common 76, 79
- cp 118
- echo 108
- endfor 113
- endfor script 111
- endif 210
- environment variable 117
- err 177
- executed 235, 256
- executes 191
- exit 111
- file manipulation 217
- following 92
- goto 109, 110, 209
- help 209
- internal 64, 92, 94
- list of 22, 189, 191
- matching endfor 191
- parse 113, 114
- pause 221
- reading 207
- script-based 22
- scripting 6
- shell-enabled 31
- shift 110, 227
- specified 209, 238, 239, 255, 256
- standard 7
- unload 169
- Command changes 196
- Command clears 197
- Command condition 191
- Command Description 190, 191, 192
- Command Details 193, 195, 197, 199, 201, 203, 205
- Command executes 208, 210
- Command execution 240
- Command exercises 169
- Command exits 207
- Command line 6, 143, 144, 233, 236, 248, 255
- Command line arguments 251
- Command Line Interface Features 6
- Command Line Parsing 243
- Command loads 215
- Command manages 194
- Command name 276
- Command parses 168
- Command Piping 234, 235
- Command processor 4, 6
- Command profiles 9, 189
- Command prompt 22
- Command Reference 189, 190, 192, 194, 196, 198, 200
- Command resets 223
- Command set 236
- Command support 7
- Command syntax 234, 235
- Command tests 170
- Command unloads 230
- Command updates 229
- Command-line 128, 131, 143, 144, 146, 155, 247
- CommandLine 255, 256
- Command-line APIs 6
- Command-line argument count 143
- Command-line arguments 139, 143, 144, 243, 244, 249, 250
- first 143
- last 143
- list of 247
- list of parsed 248, 249
- Command-line environment 13
- interactive 9
- Command-Line Handling 155
- Command-line handling advanced 155
- Command-line monitor 13
- Command-line options 115, 130, 131, 132, 155, 156, 157
- parsed 243
- Command-line parameters 28, 31, 108, 109, 110, 111, 113
- first 108, 130
- next 111
- Command-line parsing 243
- Command-line parsing functions 243, 289
- Command-line parsing in UnicodeDecode 157
- Command-line processing 151
- simple 130
- Command-line processor 6
- Command-line syntax 76, 78, 79, 80, 81, 82
- appropriate 76, 78, 80

- Command-line Usage for dblk 80
- Command-line Usage for dmem 81
- Command-line Usage for drvcfg 76
- Command-line Usage for drvdiag 78, 82
- Command-line Usage for memmap 79
- Command-name 115, 193
- CommandToGetHelpOn 276
- Common Object File Format (COFF), 182
- Comp 190, 197
- Companies 3, 48, 50, 53, 54
- Comparison function case-sensitive 144
- Comparisons 182, 211
- Complexity 5, 6
- Compliance driver model 73
- Component information 123, 125, 137, 138, 154, 163
- Component information file 139
 - simple 154, 164
- Component Name 166, 171, 172
- Component Name Protocol 172
- Component Name protocol to retrieve 171, 172
- Components 6, 37, 53, 72, 123, 124, 139
- Compressed input file 207
- Compressed output file 207
- Computer 1, 5, 85
- Concat1 output-file 112
- Concatenate Text Files 112
- Concepts 15, 16, 22, 24, 58, 75, 189
- Condition 53, 177, 191, 211
- Conditional expressions 211, 213
- Configuration 38, 44, 45, 69, 77, 214, 215
 - current TCP/IP 191
 - device's 76
 - golden 44, 47
- Configuration Access 73, 74, 75
- Configuration data 70, 72, 73, 74, 75
- Configuration event 73, 74, 75
- Configuration infrastructure 69, 72, 73, 76, 79
 - modern UEFI 71
 - platform's 76, 205
 - underlying 76
 - underlying UEFI 70
- Configuration Infrastructure Overview 71
- Configuration of Provisioned Material 69, 70, 72, 74, 76, 78, 80
- Configuration routing services 72
- Configuration services 69, 74, 75
- Configuration space device's PCI 79
- Configure 33, 70, 76, 190, 205
- Configuring 47, 50, 76, 77, 79, 81, 83
- Connect 168, 169, 170, 173, 198, 215, 216
- Connect commands 169, 170, 198
- Connect console devices 198
- Connect DeviceHandle 167
- Connect DeviceHandle DriverHandle 167
- Connect Driver Binding 166
- Connecting UEFI Drivers 169
- Connection-oriented protocol 62
- Connections 47, 198
- Console 38, 86, 190, 192, 197, 278, 279
- Console devices 49, 170, 198, 202, 276
 - standard 122
- Console output 189, 197
- CONST BOOLEAN ParameterReplacing 251
- CONST CHAR16, 249, 251, 260, 264, 269, 276, 284
- CONST DeviceName 264
- CONST KeyString 36, 248, 277, 280
- CONST LIST 246, 248, 249
- CONST SHELL 250
- &Context 65
- Control 33, 36, 37, 43, 50, 51, 107
- Control characters 135, 136
- Controller 74, 75, 205
 - device's host 52
- Conversion 54, 252, 267, 286
- Copy 5, 7, 18, 20, 31, 116, 287
- COPY command 31
- CORRUPTED 253, 255, 261, 262, 263, 282, 283
- Count 107, 197, 198, 215, 222, 287, 288
- Cp 17, 116, 117, 190, 193, 199
- CPU 36, 38, 47, 70
- CR 176
- CreateFile 238
- Creation 3
- Current directory string 263
- Current file position 209, 239, 240, 266, 281, 283, 286
- Current time zone 229
- Current working directory 196, 197, 199, 212, 220, 242, 243
- CurrentSize 287
- Cursor 275, 277
- Custom profile names 118
- Customer-visible value 85, 86

- Cyan 197
- Cycle 178

- D**
- Data
 - command-line parameter 26
 - dirty cached file 238, 245
 - optional 195, 196
 - packed 94
 - passed-in entry point 29
 - passing variable 90
 - standard entry point 28
 - string-based 72
- Data bits 224, 225
- Data files 34
- Data structures 86, 95, 96, 238, 241, 288, 289
 - variable length 6
- Date 17, 54, 117, 190, 192, 199, 229
- David 182
- Daylight saving time 229
- Days 5, 48, 50, 187
- Dblk 79, 80, 190, 200
- Dblk device 80, 200
- Dblk fs0, 80
- Dd 81, 83, 199, 203, 219
- Debug 175, 176, 177, 178, 190, 191, 192
- Debug build 175
- Debug code 175
- Debug information 180
- Debug Macro 175, 176
- Debug UEFI drivers 166
- Debugging Code Statements 175
- Declarations 96, 124, 138
 - global 127
- Decode 134, 201, 202
- Decompressed output file 207
- Default file name 207
- Defaults force 77, 174, 205
- Defects 49
- DeleteFile 238
- Deletes 190, 192, 218, 225
- Description 96, 97, 98, 99, 100, 195, 209
 - detailed 165, 173
 - user-readable 195, 196
- Destination 47, 190, 191, 199, 220, 287, 288
- Destination file/directory name 220
- Dev 221, 222

- Developer's UEFI Emulation (DUET), 18
- Device diagnostics 190
- Device drivers 72, 73
- Device Information 171, 173
- Device mappings 121, 217
- Device memory 79, 190, 191, 203
- Device names 172, 201
- Device number 203, 222
- Device path 51, 73, 195, 196, 201, 239, 272
 - associated 239
- Device path text format 227
- Device registers 52
- Device tree 172, 201
- Device type 217
- Device-handle 198, 201, 202, 203, 205, 206, 223
- DeviceHandle 77, 78, 272
- Device-handle
 - disconnect 202
 - reconnect 223
- DeviceName 263, 264
- Device-path 227
- Devices 52, 77, 78, 172, 173, 174, 198
 - add-in 73
 - bootable 18
 - byte-stream-based 179
 - current 196, 217, 256, 263
 - device's 74, 75
 - installed 50, 217
 - list of 77, 78, 190, 200
 - managed 73, 74, 75
 - single 169, 173
 - specified 190, 198, 201, 202, 223, 263, 264
 - standard error 176, 177
 - standard output 107, 220, 230
 - uninstalled 217, 218
- Devices and Drivers commands 174
- Devices command 172
- Devices command lists 172
- Devices Component Name 166
- Devtree 172, 190, 201
- DevTree commands 172, 173
- Dh 166, 167, 169, 173, 198, 201, 202
- Dh command 195, 196, 201, 202, 203, 205, 206
- DHCP (Dynamic Host Configuration Protocol), 62
- Diagnose 206

- Diagnostics 12, 13, 43, 78, 85, 86, 206
- Diagram 36, 57, 86
- Digital signature 60, 181, 182, 183, 184
- Digits 199, 208, 217
- Dir 7, 20, 190, 193, 219
- Directory 118, 194, 199, 219, 220, 224, 261
 - current 118, 197, 236, 238, 239, 263, 264
 - existing 220, 253
 - root 239
 - specified 117, 238
- Directory contents command lists 216
- Directory entries 261, 281, 283
- Directory names 194, 197, 198, 199, 216, 219, 253
 - current 263
- DirHandle 261, 262
- DirName 253, 269
- Disconnect 169, 170, 173, 190, 202, 203
- Disconnect command 169, 170, 223
- Disconnect command stops UEFI drivers 170
- Disconnect DeviceHandle DriverHandle 167
- Discovery 29, 30
- Disk 30, 42, 43, 52, 62, 85, 181
 - hard 39, 41, 217
- Disk I/O 52
- Disk Operating System 5
- Display 77, 80, 114, 198, 216, 220, 221
- Display device tree information for devices 201
- Display header and element detail information 91
- Display information 201, 216
- Display memory 82
- Display option 94
- Display output in standard format output 219
- Display statistics table 228
- Display structures 91
- Dmem 79, 81, 82, 190, 193, 203, 218
- Dmem command 81
- DMI 97, 103, 104
- DmiBIOSRevision 102, 103
- DmiStorageBase 102, 103
- DmiStorageSize 102, 103
- Dmpstore 204
- DMTF 90
- DOI 8, 9, 14, 15, 46, 47, 84
- Download 18, 58, 60, 62
- Drive
 - bootable 7
 - hard 47, 48, 50, 181
- Driver and Device Information 171, 173
- Driver Binding 166, 173, 198
- Driver Binding Protocol 168, 170
- Driver Binding Protocol of UEFI drivers 170
- Driver Configuration and Driver Diagnostics Protocols 174
- Driver Configuration ForceDefaults 166
- Driver Configuration OptionsValid 166
- Driver Configuration Protocol 174
- Driver Configuration Protocol implementation 174
- Driver Configuration SetOptions 166
- Driver Diagnostics Protocol 76, 174, 175, 206
- Driver Diagnostics Protocol in manufacturing mode 175
- Driver Diagnostics Protocol in standard mode 175
- Driver image 192, 230
- Driver information 205
- Driver initialization 73, 74, 75
- Driver list 171, 196
- Driver Model Interactions 73
- Driver type 204
- DriverBinding 167
- DriverEntryPoint 166, 167
- DriverHandle 77, 78, 167
- Drivers 72, 73, 77, 173, 190, 198, 204
 - bcfg 195
 - command reconnects 223
 - file system 266
 - loaded 168, 215
 - single 74, 121, 173
 - specified 205, 223
- Drivers command lists 171
- Drivers commands 169, 171, 174
- Drivers Component Name GetDriverName 166
- Drivers interact 72
- Drvcfg 76, 77, 78, 166, 174, 190, 205
- DrvCfg command 76, 174
- Drvdiag 76, 78, 79, 82, 174, 175, 206
- DrvDiag command 78, 82, 86, 174, 175
- DUET (Developer's UEFI Emulation), 18
- Dump 91, 195, 202

- Dumps information 171, 200, 201, 202, 204, 205, 206
 Dumps UEFI Driver 202
 DXE 34, 35, 36, 37, 42
 DXE core file 34
 DXE phase 36
 Dynamic Host Configuration Protocol (DHCP), 62
- E**
- EAP (Extensible Authentication Protocol), 57, 58, 59, 60
 EB 80, 81
 Echo 108, 109, 110, 114, 116, 117, 206
 Echo request datagram 222
 Echo1, 107, 108, 109
 Echo2, 107, 109, 110
 Echo3, 107, 110, 111, 112
 Edit 190, 207, 210
 EDK2 build files 53
 EFI 27, 35, 36, 64, 177, 255, 272
 EFI device path 73, 172
 EFI Device Path Display Format Overview section 227
 EFI file system messages 177
 EFI status code 289
 EFI SYSTEM TABLE 79, 240
 EFI system table pointer entries 81
 EFI time structure 289
 EFI variables 71, 119
 EFI API 63, 64, 256, 257, 264, 265, 269
 EFI API ShellCommandLineCheckDuplicate 246
 EFI API ShellCommandLineFreeVarList 247
 EFI API ShellCommandLineGetFlag 248
 EFI API ShellCommandLineGetRawValue 249
 EFI API ShellCommandLineGetValue 249
 EFI API ShellCommandLineParseEx 250
 EFI API ShellConvertStringToInt 252
 EFI API ShellCopySearchAndReplace 251
 EFI API Shell.CreateDirectory 253
 EFI API ShellDeleteFileByName 255
 EFI API ShellFileHandleReturnLine 257
 EFI API ShellFindFilePath 259
 EFI API ShellFindNextFile 261
 EFI API ShellGetEnvironmentVariable 264
 EFI API ShellGetExecutionBreakFlag 265
 EFI API ShellIsDecimalDigitCharacter 268
 EFI API ShellIsFileInPath 270
- EFI API ShellIsHexaDecimalDigitCharacter 270
 EFI API ShellOpenFileByDevicePath 271
 EFI API ShellOpenFileByName 273
 EFI API ShellPromptForResponse 278
 EFI API ShellPromptForResponseHii 280
 EFI API ShellSetEnvironmentVariable 284
 EfiDriver.efi 168
 EfiPciIoWidthUint 178, 179
 ISBN 8, 9, 14, 15, 46, 47, 84
 Electron 44
 Endfor 17, 107, 110, 112, 117, 208, 209
 Endif 109, 110, 112, 114, 115, 116, 213
 Engineers 48, 49, 53, 54
 Entry 246, 247, 248, 249, 259, 281, 287
 Entry point 29, 64, 124, 141, 169
 - driver’s 168
 - standard 25, 26
 Entry point function 125
 Entry Point Structure. *See* EPS
 EntryPointStructureChecksum Checksum 97
 Enumerated values following 279, 280
 En-US 200, 201, 202, 204, 205, 206
 Environment 15, 16, 18, 48, 74, 234, 256
 Environment variable lasterror 208, 225
 Environment variable name 130, 132, 151, 225, 264
 Environment variable value 225, 264
 Environment variables 107, 127, 225, 234, 236, 256, 284
 - change UEFI Shell 225
 - shellsupport 22, 189
 - specified 239, 243, 264, 284
 EnvironmentVariables 256
 EnvKey 264, 284
 Env-var-name 127, 128
 EPS (Entry Point Structure), 88, 96, 97, 102
 Eq 140, 148, 211, 212
 Error 64, 118, 130, 143, 176, 224, 276
 Error code 117, 118, 142
 Error levels 176, 177
 Error mapping functions 212
 Error message 112, 114, 117, 118, 132, 160, 161
 ErrorLevel 176, 177
 ErrorLevel parameter 176
 Event 10, 73, 75, 133, 177, 240
 Evolution 13, 33, 90

- Example 168, 170, 171, 172, 173, 174, 175
 Example exercises 168
 Example Function Declaration 36
 Example script file 107
 Example usage 187
 Executable file 185
 – signed 185
 Executable file contents 185
 Executable structure description detailed 182
 Executables 17, 35, 42, 119, 181
 Execute UEFI Shell 22, 24
 Executes 17, 107, 110, 111, 237, 238, 255
 Execution 26, 107, 109, 110, 216, 221, 235
 – command directs script file 209
 – suspends script file 221
 Execution Environment 33, 34, 36, 38, 40, 42, 44
 ExecutionBreak 133, 240
 Existing devices 217, 218
 Exit 116, 191, 258, 259, 272, 278, 279
 Exit-code 208
 Expressions 139, 146, 176, 211, 213
 Extensible Authentication Protocol (EAP), 57, 58, 59, 60
 Extract 62, 72, 113, 114, 115
- F**
- Factory 7, 47, 181
 Failure 43, 50, 85, 86, 250, 254, 255
 FALSE 63, 66, 252, 258, 259, 270, 271
 FAT parameters 200
 FAT16, 80, 81
 FE 80, 81, 83
 FFS 35, 36
 Field width 288
 File 238, 242, 254, 257, 258, 259, 281
 – build description 138
 – empty 207
 – first 242, 261
 – help 115, 116, 118, 276
 – hidden 216
 – inf 139
 – list of 238, 246, 275
 – multiple 215, 229
 – next 242, 262
 – open 242, 265
 – specified 117, 118, 204, 205, 238, 239, 272
 File Access Date 113, 114
 File Creation Date 113, 114
 File extensions 242, 260
 File formats 123, 154
 – fully-qualified Portable Executable Common Object 60
 File information 190, 191, 240, 242, 262, 265, 274
 – returned 265
 – updated 282
 File I/O 6, 241
 File I/O Functions 241
 File I/O support functions 241
 File Modification Date 113, 114
 File name 113, 194, 196, 208, 242, 260, 273
 – given 241
 File parameter 229
 File path 171, 240, 253, 272, 273, 274
 – driver's 204
 – executable 28
 – specified 242
 File position 159, 257, 258, 259
 File size 160, 161
 File System Abstractions 30
 File system intermediary 41
 File System I/O 52
 File system mappings 52, 196
 File system path 195, 239
 File system protocol 217
 File system structures 253, 255, 261, 262, 263, 282, 283
 File system style name 239
 File systems 50, 51, 52, 57, 217, 230, 231
 – current 196, 197, 230
 – current working 236
 – multiple 49
 – selected 243
 – specified 243, 263
 File Transfer Protocol. *See* FTP
 File truncates 226
 File type 156, 158, 230
 FILE2, 35
 File/directory 224
 FileExtension 260
 FileHandle 253, 254, 266, 271, 272, 273, 281
 FileInfo 114, 115, 193, 282
 FileInfo row type 114
 Filename 77, 210, 233, 255, 259, 260, 273

- File-name 128
- Filename argument 211
- Filename expansion 233
- Filename for compressed input file 207
- Filename for compressed output file 207
- Filename for decompressed output file 207
- Filename for uncompressed input file 207
- FilePath 271, 272
- FilePathName 28
- File's File attributes 272, 273
- File's information 239
 - first 260
 - next 262
- File's size 266, 267
- Files to display 113, 230
- Files/directories 224
- File-system 29
- File-System Abstractions 29
- File-system protocol simple 32
- File-systems recognized 30
- FindTarget 251, 252
- Firmware 11, 47, 50, 59, 60
- Firmware interfaces underlying UEFI 23
- First file name 197
- Flag 131, 132, 198, 199, 243, 248, 289
- Flag parameters 248
- Flash 33, 34, 48
- FLASH device 5, 48
 - platform's 16
 - unprotected 181
- Flash ROM 34, 35, 37, 42
- ForceHex 252, 271
- Format 197, 200, 203, 235, 275, 277, 288
- Format Strings 275, 288
- Formatted area 90, 97
- Formatted section 89, 91
- FreeForm type response 279, 280
- Freeing 260, 265
- Fs0, 168, 170, 171, 172, 173, 174, 175
 - blocks of 80
- FTP (File Transfer Protocol), 48, 62, 68
- FTP utility 63, 67, 68
- FTP+UEFI Shell integrated 68
- Func 167, 221, 222
- Function 133, 134, 238, 239, 240, 241, 261
 - underlying 29, 31
- Function changes 239, 284
- Function Description 241, 242, 243, 244, 245
- Function flushes 238, 262
- Function Name 241, 242, 243, 244, 245
- Function number 203, 222
- Function ParseCommandLine 130
- Function pointers 10
- Function prompts 278, 279
- Function sets 131, 240, 282, 283, 285
- Function ShellConvertStringToInt 153
- Functionality 3, 16
- G**
- GetControllerName 166, 172
- GetDeviceName 238
- GetDriverName 166, 171
- GetFileSize 159, 160, 161, 239
- GetKey 113, 126, 127, 129, 131, 134, 137
- GetKey sample application 115
- Globally Unique Identifier. *See* GUID
- Goto 64, 65, 66, 67, 107, 113, 114
- Gt 114, 149, 211
- GUID (Globally Unique Identifier), 35, 64, 86, 123, 124, 191, 226
- Guid var-guid 204
- H**
- HANDLE FileHandle 262, 265, 266, 281, 282, 283, 286
- Hard drive partition 195, 196
- Hardware 1, 49, 50, 71, 72, 85, 86
- Hardware Access 50, 51
- Header 27, 35, 88, 95, 97, 98, 185
- Header File Name 95
- Header file Uefi 122
- Header files 63, 124, 125, 140
- HelloWorld 119, 120, 121, 123, 124, 125, 126
- HelloWorld UEFI Shell application 154, 164
- Hexadecimal 195, 196, 203, 217, 224, 245, 252
- Hexadecimal bytes 210, 226, 227
- Hexadecimal characters 252, 270
 - valid 270
- Hexadecimal number 198, 200, 202, 203, 205, 206, 223
- HII (Human Interface Infrastructure), 3, 70, 71, 277, 280
- HII unique 74, 75
- HiiFormatStringId 277
- HiiHandle 101, 104
- HiiPromptStringId 280

- HOB 35
 Host-bus adapter (HBA), 42
Human Interface Infrastructure. See HII
 Hypervisor 11, 13
- I, J**
- IEPS (Intermediate Entry Point Structure), 97
 Ifconfig 191, 214, 215
 Image 9, 18, 41, 60, 62, 169, 230
 ImageHandle 27, 28, 63, 64, 256
 Imaging 41
 Implementations 9, 48, 124, 189
 Index 66, 100, 133, 196, 200, 249
 Index variable 111, 208
 Indexvar 208, 209
 Industry 1, 6, 18, 86, 90, 105
 Inf 121, 123, 125, 137, 138, 154, 163
 Inf Component Information File 126
 Infile 207
 Info 36, 241, 246, 261, 265, 281, 282
 Information 47, 86, 88, 91, 113, 114, 202
 - 32-bit Memory Error 227
 - asset 43, 87
 - decode 201
 - detailed 96, 173
 - display device 201
 - display header 91
 - display SMBIOS structure 91
 - display structure 91
 - help 209, 210, 239
 - image-related 72
 - returning string 90
 - system time zone 192
 - system's time zone 229
 - usage 209, 210
 - version 192, 230
 Infrastructure 11, 17, 18, 72, 86, 237
 - underlying 15, 20
 Init 64, 101, 177
 Init SMBIOS structure table address 101
 Initial FTP shell application 64
 Initialize log file 64
 InitializeApp 27, 28
 InitializeMiniFtp 63, 64
 Initializes 1, 3, 34, 37, 38, 64, 176
 Input 128, 155, 156, 158, 278, 279, 280
 Input command line parameters 92
 Input file 112, 156, 158, 159, 160
 Input file name 157
 – next 113
 Input parameters 9, 65
 Input redirection 235
 Install 31, 35, 36, 43, 74, 75, 115
 Install Script 115, 117
 Install services 74, 75
 InstallCmd command-name target-directory 115
 INT32, 275, 277
 Integrity 181
 Interact 20, 21, 22, 23, 25, 26, 73
 Interactions 19, 20, 24, 58, 71, 75, 76
 - basic 76, 79
 - remote 75
 Interactive Shell Environment 22, 23, 25, 27, 29, 31
 Interfaces 4, 36, 37, 72, 73, 214, 215
 - command-line 4, 5, 6
 - consistent shell/command-line 67
 - file-system 29
 - plug-and-play function 87, 88
 - programmatic 3, 4, 5, 233, 237
 - specified 214, 215
 Intermediate Entry Point Structure (IEPS), 97
 IntermediateChecksum Checksum of Intermediate Entry Point Structure 97
 Interpreter 6, 22
 Interprets 99
 INVALIDE 104
 I/O devices 33
 IP address 60, 62, 222
 Ip4, 214, 222
 IPMI Device Information 228
 IPv 191
 IPv4, 56, 59
 IPv6 network infrastructure 3
 iSCSI 57
 Isint 211, 212
 IsIPv4, 66
 IsRootShell 239
- K**
- KEK.crt 187
 Key 114, 130, 132, 133, 135, 248, 249
 - private 60, 181, 183, 187
 - public 60, 184
 Key data structures 95
 Key engineers 54
 Key text 128, 130, 134, 135, 136

Keyboard Input in UEFI Shell Scripts 126,
 127, 129, 131, 133, 135, 137
 KeyString 248, 249
 Keyword 111, 198

L

Label 109, 110, 113, 191, 209
 Lang 200, 201, 202, 204, 205, 206
 Language 6, 200, 201, 202, 204, 205, 277
 – current platform 201, 202, 204, 205, 206,
 277
 – default 171, 172
 Language codes 44, 200, 201, 202, 204,
 205, 206
 LANs (local area networks), 58
 Largest SMBIOS Structure 97, 102
 Last sector 30
 Lasterror 29, 116, 118, 212, 236
 GetLastError environment variable 28, 29
 Launch 1, 3, 4, 22, 24, 31, 44
 Launch commands 24
 Launch drivers 70
 Launch UEFI applications/drivers 22
 Launch UEFI Shell applications 22
 Launching 1, 15, 16, 22, 24, 25, 31
 Layer 30, 57, 59, 85
 Lba 80, 200
 Length 88, 90, 98, 103, 104, 208, 252
 Length field structure's 90
 Length of return buffer in bytes 103
 Levels 16, 49, 50, 117, 176, 177, 189
 Leverage 6, 10, 17, 18, 21, 73, 76
 LibGetSmbiosString 104
 Library 49, 50, 123, 124, 126, 139, 163
 Library class 124, 139
 Library function StrCmp 144
 Library header file 250
 LibSmbios 95, 100
 LibSmbiosView 100, 101, 104
 Lines 111, 113, 118, 123, 130, 146, 148
 – next 221, 259
 – single 257, 258
 Linux 41, 48, 54
 List 98, 111, 113, 244, 246, 249, 250
 – package 277, 280
 List of information 190, 204
 ListHead 246, 274, 275
 Load 166, 167, 168, 169, 191, 204, 215
 Load command 168, 171

Loaded image protocol 9, 169, 195
 LoaderCode 79
 LoaderData 79
 LOADFILE 177
 Loadpciom 167, 168, 216
 LoadPciRom command 168
 Location 17, 58, 86, 197, 275, 277
 Log 49, 63, 93
 Log file 63, 65, 67
 – name of 65, 67
 LogFilename 63, 64, 65, 66, 67
 Logical block address 41, 43, 200
 Logical size 113, 114, 115
 Loop 65, 111, 113, 131, 191, 208
 Ls 7, 113, 114, 117, 191, 193, 216
 LS command 23, 107, 113, 114, 115, 202
 Lsgrep.nsh 107, 114
 Lspa 171, 172
 Lt 114, 148, 211

M

Machine 33, 40, 41, 44, 45, 55, 62
 Macros 22, 26, 175, 176, 177, 179, 250
 Managed Network Protocol (MNP), 57
 Management Device 228
 Managing 75, 170, 173, 174, 204
 Managing UEFI Drivers 165, 166, 168, 170,
 172, 174, 176
 Man-in-the-Middle (MITM), 60, 181
 Manufacturing 41, 47, 48, 50, 52, 54, 205
 Manufacturing lines 7, 48, 49, 50
 Manufacturing process 48, 49, 52, 53, 54
 Manufacturing Test Tools 49
 Manufacturing tools 47, 49, 50, 51, 54
 – converting 53, 54
 Map 17, 85, 96, 117, 191, 193, 217
 Mappings 139, 191, 217, 218, 238, 239, 240
 – default 217, 218
 – file-system 239
 Mappings of existing devices 217, 218
 Master Boot Record (MBR), 200
 Match 113, 233, 234, 238, 239, 274, 275
 – parameter string 212
 Math 139, 140, 141, 143, 145, 152, 153
 Math Expressions 139, 141, 143, 145, 147,
 149, 151
 MaxStructureSize 97, 103
 MBR (Master Boot Record), 200
 Md 116, 118, 193, 218

- Mechanism 70, 73, 177, 178
- Media 29, 30, 253, 255, 261, 262, 282
 - bootable 17
- Medium 253, 254, 255, 263, 272, 274, 283
- MEM 35, 82, 176, 191, 193, 218, 219
- Members 96, 97, 98, 99, 100, 133, 134
- Memmap 79, 167, 191, 218
- Memory 9, 24, 33, 35, 49, 79, 82
 - block of 263
- Memory Address 81, 82
- Memory allocation 10, 35, 167, 250
- Memory Device Mapped Address 228
- Memory Module Information 227
- Memory region 210
- Messages 57, 58, 118, 119, 177, 183, 206
 - informational 176, 177
- Methods table-based 87, 88
- Migration 44, 45, 85
- Miniftp 63, 64, 65, 67
- Miscellaneous Functions 242, 243
- MITM (Man-in-the-Middle), 60, 181
- Mkdir 117, 118, 189, 191, 193, 218, 219
- MMIO 81, 82, 203, 219
- MNP (Managed Network Protocol), 57
- Mobile Internet Device 55
- Mod 151, 194, 196
- Mode 35, 220, 224, 271, 272, 273, 275
 - console output device's 191
 - page break output 238
- Modules 33, 34, 35, 94, 123, 175
 - executable 34
- MSDOS5.0, 80, 81
- MSmbiosStruct 100, 101, 102, 104

- N**
- Name 123, 214, 215, 231, 257, 269, 270
 - command's 193
 - human-readable 171, 172
 - mapped 217
 - mapping 200, 217
 - unique 29, 30, 73, 118
- Name auto 215
- Name dhcp 214
- Name of file to edit 210
- Nc option 168, 202
- NeedLog 63, 66, 67
- Network 57, 58, 59, 60, 62, 181, 191
- Network boot 41
- Network interface controller (NIC), 42, 56, 57

- Network server 48, 49
- Networking stack 56, 57
- New component information 121
- New file 207, 238
 - empty 238
- New file name 196
- NewPackageList 74, 75
- NewSize 251, 252
- NewString 251, 252
- Nextparm 112, 113
- NIC (network interface controller), 42, 56, 57
- NoFile 261, 262
- Nominal Value field 90
- Non-Script-based Programming 237, 239
- Non-volatile 130, 236, 284
- Nsh 107, 108, 109, 110, 112, 119, 120
- Nth instance 221
- Null 91, 252, 253, 259, 260, 263, 264
- Null-terminated string 248, 249, 260, 264, 269, 276, 284
- Null-terminator 227
- Number 196, 203, 204, 205, 206, 267, 286
 - large 1, 45
 - maximum 198, 200, 288
 - valid 245, 271
- NumberOfSmbiosStructures 103
- NumberOfSmbiosStructures Total number of structures 97
- Numbers output 205, 206
- Num-of-Structures 88
- NumStructures 102, 103
- Nv 226, 236

- O**
- Oemerror 211, 212
- Opened file 32, 271, 272, 273, 275, 282, 286
- Openinfo 173, 191, 220
- OpenInfo command 173
- OpenInfo DeviceHandle 167
- OpenInfo DeviceHandle Shell 167
- OpenMode 272, 273, 274, 275
- Operating system 1, 7, 13, 15, 41, 55, 70
 - traditional 29
- Operating system loaders 3, 37, 38, 42
- Operations 20, 69, 70, 71, 72, 279, 281
- Operators 145, 146, 148, 149, 150, 152, 211
 - higher priority 146
 - unary 151, 152, 153

- Option file 195
- Option filename 195
- Option number 195, 196
- Optional pointer 250, 256, 264
- Options existing 196
- OS-present application 185
- Outfile 207
- Output 107, 111, 113, 126, 192, 234, 256
- Output file names 112, 113, 155, 157, 158
 - missing 112
- Output files 155, 156, 158, 159
- Output redirection 234
- Outputfile 112, 113

- P**
- Packages 124, 126
- Page break mode 243, 285
- Param 246, 247, 250, 289
- Parameter Description 240
- Parameter passing 233
- Parameter string 212
- Parameters 66, 110, 111, 246, 250, 284, 289
 - echo 109, 110
 - first 28, 233
- Parentheses 140, 153
- Parse 144, 145, 146, 148, 192, 193, 221
- Parse Command Line in GetKey 131
- Parse shell command 193
- Parse UEFI Shell command description 193
- ParseArgs 63, 64, 65
- Parsed command-line arguments 248, 249
- ParseExpr 142, 143, 144, 146, 148, 153
- ParseExprTerm 146, 153
- Parsing 142, 143, 145, 150, 153, 155, 221
- Parsing functions 143, 144, 145
- Parties 3, 37
- Partition 30, 51, 52
- Path 116, 119, 197, 237, 269, 270, 274
 - complete 260
- Path environment variable 17, 118, 119, 242, 259, 260, 270
- Pattern 49, 113, 210, 229
- Pause 17, 117, 119, 133, 192, 221
- PC/AT BIOS 85
- PCI 36, 52, 83, 178, 179, 203, 219
- PCI configuration space 79, 191, 192, 203, 219, 222
- PCI devices 192, 221, 222
- PCI I/O Protocols 178, 179, 180
- PCI option ROM image file 216
- PCI Root Bridge I/O 178, 179, 180
- PCI Root Bridge I/O Protocol 178
- PCI shell commands 13
- PCIE 82, 83, 203, 219
- PciRootBridgelo 178, 179
- PCR (Platform Configuration Registers), 59
- Pdf 8, 9, 14, 15, 46, 47, 84
- PE (Portable Executable), 60, 182
- PE/COFF 6, 60
- PEI 34, 35, 36, 37, 42
- PEI core services 35
- PEI Modules. *See* PEIMs
- PEI Services Table 35, 36
- PEI Services Table and Example Function Declaration 36
- PEIMs (PEI Modules), 35, 36
- PEntryPointStructure 102
- Peripherals 39, 47, 50, 55
- Phases 1, 69, 70
- Phases of operation 69, 70
- PHead 102
- Physical base address 102
- Physical Size 113, 114
- PI. *See* platform initialization
- PI specifications 12, 15, 52
- Pierror 211, 212
- Ping 192, 222
- Piping 233, 234
- Placeholders 288
- Platform 7, 18, 38, 70, 71, 72, 75
 - client 56
 - headless 72
- Platform behavior changes 71
- Platform Configuration Registers (PCR), 59
- Platform firmware 37, 38, 181, 182
- Platform initialization (PI), 1, 3, 11, 69, 85, 212
- Platform Initialization Flow 34, 35
- Platform manufacturers 43, 85, 86
- Platform state 37, 38
- Platform supplier (PS), 37
- Platform vendors 16
- Plurality 59, 62, 86
- Point UEFI 41
- Pointer 99, 247, 249, 260, 263, 264, 272
 - array of 122, 144, 145
 - current protocol 268
- Pointing Device 228

- Portable Executable (PE), 60, 182
 Portion 34, 37, 195, 196
 – formatted 90, 91
 Position 44, 193, 240, 249, 266, 283, 284
 – current 239, 242, 266, 281, 283
 Positional parameters 233
 POST card 178
 PPI 35, 36
 Pragma pack 95, 96, 97
 Pre-OS state 33, 44
 PrintCommandText 276
 PrintToken 101, 104
 PrintUsage 63, 64, 67
 Priority 144, 146
 ProblemParam 250, 251
 Process 6, 48, 50, 53, 72, 183, 184
 – executable load 182
 Process script parameters 227
 Processing 22, 111, 148, 149, 155, 186
 Production build 175, 176
 Profile 116, 118, 119, 189, 190, 191, 192
 Profile names 118, 119, 189, 212, 237
 PROGRAM.EFI success echo 236
 Programmatic Shell Environment 16, 19, 21
 Programming Reference 233, 234, 236,
 238, 240
 Programs 24, 25, 132, 134, 176, 234, 235
 Prompt 127, 128, 132, 199, 278, 279, 280
 Protocol database 10
 Protocol interfaces 173
 Protocol services 10, 26, 29
 Protocol-id 202
 Protocols 9, 26, 27, 30, 31, 166, 167
 PROTOCOLShellParameters 28
 Prototype 246, 247, 249, 257, 261, 264, 265
 Provisioned Material 69, 70, 72, 74, 76, 78,
 80
 Provisioning 47, 50, 55, 62, 75
 PS (platform supplier), 37
 PSK (pre-shared key), 59
 Ptal 81
 Purpose UI designs 72
 PXE 44, 58, 62
 PXE Boot 57, 58
- Q**
- Query 62, 76
 Quit 278, 279, 280, 281
 Quotation marks 108, 109, 208, 240
- R**
- Rack 40, 44, 45
 RAID (Redundant Array of Independent Disks), 45
 Raw 38, 99, 101, 104
 Raw system board 38, 39
 Readiness 41, 43
 ReadKey 130, 134
 ReadKeyStrokeEx 134
 Read-Only (RO), 194, 220, 224, 236, 237,
 254, 255
 Read-only memory (ROM), 33, 67, 68
 ReadSize 281, 282
 Read-Write (RW), 236
 Recommended Fashion 74
 Reconnect 169, 170, 171, 223
 Reconnect commands 170, 171, 173
 Reconnect console devices 203
 Reconnect DeviceHandle 167
 Redirect 113, 150, 224, 234
 Redirection 107, 211, 233, 234
 Redundant Array of Independent Disks
 (RAID), 45
 Relationship 9, 10, 20, 34
 ReplaceWith 251, 252
 Request 24, 32, 71, 76, 278, 280, 284
 Required structures and data 91
 Reset 17, 36, 71, 209, 218, 223, 236
 ResetSystem 52, 223
 Resources 36, 250, 254, 255, 256, 273, 274
 Response 3, 62, 192, 278, 279, 280
 Result 142, 143, 146, 150, 152, 224, 286
 – structure's length 90
 Resulting answer 244, 278, 279
 Return 132, 134, 135, 144, 146, 244, 279
 – shell library 289
 – successful 251, 252
 Return DMI 103, 104, 105
 Return EFI 66, 101, 268
 Return SMBIOS structure table address 102
 Return SMBIOS Table address 102
 Return Status 28, 64, 65, 67, 101, 129, 235
 Return type SHELL 134
 Return Values 248, 249, 260, 263, 264,
 265, 267
 Returned parameter value 247
 Returned status 29, 256
 Revision 81, 97, 102, 237
 Revolution 13

- RO. *See* Read-Only
- ROM (read-only memory), 33, 67, 68
- ROM image 168, 216
- Romfile 216
- Routine Description 64, 65, 67, 101, 102, 103
- Row specified 244
- Runtime 11, 26, 27, 41, 43
- Runtime and boot services 26
- Runtime services 10, 81, 122
- S**
- SAP (Security Architectural Protocol), 186
- Scenarios 44, 55, 56, 57, 58, 68, 85
- Screen 107, 108, 198, 200, 201, 203, 285
- Script commands 24, 190, 206
- block of 107, 111, 191
 - if available 119
- Script exits 112, 117, 118
- Script files 20, 21, 22, 24, 206, 207, 209
- Script positional parameters 192
- Script Shell Environment 16
- Script-based Programming 233, 235
- ScriptFileName 63, 64, 66
- Scripting 1, 9, 17, 22, 29, 76, 117
- Scripting language 6, 48
- Scripts 22, 24, 76, 107, 108, 118, 191
- hello World 120
 - name of 65, 67
- SCSI 52
- Search 28, 29, 209, 236, 259, 260, 261
- parse command 115
- Searching 233, 242, 259, 260
- Section sectionname 210
- SectionToGetHelpOn 276
- Security 68, 182, 185
- Security Architectural Protocol (SAP), 186
- Security Considerations 181, 182, 184, 186
- Segment 219
- Serial ports 177, 224
- specified 224, 225
- Sermode 224
- Server 43, 55, 59, 62
- Server area network (SAN), 41
- ServerIpConfigured 66
- Services 9, 27, 36, 172, 174, 175, 178
- console 133
 - database 71
- Set command 225
- Set output attribute 275, 277
- Set shell command 119, 236
- Sets top nibble 212
- Settings 45, 71, 76, 121, 189, 225, 226
- Sfo 113, 197, 200, 202, 204, 216, 218
- Shell 7, 15, 16, 77, 78, 82, 167
- Shell application source code 121
- Shell applications 6, 16, 21, 25, 86
- output UEFI 123
- Shell clients 19
- Shell command cls 275, 277
- Shell command line 210
- Shell command parameters 94
- Shell command profiles 189
- Shell command source code 121
- Shell commands 114, 115, 117, 118, 119, 121, 235
- existing UEFI 193
 - lists UEFI 166
 - new 107, 237
 - new UEFI 115
 - specified 117, 118
- Shell environment 18, 22, 23, 24, 26, 237, 240
- given 189
 - running 189, 237
 - underlying UEFI 25
- Shell Environment Variables 236
- Shell executables 119
- Shell features 19, 238
- Shell interfaces calling programmatic UEFI 23
- Shell level value 189
- Shell library 155, 243, 268
- Shell parameters 107, 250, 289
- Shell Parameters Protocol 26, 28, 237, 240
- Shell Parameters Protocol Functions 240
- Shell profiles 189
- installed 118
- Shell prompt 278, 279
- Shell protocol 6, 20, 127, 133, 155, 237, 238
- Shell protocol function SetEnv 130
- Shell Protocol Functions 20, 238
- Shell protocol instance 128, 139
- Shell Script Appear 119
- Shell Script Interpreter 23
- Shell Script Resolves 31
- Shell scripts 16, 107, 193
- current UEFI 191, 208

- launching UEFI 22
- Shell Specification 4, 6, 85, 165, 197
- Shell support underlying platform 18
- Shell support levels 16, 117, 189
- Shellbook 115, 116, 118
- ShellCEEntryLib 124, 139, 141
- ShellCloseFile 241, 245
- ShellCommand 114, 221
- ShellCommandLineCheckDuplicate 246
- ShellCommandLineCheckDuplicate Detect 243
- ShellCommandLineFreeVarList 247
- ShellCommandLineGetCount 247
- ShellCommandLineGetFlag 243, 248, 249
- ShellCommandLineGetRawValue 249
- ShellCommandLineGetValue 248, 249
- ShellCommandLineGetValue Return 243
- ShellCommandLineParse 244, 250
- ShellCommandLineParseEx 244, 247, 250
- ShellConvertStringToUint 245, 252, 267, 285
- ShellCopySearchAndReplace 245, 251
- Shell.CreateDirectory 241, 253
- ShellDeleteFile 241, 254
- ShellDeleteFileName 254
- ShellDeleteFileByName Deletes 241
- ShellExecute 243, 255, 256
- ShellFileExists 241, 257
- ShellFileHandleReadLine 241, 258
- ShellFileHandleReturnLine 241, 257
- ShellFindFilePath 242, 259, 260
- ShellFindFirstFile 242, 260, 261
- ShellFindNextFile 242, 261
- ShellFlushFile 242, 262
- ShellGetCurrentDir 243, 263, 264
- ShellGetEnvironmentVariable 243, 264
- ShellGetExecutionBreakFlag 243, 265
- ShellGetFileInfo 242, 265
- ShellGetFileSize 242, 266
- ShellHexStrToIntn 245, 267
- ShellInitialize 243, 268
- ShellIsDecimalDigitCharacter 245, 268
- ShellIsDirectory 242, 269
- ShellIsFile 242, 269
- ShellIsFileInPath 242, 270
- ShellIsHexaDecimalDigitCharacter 245, 270
- ShellIsHexOrDecimalNumber 245, 271
- ShellLib 139, 141
- ShellOpenFile 242
- ShellOpenFileByDevicePath 242, 271
- ShellOpenFileByName 242, 273
- ShellOpenFileMetaArg 241, 274
- &ShellParameters 28
- ShellPkg 121, 124, 139
- ShellPrintEx 244, 275
- ShellPrintHelp 244, 276
- ShellPrintHiiEx 244, 277
- ShellPromptResponseAll 279, 281
- ShellSetEnvironmentVariable 284
- ShellSetEnvironmentVariable Changes 243
- ShellSetFileInfo 242, 282
- ShellStrToIntn 245, 285
- Shellsupport 115, 117
- ShellWriteFile 242, 286
- Shift 107, 109, 110, 112, 117, 192, 227
- Shifts positional command-line parameters 107
- Sign 181, 185, 186
- Signature 61, 176, 183, 184, 185, 186
- Signed Executable Overview 182, 183
- Signed Executable Processing 185
- Signed UEFI Executable 182, 183
- SignTool 185, 186, 187
- Simple File System 52
- Simple Network Protocol (SNP), 57
- Simple Standard Application 124, 125
- Simple Text Input 128, 134, 135
- Simple UEFI Shell Application 121, 123
- Simple UEFI Shell Application Component Information File 123
- Simple UEFI Shell Application Source Code 122
- Smbios 95, 96, 97, 98, 99, 100, 102
- &Smbios 104
- Smbios members of 97, 98, 99
- SMBIOS core driver 88
- SMBIOS data 87, 101
- SMBIOS Information 88, 102, 103, 227 – system's 192
- SMBIOS specification 87, 88, 90, 91, 97
- SMBIOS Structure Table 97, 100
- SMBIOS structures 87, 88, 89, 90, 91, 97, 98
- SMBIOS table 86, 87, 88, 92, 94, 96, 101
- SMBIOS Table Organization 87, 89
- Smbios Utility Components Architecture 94, 95
- SmbiosEnd.Raw 104

SmbiosHandle 227, 228
Smbios.Hdr 104
Smbios.Raw 104
SmbiosType 227
SmbiosView 87, 88, 91, 92, 93, 101, 227
SmbiosView display 92, 192
SMBIOSVIEW shell command 92
Sname 217, 225
SNP (Simple Network Protocol), 57
Software components 33
Solutions 17, 18, 68, 179
Source 47, 220, 234, 235, 245, 287, 288
Source Code 100, 101, 103, 121
Source File 121, 123, 124, 127, 155, 163, 199
Source file name 221
Source file/directory name 220
SourceString 251, 252
Source.txt 31, 32
Spaces 7, 208, 211, 214, 217, 251, 252
Specification 4, 5, 69, 85, 86, 87, 90
Specified file name 269
Src 199, 220
Ss 83, 219, 228
Standard Entry description 27, 28
Standard Error 27, 121, 234
Standard format output 192, 193, 200, 201, 202, 218, 219
Standard input 27, 117, 121, 234, 235, 240
Standard mode diagnostics in 78, 79, 206
Standard output 113, 117, 128, 130, 190, 192, 234
Standard UEFI data type 130
Standard UEFI driver/application 21
Standard-format output 113, 114, 115, 197, 204, 205
Start 166, 167, 168, 169, 170, 208, 209
Start UEFI drivers 167
STATIC SMBIOS 100
Statistics 92, 95, 99, 100
Statistics information display structure table 91
Status 28, 64, 66, 101, 178, 179, 256
Status code 36, 134, 256
Status Codes Returned 246, 247, 252, 253, 262, 269, 276
Step 70, 71, 74, 75, 76, 208, 209
Stop 166, 167, 170, 185
Stop bits 224, 225
Storage devices 5, 29, 52
STR 101, 104
Strcpy 65, 66, 124
String 90, 211, 245, 251, 252, 271, 277
– empty 284
– printf-style 244
– quoted 6, 193, 195, 196
– specified 275, 277
– user-typed 278, 280
String fields 91
String Functions 244, 245
String identifier 277, 280
String-containing example 91
Structure data 103, 104
Structure Evolution 90, 91
Structure header 99, 100
Structure Standards 89
Structure table 88, 90, 97, 98
Structure Table Address 97
Structure table start address 100
Structure type 89, 98, 99, 100
Structures 88, 90, 91, 95, 96, 99, 100
– last 88
– members of 100
– next 103, 104
– table 97
StructureSize 102, 103
Structure-termination examples 90
Subdirectories 199, 216, 220, 224, 229
Subsystem 52
Success 65, 101, 103, 246, 252, 253, 262
Support 16, 17, 18, 21, 22, 174, 217
– basic scripting 16, 117
– command piping 235
– inherent 72
– underlying UEFI Shell's function 16
Support file information 283
Support levels 22, 76, 117, 189
– given shell 189
Syntax 112, 113, 127, 154
System 7, 27, 50, 53, 72, 73, 75
– contents of 190, 191
System board 33, 34, 37, 39, 55
System files 216
System firmware 55, 223
System information 91, 227
System operation 74, 75
System reset 70, 113, 130, 223
System Table above-referenced UEFI 27
SystemTable 27, 28, 63, 64

T

Table 27, 95, 96, 97, 99, 100, 235
 Table entry point 87
 Table header 88
 Table Name 192, 193
 Table organization graph 88
 TableAddress 97, 101, 103
 TableLength 97, 103, 104
 Tag 29, 30, 115
 Target directory 115, 116, 117, 118, 119
 Target file 199, 226
 Target-ip 222, 223
 TCP (Transmission Control Protocol), 56, 62
 Testing 47, 48, 49, 52, 166, 170, 174
 Tests 49, 50, 52, 53, 166, 167, 174
 – subsystem functionality 47
 Text 72, 94, 126, 128, 130, 135, 136
 – text I/O functions display 244
 Text files 112, 115
 – new 107, 112
 Text strings 88, 90, 91, 97, 137, 208
 Text-mode VGA display 179
 Text-Mode VGA Frame Buffer 179
 TFTP (trivial file transfer protocol), 48, 62, 63
 Time 49, 52, 54, 117, 200, 228, 229
 – current 192, 228, 229
 Time commands 117
 Timezone 17, 117, 192, 229
 TNC (Trusted Network Connect), 59
 ToExit 63, 65
 Token 101, 104, 144
 Tools 48, 53, 54, 87, 187, 192
 TPMs (Trusted Platform Modules), 45, 59, 62
 Transmission Control Protocol. *See* TCP
 Trivial file transfer protocol (TFTP), 48, 62,
 63
 Trivial File Transport Protocol 62
 Truncates 226, 258, 259
 Trusted Network Connect (TNC), 59
 Trusted Platform Modules (TPMs), 45, 59, 62
 Type 22, 88, 124, 176, 205, 217, 278
 Type command outputs 113
 Type commands 4
 Type EFI 129, 130
 Type parameter 278, 279
 Type SHELL 129
 Type0, 98
 Type1, 98

U

UCS-2, 155, 159, 160, 161, 227, 230
 UCS-2 and ASCII file types 207
 UDP (user datagram protocol), 56, 62, 63
 UEFI 24, 26, 49, 53, 59, 62, 85
 UEFI abstracts 52
 UEFI abstracts access 52
 UEFI APIs 10, 37
 UEFI app 27, 28
 UEFI application/driver 196
 UEFI applications 9, 24, 25, 122, 123, 236,
 237
 – standard 6, 119
 UEFI BIOS interfaces 21
 UEFI boot manager 119, 120
 UEFI Boot services 12, 27
 UEFI client 55, 59, 60, 62
 UEFI Component Name protocol 201
 UEFI Compression algorithm 190
 UEFI configuration infrastructure 3, 69, 70,
 72, 73
 UEFI Driver Configuration protocol 190
 UEFI Driver Diagnostics protocol 190
 UEFI driver entry 168
 UEFI driver executes 179
 UEFI driver files 168
 UEFI driver image 168
 UEFI driver image EfiDriver.efi 168
 UEFI Driver Model 170, 171, 172, 173, 190,
 200, 201
 UEFI drivers 168, 169, 170, 171, 172, 173,
 174
 – networking 56
 UEFI drivers in option ROMs and operating
 system loaders 37
 UEFI environment 25, 26, 29, 30, 190, 191,
 218
 UEFI file image 18
 UEFI firmware 55, 56, 60
 UEFI firmware services underlying 26
 UEFI for Diagnostics 85, 86, 88, 90, 92, 94,
 96
 UEFI Forum 2, 4, 12
 UEFI functions 125
 UEFI implementation 12, 186
 UEFI infrastructure 19, 70, 71
 – underlying 19
 UEFI IP4 and UEFI IP6 stack applications 57
 UEFI IP6 stack applications 57

- UEFI IPv4 network stack 214, 222
- UEFI IPv6 network stack 222
- UEFI libraries 125, 244
- UEFI Load Image 186
- UEFI LoadImage 9
- UEFI Networking Stack 49, 56
- UEFI platform initialization 11
- UEFI protocol definitions 127
- UEFI protocol services 9
- UEFI protocols 29, 30, 71, 133
- UEFI PXE application 60
- UEFI script 23
- UEFI service 20, 237
 - appropriate 85
- UEFI services basic 124
- UEFI Shell 5, 6, 7, 9, 16, 18, 49
 - current 240
 - underlying 21
- UEFI Shell APIs 6, 241
- UEFI Shell application entry point 142
- UEFI Shell applications 12, 21, 48, 52, 121, 124, 193
- UEFI Shell Binary Integrity 181
- UEFI Shell command Connect 168, 169
- UEFI Shell command profiles 237
- UEFI Shell commands 20, 21, 113, 165, 167, 192, 236
- UEFI Shell environment 28, 29, 31, 32, 190, 191, 237
- UEFI Shell environment interfaces 20
- UEFI Shell environment variable 128, 192
- UEFI Shell environment’s initialization 29
- UEFI Shell for diagnostics 85
- UEFI Shell for provisioning 55
- UEFI Shell for SMBIOS 105
- UEFI Shell implementation details 126
- UEFI Shell infrastructure 21, 24
- UEFI Shell initialization 30
- UEFI Shell interfaces and UEFI BIOS interfaces 21
- UEFI Shell interpreter 22, 24
- UEFI Shell Levels of Support 17
- UEFI Shell Library 241, 242, 244, 246, 248, 250, 288
- UEFI Shell profiles 118
 - custom 118
- UEFI Shell Programming 121, 122, 124, 126, 128, 130, 132
- UEFI Shell programming environment 233
- UEFI Shell programs 6
- UEFI Shell prompt 16
- UEFI Shell protocol 242
- UEFI Shell script commands 20
- UEFI Shell script name 233
- UEFI Shell Scripting 107, 108, 110, 112, 114, 116, 118
- UEFI Shell Scripts 126, 127, 129, 131, 133, 227, 233
- UEFI Shell script’s positional parameters 227
- UEFI Shell searches for shell scripts 107
- UEFI Shell services 19, 141
- UEFI Shell shares 49
- UEFI Shell signals 133
- UEFI Shell Specification 117, 118, 193, 201, 202, 204, 205
- UEFI Shell specification codifies usage 86
- UEFI Shell specification for details 200, 218
- UEFI Shell Sub-team 3
- UEFI Shell Support Levels 16, 115, 117
- UEFI Shell usage models 19
- UEFI Shell utility 100
- UEFI Shell version 230
- UEFI Shell’s resources 127
- UEFI Shell’s scripting language 5
- UEFI Shell’s shellsupport level 18
- UEFI Simple File 20
- UEFI Simple File System 194
- UEFI Specification 3, 37, 181, 201, 204, 218, 237
- UEFI specification driver model 24
- UEFI StartImage 9
- UEFI system partition 37, 38, 42, 68, 181
- UEFI System Table 9, 10, 86, 133, 203, 230
- UEFI systems 17, 30, 85
- UEFI variables 11, 190, 192, 194, 198, 204, 226
- UEFI-compliant 3, 15, 24, 25
- UEFI-compliant system 5
- UefiError 212
- UefiLib 124, 139, 141, 275, 277
 - library header files 122
- UINT8, 90, 99, 101, 103, 104, 178, 179
- UINT8 EntryPointStructureChecksum 95
- UINT16, 102, 103, 104
- UINT16 Handle 97, 99
- UINT16 NumberOfSmbiosStructures 96
- UINT64, 252, 266, 272, 273, 274, 283

- UINTN 247, 249, 251, 267, 285, 286, 287
- UNDI (Universal Network Device Interface), 57, 177
- Unicode 154, 155, 156, 157, 159, 161, 241
- Unicode ASCII Unicode Variable ASCII Variable 234, 235
- Unicode byte order marks 257, 258, 259
- Unicode character 163, 268, 270, 289
- Unicode string 288
- Unicode UCS-2, 154, 257, 258
- UnicodeDecode 154, 155, 156, 157, 159, 161, 163
- UnicodeDecode Component Information File 164
- Universal Network Device Interface (UNDI), 57, 177
- Unload 166, 167, 169, 192, 230
- Unloading UEFI Drivers 167
- Unsigned integer 143, 153, 198, 205, 245, 248, 249
- Updates 50, 104, 107, 115, 192, 194, 195
- Usage 65, 67, 158, 159, 209, 210, 211
- Usage Guidelines 90, 91
- USB 36, 49, 52
- USB device 181
- Use of UEFI for Diagnostics 85, 86, 88, 90, 92, 94, 96
- Use ShellFindNextFile 260
- User datagram protocol. *See UDP*
- User input 94, 114, 244
- User interface firmware's 223, 224
- Users 1, 4, 5, 15, 43, 92, 177
- UseScript 63, 64, 65, 66
- Utilities 5, 17, 18, 62, 63, 100, 101
 - diagnostic UEFI Shell-based 87
 - executable 17
- V**
- Valid file names 233
- Valid values 117, 197, 229, 278, 280, 289
- Value 111, 178, 179, 208, 225, 248, 284
 - current 219, 226, 239, 284
 - enumerated 278, 279, 280, 289
 - following 224, 227
 - hash 183
 - initial 111
 - integer 214
 - invalid 253, 255, 272, 273, 284
 - raw 243, 249
 - resulting 115, 143
- Value of ShellPromptResponseTypeYesNo 278, 279
- Value parameters 248
- Variable 135, 194, 195, 204, 226, 236, 237
 - global 63, 128, 130, 131, 140, 155
 - lasterror shell 235
 - local 129, 130, 133, 135
 - new 226
 - shellsupport 16
- Variable argument list 276, 278
- Variable contents 204
- Variable-name 226, 236
- Var-name 204
- Verbose 202, 209, 210, 230
- Volatile 130, 226, 236, 284
- Voltage Probe structure 90
- Volume 253, 254, 263, 273, 274, 283, 287
- Volume label 192, 230, 231
- V/RO uefishellversion 237
- W, X, Y**
- Wide-area networks (WANs), 58
- Width 82, 288
- Wildcards 199, 208, 215, 216, 217, 220, 224
 - command support 217
- Windows command prompt 107
- Work 1, 24, 52, 53, 107, 179
- World 107, 108, 119, 120
- Wrappers 122, 124
- WriteFile 159, 162, 240
- Www.uefi.org 2, 4, 15, 85
- Z**
- Zimmer Vincent 3