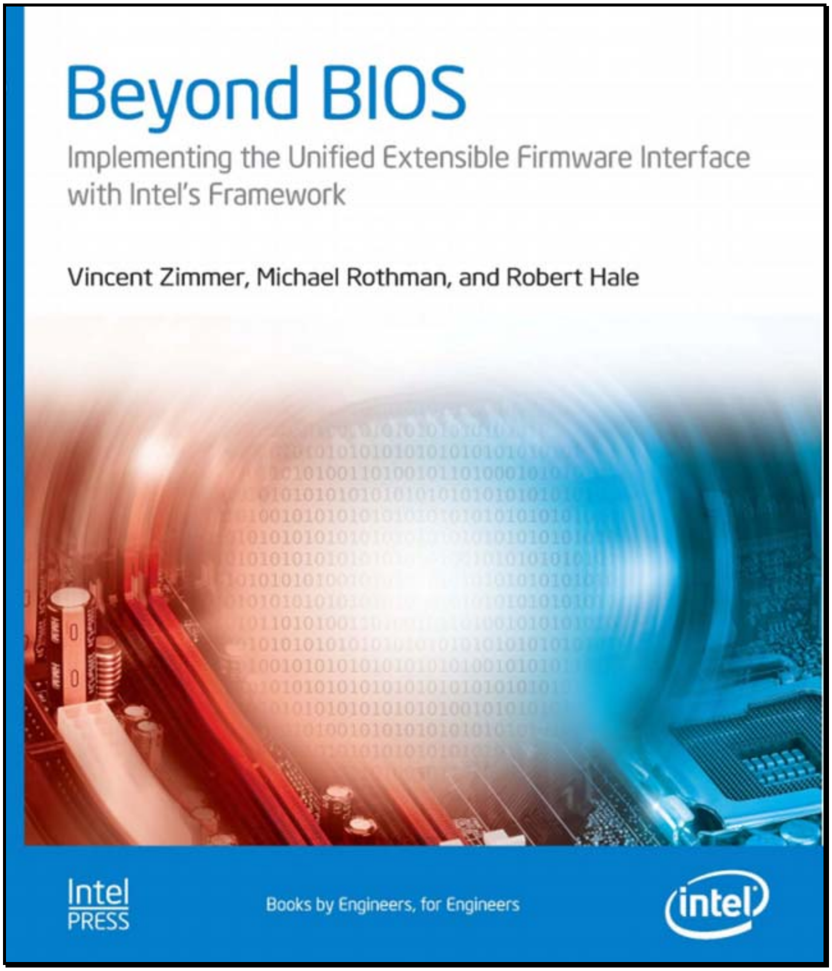
Digital Edition

Digital Editions of selected Intel Press books are in addition to and complement the printed books.



Beyond BIOS

Implementing the Unified Extensible Firmware Interface with Intel's Framework

Vincent Zimmer, Michael Rothman, and Robert Hale

Intel PRESS

Books by Engineers, for Engineers

intel

Recommended Reading
Essential for Developers

Click the icon to access information on other essential books for Developers

Visit our website at www.intel.com/intelpress

Beyond BIOS:
Implementing the Unified Extensible Firmware Interface with Intel's Framework

Vincent Zimmer
Michael Rothman
Robert Hale

Intel
PRESS

ISBN 0-9743649-0-8

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

Fictitious names of companies, products, people, characters, and/or data mentioned herein are not intended to represent any real individual, company, product, or event.

Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

This book is printed on acid-free paper.

Publisher: Richard Bowles

Editor: David J. Clark

Program Management: Ashwood Group and Douglas Technology Group

Text Design & Composition: Horizon Interactive

Graphic Art: Kirsten Foote (illustrations), Ted Cyrek (cover)

Library of Congress Cataloging in Publication Data:

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

First printing, September 2006

To my wife Jan who is always there for me, my daughters Ally and Zoe whose smiles never fail to make me smile, and to my parents who taught me the value of hard work.

—Vincent Zimmer

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.

—Mike Rothman

To my late mother, Patricia, who taught me more than I know, to my father, who continues to teach me, and to my UPSD BIOS colleagues, who continue to amaze in transition.

—Robert Hale

Foreword

Beyond BIOS. Those two words began to circulate through the elite firmware architects and developers in the industry standard computing circle around 1998, when Intel, Microsoft, HP and a number of other companies began to lay out the plan for bringing up the first Intel® Itanium® systems. The plan was originally called IBI, the Intel Boot Initiative. Mainstream PCs had been using BIOS ever since the beginning of the IBM PC. Its drawbacks and limitations were magnified in the "big iron" machines based on the Itanium processors. For example, BIOS depends on many of the PC-AT hardware such as the 8254 timer and 8259 interrupt controller, which were not designed to scale to larger servers like the HP Integrity Superdome† servers. Worse, BIOS assumes a 1MB execution memory limit and has very limited memory space to execute the Option ROMs on the add-in cards. BIOS' 16-bit nature stifles the platform advancement for Itanium systems that are 64-bit based.

There have been non-BIOS solutions in the more proprietary vertical integrated systems design, such as Open Firmware used by IBM Power†, SUN SPARC†, and Apple PowerPC†; ARCS† by DEC Alpha, and PDC/IODC† by HP PA-RISC. Open Firmware is Forth-based, it is difficult to find the talent, and its specifications have not kept up with the evolution of the technology. ARCS lacks the driver model to support add-in cards. With BIOS hitting the wall and no clear alternative that can be brought into the industry standard arena, Intel spearheaded the IBI, which at this stage is named Extensible Firmware Interface (EFI), to reflect the objective of the effort. EFI brought the modern computer software architectural concepts into firmware. EFI enables firmware development in high-level languages like C, provides proper abstraction of hardware, and enables extensibility through the GUID concept. The benefits of EFI were so convincing that

Microsoft and the industry made it the only boot mechanism for the Itanium-based systems.

Now that the IA-32 processors are also extended to x64, the industry is working on Unified EFI (UEFI) as the standard pre-boot firmware infrastructure going forward. Intel also spearheaded the effort on the Intel Platform Innovation Framework for EFI (Framework). The Framework is Intel's implementation of EFI, and it is also the starting point for the industry to define the Platform Initialization (PI) specifications that establish the firmware internal interface architecture as well as firmware-to-silicon interfaces that enable silicon driver modularity and interoperability. With Framework's implementation of UEFI and PI, Intel has completed the task of replacing BIOS, thus enabling the industry to move *beyond BIOS*.

This book is a landmark in the development of the UEFI and PI firmware. It couples a powerful, modern firmware infrastructure with a unique look into the mind of a few UEFI/PI architects who have made the Framework implementation into reality and as a reference. It's accessible to the student in firmware development, conveys a deep technical understanding of the UEFI/PI architecture and the Framework implementation. It covers all major areas of the Framework implementation that the firmware developers need to understand. Using this book along with the EFI Development Kit (EDK) code on the TianoCore.org open source project would be the excellent tutorial for the firmware engineers to move beyond BIOS.

Vincent Zimmer has been running the PI Working Group meetings and Michael Rothman has been running the UEFI Specification Working Group meetings, helping the Chair of both groups, Mark Doran. Vincent has also been very involved in the Trusted Computing Group, defining security related extensions for EFI. Michael Rothman and Robert Hale are also the lead forces in the development of the Human Interface Infrastructure coming up in the next generation of the UEFI Specification. These authors are some of the elite EFI architects who have helped bring the dream of moving beyond BIOS into reality, thus among the best possible teachers of this subject matter.

This book is for you if you'd like to understand the UEFI/PI architecture and the Framework implementation; that is, to understand how to move beyond BIOS. It gives you all the technical background to understand Bill Gates' WINHEC 2006 Keynote Speech when he said: "There are changes across the board, in terms of how hardware and software work together. If we think about boot, we're finally moving away from the old BIOS to this unified extensible firmware interface, and that gives us new flexibility and capability, and it's got a rich API set to build on, so many of you are working with us on that." For any student in this field, this book provides an important bridge between normative specifications and the informative details of the development.

Today, Itanium-based systems are no longer the only machines supporting EFI. All the Intel®-based Apple Mac† systems are supporting EFI. Systems based on x64 processors are also in the process of supporting UEFI. Embedded systems are also making use of the UEFI/PI architecture in its specialized environment. Operating systems, such as Windows†, Linux†, HP-UX†, Open VMS, FreeBSD, and so on are already EFI-based on Itanium-based systems. Windows, Linux and OS-X† are in the process of supporting UEFI on x64 systems. Currently OS-X already supports EFI on the IA-32 processor-based Mac systems.

This book is the first to describe in detail the Framework implementation of UEFI and PI architecture. I am very pleased to recommend this new must-read to you who may have been living in the BIOS world for so long to see the life beyond BIOS as envisioned by Bill Gates. I also recommend that readers take full advantage of the open source TianoCore.org. Sample code is worth a thousand words. The EDK is a great companion for the book.

Dong Wei

Vice President and Chief Executive, the Unified EFI Forum

HP Distinguished Technologist

Granite Bay, California

June 18, 2006

Contents

Preface

A plan is a common basis for change.

—Dan Laner

This is a book about a new way to solve an old set of problems that are persistent as well as fundamental, but not always well understood: How should you boot a computer? What sits at the reset vector? What can the operating system count on when it is loaded and initially receives control? What should the internal structures be between these two endpoints? How can the same basic structure work for handhelds and megaservers? How do we convince ourselves today's design will work 10 or 20 years from now? How much will it cost to switch? How much will it cost steady state? What comes after BIOS?

Beyond BIOS is a book about a largely invisible subject. The general user, if they have any view of BIOS at all, tends to view it as 10 unnecessary seconds on the way to booting the operating system or as setup. The community that knows and uses the BIOS has tended to view it as an uncontrolled place of kludge, myth, bug, and legend. The very small community of BIOS developers has viewed their code not only as highly mutable and embodying much of the compatibility that has made the PC and its offspring so successful, but also as their livelihood.

This is a book that is about what comes after BIOS, which we call **Extensible Firmware Interface** (EFI) and the Framework. In doing so, it must also be a book at least partly about what a BIOS or its replacement is called upon to do. It is not a cookbook on how to port the Framework from platform to platform. It is not a rehash of the specifications. Instead, it tries to fit in the middle ground between specifications and cookbook. It tries to focus on the concepts and constructs that are cross-platform and implied, if not stated, by the architecture. It is supposed to help to get to some of the "why" behind the specs and make the porting work make some sense.

This book is a child of its time. Both the EFI and the Framework are now substantially under the control of the Unified EFI Forum, an industry-wide group that you are encouraged to participate in. *Beyond BIOS* mainly focuses on the state of the Framework and EFI before being modified by that Forum, its working groups, and its sub-teams. This is not to say that this is only a history book. We believe it remains valuable as an introduction to the newer versions of the specifications no matter who "has the pen."

The Chapters

Chapter 1 provides a historical introduction and delves deeper into the motivation behind changing the way the system boots. It then describes the goals that precipitated the architecture and the non-goals, which are as important.

This rest of the book is organized into two major sections. The earlier chapters present an introduction to EFI, and the later chapters cover the Framework.

Chapter 2 provides an overview of the basic EFI architecture. This is a must-read for anyone seeking an understanding of the Extensible Firmware Interface (EFI).

Chapter 3 describes the EFI driver model. This is important for vendors writing device drivers for output devices (such as video), input devices (such as keyboards or mice), networking adapters, and block devices. These drivers can be stored in the host-bus adapter, the platform ROM, or loaded from the EFI system partition.

Chapter 4 describes of series of commonly used EFI protocols. This chapter complements the earlier two chapters and includes data on additional boot services application interfaces.

Chapter 5 includes information on the EFI runtime operational environment. This chapter is important for operating system vendors who need to interact with the platform during the operating system execution.

Chapter 6 describes EFI input and output console services. This chapter provides details on the particular capabilities, interfaces, and relationships of the console services.

Chapter 7 includes a list of different platforms and the Framework implementations. This chapter demonstrates the flexibility of the Framework by mapping the infrastructure to widely varying hardware platforms.

Chapter 8 describes how the Framework manages where it lives. It delves into the operating-system–like model used to store code and data and where the Framework's design differs and why.

Chapter 9 describes the basics of the Driver Execution Environment (DXE). This is important to read for anyone working on the phase of execution prior to EFI service availability but after early platform initialization (PEI).

Chapter 10 describes some common EFI interfaces. This chapter includes information on interfaces that are important for both EFI and DXE development.

Chapter 11 describes the information paths, roads, and highways that interlink the phases and modules of the Framework.

Chapter 12 provides information on the differences between DXE and EFI drivers.

Chapter 13 de scribes Boot Device Selection (BDS). This includes the policy by which Framework platforms decide look-and-feel, in addition to how to boot.

Chapter 14 describes the various boot flows that can occur within a platform. These include power-event restarts, and so on.

Chapter 15 describes the Pre-EFI Initialization environment. This is the phase of execution that occurs after reset and is responsible for the early hardware state and memory initialization.

Chapter 16 includes information on emulation of a firmware environment within an operating system.

Chapter 17 describes the Compatibility Support Module, or how to support today's BIOS boot environment.

The Appendixes include source code data types and commonly-used interfaces.

Acknowledgements

The authors are grateful to Debkumar De, who wrote the CSM chapter. We are overwhelmed with Deb's knowledge, diligence, and helpfulness.

The authors recognize the efforts and contribution of the two men and a dog: Mark Doran, Ken Reneris, and Andrew Fish, who conceived and hatched EFI.

The authors recognize and thank the other original Framework (Tiano) architects Andrew Fish, Mike Kinney, Barnes Cooper, Will Stevens, Krithivas, ER Uber, Mahesh Natu, Rahul Khanna, Jim Ewertz, Kirk Brannock, and others whose names are lost to time and the team's intrepid leader, Mark Doran. We thank Isaac Oram, John Lambino, and the entire TAT team for fleshing out and enhancing the architecture. Thank you to the Tiano engineering team for their patience while implementing the first versions and to our internal and external customers. The innovation in this book is from these fertile brains.

We thank our managers, past and present, for giving us the chance and the time to work on the architecture and this book including Tom Yan, Doug Fisher, Richard Wirt, Stu Goossen, Mike Richmond, and Ron Story. We particularly want to acknowledge the father of Tiano, the late Mil Travnicek. We hope this is good.

We acknowledge the ever-supportive marketing team: Yosi, Laurie, Harry, and Marty.

No Intel Press book is published without peer review. We'd like to thank all the reviewers for identifying errors and for providing valuable insight and encouragement along the way. Without their help, this book would not have been a success. From Intel, these individuals participated, at one time or another, in the review of this project: Rob Branch, Mallik Bulusu, Brad Davis, Michael Krau, John Suresh Kumar, Matthew Parrish, Mike Richmond, Lee Rosenbaum, and Sudhakar Otturu. Other reviewers included Cameron Esfahani from Apple Computer Corporation, Todd Greene from QLogic Corporation, Penny Huang from Micro-Star International Company, Limited, Jimmy Hwang from American Megatrends, Incorporated, Jiming Sun from Advanced Micro Devices, Incorporated, and Dong Wei from Hewlett-Packard Development Company, L.P.

A book like this describes the efforts of a large number of talented individuals. The authors would like to thank all of them for their efforts and support. Please accept our apologies if we missed you. We can only say that space is as limited here as it is in ROMs and time as limited here as it is in schedules. We'll try to fix it in the next release.

Finally, we'd like to thank the team from Intel Press. Stuart Douglas was the content architect for this project—his steady guidance and ability to find resources as we needed them kept this book on track. David Clark was the editor on the project, and helped take our engineering prose and turn it into a real book. David Spencer and Matt Wangler were involved in the early stages of this project, getting us started on the right foot.

Chapter 1

Introduction

> *Come visit me, here in this better street. Pretend you are a white rabbit...*
>
> —Harlan Ellison

The average user is unaware of how much goes on behind the scenes in the operation of a modern computer. This is by design, Developers put a lot of time, thought, and energy into abstracting the disparate parts of the computer into a more-or-less seamless whole. Nowhere is this abstraction more obscure, even to those who program the rest of the computer, than in the program that starts the computer running.

The complexity of *boot firmware* has evolved to keep pace with the ever-increasing complexity of the underlying hardware and of the software that this firmware loads. Defined by its interfaces to the operating system and option ROMs, the latest architecture for boot firmware is the Extensible Firmware Interface (EFI). An almost infinite number of software architectures may implement EFI. This book describes the components and characteristics of EFI itself, then explains a rich architecture that implements EFI that is known as the Intel® Platform Innovation Framework for EFI, or simply as the Framework.

Both EFI and the Framework use concepts and techniques derived from the now-standard software disciplines. Although EFI and the Framework do not define an operating system, in any traditional sense, they use many concepts borrowed from that area of study. The design of EFI and Tiano also reflect the state of knowledge and trends in software engineering as well as the hard-nosed experience gained by the embedded community. Both streams of thought have required tempering in the light of experience hard gained with previous boot firmware architectures.

History

Initially, computers had no boot firmware. Instead, the user had to enter a boot program by hand using switches on the front panel of the computer. This process was slow, quite laborious, and error prone. Some computers used complex instructions, so designers simplified the process, reading in the program from a paper tape, for example. One rather peculiar piece of "automation" during the era—this method was fairly commonly used—was a piece of plywood which had notches for each switch. The notches were designed such that, when you aligned the board with the switch panel, then slid it first downward and then upwards, the switches would end up in the right positions. It was also common for the operators to hold contests to define smaller programs or, more usually, ones with fewer switch changes, to boot the system.

Later, the initial switch positions were encoded as diodes on cards and were treated somewhat like a peripheral. Only a fairly trivial program to copy the diode "memory" to RAM (core) had to be entered.

With the advent of smaller computers, small programs, typically no more than 256 words long, were stored in newly available ROMs, which allowed boot from a very limited number of devices. If the operator needed to boot from a different device—from the paper tape reader rather than the drum or disk, for example—he would physically replace the ROM.

Once loaded, the operating system usually did not use the code in these ROMs. Instead, most operating systems were linked with the drivers for the boot devices so that they could initialize them immediately.

Firmware as Hardware Abstraction

A major change in this process was defined in the early 1970s by, among others, Gary Kildall. Kildall, later to become famous as the author of CP/M and owner of Digital Research, proposed that the boot firmware provide an abstraction layer between the system hardware and the operating system. This layer was to be used both to boot the system and to provide low-level communication with the system's basic peripherals. Basically, this firmware was an attempt to solve a basic operating system dilemma: how do you get drivers for the peripherals you are loading drivers from?

When the PC was developed, developers implemented an extended version of the same concept called the Basic Input/Output System, shortened to BIOS (IBM 1985). The BIOS consisted of two main pieces: the

Power On Self Test (POST) and the run-time, which served as an abstraction layer for early PC-based operating systems.

The basic goal of the BIOS was to test and initialize the system, to find an input device, which usually meant the keyboard, an output device that was most often a graphical monitor, and a boot device, usually a hard disk. Then, BIOS starts the operating system boot process by loading the operating system's 512-bytes first-stage loader from the hard drive and passing control to it. The BIOS then played a supporting hardware abstraction role for the devices of which it was aware.

Over time, the basic flaws in the concept became apparent. The abstractions provided by the BIOS were extremely primitive and provided no synchronization. Due to the lack of synchronization, the BIOS abstractions were polled rather than interrupt driven. Further, the processor mode assumed by the abstractions assumed that the processor was in real mode, which was not the mode the operating systems ran in. These same abstractions, however, stayed in use but only during the OS bootstrap process.

The abstractions had other issues with extensibility:, the ability for the abstraction to be upgraded and extended as new technologies become available. For example, the video abstraction (INT 10h) relied on enumerated modes for video resolution. Most of the early common modes, such as 320x200, became obsolete almost immediately. IBM managed extensions initially, since it invented the interface. As ownership dispersed, the extensions were managed either by standards associations or by corporations, who simply extended them without regard to their meaning, if any, for other products.

The abstractions have allowed for implementation of numerous underlying architectures, all of which are considered compatible with one-another. The implementations have ranged from the very monolithic to the very modular. A common thread is that almost all have been written primarily in assembly language, due to a number of factors including:

- Tradition
- Size constraints
- The general processor memory model, also called big real mode, not being a target of any compiler
- The fact that the interfaces must work with limited stack space and are not high-level language friendly

Notably, many more recent BIOS implementations have used C for certain modules, particularly for setup. Typical BIOS development tools have also been written in high-level languages, including C, C++, and Perl.

Option ROMs

The PC BIOS provided an important extension to the concept of hardware abstraction in firmware: Option ROMs. An Option ROM is a BIOS extension that resides on an add-in card. Although not presented as such at the time, the Option ROM serves the same purpose as a device driver in an operating system: to allow the base software to access peripherals that it does not intrinsically know about.

The PC BIOS Option ROM implementation had several flaws:

- *No standard way for option ROMs to obtain and keep RAM or other system resources.* Resource utilization was not well defined, particularly the maximum stack space utilization allowed during calls.
- *No clean way for the option ROM to add its features into the hardware abstraction.* For example, it offered no feature for SCSI cards to add abstractions for the drives attached to the card to the system's hardware abstractions. The option ROM was stuck with trapping and redirecting the standard INT 13h disk interrupt.
- *No standard way to control boot order* between the devices controlled by option ROMs and the devices owned by the motherboard.
- *No common way to enter ROM configuration.* Although many option ROMs required configuration, remote access to the Option ROM configuration was not easy because the poor video interface support defined by the BIOS video abstraction required direct hardware access in order to achieve reasonable performance.

Motherboard Hardware Initialization

The advent of POST was indicative of a trend towards hardware that was less initialized at reset. Software began to initialize the system's hardware. This approach has several advantages that have become more predominant over time:

- Hardware, particularly chips, can be made generic by allowing the platform firmware to initialize the chip to the configuration required by the platform.
- Hardware initialization of large parts of the hardware to known non-simplistic default values has proved error prone and expensive, as measured by the amount of space on the die the initialization occupies.
- Hardware-initialized configurations require expensive hardware changes in order to fix bugs. Firmware requires at most an updated ROM, which is significantly cheaper. As such the boot firmware has become the repository of work-arounds for chipset issues.

The main function of POST in the early PC and AT systems was, however, not initialization but test. As the components have become more reliable and more highly integrated, the necessity for testing has lessened, although it has not disappeared.

Other requirements were added to this largest piece of the BIOS as time progressed. For example, the BIOS is responsible for loading processor microcodes and has become responsible for the description of motherboard features that cannot otherwise be discovered by the operating system.

When the PC was first produced in 1980, the motherboard and almost all add-in cards had banks of DIP-switches that you used to configure them. The configuration ranged from the basic actions, such as interrupt and I/O range allocation, to the description of specific equipment characteristics like motherboard memory size, LAN "MAC" address, and so forth. Systems were difficult to configure since it one could easily set cards to conflicting I/O ranges and interrupts.

As the cost of transistors on silicon dropped dramatically, it became more and more possible for cards and on-board devices to be soft-configured. Buses became enumerable: software, including the BIOS, could locate add-in devices, determine their resource requirements, and satisfy them, in most cases.

The longer term trend is towards peripherals hung off of serial buses, which do not use the number of traditional resources that parallel devices do. Examples are 1394 and USB. A serial bus can support many USB devices using the same fairly minimal hardware resources as a single USB keyboard.

The bus initialization in high-end servers can be extremely complex, but this sophistication is required to locate possible input, output, and boot devices.

BIOS as Differentiation

Early on, planners at many companies producing systems noted that the BIOS had two main points of unexpected value. First, the BIOS was the only software that the manufacturer produced that they could be sure the user actually ran, since it was required to boot the system. Secondly, the firmware was the only piece of software that was absolutely tied to the platform, since it was soldered on the motherboard or, at the very least, installed in a socket on the platform.

This certainty, plus intensifying competition, drove the companies producing systems to see BIOS as a place to put what would be called value-add or product differentiation features.

The most obvious place this change occurred was the replacement of the on-board DIP switches with a system Setup application that was accessible with a hot-key. The addition of Setup caused interesting side-effects when developers tried to localize systems to various languages and geographies. English-only Setup, along with BIOS error messages and miscellaneous prompts, had to start speaking French, Italian, Chinese, Japanese, and the like. Support for localization became a major effort for most BIOS developers.

A more user-friendly Setup was no good without some place to store the information. The IBM PC/AT introduced the idea of storing the configuration in the same non-volatile part that stored the time-of-day clock (IBM 1985). Over time, this became known as "CMOS" storage. It also became a management headache, since the amount of storage[1] was always so limited that it became impossible to allocate different ranges to different modules. Instead, the task of CMOS allocation became a part of the build process. CMOS was allocated per bit, not per byte.

If One Abstraction Is Good...

The BIOS run-time saw two almost contradictory trends. First, the operating systems ceased using the run-time interfaces, using them only during bootstrap. The second trend was the proliferation of various new abstractions. Certain abstractions were required to replace older abstractions

[1] Storage at first was 48 bytes, then 116 bytes, and finally in some implementations 240 bytes!

that could not support increasingly complex and large systems. For example, the initial abstraction for reporting memory size was limited. The maximum amount of memory it could report was 640 kilobytes and later 24 megabytes. As processor address ranges passed these sizes, new abstractions were required. Similar issues occurred with hard drives.

Other abstractions covered new topics. Most of them centered around manageability. Developers noticed that the BIOS was in a unique position to report system data, ranging from the manufacturer to slot information to configuration information. Specifications ranging from DMI and SMBIOS to PnP BIOS to ACPI proliferated.

The BIOS business evolved into a number of larger companies with their own staff, and four vendors offered their code in source or binary to those large companies as well as to smaller companies. Each BIOS base defined its own set of unique and sometimes conflicting interfaces.

The basis of the conflicting interfaces goes back to the basic definition of the calling conventions for the abstractions: the use of the INT instruction. This instruction is analogous to a hardware interrupt, causing an immediate vectoring to a known location. In the base 8086 architecture, 256 interrupts were available. In the PC architecture, interrupts between 0x10 and 0x1F were allocated to BIOS. Sub-functions were defined by the contents of the AX register. No single body controlled the allocation of sub-function values, with VESA being a notable although eventually unsuccessful exception. The issue was most obvious on INT 15h, which is the miscellaneous interrupt.

Highest Cost Per Bit

The BIOS was originally stored in ROMs. As the size of the BIOS increased, the size of the ROMs also increased. As the complexity of the BIOS increased, so did the likelihood of bugs in the BIOS, although even the original PC BIOS contained a few. As the number of bugs increased, it became more and more common for people to have to change BIOSs in their systems. This process was error prone; physical chips had to be removed from sockets and new ones added.

In the late 1980s and early 1990s, ROMs were replaced by flash, which was the code name for a non-volatile, alterable form of memory that acted like ROMs. The storage of BIOS in flash has become the boon and curse of developers. Flash allows for updates to be sent, programs run, and BIOSs to be updated. On the other hand, flash is not a friendly medium to program.

Generally but not always, flash is divided into regions called *blocks*, which function much as sectors do on hard drives. However, due to characteristics of the technology used to implement flash, the blocks are much larger than the typical 512 bytes of hard-drive sectors. One common flash part for example was 512 kilobytes divided into only eight 64-kilobyte blocks.

Flash has the general property that changes from ones to zeros may be made by simple programming while any change from a zero to a one requires rewriting the entire block. Rewriting the entire block can also take many hundreds of milliseconds. Consider the apparently simple task of updating the block of BIOS flash to which the boot vector points, that is, the highest block in the 4-gigabyte address range on most PCs. To update the block, you must first erase the block since you are going to change some zeroes to ones. If the system then loses power, it could never boot again unless you make a trip to a rework station.

How Does It Work At All?

With this list of issues (and several more), it is a wonder that the PCs the world increasingly depends on boot at all. It has taken the efforts and dedication of hardworking developers at companies worldwide to evolve the initial efforts of the BIOS over the years. Cracks have appeared and have been patched. A notable example was that the initial maximum size of a hard drive was 540 megabytes, a figure that now seems quite laughably small but in 1980 was well into mainframe territory. New calls were defined by the industry and enforced by the operating system vendors.

Some cracks proved more difficult to patch. Through the 1990s, the industry used 16-bit code and interfaces to boot 32-bit operating systems. With the advent of 64-bit processors that were practical enough for powering PCs, the stretched BIOS was clearly at its end. The actual breaking point came when the 512-byte boot block was shown to be too small to fit the required 64-bit Itanium® instructions.

It would be remiss not to note that the original IBM design was for what was expected to be a well-defined product without an extended future. The original designers of the BIOS deserve full credit for designing something flexible enough to last 25 years and in systems from a few megahertz to many gigahertz and from parts of a megabyte of RAM to many gigabytes. We can only hope EFI and Framework will be useful for as long.

Non-BIOS Alternatives

Other firmware designs appeared since the design of the BIOS. Most notable of these were Open Firmware and ARC.

Open Firmware (IEEE 1275) is a rich boot environment built around the Forth language. Forth is interpreted, making the job of developing Instruction Set Architecture (ISA) neutral code much simpler. Required support for legacy BIOS interfaces such as Option ROMs, plus the complexity surrounding Forth itself (described by many wags in the industry as "the world's only write-only language" due to its cryptic nature) drove the decision to look elsewhere for the EFI and Framework designs.

One of the trends that appears in these more recent implementations is support for file-system-based operations rather than the limited block I/O operations supported in BIOS. This means that the firmware understands the disk at the directory and file level rather than just the sector level.

EFI Goals

After 25 years of living with one set of interfaces, it is inevitable that many of the goals for the new interfaces revolve around addressing the known warts of the old interfaces. It is important however to go beyond resolving old issues. The following is a subset of the goals the designers of EFI had in mind. As with any engineering effort, it is inevitable that the goals conflict. One of the important details to study in the remainder of the book is the tradeoffs that were discovered and how those tradeoffs were handled.

Operating System Neutrality

The BIOS has been operating system neutral throughout its life, enabling remarkable innovation in the operating system community while enabling a vibrant community of platform sellers. The alternative has also been successful: a platform developed by the same company as the operating system, allowing close, often seamless, integration between software and firmware.

Crisply Defined and Extensible Interfaces

Compatibility should be able to be well-documented and developers should be able to test for it, not a set of disparate documents and conventions. Compatibility should be limited to interfaces, not the implementations underlying them.

Interfaces should be flexible enough to withstand changes of the kind that are frequent in the industry. Innovation should be enabled, even encouraged, by allowing developers to define new private interfaces without going to a standards body. This flexibility is accomplished by naming each interface by GUID, a unique identifier that can be automatically generated without fear of collision.

Modularity

The programming environment should lend itself to a cooperating set of modules that may be created by different organizations or companies. EFI enables this cooperation by supporting drivers and applications and common mechanisms for inter-module cooperation.

Known Set of Intrinsic Services

Modules should have a set of the basic services that are provided by any common operating environment that is available to them. These services are focused on the common jobs resolved by EFI. For example, memory management and priority levels are common commands found in most operating systems whereas management of the global coherency domain (GCD), the allocation of the system's consumable resources to devices and processes, is more focused on the job of booting a system.

Instruction Set Independent

The design of the firmware should permit the common pieces to be retargeted from platform to platform and even from one instruction set to another. EFI has been successfully retargeted thusfar to at least three separate instruction sets.

High Level Language Friendly

The values of programming in high-level languages are well known. The interfaces should be able to be called from procedures or functions, and the supporting code should be written in the most commonly used systems-based high-level language, C. The change in language from assembly to C does not mean that anyone who writes C can develop firmware. Expertise in developing in the pre-boot space is about much more than just the language used.

Option ROMs as Full Partners

Option ROMs should be viewed simply as other modules, with the same rights and responsibilities as those other modules. Any given option ROM should be able to count on the basic services and other known protocols and use them as an integrated component.

Scalability

The PC BIOS was defined for what would today be called desktop systems and has had to be wedged, crunched, squished and otherwise jammed into everything from PDAs and sub-notebooks to many-way servers. EFI should support all levels. This requirement doesn't mean that all modules the majority of which are known as drivers, are applicable to all environments. A bus enumerator for a desktop might be inadequate for server applications even though the interfaces are identical.

Long-Lived Abstractions

Experience with BIOS indicates little gain in size savings from abstractions that attempt to minimize space by skimping on headroom, but these abstractions can lead to headaches as peripheral capabilities grow. For example, the maximum address space for hard disks in the original BIOS was 540 megabytes. Using a 64-bit number to describe the linear sector index means we won't have to worry about the problem.

Boot Manager

EFI needs a central focus for user control of boot order and, at the same time, a programmatic control for optimizations that can speed up boot time, particularly the initialization of only parts of the platform. These functions are owned in EFI by the boot manager.

Rich "Pre-Boot" Environment, Limited Run-Time Environment

EFI was designed to follow the current and expected long-term model for boot operations. The firmware provides basic abstractions to support the OS boot loader and to provide a description of the system. The defined set of interfaces is rich enough that simple command shells have been defined that use EFI as their interface to hardware.

Successful use of the firmware in OS run-time environments has been limited, and usually, it consists of tabular data, which may be data structures such as SMBIOS or interpreted data, as in the case of ACPI,. Support of call-based interfaces is complex, given the requirement that EFI must be operating system neutral. Operating systems have varying expectations for a variety of parameters, including interrupt latency, memory configuration, and peripheral usage, thereby making call-based interfaces complex to code and of questionable reliability in the best of cases. EFI defines a minimal number of these parameters to enable communication between OS and firmware and a few others for non-OS purposes.

Framework Goals

Not surprisingly, EFI's goals are also the Framework's goals. Other goals, or modifications and clarifications to EFI goals, drove Framework design.

Firmware for the Next 20 Years

The basic BIOS model has survived since 1981. However, the underlying code has undergone enough changes to be unrecognizable. Each BIOS-developing entity has gone through several iterations of restructuring to improve the code and make it better suited to supporting the requirements of the time. This development has led to a common set of basic interfaces for external customers but considerable internal fragmentation. A piece of code written for one restructuring is unlikely to run in another version of the same company's code, let alone in that of a different organization. The variability in calling conventions, module construction, and supporting infrastructure makes such interoperability impossible.

A driving concept behind the Framework is to provide a basic infrastructure for the levels between the boot vector and EFI. This common denominator does not stifle innovation. An operating system defines a similar structure. If the operating system provides a reasonably well–defined, well thought out set of services, the applications running on that operating system are, in fact, freed to go do new things.

With the Framework, one might expect the defined common interfaces to make something similar happen. For example, a chipset vendor could provide a single more fully validated set of modules supporting the newest chipset, and those modules would be closer to production quality. Being "only software," the Framework can only enable technologies. Vendors must determine the business strategies that are best for themselves.

Modularity Using an Object Model

It is probably safe to say that every new BIOS core developed over the last ten or fifteen years claims that it is more modular. Why spend so much time, effort, and money on modularity? Code reuse! Well-isolated modules can be reused from product to product, saving development costs and validation costs, which are increasingly important.

Of course, modularity is an overused, under-defined term. The Framework model follows the EFI driver model. In design recommendations, however, the Framework model goes further. Consider a request for a module abstracting a combination I/O device, such as a parallel or serial printer, a keyboard, and a mouse. A model that demands strict modularity would require the drivers that abstract the device also to provide all of the configuration support, up to and including the Setup questions. The recommended Framework module model goes more towards a modern object model. The drivers supporting the device provide interfaces to configure the abstracted devices. The configuration may take the form of constants, derived values, or Setup questions.

At first, this model would not seem to be very modular. However, if the goal is code reuse, the Framework model increases modularity. The developers of each platform do have to make decisions about the level of configurability. The complex problems that come from actually talking to the hardware are left up to the combination I/O driver stack whereas the sometimes equally complex tasks of interfacing to the user are left up to the platform experts. Modularity should follow expertise.

A Framework, Not a Straight Jacket

The Framework defines a set of primitive functions strongly overlapping those for EFI. It then defines a series of structures and protocols that make the job of the Framework possible. For example, the Framework defines abstractions for the storage of modules, which abstracts the Framework from details of where the modules are stored and allows the Framework elements to support the controlled dispatch of drivers and the like.

As with EFI, the Framework defines driver protocols for a number of common tasks. Not every platform performs all tasks, so the requirement is more along the lines of "If you are going to do this, use these interfaces." When writing interoperable modules, it is important for you to understand which interfaces are required and which are not.

Phasing

As noted previously, economics have driven designers to define the initial state of most systems as one that provides only the bare essentials to start execution. In an environment like this, particularly without the probability that system RAM will be initialized, an EFI-based infrastructure cannot function. The Framework sets out phases prior to those that are EFI-based. The major one is called Pre-EFI Initialization (PEI) for this reason. It is in fact subset of EFI that specifies a minimal RAM and minimal code size. Designed to be more focused than subsequent phases, the PEI's job is to get enough RAM to run the next, EFI-based phase, Driver Execution Environment (DXE). The names are different to help programmers to keep the two separate.

Phasing is also important in DXE. Many of the EFI primitives—Notify, TPL, and Coherency Domain are examples—rely entirely on RAM and are, as such, available when DXE starts. Many others do not, including stall, clocks, and security. When invoked, these services are not yet available. Drivers that use these services must wait until the drivers that provide the Architectural Protocols (APs) have been executed. The DXE becomes ready to support EFI rather than being ready to do so from the start.

Both PEI and DXE provide rich primitives to support modularity. Drivers from different organizations are expected to be grouped together to form a product. Without some mechanism to know which driver requires what protocols to be available, the system would be unworkable. A *dispatcher* provides the mechanism for ordering of execution of drivers. Although the word is the same and the actions are similar, the functionality here is different from that performed by an operating system's scheduler or dispatcher. The Framework defines *dependency expressions* that allow drivers or their PEI counterparts, PEI Modules, to describe their initial execution requirements. The dispatcher then determines the order in which the modules are executed.

Support the Transition from Old to New

After the introduction of PCI in around 1990, it took at least 10 years for the previously dominant ISA bus to disappear from most desktops. USB was introduced in the late 1990s and PS/2 keyboard and mice ports had not disappeared completely from most systems 8 years later. People aren't particularly surprised when it takes many years for an old bus to disappear in favor of a new one. Interfaces perform the same function for software as bus specifications do for hardware. As such, one might expect the transition from BIOS to EFI and Framework to take many years.

It would be naïve if the Framework didn't take steps to support the transition, supporting EFI, or BIOS via a Compatibility Support Module (CSM), or both. While the task is not trivial, it is not as daunting as it might first appear.

- Several major chunks of data are required both by EFI and BIOS, particularly ACPI and SMBIOS.
- EFI requires many of the hardware enumeration and initialization tasks that BIOS does.
- Many of the structures provided to the operating system by EFI have parallels in BIOS. A notable example is the memory usage tables in the Global Coherency Domain. The CSM lets the Framework develop the table and then translates it from the GCD format to the BIOS format (INT 15h, E820h).
- Lacking EFI option ROMs for a card, EFI uses the legacy option ROMs.

And so it goes. The preceding list is not meant to be comprehensive, only to give a taste of how the CSM goes about supporting using the Framework to support an alternate OS interface, BIOS.

To OS or Not to OS

Is EFI an OS? Isn't the Framework an OS? Technically, the answer to both questions is "No" since EFI and the Framework are simply definitions of interfaces rather than actual programs. That answer is a little unfair since most users view the OS through its interfaces. The answer gets murkier if the real question concerns the implementations of EFI and the Framework, and whether or not those implementations inevitably constitute operating systems.

At its most basic, an operating system is a program or set of programs that manage the resources of the system. This definition doesn't set the bar very high. Both EFI and the Framework define mechanisms for prioritization of execution (managing the processor), mechanisms for management of devices (device I/O protocols), and mechanisms for management of memory. As such, in the most general sense, one would have to answer "Yes, they sure look like operating systems."

If, however, we take the more common user-centered definition of operating system that involves paging, multi-user, task synchronization, even complex GUI interfaces, then the answer is clearly "No." In that sense, EFI and Framework are more nearly related to embedded operating environments.

What is actually more important here is that the design uses many concepts initially developed for operating systems. The Framework defines a dispatcher and manages its non-volatile storage via a file system, for example. It would be irresponsible not to use the 50-plus years of OS experience in designing any new software.

Not all of the experience used is from operating systems. The dependency expressions come from the study of languages and compilers. The Internal Forms Representation (IFR) in the Framework's Human Interface Infrastructure is based on learnings from the Web.

A Note on Ordering

The normal ordering for a book like this one is chronological, starting with the boot vector and ending up with the OS run-time calls. We have chosen instead to start with OS interfaces and follow with main firmware the the first phases. While at first glance, it would seem about as rational as presenting a compiler book from code generation back through to lexical analyzer, we have done so for good reasons.

The operating system (EFI) interfaces were defined first. The Framework interfaces were designed later, knowing that the goal was to boot EFI. Many of the same constructs appear in both and, in fact, the foundation of the major phases of the pre-boot phases use the same infrastructure as EFI. The initial phases simply cannot be guaranteed to have the resources required for the full framework, and so they use a subset. Understanding the Framework is thus essential for understanding the earliest phases.

As well, if you think back to the compiler analogy at the start of this section, then EFI is the equivalent of the language on which the compiler book's discussion of program code would depend.

Finally, a subset of our intended audience could be most interested in the EFI sections only. Those who are interested in the lower levels of the design need to understand the EFI sections thoroughly.

Chapter 2

Basic EFI Architecture

I believe in standards. Everyone should have one.

—George Morrow

The Extensible Firmware Interface (EFI) describes a programmatic interface to the platform. The platform includes the motherboard, chipset, central processing unit (CPU), and other components. EFI allows for pre-operating system (pre-OS) agents. Pre-OS agents are OS loaders, diagnostics, and other applications that the system needs for applications to execute and interoperate, including EFI drivers and applications. EFI represents a pure interface specification against which the drivers and applications interact, and this chapter highlights some of the architectural aspects of the interface. These architectural aspects include a set of objects and interfaces described by the EFI Specification.

The cornerstones for understanding EFI applications and drivers are several EFI concepts that are defined in the *EFI 1.1 Specification*. Assuming you are new to EFI, the following introduction explains a few of the key EFI concepts in a helpful framework to keep in mind as you study the specification:

- Objects managed by EFI-based firmware - used to manage system state, including I/O devices, memory, and events
- The EFI System Table - the primary data structure with data information tables and function calls to interface with the systems
- Handle database and protocols - the means by which callable interfaces are registered

- EFI images - the executable content format by which code is deployed
- Events - the means by which software can be signaled in response to some other activity
- Device paths - a data structure that describes the hardware location of an entity, such as the bus, spindle, partition, and file name of an EFI image on a formatted disk.

Objects Managed by EFI-based Firmware

Several different types of objects can be managed through the services provided by EFI. Some EFI drivers may need to access environment variables, but most do not. Rarely do EFI drivers require the use of a monotonic counter, watchdog timer, or real-time clock. The EFI System Table is the most important data structure, because it provides access to all EFI-provided the services and to all the additional data structures that describe the configuration of the platform.

EFI System Table

The EFI System Table is the most important data structure in EFI. A pointer to the EFI System Table is passed into each driver and application as part of its entry-point handoff. From this one data structure, an EFI executable image can gain access to system configuration information and a rich collection of EFI services that includes the following:

- EFI Boot Services
- EFI Runtime Services
- Protocol services

The EFI Boot Services and EFI Runtime Services are accessed through the EFI Boot Services Table and the EFI Runtime Services Table, respectively. Both of these tables are data fields in the EFI System Table. The number and type of services that each table makes available is fixed for each revision of the EFI specification. The EFI Boot Services and EFI Runtime Services are defined in the *EFI 1.10 Specification*. Chapter 4 of the *EFI 1.10 Specification* describes the common uses that EFI drivers make of these services.

Protocol services are groups of related functions and data fields that are named by a Globally Unique Identifier (GUID), a 16-bit, statistically-unique entity defined in Appendix A of the *EFI 1.10 Specification*. Typically, protocol services are used to provide software abstractions for devices such as consoles, disks, and networks, but they can be used to extend the number of generic services that are available in the platform. Protocols are the mechanism for extending the functionality of EFI firmware over time. The *EFI 1.10 Specification* defines over 30 different protocols, and various implementations of EFI firmware and EFI drivers may produce additional protocols to extend the functionality of a platform.

Handle Database

The *handle database* is composed of objects called handles and protocols. *Handles* are a collection of one or more protocols, and *protocols* are data structures that are named by a GUID. The data structure for a protocol may be empty, may contain data fields, may contain services, or may contain both services and data fields. During EFI initialization, the system firmware, EFI drivers, and EFI applications create handles and attach one or more protocols to the handles. Information in the handle database is global and can be accessed by any executable EFI image.

The handle database is the central repository for the objects that are maintained by EFI-based firmware. The handle database is a list of EFI handles, and each EFI handle is identified by a unique handle number that is maintained by the system firmware. A handle number provides a database "key" to an entry in the handle database. Each entry in the handle database is a collection of one or more protocols. The types of protocols, named by a GUID, that are attached to an EFI handle determine the handle type. An EFI handle may represent components such as the following:

- Executable images such as EFI drivers and EFI applications
- Devices such as network controllers and hard drive partitions
- EFI services such as EFI Decompression and the EBC Virtual Machine

Figure 2.1 below shows a portion of the handle database. In addition to the handles and protocols, a list of objects is associated with each protocol. This list is used to track which agents are consuming which protocols. This information is critical to the operation of EFI drivers, because this information is what allows EFI drivers to be safely loaded, started, stopped, and unloaded without any resource conflicts.

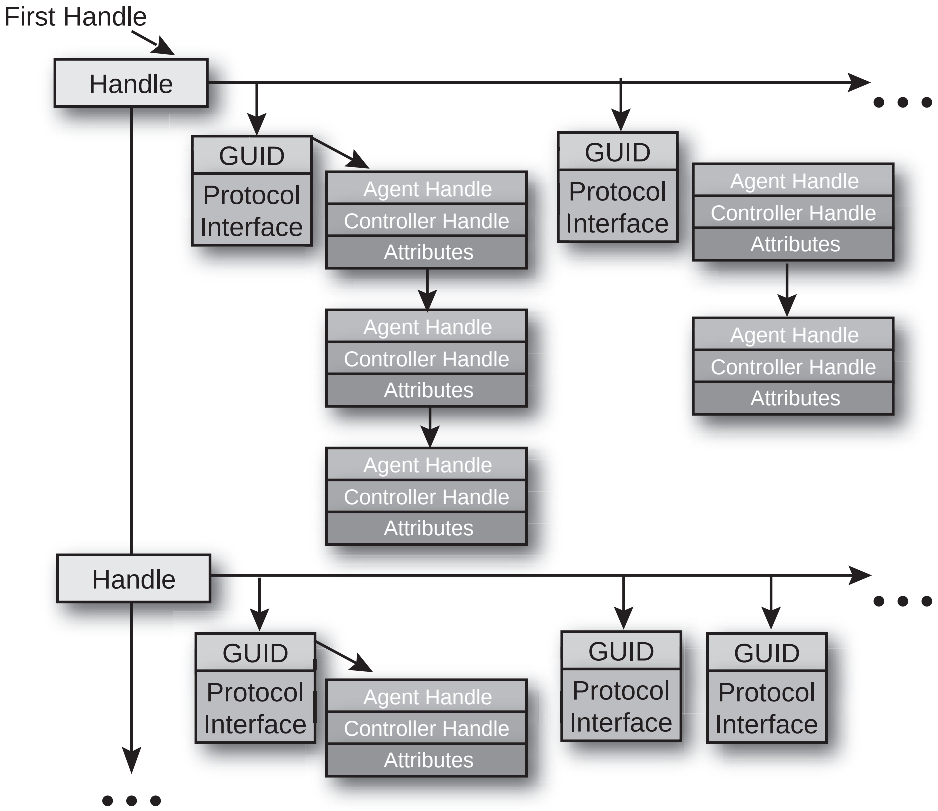


Figure 2.1 Handle Database

Figure 2.2 shows the different types of handles that can be present in the handle database and the relationships between the various handle types. All handles reside in the same handle database and the types of protocols that are associated with each handle differentiate the handle type. Like file system handles in an operating system context, the handles are unique for the session, but the values can be arbitrary. Also, like the handle returned from an `fopen` function in a C library, the value does not necessarily serve a useful purpose in a different process or during a subsequent restart in the same process. The handle is just a transitory value to manage state.

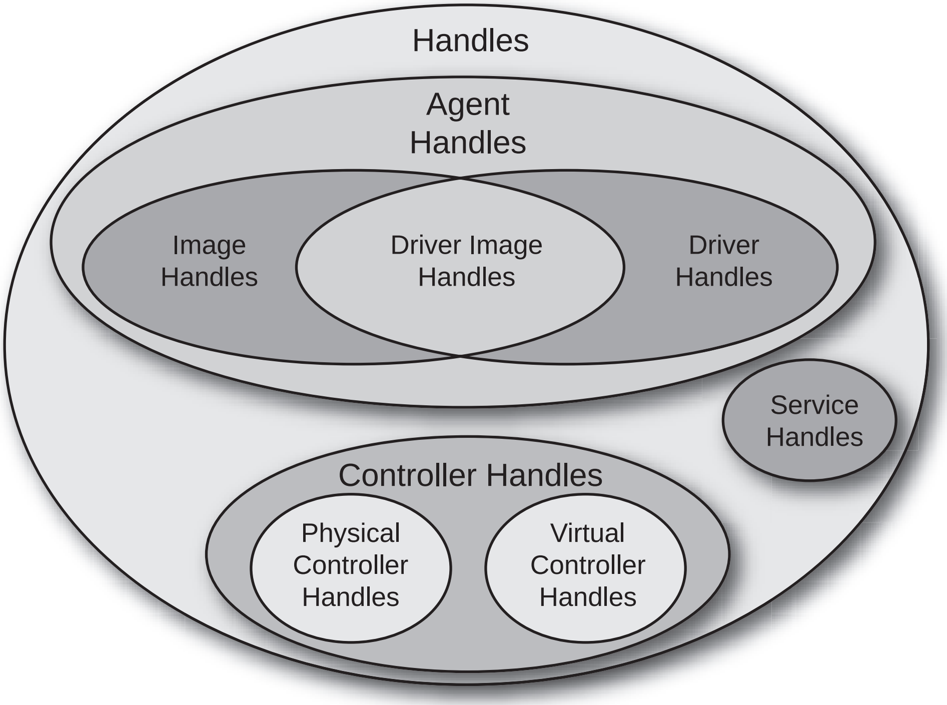


Figure 2.2 Handle Types

Protocols

The extensible nature of EFI is built, to a large degree, around protocols. EFI drivers are sometimes confused with EFI protocols. Although they are closely related, they are distinctly different. An *EFI driver* is an executable EFI image that installs a variety of protocols of various handles to accomplish its job.

An *EFI protocol* is a block of function pointers and data structures or APIs that have been defined by a specification. At a minimum, the specification must define a GUID. This number is the protocol's real name; boot services like LocateProtocol uses this number to find his protocol in the handle database. The protocol often includes a set of procedures and/or data structures, called the *protocol interface structure*. The following code sequence is an example of a protocol definition from section 9.6 of the *EFI 1.10 Specification*. Notice how it defines two function definitions and one data field.

Sample GUID

```
#define EFI_COMPONENT_NAME_PROTOCOL_GUID \
{0x107a772c,0xd5e1,0x11d4,0x9a,0x46,0x0,0x90,
 0x27,0x3f,0xc1,0x4d}
```

Protocol Interface Structure

```
typedef struct _EFI_COMPONENT_NAME_PROTOCOL {
 EFI_COMPONENT_NAME_GET_DRIVER_NAME
  GetDriverName;
 EFI_COMPONENT_NAME_GET_CONTROLLER_NAME
  GetControllerName;
 CHAR8
  *SupportedLanguages;
} EFI_COMPONENT_NAME_PROTOCOL;
```

Figure 2.3 shows a single handle and protocol from the handle database that is produced by an EFI driver. The protocol is composed of a GUID and a protocol interface structure. Many times, the EFI driver that produces a protocol interface maintains additional private data fields. The protocol interface structure itself simply contains pointers to the protocol function. The protocol functions are actually contained within the EFI driver. An EFI driver might produce one protocol or many protocols depending on the driver's complexity.

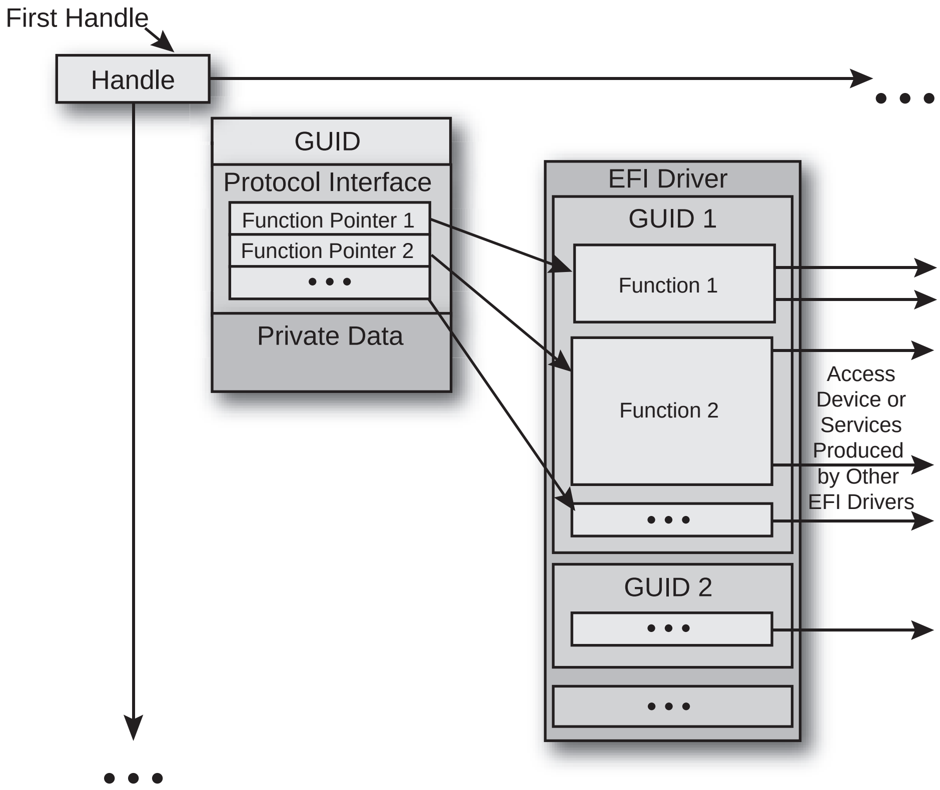


Figure 2.3 Construction of a Protocol

Not all protocols are defined in the *EFI 1.10 Specification*. The EFI Developer Kit (EDK) includes many protocols that are not part of the *EFI 1.10 Specification*. These protocols provide the wider range of functionality that might be needed in any particular implementation, but they are not defined in the *EFI 1.10 Specification* because they do not present an external interface that is required to support booting an OS or writing an EFI driver. The creation of new protocols is how EFI-based systems can be extended over time as new devices, buses, and technologies are introduced. For example, some protocols that are in the *EDK* but not in the *EFI 1.10 Specification* are*:*

- Varstore – interface to abstract storage of EFI persistent binary objects
- ConIn – service to provide a character console input

- ConOut – service to provide a character console output
- StdErr – service to provide a character console output for error messaging
- PrimaryConIn – the console input with primary view
- VgaMiniPort – a service that provides Video Graphics Array output
- UsbAtapi – a service to abstract block access on USB bus
- The EFI Application Toolkit also contains a number of EFI protocols that may be found on some platforms, such as,:
- PPP Deamon – Point-to-Point Protocol driver
- Ramdisk – file system instance on a Random Access Memory buffer
- TCP/IP – Transmission Control Protocol / Internet Protocol

The OS loader and drivers should not depend on these types of protocols because they are not guaranteed to be present in every EFI-compliant system. OS loaders and drivers should depend only on protocols that are defined in the *EFI 1.10 Specification* and protocols that are required by platform design guides such as *Design Implementation Guide for 64-bit Server*.

The extensible nature of EFI allows the developers of each platform to design and add special protocols. Using these protocols, they can expand the capabilities of EFI and provide access to proprietary devices and interfaces in congruity with the rest of the EFI architecture.

Because a protocol is "named" by a GUID, no other protocols should have that same identification number. Care must be taken when creating a new protocol to define a new GUID for it. EFI fundamentally assumes that a specific GUID exposes a specific protocol interface. Cutting and pasting an existing GUID or hand-modifying an existing GUID creates the opportunity for a duplicate GUID to be introduced. A system containing a duplicate GUID inadvertently could find the new protocol and think that it is another protocol, crashing the system as a result. For these types of bugs, finding the root cause is also very difficult. The GUID allows for naming APIs without having to worry about namespace collision. In systems such as PC/AT BIOS, services were added as an enumeration. For example, the venerable `Int15h` interface would pass the service type in `AX`. Since no central repository or specification managed the evolution of Int15h services, several vendors defined similar service numbers, thus making interoperability with operating systems and pre-OS applications difficult. Through the judicious use of GUIDs to name APIs and an association to develop the specification, EFI balances the need for API evolution with interoperability.

Working with Protocols

Any EFI code can operate with protocols during boot time. However, after `ExitBootServices()` is called, the handle database is no longer available. Several EFI boot time services work with EFI protocols.

Multiple Protocol Instances

A handle may have many protocols attached to it. However, it may have only one protocol of each type. In other words, a handle may not have more than one instance of the exact same protocol. Otherwise, it would make requests for a particular protocol on a handle nondeterministic.

However, drivers may create multiple instances of a particular protocol and attach each instance to a different handle. The PCI I/O Protocol fits this scenario, where the PCI bus driver installs a PCI I/O Protocol instance for each PCI device. Each instance of the PCI I/O Protocol is configured with data values that are unique to that PCI device, including the location and size of the EFI Option ROM (OpROM) image.

Also, each driver can install customized versions of the same protocol as long as they do not use the same handle. For example, each EFI driver installs the Component Name Protocol on its driver image handle, yet when the `EFI_COMPONENT_NAME_PROTOCOL.GetDriverName` function is called, each handle returns the unique name of the driver that owns that image handle. The `EFI_COMPONENT_NAME_PROTOCOL.GetDriverName()` function on the USB bus driver handle returns "USB bus driver" for the English language, but on the PXE driver handle it returns "PXE base code driver."

Tag GUID

A protocol may be nothing more than a GUID. In such cases, the GUID is called a *tag GUID*. Such protocols can serve useful purposes such as marking a device handle as special in some way or allowing other EFI images to easily find the device handle by querying the system for the device handles with that protocol GUID attached. The *EDK* uses the `HOT_PLUG_DEVICE_GUID` in this way to mark device handles that represent devices from a hot-plug bus such as USB.

EFI Images

All EFI images contain a PE/COFF header that defines the format of the executable code as required by the *Microsoft Portable Executable and Common Object File Format Specification* (Microsoft 1997). The target for this code can be an IA-32 processor, an Itanium® processor, or a processor agnostic, generic EFI Byte Code. The header defines the processor type and the image type. Presently there are three processor types and the following three image types defined:

- EFI applications –images that have their memory and state reclaimed upon exit.
- EFI Boot Service drivers –images that have their memory and state preserved throughout the pre-operating system flow. Their memory is reclaimed upon invocation of ExitBootServices() by the OS loader.
- EFI Runtime drivers –images whose memory and state persist throughout the evolution of the machine. These images coexist with and can be invoked by an EFI-aware operating system.

The value of the EFI Image format is that various parties can create binary executables that interoperate. For example, the operating system loader for Microsoft Windows† and Linux for an EFI-aware OS build is simple an EFI application. In addition, third parties can create EFI drivers to abstract their particular hardware, such as a networking interface host bus adapter (HBA) or other device. EFI images are loaded and relocated into memory with the Boot Service `gBS->LoadImage()`. Several supported storage locations for EFI images are available, including the following:

- Expansion ROMs on a PCI card
- System ROM or system flash
- A media device such as a hard disk, floppy, CD-ROM, or DVD
- A LAN boot server

In general, EFI images are not compiled and linked at a specific address. Instead, the EFI image contains relocation fix-ups so the EFI image can be placed anywhere in system memory. The Boot Service `gBS->LoadImage()` does the following:

- Allocates memory for the image being loaded
- Automatically applies the relocation fix-ups to the image

- Creates a new image handle in the handle database, which installs an instance of the `EFI_LOADED_IMAGE_PROTOCOL`

This instance of the `EFI_LOADED_IMAGE_PROTOCOL` contains information about the EFI image that was loaded. Because this information is published in the handle database, it is available to all EFI components.

After an EFI image is loaded with `gBS->LoadImage()`, it can be started with a call to `gBS->StartImage`. The header for an EFI image contains the address of the entry point that is called by `gBS->StartImage()`. The entry point always receives the following two parameters:

- The image handle of the EFI image being started
- A pointer to the EFI System Table

These two items allow the EFI image to do the following:

- Access all of the EFI services that are available in the platform.
- Retrieve information about where the EFI image was loaded from and where in memory the image was placed.

The operations that the EFI image performs in its entry point vary depending on the type of EFI image. Figure 2.4 shows the various EFI image types and the relationships between the different levels of images.

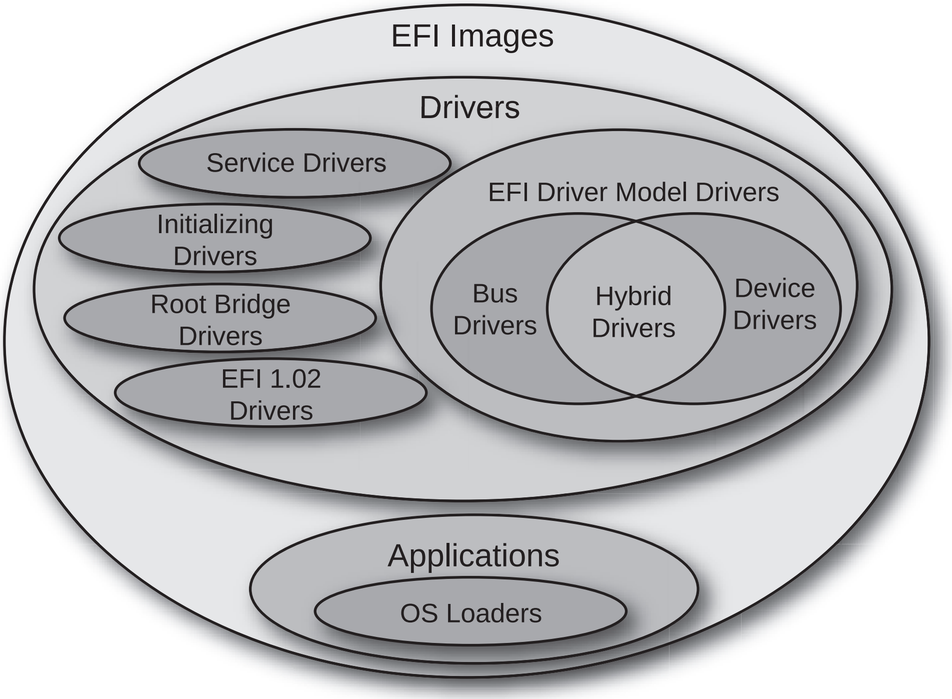


Figure 2.4 Image Types and Their Relationship to One Another

Table 2.1 Description of Image Types

Type of Image	Description
Application	An EFI image of type `EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION`. This image is executed and automatically unloaded when the image exits or returns from its entry point.
OS loader	A special type of application that normally does not return or exit. Instead, it calls the EFI Boot Service `gBS->ExitBootServices()` to transfer control of the platform from the firmware to an operating system.
Driver	An EFI image of type `EFI_IMAGE_SUBSYSTEM_BOOT_SERVICE_DRIVER` or `EFI_IMAGE_SUBSYSTEM_RUNTIME_DRIVER`. If this image returns `EFI_SUCCESS`, then the image is not unloaded. If the image returns an error code other than `EFI_SUCCESS`, then the image is automatically unloaded from system memory. The ability to stay resident in system memory is what differentiates a driver from an application. Because drivers can stay resident in memory, they can provide services to other drivers, applications, or an operating system. Only the services produced by runtime drivers are allowed to persist past `gBS->ExitBootServices()`.
Service driver	A driver that produces one or more protocols on one or more new service handles and returns `EFI_SUCESS` from its entry point.
Initializing driver	A driver that does not create any handles and does not add any protocols to the handle database. Instead, this type of driver performs some initialization operations and returns an error code so the driver is unloaded from system memory.
Root bridge driver	A driver that creates one or physical controller handles that contain a Device Path Protocol and a protocol that is a software abstraction for the I/O services provided by a root bus produced by a core chipset. The most common root bridge driver is one that creates handles for the PCI root bridges in the platform that support the Device Path Protocol and the PCI Root Bridge I/O Protocol.
EFI 1.02 driver	A driver that follows the *EFI 1.02 Specification*. This type of driver does not use the EFI Driver Model. These types of drivers are not discussed in detail in this document. Instead, this document presents recommendations on converting EFI 1.02 drivers to drivers that follow the EFI Driver Model.

Table 2.1 Description of Image Types (continued)

Type of Image	Description
EFI Driver Model driver	A driver that follows the EFI Driver Model that is described in detail in the *EFI 1.10 Specification*. This type of driver is fundamentally different from service drivers, initializing drivers, root bridge drivers, and EFI 1.02 drivers because a driver that follows the EFI Driver Model is not allowed to touch hardware or produce device-related services in the driver entry point. Instead, the driver entry point of a driver that follows the EFI Driver Model is allowed only to register a set of services that allow the driver to be started and stopped at a later point in the system initialization process.
Device driver	A driver that follows the EFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol into the handle database. This type of driver does not create any child handles when the `Start()` service of the Driver Binding Protocol is called. Instead, it only adds additional I/O protocols to existing controller handles.
Bus driver	A driver that follows the EFI Driver Model. This type of driver produces one or more driver handles or driver image handles by installing one or more instances of the Driver Binding Protocol in the handle database. This type of driver creates new child handles when the `Start()` service of the Driver Binding Protocol is called. It also adds I/O protocols to these newly created child handles.
Hybrid driver	A driver that follows the EFI Driver Model and shares characteristics with both device drivers and bus drivers. This distinction means that the `Start()` service of the Driver Binding Protocol will add I/O protocols to existing handles and it will create child handles.

Applications

An EFI application starts execution at its entry point, then continues execution until it reaches a return from its entry point or it calls the `Exit()` boot service function. When done, the image is unloaded from memory. Some examples of common EFI applications include the EFI shell, EFI shell commands, flash utilities, and diagnostic utilities. It is perfectly acceptable to invoke EFI applications from inside other EFI applications.

OS Loader

A special type of EFI application, called an *OS boot loader*, calls the `ExitBootServices()` function when the OS loader has set up enough of the OS infrastructure to be ready to assume ownership of the system resources. At `ExitBootServices()`, the EFI core frees all of its boot time services and drivers, leaving only the run-time services and drivers.

Drivers

EFI drivers differ from EFI applications in that the driver stays resident in memory unless an error is returned from the driver's entry point. The EFI core firmware, the boot manager, or other EFI applications may load drivers.

EFI 1.02 Drivers

Several types of EFI drivers exist, having evolved with subsequent levels of the specification. In EFI 1.02, drivers were constructed without a defined driver model. The *EFI 1.10 Specification* provides a driver model that replaces the way drivers were built in EFI 1.02 but that still maintains backward compatibility with EFI 1.02 drivers. EFI 1.02 immediately started the driver inside the entry point. Following this method meant that the driver searched immediately for supported devices, installed the necessary I/O protocols, and started the timers that were needed to poll the devices. However, this method did not give the system control over the driver loading and connection policies, so the EFI Driver Model was introduced in section 1.6 of the *EFI 1.10 Specification* to resolve these issues.

The Floating-Point Software Assist (FPSWA) driver is a common example of an EFI 1.02 driver; other EFI 1.02 drivers can be found in the EFI Application Toolkit 1.02.12.38. For compatibility, EFI 1.02 drivers can be converted to EFI 1.10 drivers that follow the EFI Driver Model.

Boot Service and Runtime Drivers

Boot-time drivers are loaded into area of memory that are marked as `EfiBootServicesCode`, and the drivers allocate their data structures from memory marked as `EfiBootServicesData`. These memory types are converted to available memory after `gBS->ExitBootServices()` is called.

Runtime drivers are loaded in memory marked as `EfiRuntimeServicesCode`, and they allocate their data structures from memory marked as `EfiRuntimeServicesData`. These types of memory are preserved after `gBS->ExitBootServices()` is called, thereby enabling the runtime driver to provide services to an operating system while the operating system is running. Runtime drivers must publish an alternative calling mechanism, because the EFI handle database does not persist into OS runtime. The most common examples of EFI runtime drivers are the Floating-Point Software Assist driver (`FPSWA.efi`) and the network Universal Network Driver Interface (UNDI) driver. Other than these examples, runtime drivers are not very common. In addition, the implementation and validation of runtime drivers is much more difficult than boot service drivers because EFI supports the translation of runtime services and runtime drivers from a physical addressing mode to a virtual addressing mode. With this translation, the operating system can make virtual calls to the runtime code. The OS typically runs in virtual mode, so it must transition into physical mode to make the call. Transitions into physical mode for modern, multiprocessor operating systems are expensive because they entail flushing translation look-up blocks (TLB), rendezvousing coordinating all CPUs, and other tasks. As such, EFI runtime offers an efficient invocation mechanism because no transition is required.

Events and Task Priority Levels

Events are another type of object that is managed through EFI services. An event can be created and destroyed, and an event can be either in the waiting state or the signaled state. An EFI image can do any of the following:

- Create an event.
- Destroy an event.
- Check to see if an event is in the signaled state.
- Wait for an event to be in the signaled state.
- Request that an event be moved from the waiting state to the signaled state.

Because EFI does not support interrupts, it can present a challenge to driver writers who are accustomed to an interrupt-driven driver model. Instead, EFI supports polled drivers. The most common use of events by an EFI driver is the use of timer events that allow drivers to periodically poll a device. Figure 2.5 shows the different types of events that are supported in EFI and the relationships between those events.

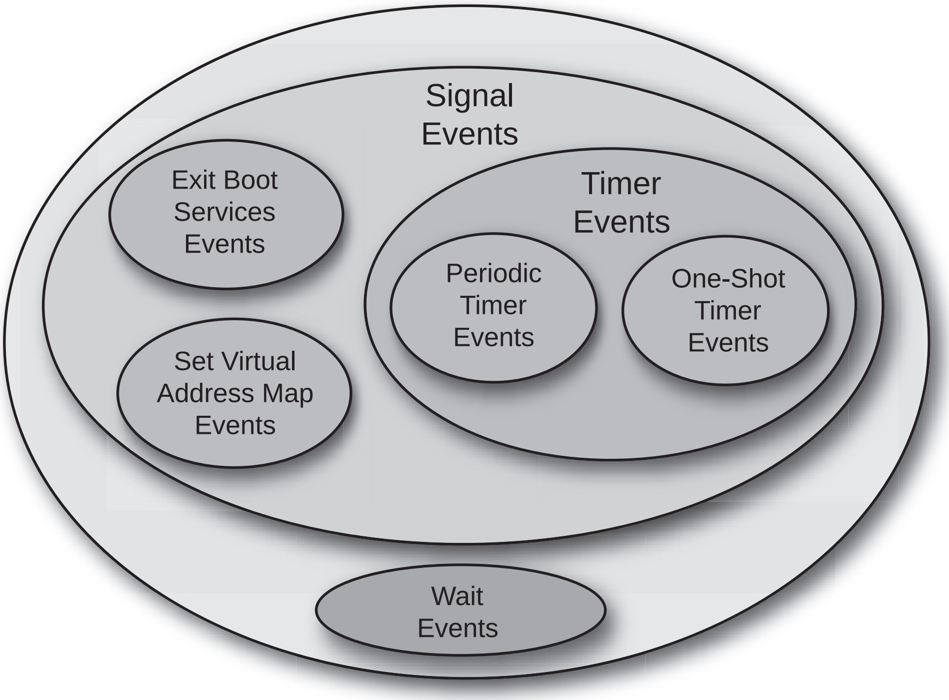


Figure 2.5 Event Types and Relationships

Table 2.2 Description of Event Types

Type of Events	Description
Wait event	An event whose notification function is executed whenever the event is checked or waited upon.
Signal event	An event whose notification function is scheduled for execution whenever the event goes from the waiting state to the signaled state.
Exit Boot Services event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Boot Service ExitBootServices() is called. This call is the point in time when ownership of the platform is transferred from the firmware to an operating system. The event's notification function is scheduled for execution when ExitBootServices() is called.
Set Virtual Address Map event	A special type of signal event that is moved from the waiting state to the signaled state when the EFI Runtime Service SetVirtualAddressMap() is called. This call is the point in time when the operating system is making a request for the runtime components of EFI to be converted from a physical addressing mode to a virtual addressing mode. The operating system provides the map of virtual addresses to use. The event's notification function is scheduled for execution when SetVirtualAddressMap() is called.
Timer event	A type of signal event that is moved from the waiting state to the signaled state when at least a specified amount of time has elapsed. Both periodic and one-shot timers are supported. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
Periodic timer event	A type of timer event that is moved from the waiting state to the signaled state at a specified frequency. The event's notification function is scheduled for execution when a specific amount of time has elapsed.
One-shot timer event	A type of timer event that is moved from the waiting state to the signaled state after the specified timer period has elapsed. The event's notification function is scheduled for execution when a specific amount of time has elapsed.

The following three elements are associated with every event:

- The Task Priority Level (TPL) of the event
- A notification function
- A notification context

The notification function for a wait event is executed when the state of the event is checked or when the event is being waited upon. The notification function of a signal event is executed whenever the event transitions from the waiting state to the signaled state. The notification context is passed into the notification function each time the notification function is executed. The TPL is the priority at which the notification function is executed. Table 2.3 lists the four TPL levels that are defined today. Additional TPL's could be added later. An example of a compatible addition to the TPL list could include a series of "Interrupt TPL's" between TPL_NOTIFY and TPL_HIGH_LEVEL in order to provide interrupt-driven I/O support within EFI..

Table 2.3 Task Priority Levels Defined in EFI

Task Priority Level	Description
TPL_APPLICATION	The priority level at which EFI images are executed.
TPL_CALLBACK	The priority level for most notification functions.
TPL_NOTIFY	The priority level at which most I/O operations are performed.
TPL_HIGH_LEVEL	The priority level for the one timer interrupt supported in EFI.

TPLs serve the following two purposes:

- To define the priority in which notification functions are executed
- To create locks

For priority definition, you use this mechanism only when more than one event is in the signaled state at the same time. In these cases, the application executes the notification function that has been registered with the higher priority first. Also, notification functions at higher priorities can interrupt the execution of notification functions executing at a lower priority.

For creating locks, code running in normal context and code in an interrupt context can access the same data structure because EFI does support a single-timer interrupt. This access can cause problems and unexpected results if the updates to a shared data structure are not atomic. An EFI application or EFI driver that wants to guarantee exclusive access to a shared data structure can temporarily raise the task priority level to prevent simultaneous access from both normal context and interrupt context. The application can create a lock by temporarily raising the task priority level to `TPL_HIGH_LEVEL`. This level blocks even the one-timer interrupt, but you must take care to minimize the amount of time that the system is at `TPL_HIGH_LEVEL`. Since all timer-based events are blocked during this time, any driver that requires periodic access to a device is prevented from accessing its device. A TPL is similar to the IRQL in Microsoft Windows and the SPL in various Unix implementations. A TPL describes a prioritization scheme for access control to resources.

Chapter 3

EFI Driver Model

Things should be made as simple as possible—but no simpler.

—Albert Einstein

The Extensible Firmware Interface (EFI) provides a driver model for support of devices that attach to today's industry-standard buses, such as PCI and USB, and architectures of tomorrow. The EFI Driver Model is intended to simplify the design and implementation of device drivers and produce small executable image sizes. As a result, some complexity has been moved into bus drivers and to a greater extent into common firmware services. A device driver needs to produce a Driver Binding Protocol on the same image handle on which the driver was loaded. It then waits for the system firmware to connect the driver to a controller. When that occurs, the device driver is responsible for producing a protocol on the controller's device handle that abstracts the I/O operations that the controller supports. A bus driver performs these exact same tasks. In addition, a bus driver is also responsible for discovering any child controllers on the bus, and creating a device handle for each child controller found.

The combination of firmware services, bus drivers, and device drivers in any given platform is likely to be produced by a wide variety of vendors including OEMs, IBVs, and IHVs. These different components from different vendors are required to work together to produce a protocol for an I/O device than can be used to boot an EFI compliant operating system. As a result, the EFI Driver Model is described in great detail in order to increase the interoperability of these components.

This chapter gives a brief overview of the EFI Driver Model. It describes the entry point of a driver, host bus controllers, properties of device drivers, properties of bus drivers, and how the EFI Driver Model can accommodate hot plug events.

Why a Driver Model Prior to OS Booting?

Under the EFI Driver Model, only the minimum number of I/O devices needs to be activated. For example, with today's BIOS-based systems, a server with dozens of SCSI drives needs to have those drives "spun-up" or activated. This is because the BIOS Int19h code does not know a priori which device will contain the operating system loader. The EFI Driver Model allows for only activating the subset of devices that are necessary for booting. This makes a rapid system restart possible and pushes the policy of which additional devices need activation up into the operating system. With the more aggressive boot time requirements more along the lines of consumer electronics devices being pushed to all open platforms, this capability is imperative.

Driver Initialization

The file for a driver image must be loaded from some type of media. This could include ROM, flash, hard drives, floppy drives, CD-ROM, or even a network connection. Once a driver image has been found, it can be loaded into system memory with the Boot Service `LoadImage()`. `LoadImage()` loads a PE/COFF formatted image into system memory. A handle is created for the driver, and a Loaded Image Protocol instance is placed on that handle. A handle that contains a Loaded Image Protocol instance is called an *Image Handle*. At this point, the driver has not been started. It is just sitting in memory waiting to be started. Figure 3.1 shows the state of an image handle for a driver after `LoadImage()` has been called.

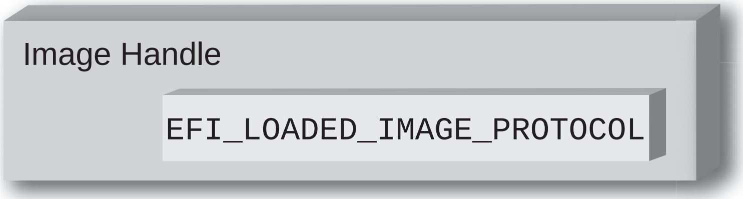


Figure 3.1 Image Handle

After a driver has been loaded with the Boot Service `LoadImage()`, it must be started with the Boot Service `StartImage()`. This is true of all types of EFI applications and EFI drivers that can be loaded and started on an EFI compliant system. The entry point for a driver that follows the EFI Driver Model must follow some strict rules. First, it is not allowed to touch any hardware. Instead, it is only allowed to install protocol instances onto its own Image Handle. A driver that follows the EFI Driver Model is *required* to install an instance of the Driver Binding Protocol onto its own Image Handle. It may optionally install the Driver Configuration Protocol, the Driver Diagnostics Protocol, or the Component Name Protocol. In addition, if a driver wishes to be unloadable it may optionally update the Loaded Image Protocol to provide its own `Unload()` function. Finally, if a driver needs to perform any special operations when the Boot Service `ExitBootServices()` is called, it may optionally create an event with a notification function that is triggered when the Boot Service `ExitBootServices()` is called. An Image Handle that contains a Driver Binding Protocol instance is known as a *Driver Image Handle*. Figure 3.2 shows a possible configuration for the Image Handle from Figure 3.1 after the Boot Service `StartImage()` has been called.

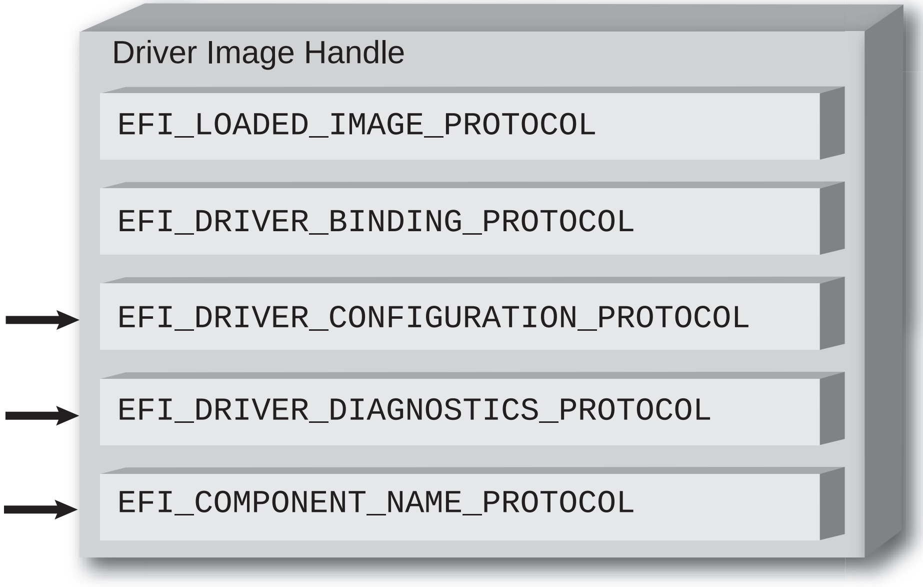

Driver Image Handle
EFI_LOADED_IMAGE_PROTOCOL
EFI_DRIVER_BINDING_PROTOCOL
EFI_DRIVER_CONFIGURATION_PROTOCOL
EFI_DRIVER_DIAGNOSTICS_PROTOCOL
EFI_COMPONENT_NAME_PROTOCOL

Figure 3.2 Driver Image Handle

Host Bus Controllers

Drivers are not allowed to touch any hardware in the driver's entry point. As a result, drivers are loaded and started, but they are all waiting to be told to manage one or more controllers in the system. A platform component, like the EFI Boot Manager, is responsible for managing the connection of drivers to controllers. However, before even the first connection can be made, some initial collection of controllers must be present for the drivers to manage. This initial collection of controllers is known as the *Host Bus Controllers*. The I/O abstractions that the Host Bus Controllers provide are produced by firmware components that are outside the scope of the EFI Driver Model. The device handles for the Host Bus Controllers and the I/O abstraction for each one must be produced by the core firmware on the platform, or an EFI Driver that may not follow the EFI Driver Model. See the *PCI Host Bridge I/O Protocol Specification* for an example of an I/O abstraction for PCI buses.

A platform can be viewed as a set of CPUs and a set of core chip set components that may produce one or more host buses. Figure 3.3 shows a platform with *n* CPUs, and a set of core chipset components that produce *m* host bridges.

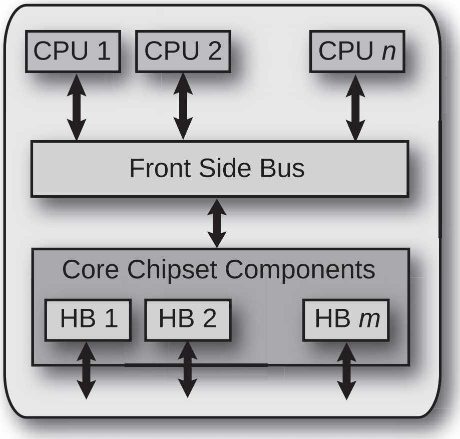


Figure 3.3 Host Bus Controllers

Each host bridge is represented in EFI as a device handle that contains a Device Path Protocol instance, and a protocol instance that abstracts the I/O operations that the host bus can perform. For example, a PCI Host Bus Controller supports the PCI Host Bridge I/O Protocol. Figure 3.4 shows an example device handle for a PCI Host Bridge.

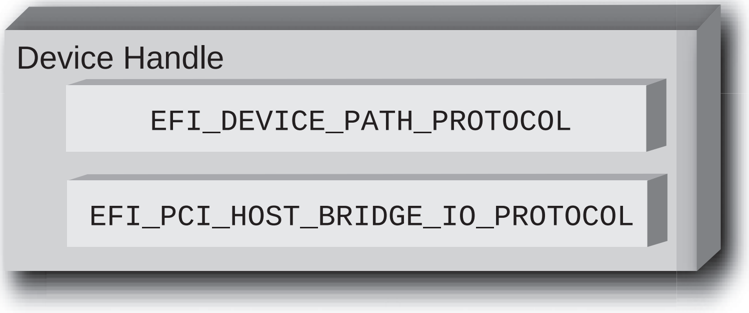


Figure 3.4 Host Bus Device Handle

A PCI Bus Driver could connect to this PCI Host Bridge, and create child handles for each of the PCI devices in the system. PCI Device Drivers should then be connected to these child handles, and produce I/O abstractions that may be used to boot an EFI compliant OS. The following section describes the different types of drivers that can be implemented within the EFI Driver Model. The EFI Driver Model is very flexible, so all the possible types of drivers are not discussed here. Instead, the major types are covered that can be used as a starting point for designing and implementing additional driver types.

Device Drivers

A device driver is not allowed to create any new device handles. Instead, it installs additional protocol interfaces on an existing device handle. The most common type of device driver attaches an I/O abstraction to a device handle that has been created by a bus driver. This I/O abstraction may be used to boot an EFI compliant OS. Some example I/O abstractions would include Simple Text Output, Simple Input, Block I/O, and Simple Network Protocol. Figure 3.5 shows a device handle before and after a device driver is connected to it. In this example, the device handle is a child of the XYZ Bus, so it contains an XYZ I/O Protocol for the I/O services that the XYZ bus supports. It also contains a Device Path Protocol that was placed there by the XYZ Bus Driver. The Device Path Protocol is not required for all device handles. It is only required for device handles that represent physical devices in the system. Handles for virtual devices do not contain a Device Path Protocol.

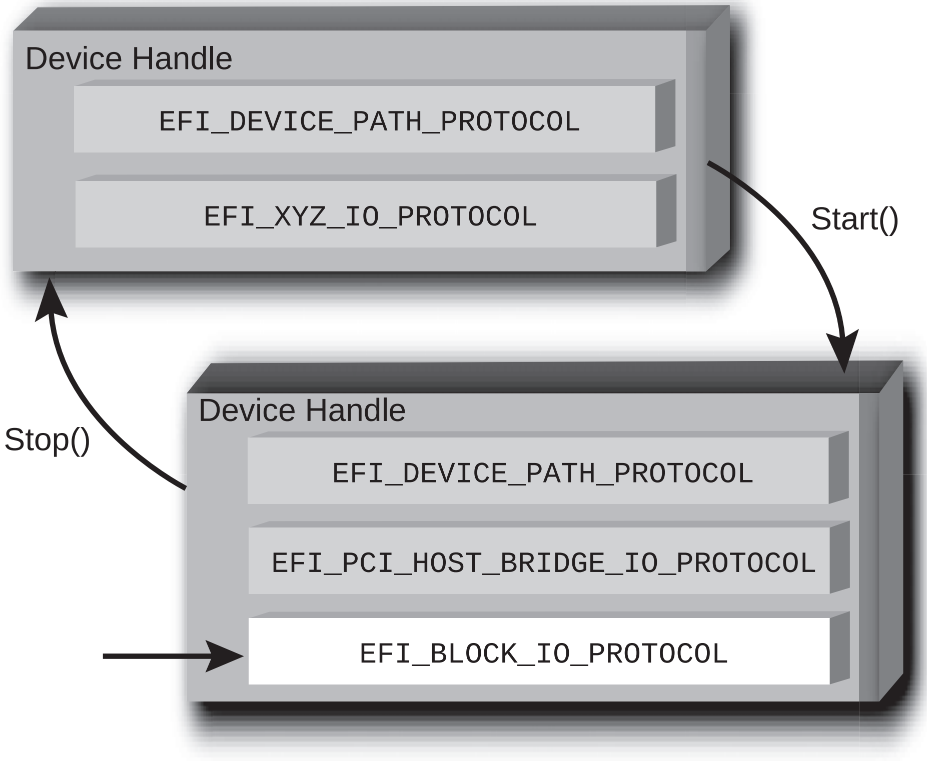


Figure 3.5 Connecting Device Drivers

The device driver that connects to the device handle in Figure 3.5 must have installed a Driver Binding Protocol on its own image handle. The Driver Binding Protocol contains three functions called `Supported()`, `Start()`, and `Stop()`. The `Supported()` function tests to see if the driver supports a given controller. In this example, the driver will check to see if the device handle supports the Device Path Protocol and the XYZ I/O Protocol. If a driver's `Supported()` function passes, then the driver can be connected to the controller by calling the driver's `Start()` function. The `Start()` function is what actually adds the additional I/O protocols to a device handle. In this example, the Block I/O

Protocol is being installed. To provide symmetry, the Driver Binding Protocol also has a Stop() function that forces the driver to stop managing a device handle. This causes the device driver to uninstall any protocol interfaces that were installed in Start().

The Support(), Start(), and Stop() functions of the EFI Driver Binding Protocol are required to make use of the new Boot Service OpenProtocol() to get a protocol interface and the new Boot Service CloseProtocol() to release a protocol interface. OpenProtocol() and CloseProtocol() update the handle database maintained by the system firmware to track which drivers are consuming protocol interfaces. The information in the handle database can be used to retrieve information about both drivers and controllers. The new Boot Service OpenProtocolInformation() can be used to get the list of components that are currently consuming a specific protocol interface.

Bus Drivers

Bus drivers and device drivers are virtually identical from the EFI Driver Model's point of view. The only difference is that a bus driver creates new device handles for the child controllers that the bus driver discovers on its bus. As a result, bus drivers are slightly more complex than device drivers, but this in turn simplifies the design and implementation of device drivers. There are two major types of bus drivers. The first creates handles for all the child controllers on the first call to Start(). The second type allows the handles for the child controllers to be created across multiple calls to Start(). This second type of bus driver is very useful in supporting a rapid boot capability. It allows a few child handles or even one child handle to be created. On buses that take a long time to enumerate all of their children (such as SCSI), this can lead to a very large time savings in booting a platform. Figure 3.6 shows the tree structure of a bus controller before and after Start() is called. The dashed line coming into the bus controller node represents a link to the bus controller's parent controller. If the bus controller is a Host Bus Controller, then it does not have a parent controller. Nodes A,B,C,D, and E represent the child controllers of the bus controller.

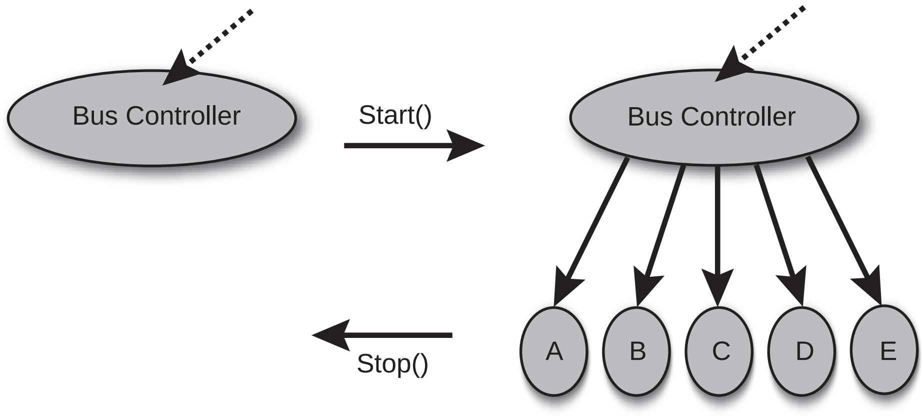


Figure 3.6 Connecting Bus Drivers

A bus driver that supports creating one child on each call to `Start()` might choose to create child C first, and then child E, and then the remaining children A,B, and D. The `Supported()`, `Start()`, and `Stop()` functions of the Driver Binding Protocol are flexible enough to allow this type of behavior.

A bus driver must install protocol interfaces onto every child handle that is creates. At a minimum, it must install a protocol interface that provides an I/O abstraction of the bus's services to the child controllers. If the bus driver creates a child handle that represents a physical device, then the bus driver must also install a Device Path Protocol instance onto the child handle. A bus driver may optionally install a Bus Specific Driver Override Protocol onto each child handle. This protocol is used when drivers are connected to the child controllers. A new Boot Service `ConnectController()` uses architecturally defined precedence rules to choose the best set of drivers for a given controller. The Bus Specific Driver Override Protocol has higher precedence than a general driver search algorithm, and lower precedence than platform overrides. An example of a bus specific driver selection occurs with PCI. A PCI Bus Driver gives a driver stored in a PCI controller's option ROM a higher precedence than drivers stored elsewhere in the platform. Figure 3.7 shows an example child device handle that has been created by the XYZ Bus Driver that supports a bus specific driver override mechanism.

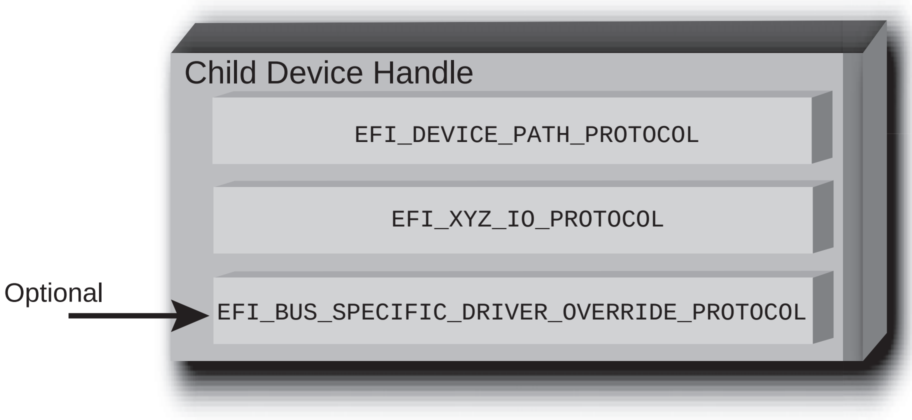


Figure 3.7 Child Device Handle with a Bus Specific Override

Platform Components

Under the EFI Driver Model, the act of connecting and disconnecting drivers from controllers in a platform is under the platform firmware's control. This will typically be implemented as part of the EFI Boot Manager, but other implementations are possible. The new Boot Services `ConnectController()` and `DisconnectController()` can be used by the platform firmware to determine which controllers get started and which ones do not. If the platform wishes to perform system diagnostics or install an operating system, then it may choose to connect drivers to all possible boot devices. If a platform wishes to boot a pre-installed operating system, it may choose to only connect drivers to the devices that are required to boot the selected operating system. The EFI Driver Model supports both of these modes of operation through the new Boot Services `ConnectController()` and `DisconnectController()`. In addition, since the platform component that is in charge of booting the platform has to work with device paths for console devices and boot options, all of the services and protocols involved in the EFI Driver Model are optimized with device paths in mind.

The platform may also choose to produce an optional protocol named the Platform Driver Override Protocol. This is similar to the Bus

Specific Driver Override Protocol, but it has higher priority. This gives the platform firmware the highest priority when deciding which drivers are connected to which controllers. The Platform Driver Override Protocol is attached to a handle in the system. The new Boot Service `ConnectController()` will make use of this protocol if it is present in the system.

Hot Plug Events

In the past, system firmware has not had to deal with hot plug events in the pre-boot environment. However, with the advent of buses like USB, where the end user can add and remove devices at any time, it is important to make sure that it is possible to describe these types of buses in the EFI Driver Model. It is up to the bus driver of a bus that supports the hot adding and removing of devices to provide support for such events. For these types of buses, some of the platform management is going to have to move into the bus drivers. For example, when a keyboard is added hot to a USB bus on a platform, the end user would expect the keyboard to be active. A USB Bus driver could detect the hot add event and create a child handle for the keyboard device. However, since drivers are not connected to controllers unless `ConnectController()` is called the keyboard would not become an active input device. Making the keyboard driver active requires the USB Bus driver to call `ConnectController()` when a hot add event occurs. In addition, the USB Bus Driver would have to call `DisconnectController()` when a hot remove event occurs.

Device drivers are also affected by these hot plug events. In the case of USB, a device can be removed without any notice. This means that the `Stop()` functions of USB device drivers must deal with shutting down a driver for a device that is no longer present in the system. As a result, any outstanding I/O requests must be flushed without actually being able to touch the device hardware.

In general, adding support for hot plug events greatly increases the complexity of both bus drivers and device drivers. Adding this support is up to the driver writer, so the extra complexity and size of the driver must be weighed against the need for the feature in the pre-boot environment.

The two example code sequences below provide guidance on how a device driver writer might discover if it in fact manages the candidate hardware device. These mechanisms include looking at the controller handle in the first example and examining the device path in the second example.

```
extern EFI_GUID              gEfiDriverBindingProtocolGuid;
EFI_HANDLE                   gMyImageHandle;
EFI_HANDLE                   DriverImageHandle;
EFI_HANDLE                   ControllerHandle;
EFI_DRIVER_BINDING_PROTOCOL  *DriverBinding;
EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath;

//
// Use the DriverImageHandle to get the Driver Binding Protocol
// instance
//
Status = gBS->OpenProtocol (
                 DriverImageHandle,
                 &gEfiDriverBindingProtocolGuid,
                 &DriverBinding,
                 gMyImageHandle,
                 NULL,
                 EFI_OPEN_PROTOCOL_HANDLE_PROTOCOL
                 );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// EXAMPLE #1
//
// Use the Driver Binding Protocol instance to test to see if
// the driver specified by DriverImageHandle supports the
// controller specified by ControllerHandle
//
Status = DriverBinding->Supported (
                         DriverBinding,
                         ControllerHandle,
                         NULL
                         );
if (!EFI_ERROR (Status)) {
  Status = DriverBinding->Start (
                           DriverBinding,
                           ControllerHandle,
                           NULL
                           );
}

return Status;

//
// EXAMPLE #2
```

```
//
// The RemainingDevicePath parameter can be used to initialize
// only the minimum devices required to boot. For example,
// maybe we only want to initialize 1 hard disk on a SCSI
// channel. If DriverImageHandle is a SCSI Bus Driver, and
// ControllerHandle is a SCSI Controller, and we only want to
// create a child handle for PUN=3 and LUN=0, then the
// RemainingDevicePath would be SCSI(3,0)/END. The following
// example would return EFI_SUCCESS if the SCSI driver supports
// creating the child handle for PUN=3, LUN=0. Otherwise it
// would return an error.
//
Status = DriverBinding->Supported (
                          DriverBinding,
                          ControllerHandle,
                          RemainingDevicePath
                          );
if (!EFI_ERROR (Status)) {
  Status = DriverBinding->Start (
                            DriverBinding,
                            ControllerHandle,
                            RemainingDevicePath
                            );
}

return Status;
```

Pseudo Code

The algorithms for the Start() function for three different types of drivers are presented here. How the Start() function of a driver is implemented can affect how the Supportred() function is implemented. All of the services in the EFI_DRIVER_BINDING_PROTOCOL need to work together to make sure that all resources opened or allocated in Supported() and Start() are released in Stop().

The first algorithm is a simple device driver that does not create any additional handles. It only attaches one or more protocols to an existing handle. The second is a simple bus driver that always creates all of its child handles on the first call to Start(). It does not attach any additional protocols to the handle for the bus controller. The third is a more advanced bus driver that can either create one child handles at a time on successive calls to Start(), or it can create all of its child handles or all of the remaining child handles in a single call to Start(). Once again, it does not attach any additional protocols to the handle for the bus controller.

Device Driver

1. Open all required protocols with `OpenProtocol()`. If this driver allows the opened protocols to be shared with other drivers, then it should use an *`Attribute`* of `EFI_OPEN_PROTOCOL_BY_DRIVER`. If this driver does not allow the opened protocols to be shared with other drivers, then it should use an *`Attribute`* of `EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE`. It must use the same *`Attribute`* value that was used in `Supported()`.
2. If any of the calls to `OpenProtocol()` in Step 1 returned an error, then close all of the protocols opened in Step 1 with `CloseProtocol()`, and return the status code from the call to `OpenProtocol()` that returned an error.
3. Ignore the parameter *`RemainingDevicePath`*.
4. Initialize the device specified by *`ControllerHandle`*. If an error occurs, close all of the protocols opened in Step 1 with `CloseProtocol()`, and return `EFI_DEVICE_ERROR`.
5. Allocate and initialize all of the data structures that this driver requires to manage the device specified by *`ControllerHandle`*. This would include space for public protocols and space for any additional private data structures that are related to *`ControllerHandle`*. If an error occurs allocating the resources, then close all of the protocols opened in Step 1 with `CloseProtocol()`, and return `EFI_OUT_OF_RESOURCES`.
6. Install all the new protocol interfaces onto *`ControllerHandle`* using `InstallProtocolInterface()`. If an error occurs, close all of the protocols opened in Step 1 with `CloseProtocol()`, and return the error from `InstallProtocolInterface()`.
7. Return `EFI_SUCCESS`.

Bus Driver that Creates All of Its Child Handles on the First Call to Start()

1. Open all required protocols with `OpenProtocol()`. If this driver allows the opened protocols to be shared with other drivers, then it should use an *`Attribute`* of `EFI_OPEN_PROTOCOL_BY_DRIVER`. If this driver does not allow the opened protocols to be shared with other drivers, then it should use an *`Attribute`* of `EFI_OPEN_PROTOCOL_BY_DRIVER | EFI_OPEN_PROTOCOL_EXCLUSIVE`. It must use the same *`Attribute`* value that was used in `Supported()`.
2. If any of the calls to `OpenProtocol()` in Step 1 returned an error, then close all of the protocols opened in Step 1 with `Close-`

Protocol(), and return the status code from the call to OpenProtocol() that returned an error.

3. Ignore the parameter *RemainingDevicePath*.
4. Initialize the device specified by *ControllerHandle*. If an error occurs, close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_DEVICE_ERROR.
5. Discover all the child devices of the bus controller specified by *ControllerHandle*.
6. If the bus requires it, allocate resources to all the child devices of the bus controller specified by *ControllerHandle*.
7. FOR each child C of *ControllerHandle*
    8. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.
    9. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to *ControllerHandle*.
    10. Initialize the child device C. If an error occurs, close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_DEVICE_ERROR.
    11. Create a new handle for C, and install the protocol interfaces for child device C. This may include the EFI_DEVICE_PATH_PROTOCOL.
    12. Call OpenProtocol() on behalf of the child C with an *Attribute* of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.
13. END FOR
14. Return EFI_SUCCESS.

Bus Driver that Is Able to Create All or One of Its Child Handles on Each Call to Start():

1. Open all required protocols with OpenProtocol(). If this driver allows the opened protocols to be shared with other drivers, then it should use an *Attribute* of EFI_OPEN_PROTOCOL_BY_DRIVER. If this driver does not allow the opened protocols to be shared with other drivers, then it should use an *Attribute* of EFI_OPEN_PROTOCOL_BY_DRIVER

| EFI_OPEN_PROTOCOL_EXCLUSIVE. It must use the same *Attribute* value that was used in Supported().

2. If any of the calls to OpenProtocol() in Step 1 returned an error, then close all of the protocols opened in Step 1 with CloseProtocol(), and return the status code from the call to OpenProtocol() that returned an error.
3. Initialize the device specified by *ControllerHandle*. If an error occurs, close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_DEVICE_ERROR.
4. IF *RemainingDevicePath* is not NULL, THEN
    5. C is the child device specified by *RemainingDevicePath*.
    6. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.
    7. If the bus driver creates device paths for the child devices, then create a device path for the child C based upon the device path attached to *ControllerHandle*.
    8. Initialize the child device C.
    9. Create a new handle for C, and install the protocol interfaces for child device C. This may include the EFI_DEVICE_PATH_PROTOCOL.
    10. Call OpenProtocol() on behalf of the child C with an *Attribute* of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.
11. ELSE
    12. Discover all the child devices of the bus controller specified by *ControllerHandle*.
    13. If the bus requires it, allocate resources to all the child devices of the bus controller specified by *ControllerHandle*.
    14. FOR each child C of *ControllerHandle*
        15. Allocate and initialize all of the data structures that this driver requires to manage the child device C. This would include space for public protocols and space for any additional private data structures that are related to the child device C. If an error occurs allocating the resources, then close all of the protocols opened in Step 1 with CloseProtocol(), and return EFI_OUT_OF_RESOURCES.

```
        16. If the bus driver creates device paths for the child devices,
            then create a device path for the child C based upon the device
            path attached to ControllerHandle.
        17. Initialize the child device C.
        18. Create a new handle for C, and install the protocol interfaces
            for child device C. This may include the
            EFI_DEVICE_PATH_PROTOCOL.
        19. Call OpenProtocol() on behalf of the child C with an Attribute
            of EFI_OPEN_PROTOCOL_BY_CHILD_CONTROLLER.
      20. END FOR
    21. END IF
    22. Return EFI_SUCCESS.
```

Listed below is sample code of the `Start()` function of device driver for a device on the XYZ bus. The XYZ bus is abstracted with the `EFI_XYZ_IO_PROTOCOL`. This driver does allow the `EFI_XYZ_IO_PROTOCOL` to be shared with other drivers, and just the presence of the `EFI_XYZ_IO_PROTOCOL` on *`ControllerHandle`* is enough to determine if this driver supports *`ControllerHandle`*. This driver installs the `EFI_ABC_IO_PROTOCOL` on *`ControllerHandle`*. The `gBS` and `gMyImageHandle` variables are initialized in this driver's entry point

The following code sequence provides a generic example of what a driver can do in its start routine in the hope of particularizing the guidance listed above.

```
extern EFI_GUID           gEfiXyzIoProtocol;
extern EFI_GUID           gEfiAbcIoProtocol;
EFI_BOOT_SERVICES_TABLE   *gBS;
EFI_HANDLE                gMyImageHandle;

EFI_STATUS
AbcStart (
  IN EFI_DRIVER_BINDING_PROTOCOL  *This,
  IN EFI_HANDLE                   ControllerHandle,
  IN EFI_DEVICE_PATH_PROTOCOL     *RemainingDevicePath
OPTIONAL
)

{
  EFI_STATUS            Status;
  EFI_XYZ_IO_PROTOCOL   *XyzIo;
  EFI_ABC_DEVICE        AbcDevice;
```

```
//
// Open the Xyz I/O Protocol that this driver consumes
//
Status = gBS->OpenProtocol (
                ControllerHandle,
                &gEfiXyzIoProtocol,
                &XyzIo,
                gMyImageHandle,
                ControllerHandle,
                EFI_OPEN_PROTOCOL_BY_DRIVER
                );
if (EFI_ERROR (Status)) {
  return Status;
}

//
// Allocate and zero a private data structure for the Abc
// device.
//
Status = gBS->AllocatePool (
                EfiBootServicesData,
                sizeof (EFI_ABC_DEVICE),
                &AbcDevice
                );
if (EFI_ERROR (Status)) {
  goto ErrorExit;
}
ZeroMem (AbcDevice, sizeof (EFI_ABC_DEVICE));

//
// Initialize the contents of the private data structure for
// the Abc device. This includes the XyzIo protocol instance
// and other private data fields and the EFI_ABC_IO_PROTOCOL
// instance that will be installed.
//
AbcDevice->Signature       = EFI_ABC_DEVICE_SIGNATURE;
AbcDevice->XyzIo           = XyzIo;

AbcDevice->PrivateData1    = PrivateValue1;
AbcDevice->PrivateData1    = PrivateValue2;
. . .
AbcDevice->PrivateData1    = PrivateValueN;

AbcDevice->AbcIo.Revision  = EFI_ABC_IO_PROTOCOL_REVISION;
AbcDevice->AbcIo.Func1     = AbcIoFunc1;
AbcDevice->AbcIo.Func2     = AbcIoFunc2;
. . .
AbcDevice->AbcIo.FuncN     = AbcIoFuncN;
```

```
  AbcDevice->AbcIo.Data1     = Value1;
  AbcDevice->AbcIo.Data2     = Value2;
  . . .
  AbcDevice->AbcIo.DataN     = ValueN;

  //
  // Install protocol interfaces for the ABC I/O device.
  //
  Status = gBS->InstallProtocolInterface (
                  &ControllerHandle,
                  &gEfiAbcIoProtocolGuid,
                  EFI_NATIVE_INTERFACE,
                  &AbcDevice->AbcIo
                  );
  if (EFI_ERROR (Status)) {
    goto ErrorExit;
  }

  return EFI_SUCCESS;

ErrorExit:
  //
  // When there is an error, the provate data structures need
  // to be freed and the protocols that were opened need to be
  // closed.
  //
  if (AbcDevice != NULL) {
    gBS->FreePool (AbcDevice);
  }
  gBS->CloseProtocol (
         ControllerHandle,
         &gEfiXyzIoProtocolGuid,
         gMyImageHandle,
         ControllerHandle
         );
  return Status;
}
```

Chapter 4

Protocols You Should Know

Common sense ain't common.

—Will Rogers

This chapter describes protocols that everyone who is working with the Extensible Firmware Interface, whether creating device drivers, EFI pre-OS applications, or platform firmware, should know. The protocols are illustrated by a few examples, beginning with the most common exercise from any programming text, namely "Hello world." The test application listed here is the simplest possible application that can be written. It does not depend upon any EFI Library functions, so the EFI Library is not linked into the executable that is generated. This test application uses the *`SystemTable`* that is passed into the entry point to get access to the EFI console devices. The console output device is used to display a message using the `OutputString()` function of the **`SIMPLE_TEXT_OUTPUT_INTERFACE`** protocol, and the application waits for a keystroke from the user on the console input device using the `WaitForEvent()` service with the *`WaitForKey`* event in the **`SIMPLE_INPUT_INTERFACE`** protocol. Once a key is pressed, the application exits.

```
/*++

Module Name:
helloworld.c

Abstract:
This is a simple module to display behavior of a basic
```

```
EFI application.

Author:
Waldo

Revision History
--*/

#include "efi.h"

EFI_STATUS
InitializeHelloApplication (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
{
    UINTN Index;

    //
    // Send a message to the ConsoleOut device.
    //

    SystemTable->ConOut->OutputString (
      SystemTable->ConOut,
      L"Hello application started\n\r");

    //
    // Wait for the user to press a key.
    //

    SystemTable->ConOut->OutputString (
      SystemTable->ConOut,
      L"\n\r\n\r\n\rHit any key to exit\n\r");

    SystemTable->BootServices->WaitForEvent (
      1,
      &(SystemTable->ConIn->WaitForKey),
      &Index);

    SystemTable->ConOut->OutputString (
      SystemTable->ConOut,L"\n\r\n\r");

    //
    // Exit the application.
    //

    return EFI_SUCCESS;
}
```

To execute an EFI application, type the program's name at the EFI Shell command line. The following examples show how to run the test application described above from the EFI Shell. The application waits for the user to press a key before returning to the EFI Shell prompt. It is assumed that `hello.efi` is in the search path of the EFI Shell environment.

Example

```
Shell> hello

Hello application started



Hit any key to exit this image
```

EFI OS Loaders

This section discusses the special considerations that are required when writing an OS loader. An *OS loader* is a special type of EFI application responsible for transitioning a system from a firmware environment into an OS environment. To accomplish this task, several important steps must be taken:

1. The OS loader must determine where it was loaded from. This determination allows an OS loader to retrieve additional files from the same location.
2. The OS loader must determine where in the system the OS exists. Typically, the OS resides on a partition of a hard drive. However, the partition where the OS exists may not use a file system that is recognized by the EFI environment. In this case, the OS loader can only access the partition as a block device using only block I/O operations. The OS loader will then be required to implement or load the file system driver to access files on the OS partition.
3. The OS loader must build a memory map of the physical memory resources so that the OS kernel can know what memory to manage. Some of the physical memory in the system must remain untouched by the OS kernel, so the OS loader must use the EFI APIs to retrieve the system's current memory map.
4. An OS has the option of storing boot paths and boot options in nonvolatile storage in the form of environment variables. The OS loader may need to use some of the environment variables that are stored in nonvolatile storage. In addition, the OS loader may

be required to pass some of the environment variables to the OS kernel.

5. The next step is to call `ExitBootServices()`. This call can be done from either the OS loader or from the OS kernel. Special care must be taken to guarantee that the most current memory map has been retrieved prior to making this call. Once `ExitBootServices()` had been called, no more EFI Boot Services calls can be made. At some point, either just prior to calling `ExitBootServices()` or just after, the OS loader will transfer control to the OS kernel.
6. Finally, after `ExitBootServices()` has been called, the EFI Boot Services calls are no longer available. This lack of availability means that once an OS kernel has taken control of the system, the OS kernel may only call EFI Runtime Services.

A complete listing of a sample application for an OS loader can be found below. The code fragments in the following sections do not perform any error checking. Also, the OS loader sample application makes use of several EFI Library functions to simplify the implementation.

The output shown below starts by printing out the device path and the file path of the OS loader itself. It also shows where in memory the OS loader resides and how many bytes it is using. Next, it loads the file `OSKERNEL.BIN` into memory. The file `OSKERNEL.BIN` is retrieved from the same directory as the image of the OS loader sample of Figure 4.1.

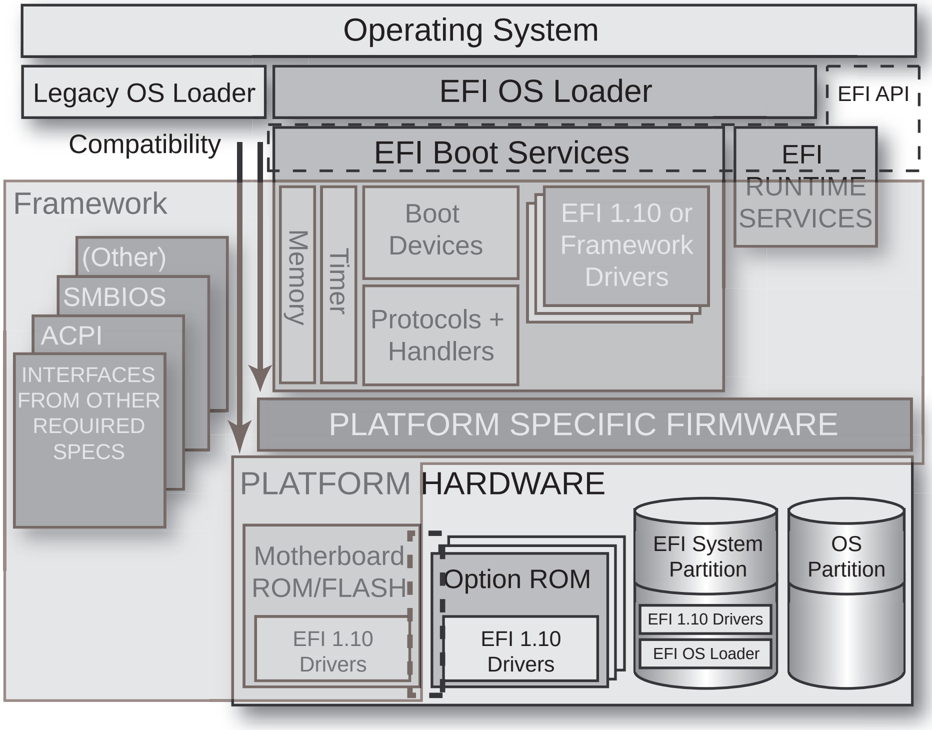


Figure 4.1 EFI Loader in System Diagram

The next section of the output shows the first block of several block devices. The first one is the first block of the floppy drive with a FAT12 file system. The second one is the Master Boot Record (MBR) from the hard drive. The third one is the first block of a large FAT32 partition on the same hard drive, and the fourth one is the first block of a smaller FAT16 partition on the same hard drive.

The final step shows the pointers to all the system configuration tables, the system's current memory map, and a list of all the system's environment variables. The very last step shown is the OS loader calling `ExitBootServices()`.

Device Path and Image Information of the OS Loader

The following code fragment shows the steps that are required to get the device path and file path to the OS loader itself. The first call to HandleProtocol() gets the LOADED_IMAGE_PROTOCOL interface from the *ImageHandle* that was passed into the OS loader application. The second call to HandleProtocol() gets the DEVICE_PATH_PROTOCOL interface to the device handle of the OS loader image. These two calls transmit the device path of the OS loader image, the file path, and other image information to the OS loader itself.

```
BS->HandleProtocol(
        ImageHandle,
        &LoadedImageProtocol,
        LoadedImage
       );



BS->HandleProtocol(
        LoadedImage->DeviceHandle,
        &DevicePathProtocol,
        &DevicePath
      );



Print (
   L"Image device : %s\n",
   DevicePathToStr (DevicePath)
  );

Print (
    L"Image file   : %s\n",
    DevicePathToStr (LoadedImage->FilePath)
  );

Print (
    L"Image Base   : %X\n",
    LoadedImage->ImageBase
  );

Print (
    L"Image Size   : %X\n",
    LoadedImage->ImageSize
  );
```

Accessing Files in the Device Path of the OS Loader

The previous section shows how to retrieve the device path and the image path of the OS loader image. The following code fragment shows how to use this information to open another file called OSKERNEL.BIN that resides in the same directory as the OS loader itself. The first step is to use HandleProtocol() to get the FILE_SYSTEM_PROTOCOL interface to the device handle retrieved in the previous section. Then, the disk volume can be opened so file access calls can be made. The end result is that the variable *CurDir* is a file handle to the same partition in which the OS loader resides.

```
BS->HandleProtocol(
   LoadedImage->DeviceHandle,
   &FileSystemProtocol,
   &Vol
);


Vol->OpenVolume (
        Vol,
        &RootFs
       );


CurDir = RootFs;
```

The next step is to build a file path to OSKERNEL.BIN that exists in the same directory as the OS loader image. Once the path is built, the file handle *CurDir* can be used to call Open(), Close(), Read(), and Write() on the OSKERNEL.BIN file. The following code fragment builds a file path, opens the file, reads it into an allocated buffer, and closes the file.

```
StrCpy(FileName,DevicePathToStr(LoadedImage->FilePath));
for(i=StrLen(FileName);i>=0 && FileName[i]!='\\';i--);

FileName[i] = 0;

StrCat(FileName,L"\\OSKERNEL.BIN");
      CurDir->Open (CurDir, &FileHandle, FileName,
EFI_FILE_MODE_READ, 0);
Size = 0x00100000;
BS->AllocatePool(EfiLoaderData, Size, &OsKernelBuffer);

FileHandle->Read(FileHandle, &Size, OsKernelBuffer);

FileHandle->Close(FileHandle);
```

Finding the OS Partition

The EFI sample environment materializes a BLOCK_IO_PROTOCOL instance for every partition that is found in a system. An OS loader can search for OS partitions by looking at all the BLOCK_IO devices. The following code fragment uses LibLocateHandle() to get a list of BLOCK_IO device handles. These handles are then used to retrieve the first block from each one of these BLOCK_IO devices. The HandleProtocol() API is used to get the DEVICE_PATH_PROTOCOL and BLOCK_IO_PROTOCOL instances for each of the BLOCK_IO devices. The variable *BlkIo* is a handle to the BLOCK_IO device using the BLOCK_IO_PROTOCOL interface. At this point, a ReaddBlocks() call can be used to read the first block of a device. The sample OS loader just dumps the contents of the block to the display. A real OS loader would have to test each block read to see if it is a recognized partition. If a recognized partition is found, then the OS loader can implement a simple file system driver using the EFI API ReadBlocks() to load additional data from that partition.

```
NoHandles = 0;

HandleBuffer = NULL;

LibLocateHandle(ByProtocol, &BlockIoProtocol, NULL,
&NoHandles, &HandleBuffer);

for(i=0;i<NoHandles;i++) {

    BS->HandleProtocol (
            HandleBuffer[i],
            &DevicePathProtocol,
            &DevicePath
           );

    BS->HandleProtocol (
            HandleBuffer[i],
            &BlockIoProtocol,
            &BlkIo
           );

    Block = AllocatePool (BlkIo->BlockSize);

    MediaId = BlkIo->MediaId;

    BlkIo->ReadBlocks(
             BlkIo,
              MediaId,
              (EFI_LBA)0,
              BlkIo->BlockSize,
              Block
             );

    Print(
        L"\nBlock #0 of device
%s\n",DevicePathToStr(DevicePath));

    DumpHex(0,0,BlkIo->BlockSize,Block);

}
```

Getting the Current System Configuration

The system configuration is available through the *SystemTable* data structure that is passed into the OS loader. The operating system loader is an EFI application that is responsible for bridging the gap between the platform firmware and the operating system runtime. The System Table informs the loader of many things: the services available from the platform firmware (such as block and console services for loading the OS kernel binary from media and interacting with the user prior to the OS drivers are loaded, respectively) and access to industry standard tables like ACPI, SMBIOS, and so on. Five tables are available, and their structure and contents are described in the appropriate specifications.

```
LibGetSystemConfigurationTable(
               &AcpiTableGuid,&AcpiTable
              );

LibGetSystemConfigurationTable(
               &SMBIOSTableGuid,&SMBIOSTable
              );

LibGetSystemConfigurationTable(
               &SalSystemTableGuid,&SalSystemTable
              );

LibGetSystemConfigurationTable(
               &MpsTableGuid,&MpsTable
              );
```

```
Print(
    L"  ACPI Table is at address           :
    %X\n",AcpiTable
    );

Print(
    L"  SMBIOS Table is at address         :
    %X\n",SMBIOSTable
    );

Print(
    L"  Sal System Table is at address     :
    %X\n",SalSystemTable
   );

Print(
    L"  MPS Table is at address            :
    %X\n",MpsTable
   );
```

Getting the Current Memory Map

One EFI Library function can retrieve the memory map maintained by the EFI environment. While the loader is running, the memory has been managed by the platform firmware. It has allocated memory for both firmware usage (boot services memory) and other memory that needs to persist into the OS runtime (runtime memory). Until the loader passes final control to the OS kernel and invokes ExitBootServices(), the EFI platform firmware manages the allocation of memory. The means by which the OS loader and other EFI applications can ascertain the allocation of memory if via the memory map services. The following code fragment shows the use of this function to ascertain the memory map, and it displays the contents of the memory map. An OS loader must pay special attention to the *MapKey* parameter. Every time that the EFI environment modifies the memory map that it maintains, the *MapKey* is incremented. An OS loader needs to pass the current memory map to the OS kernel. Depending on what functions the OS loader calls between the time the memory map is retrieved and the time that `ExitBootServices()` is called, the memory map may be modified. In general, the OS loader should retrieve the memory map just before calling `ExitBootServices()`. If `ExitBootServices()` fails because the *MapKey* does not match, then the OS loader must get a new copy of the memory map and try again.

```
MemoryMap = LibMemoryMap(
              &NoEntries,
              &MapKey,
              &DescriptorSize,
              &DescriptorVersion
             );

Print(
   L"Memory Descriptor List:\n\n"
  );

Print(
   L"  Type        Start Address     End Address
Attributes      \n"
    );
Print(
  L"  ==========  ================  ================
================\n");

MemoryMapEntry = MemoryMap;

for(i=0;i<NoEntries;i++) {
    Print(L"  %s  %lX  %lX  %lX\n",
          OsLoaderMemoryTypeDesc[MemoryMapEntry->Type],
          MemoryMapEntry->PhysicalStart,
          MemoryMapEntry->PhysicalStart +
              LShiftU64(
                    MemoryMapEntry->NumberOfPages,
                    PAGE_SHIFT)-1,
                    MemoryMapEntry->Attribute
                    );
    MemoryMapEntry = NextMemoryDescriptor(
                           MemoryMapEntry,
                           DescriptorSize
                          );
}
```

Getting Environment Variables

The following code fragment shows how to extract all the environment variables maintained by the EFI environment. It uses the `GetNextVariableName()` API to walk the entire list.

```
VariableName[0] = 0x0000;

VendorGuid = NullGuid;
```

```
Print(
   L"GUID                                Variable Name
   Value\n");
Print(
    L"===================================
====================
    ========\n");
do {
  VariableNameSize = 256;
  Status = RT->GetNextVariableName(
                 &VariableNameSize,
                 VariableName,
                 &VendorGuid
               );
  if (Status == EFI_SUCCESS) {
    VariableValue = LibGetVariable(
                        VariableName,
                        &VendorGuid
                       );
    Print(
      L"%.-35g %.-20s
      %X\n",&VendorGuid,VariableName,VariableValue
     );
  }
} while (Status == EFI_SUCCESS);
```

Transitioning to an OS Kernel

A single call to `ExitBootServices()` terminates all the EFI Boot Services that the EFI environment provides. From that point on, only the EFI Runtime Services may be used. Once this call is made, the OS loader needs to prepare for the transition to the OS kernel. It is assumed that the OS kernel has full control of the system and that only a few firmware functions are required by the OS kernel. These functions are the EFI Runtime Services. The OS loader must pass the *`SystemTable`* to the OS kernel so that the OS kernel can make the Runtime Services calls. The exact mechanism that is used to transition from the OS loader to the OS kernel is implementation-dependent. It is important to note that the OS loader could transition to the OS kernel prior to calling `ExitBootServices()`. In this case, the OS kernel would be responsible for calling `ExitBootServices()` before taking full control of the system.

Chapter 5

EFI Runtime

Adding manpower to a late software project makes it later.

—Brook's Law

This chapter describes the fundamental services that are made available in an EFI-compliant system. The services are defined by interface functions that may be used by code running in the EFI environment. Such code may include protocols that manage device access or extend platform capabilities. In this chapter, the Runtime services will be the focus of discussion. These runtime services are functions that are available both during EFI operation and when the OS has been launched and running.

During boot, system resources are owned by the firmware and are controlled through a variety of system services that expose callable APIs. In EFI there are two primary types of services:

- Boot Services – Functions that are available prior to the launching of the boot target (such as the OS), and prior to the calling of the `ExitBootServices()` function.
- Runtime Services – Functions that are available both during the boot phase prior to the launching of the boot target and after the boot target is executing. :

Figure 5.1 illustrates the phases of boot operation that a platform evolves through.

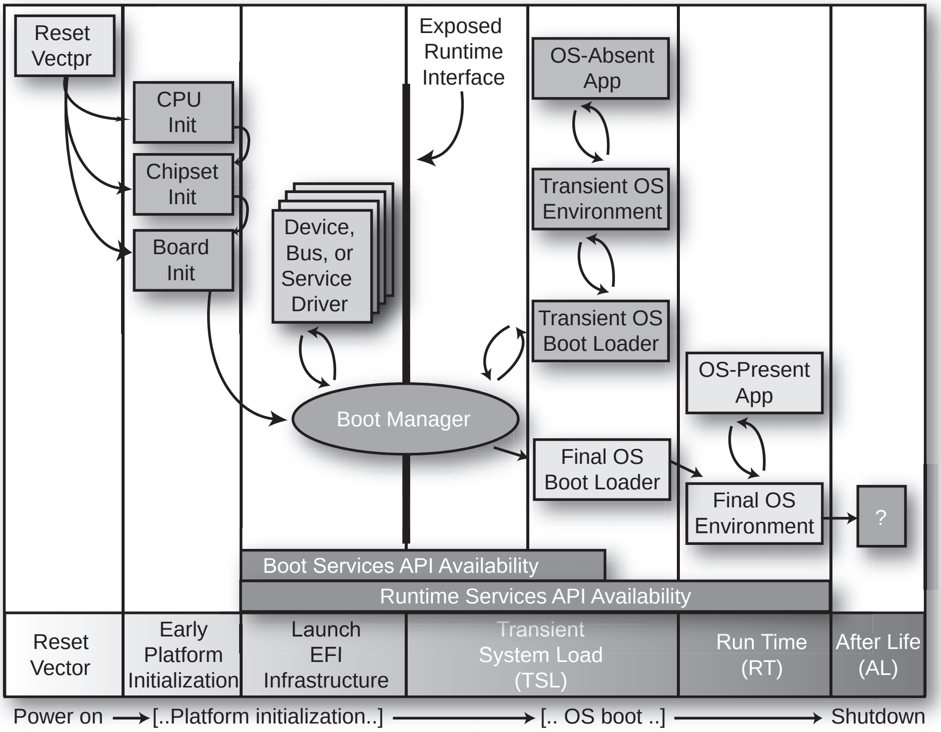


Figure 5.1 Phases of Boot Operation

In Figure 5.1, it is clearly evident that the two previously mentioned forms of services (Boot Services and Runtime Services) are available during the early launch of the EFI infrastructure and only the runtime services are available after the remainder of the firmware stack has relinquished control to an OS loader. Once an OS loader has loaded enough of its own environment to take control of the system's continued operation it can then terminate the boot services with a call to `ExitBootServices()`.

In principle, the `ExitBootServices()` call is intended for use by the operating system to indicate that its loader is ready to assume control of the platform and all platform resource management. Thus boot services are available up to this point to assist the OS loader in preparing to boot the operating system. Once the OS loader takes control of the system and completes the operating system boot process, only runtime services may be called. Code other than the OS loader, however, may or may not choose to call `ExitBootServices()`. This choice may in part depend upon whether or not such code is designed to make continued use of EFI boot services or the boot services environment.

Isn't There Only One Kind of Memory?

When EFI memory is allocated, it is "typed" according to certain classifications which designate the general purpose of a particular memory type. For instance, one might choose to allocate a buffer as an `EfiRuntimeServicesData` buffer if it was desired that a buffer containing some data remained available into the runtime phase of platform operations. When allocated memory, one might think "Why not allocate everything as a runtime memory type 'just in case'?" The reason that such activity is hazardous is that when the platform transitions from Boot Services phase into Runtime phase, all of the buffers which might have been allocated as runtime as now frozen and unavailable to the OS. Since there is an implicit assumption that items which request runtime-enabled memory know what they are doing, one can imagine a proliferation of memory leaks if we simply assumed a single type of memory usage. With this situation in mind, EFI establishes a certain set of memory types with certain expected usage associated with each.

Table 5.1 EFI Memory Types and Usage Prior to ExitBootServices()

Mnemonic	Description
EfiReservedMemoryType	Not used.
EfiLoaderCode	The code portions of a loaded application. (Note that EFI OS loaders are EFI applications.)
EfiLoaderData	The data portions of a loaded application and the default data allocation type used by an application to allocate pool memory.
EfiBootServicesCode	The code portions of a loaded Boot Services Driver.
EfiBootServicesData	The data portions of a loaded Boot Serves Driver, and the default data allocation type used by a Boot Services Driver to allocate pool memory.
EfiRuntimeServicesCode	The code portions of a loaded Runtime Services Driver.
EfiRuntimeServicesData	The data portions of a loaded Runtime Services Driver and the default data allocation type used by a Runtime Services Driver to allocate pool memory.

Mnemonic	Description
EfiConventionalMemory	Free (unallocated) memory.
EfiUnusableMemory	Memory in which errors have been detected.
EfiACPIReclaimMemory	Memory that holds the ACPI tables.
EfiACPIMemoryNVS	Address space reserved for use by the firmware.
EfiMemoryMappedIO	Used by system firmware to request that a memory-mapped IO region be mapped by the OS to a virtual address so it can be accessed by EFI runtime services.
EfiMemoryMappedIO-PortSpace	System memory-mapped IO region that is used to translate memory cycles to IO cycles by the processor.
EfiPalCode	Address space reserved by the firmware for code that is part of the processor.

Table 5.1 lists memory types and their corresponding usage prior to launching a boot target (such as an OS). The memory types that would be used by most runtime drivers would be those with the keyword "runtime" in them.

However, to better illustrate how these memory types are used in the runtime phase of the platform evolution, Table 5.2 illustrates how these EFI Memory types are used after the OS loader has called `ExitBootServices()` to indicate the transition from the pre-boot, to the runtime phase of operations.

Table 5.2 EFI Memory Types and Usage after `ExitBootServices()`

Mnemonic	Description
EfiReservedMemoryType	Not used.
EfiLoaderCode	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called **`ExitBootServices()`** is utilizing one or more **`EfiLoaderCode`** ranges.
EfiLoaderData	The Loader and/or OS may use this memory as they see fit. Note: the OS loader that called **`ExitBootServices()`** is utilizing one or more **`EfiLoaderData`** ranges.
EfiBootServicesCode	Memory available for general use.
EfiBootServicesData	Memory available for general use.

Mnemonic	Description
EfiRuntimeServicesCode	The memory in this range is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiRuntimeServicesData	The memory in this range is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiConventionalMemory	Memory available for general use.
EfiUnusableMemory	Memory that contains errors and is not to be used.
EfiACPIReclaimMemory	This memory is to be preserved by the loader and OS until ACPI is enabled. Once ACPI is enabled, the memory in this range is available for general use.
EfiACPIMemoryNVS	This memory is to be preserved by the loader and OS in the working and ACPI S1–S3 states.
EfiMemoryMappedIO	This memory is not used by the OS. All system memory-mapped IO information should come from ACPI tables.
EfiMemoryMappedIOPortSpace	This memory is not used by the OS. All system memory-mapped IO port space information should come from ACPI tables.
EfiPalCode	This memory is to be preserved by the loader and OS in the working and ACPI S1–S3 states. This memory may also have other attributes that are defined by the processor implementation.

In Table 5.2, one can see how the runtime memory types are preserved, and the BootServices type of memory is available for the OS to reclaim as its own.

How Are Runtime Services Exposed?

In EFI, firmware services are exposed through a set of EFI protocol definitions, a series of function pointers in some special purpose service tables, and finally in the EFI configuration table. Of these mechanisms that are used to expose firmware APIs, only the following two are persistent into the runtime phase of computer operations.

- Runtime Services Table - The EFI Runtime Services Table contains pointers to all of the runtime services. All elements in the EFI

Runtime Services Table are prototypes of function pointers that are valid after the operating system has taken control of the platform with a call to `ExitBootServices()`.

- EFI Configuration Table - The EFI Configuration Table contains a set of GUID/pointer pairs. The number of entries in this table can easily grow over time. That is why a GUID is used to identify the configuration table type. This table may contain at most one instance of each table type.

The runtime services that are exposed in the EFI Runtime Services Table at minimum define the core required runtime API capabilities of an EFI-compliant platform. These functions include services that expose time, virtual memory, and variable services at a minimum.

The information exposed through the EFI Configuration Table is going to vary widely between platform implementations. One key thing to note, however, is that the GUID associated with the GUID/pointer pair defines how one interprets the data to which the pointer is pointing. The content to which the pointer is pointed could be a function/API, a table of data, or practically anything else. Some examples of the type of information that can be exposed through this table are SMBIOS, ACPI, and MPS tables, as well as function prototypes for an UNDI-compliant network card. Figure 5.2 is an example diagram of the interactions between the EFI Configuration Table and an example function prototype.

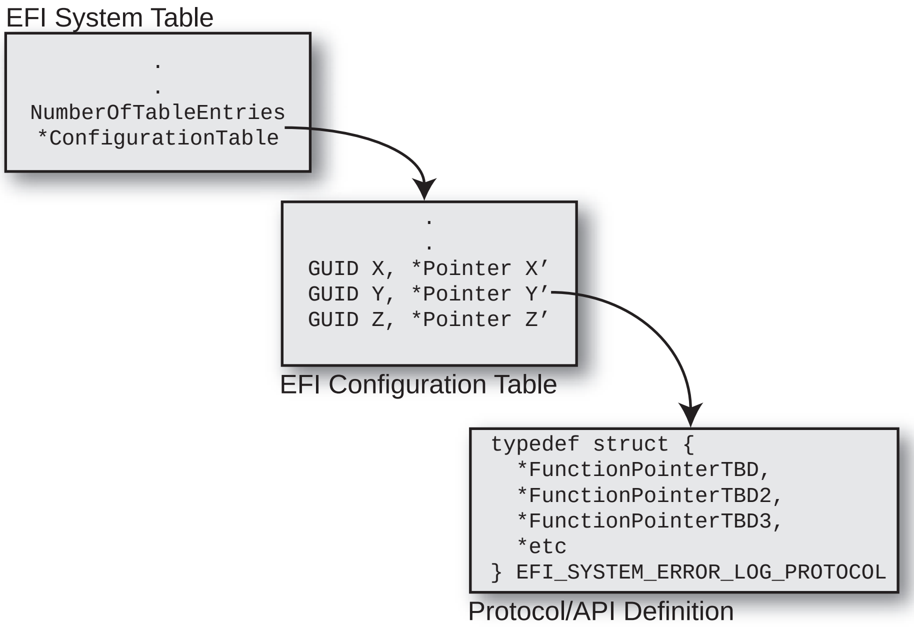


Figure 5.2 Interactions between the EFI Configuration Table and a Function Prototype

Time Services

This section describes the core EFI definitions for time-related functions that are specifically needed by operating systems at runtime to access underlying hardware that manages time information and services. The purpose of these interfaces is to provide runtime consumers of these services an abstraction for hardware time devices, thereby relieving the need to access legacy hardware devices directly. The functions listed in Table 5.3 reside in the EFI Runtime Services table.

Table 5.3 Time-based Functions in the EFI Runtime Services Table

Name	Type	Description
GetTime	Runtime	Returns the current time and date, and the time-keeping capabilities of the platform.
SetTime	Runtime	Sets the current local time and date information.
GetWakeupTime	Runtime	Returns the current wakeup alarm clock setting.
SetWakeupTime	Runtime	Sets the system wakeup alarm clock time.

Why Abstract Time?

For a variety of reasons one might choose to abstract the access to the platform RealTime Clock (RTC). First, very poor standard mechanisms (if any) exist to access the platform's RTC. A variety of legacy interrupts might serve some purposes, but typically might not abstract sufficient information to be particularly useful. If a user wanted to talk to the RTC directly, the user would not typically know how to with the exception of using some of the standard IBM CMOS directives. Ultimately, how one might gain access to this fundamental piece of information ("What time is it?") could change over time. With that in mind, one needed the platform to provide a set of abstractions so that the caller would not have to worry about the vagaries of varying programming some RTC to acquire time information or to depend on some poorly documented and completely nonstandard set of legacy interrupts to abstract this same data.

Get Time

Even though this function is called "GetTime", it is intended to return the current time as well as the date information along with the capabilities of the current underlying time-based hardware. This service is not intended to provide highly accurate timings beyond certain described levels. During the Boot Services phase of platform initialization, there are other means by which to do accurate time stall measurements (for example, see the `Stall()` boot services function in the EFI specification).

Even though Figure 5.3 shows the smallest granularity of time measurement in nanoseconds, this by no means is intended as an indication of the accuracy of the time measurement of which the function is capable. The only thing that is guaranteed by the call to this function is that it returns a time that was valid during the call to the function. This guarantee is more understandable when one thinks about the processing time for the call to traverse various levels of code between the caller and the service function actually talking to the hardware device and this data then being passed back to the caller. Since this is a call initiated during the runtime phase of platform operations, the highly accurate timers that are needed for small granularity timing events would be provided by alternate (likely OS-based) solutions.

```
//********************************************************

//EFI_TIME

//********************************************************

// This represents the current time information

typedef struct {
      UINT16            Year;                 // 1998 - 20XX
      UINT8             Month;                // 1 - 12
      UINT8             Day;                  // 1 - 31
      UINT8             Hour;                 // 0 - 23
      UINT8             Minute;               // 0 - 59
      UINT8             Second;               // 0 - 59
      UINT8             Pad1;
      UINT32            Nanosecond;        // 0 - 999,999,999
      INT16             TimeZone;          // -1440 to 1440 or 2047
      UINT8             Daylight;
      UINT8             Pad2;
} EFI_TIME;
```

Figure 5.3 Example Time Definition

Set Time

This function provides the ability to set the current time and date information on the platform.

Get Wakeup Time

This function provides the abstraction for obtaining the alarm clock settings for the platform. This is often used to determine if a platform has been set for being woken up, and if so, at what time it should be woken up.

Set Wakeup Time

Setting a system wakeup alarm causes the system to wake up or power on at the set time. When the alarm fires, the alarm signal is latched until acknowledged by calling `SetWakeupTime()` to disable the alarm. If the alarm fires before the system is put into a sleeping or off state, since the alarm signal is latched the system will immediately wake up.

Virtual Memory Services

This section contains function definitions for the virtual memory support that may be optionally used by an operating system at runtime. If an operating system chooses to make EFI runtime service calls in a virtual addressing mode instead of the flat physical mode, then the operating system must use the services in this section to switch the EFI runtime services from flat physical addressing to virtual addressing. Table 5.4 lists the virtual memory services functions that EFI provides.

Table 5.4 Virtual Memory Services

Name	Type	Description
SetVirtualAddressMap	Runtime	Used by an OS loader to convert from physical addressing to virtual addressing.
ConvertPointer	Runtime	Used by EFI components to convert internal pointers when switching to virtual addressing.

By using these functions, the platform provides a mechanism by which components that will exist during the runtime phase of operations can adjust their own data references to the new virtual addresses that the runtime caller has supplied. This makes it possible for the underlying firmware component(s) to adjust from a physical address mode to virtual address mode entity.

This conversion applies to all functions in the runtime services table as well as the pointers in the EFI System Table. However, this is not necessarily the case for the EFI Configuration Table. In the EFI Configuration Table, one is dealing with GUID/pointer pairs, and since the pointers are all physical to start with in the firmware, one might think that the pointers are converted during the transition to the runtime phase of platform operations, right? In this particular case, you would be wrong.

The GUID portion of the GUID/pointer pair defines the state of the pointer itself. In theory, one might have a particular GUID that during runtime has a virtual address pointer paired with it, but the next GUID' in the table might very well be a physical pointer. This is because the EFI Configuration Table can often be used to advertise certain pieces of information and the consumer of this information might have reason for interpreting the pointer as a physical pointer even though the OS has converted all other pertinent data to virtual addresses. In addition, the EFI Configuration Table often might be pointing to a runtime enabled function prototype. In most cases, the pointers for this function would be converted, while other items that might be pointed at by the EFI Configuration Table (Data Tables, for instance) might have no reason to have any data be converted.

Set Virtual Address Map

By calling this service, the agent that is the owner of the system's memory map (the component that called `ExitBootServices()`) can change the runtime addressing mode of the underlying EFI firmware from physical to virtual. The inputs of course are the new virtual memory map which shows an array of memory descriptors that have mapping information for all runtime memory ranges.

When this service is called, all runtime-enabled agents will in turn be called through a notification event triggered by the `SetVirtualAddressMap()` function.

ConvertPointer

The `ConvertPointer` function is used by an EFI component during the `SetVirtualAddressMap()` operation. When the platform has passed control to an OS loader and it in turn calls `SetVirtualAddressMap()`, a function is be called in most runtime drivers that responds to the virtual address change event that is triggered. This function uses the ConvertPointer service to convert the current physical pointer to an appropriate virtual address pointer. All pointers that the component has allocated should be updated using this mechanism.

Variable Services

Variables are defined as key/value pairs that consist of identifying information, attributes, and some quantity of data. Variables are intended for use as a means to store data that is passed between the EFI environment implemented in the platform and EFI OS loaders and other applications that run in the EFI environment.

Although the implementation of variable storage is not specifically defined for a given platform, variables must be able to persist across reboots of the platform. This implies that the EFI implementation on a platform must arrange it so that variables passed in for storage are retained and available for use each time the system boots, at least until they are explicitly deleted or overwritten. Provision of this type of nonvolatile storage may be very limited on some platforms, so variables should be used sparingly in cases where other means of communicating information cannot be used. Table 5.5 lists the variable services functions that EFI provides.

Table 5.5 Variable Services

Name	Type	Description
GetVariable	Runtime	Returns the value of a variable.
GetNextVariableName	Runtime	Enumerates the current variable names.
SetVariable	Runtime	Sets the value of a variable.

GetVariable

This function returns the value of a given EFI variable. Since a fully qualified EFI variable name is composed of both a human-readable text value paired with a GUID, a vendor can create and manage its own variables without the risk of name conflicts by using its own unique GUID value. For instance, one can easily have three variables named "Setup" that are wholly unique assuming that each of these "Setup" variables has a different numeric GUID value.

One of the key items to note in the definition of an EFI variable is that each one has some attributes associated with it. These attributes are treated as a bit field, which implies that none, any, or all of the bits can be activated at any given time. In the case of EFI variables, however, there are three defined attribute bits to be aware of:

- Nonvolatile – a variable that has this attribute activated is defined to be persistent across platform resets. It should also be noted that the explicit absence of this bit being activated indicates that the variable is volatile, and is therefore a temporary variable that will be absent once the system resets or the variable is deleted.
- BootService – a variable that has this attribute activate provides read/write access to it during the BootService phase of the plat-

form evolution. This simply means that once the platform enters the runtime phase, the data will no longer be able to be set through the SetVariable service.

- Runtime – a variable that has this attribute activated must also have the BootService attribute activated. With this, the variable is accessible during all phases of the platform evolution.

GetNextVariableName

Since the EFI variable repository is very similar in concept to a file system, the ability to parse the repository is provided by the GetNextVariableName service. This service enumerates the current variable names in the platform, and with each subsequent call to the service the previous results can be passed into the interface, and on output the interface returns the next variable name data. Once the entire list of variables has been returned, a subsequent call into the service providing the previous "last" variable name will provide the equivalent of a "Not Found" error.

It should be noted that this service is affected by the phase of platform operations. Variables that do not have the runtime attribute activated are allocated typically from some type of BootServices memory. Since this is the case, once `ExitBootServices()` is performed to signify the transition into the runtime phase, these variables will no longer show up in the search list that GetNextVariableName provides.

One other behavior that should be noted is that one might conceive that if a variable has the ability to be named the same human-readable name (such as "Setup") and the only thing that differs is the GUID, one could seed the search mechanism for this service by walking a common GUID-based list of variables. This is not the case. The usage of this service is typically initiated with a call that starts with a pointer to a Null Unicode string as the human-readable name; the GUID is ignored. Instead, the entire list of variables must be retrieved, and the caller may act as a filter if you choose to have it do so.

SetVariable

EFI variables are often used to provide a means by which to save platform-based context information. For instance, when the platform initializes the I/O infrastructure and has probed for all known console output devices, it will likely construct a ConOutDev global variable. These global variables have a unique purpose in the platform since they have a specific architectural role to play with a specific purpose. Table 5.6 shows some of the defined global variables.

Table 5.6 Global Variables

Variable Name	Attribute	Description
LangCodes	BS, RT	The language codes that the firmware supports. This value is deprecated.
Lang	NV, BS, RT	The language code that the system is configured for. This value is deprecated.
Timeout	NV, BS, RT	The firmware's boot managers timeout, in seconds, before initiating the default boot selection.
PlatformLangCodes	BS, RT	The language codes that the firmware supports.
PlatformLang	NV, BS, RT	The language code that the system is configured for.
ConIn	NV, BS, RT	The device path of the default input console.
ConOut	NV, BS, RT	The device path of the default output console.
ErrOut	NV, BS, RT	The device path of the default error output device.
ConInDev	BS, RT	The device path of all possible console input devices.
ConOutDev	BS, RT	The device path of all possible console output devices.
ErrOutDev	BS, RT	The device path of all possible error output devices.

The examples in Table 5.6 show some of the common global variables, their descriptions, and their attributes. Some of the noted differences are the presence or absence of the NV (nonvolatile) attribute. This simply means that the values associated with these variables are not persistent across platform resets and their values are determined during the initialization phase of platform operations. Unlike variables that are persistent, robust implementations of EFI enable the setting of volatile variables in memory-backed store, and do not necessarily have the storage size sensitivities that the other variables have that are stored in a fixed hardware with often very limited storage capacity.

Software should only use a nonvolatile variable when absolutely necessary. It should be noted that a variable has no concept of a zero-byte data payload. All variables must contain at least 1 byte of data, since the service definition stipulates that the means by which you delete a target variable is by calling the SetVariable() service with a zero byte data payload.

There are certain rules that should definitely be noted when it comes to the use of the attributes:

- Attributes are only applied to a variable when the variable is created. If a preexisting variable is rewritten with different attributes, the result is indeterminate and may vary between implementations. The correct method of changing the attributes of a variable is to delete the variable and recreate it with different attributes.
- Setting a data variable with no access attributes or a zero size data payload causes it to be deleted.
- Runtime access to a data variable implies boot service access.
- Once ExitBootServices() is performed, data variables that did not have the runtime access attribute set are no longer visible. This simply enforces the paradigm that once in runtime phase, variables without the runtime attribute are not to be read from.
- Once ExitBootServices() is performed, only variables that have the runtime and the nonvolatile access attributes set can be set with a call to the SetVariable() service. In addition, variables that have runtime access but that are not nonvolatile are now read-only data variables. The reason for this situation is that once the platform firmware has handed off control to another agent (such as the OS), it no longer controls the memory services and cannot further allocate services that might be backed by memory. Since the SetVariable service typically uses memory to spill content to store a volatile variable, this capability is no longer available during the runtime phase of operations.

By providing a mechanism for shared data content such as an EFI variable, the use of variables can be seen as a fairly flexible and highly available mechanism for firmware components to communicate. The variables shown in Table 5.6 are some of the architectural variables that steer the behavior of a platform. In this case aspects of the platform configuration can be seen in the data reflected by these variables. Another usage of the variable services can be to use the volatile (one must stress volatile, and not nonvolatile) variable as means by which two disparate components can have a common repository that is independent of a nonvolatile backing store (such as a hard-disk), yet can act as a temporary repository of data such as registry content that is discovered by one agent and retrieved by another. This infrastructure provides for a lot of flexibility in implementation.

Miscellaneous Services

This section contains the remaining function definitions for runtime services that were not talked about in previous sections but are required to complete a compliant implementation of an EFI environment. The services that are in this section are as listed in Table 5.7.

Table 5.7 Miscellaneous Services

Name	Type	Description
GetNextHighMonotonicCount	Runtime	Returns the next high 32 bits of the platform's monotonic counter.
ResetSystem	Runtime	Resets the entire platform.

Reset System

This service provides a caller the ability to reset the entire platform including all processors and devices, and reboots the system. This service provides the ability to stipulate three types of rests:

- Cold Reset – A call to the ResetSystem() service stipulating a cold reset will cause a system-wide reset. This sets all circuitry within the system to its initial state. This type of reset is asynchronous to system operation and operates without regard to cycle boundaries. This is tantamount to a system power cycle.
- Warm Reset – Calling the ResetSystem() service stipulating a warm reset will also cause a system-wide initialization. The processors are set to their initiate state, and pending cycles are not corrupted. This difference should be noted, since memory is not typically reinitialized and that the machine may be rebooting without having cleared memory that previously existed. There are a lot of examples of this usage model, and implementations vary on exactly what platforms choose to do with this type of feature. If the system does not support this reset type, then a Cold Reset must be performed.
- Reset Shutdown – Calling the ResetSystem() service stipulating a Reset Shutdown will cause the system to enter a power state equivalent to the ACPI G2/S5 or G3 states. If the system does not

support this reset type, then when the system is rebooted, it should exhibit the same attributes as having booted from a Cold Reset.

Get Next High Monotonic Count

The platform provides a service to get the platform monotonic counter. The platform's monotonic counter is comprised of two 32-bit quantities: the high 32 bits and the low 32 bits. During boot service time the low 32-bit value is volatile: it is reset to zero on every system reset and is increased by 1 on every call to GetNextMonotonicCount(). The high 32-bit value is nonvolatile is increased by 1 whenever the system resets or whenever the low 32-bit count overflows.

Since the GetNextMonotonicCount() service is available only at boot services time, and if the operating system wishes to extend the platform monotonic counter to runtime, it may do so by utilizing the GetNextHighMonotonicCount() runtime service. To do this, before calling ExitBootServices() the operating system would call GetNextMonotonicCount() to obtain the current platform monotonic count. The operating system would then provide an interface that returns the next count by:

- Adding 1 to the last count.
- Before the lower 32 bits of the count overflows, call GetNextHighMonotonicCount(). This will increase the high 32 bits of the platform's nonvolatile portion of the monotonic count by 1.

This function may only be called at runtime.

Chapter 6

EFI Console Services

Never test for an error condition you don't know how to handle.

—Steinbach's Guideline for Systems Programming

This chapter describes how EFI extends the traditional boundaries of console support in the pre-boot phase and provides a series of software layering approaches that are commonly used in EFI-compliant platforms. Most platforms, at minimum, would have a text-based console for a user to either locally or remotely interact with the system. A variety of mechanisms can accomplish this communication in EFI. Whether it is through a remote interface, through a local keyboard and monitor, or even a remote network connection, each has a common root that can be thought of as the basic EFI console support. This support is used to handle input and output of text-based information intended for the system user during the operation of code in the EFI boot services environment. These console definitions are split into three types of console devices: one for input, and one each for normal output and errors.

These interfaces are specified by function call definitions to allow maximum flexibility in implementation. For example, a compliant system is not required to have a keyboard or screen directly connected to the system. As long as the semantics of the functions are preserved, implementations may direct information using these interfaces in any way that succeeds in passing the information to the system user

The EFI console is built out of two primary protocols, and these are EFI Simple Text Input and EFI Simple Text Output protocols. These two protocols implement a basic text-based console that allows platform firmware, EFI applications, and EFI OS loaders to present information to and receive input from a system administrator. The EFI console consists of 16-bit Unicode characters, a simple set of input control characters known as scan codes, and a set of output-oriented programmatic interfaces that give functionality equivalent to an intelligent terminal. This text-based set of interfaces does not inherently support pointing devices on input or bitmaps on output.

To ensure greatest interoperability, the EFI Simple Text Output protocol is recommended to support at least the printable basic Latin Unicode character set to enable standard terminal emulation software to be used with an EFI console. The basic Latin Unicode character set implements a superset of ASCII that has been extended to 16-bit characters. This provides the maximum interoperability with external terminal emulations that might otherwise require the conversion of text encoding to be down-converted to a set of ASCII equivalents.

EFI has a variety of system-wide references to consoles. The EFI System Table contains six console-related entries:

- ConsoleInHandle – The handle for the active console input device. This handle must support the EFI Simple Text Input protocol.
- ConIn – A pointer to the EFI Simple Text Input protocol interface that is associated with ConsoleInHandle.
- ConsoleOutHandle – The handle for the active console output device. This handle must support the EFI Simple Text Output protocol.
- ConOut – A pointer to the EFI Simple Text Output protocol interface that is associated with ConsoleOutHandle.
- StandardErrorHandle – The handle for the active standard error console device. This handle must support the EFI Simple Text Output protocol.
- StdErr – A pointer to the EFI Simple Text Output protocol interface that is associated with StandardErrorHandle.

Other system-wide references to consoles in EFI are contained within the global variable definitions. Some of the pertinent global variable definitions in EFI are:

- ConIn – The EFI global variable that contains the device path of the default input console.

- ConInDev – The EFI global variable that contains the device path of all possible console input devices.
- ConOut – The EFI global variable that contains the device path of the default output console.
- ConOutDev – The EFI global variable that contains the device path of all possible console output devices.
- ErrOut – The EFI global variable that contains the device path of the default error console.
- ErrOutDev – The EFI global variable that contains the device path of all possible console output devices.

Figure 6.1 illustrates the software layering discussed so far. An EFI application or driver that wants to communicate through a text interface can use the active console shown in the EFI System Table to call the interface that supports the appropriate text input or text output protocol. During initialization, the system table is passed to the launched EFI application or driver, and this component can then immediately start using the console in question.

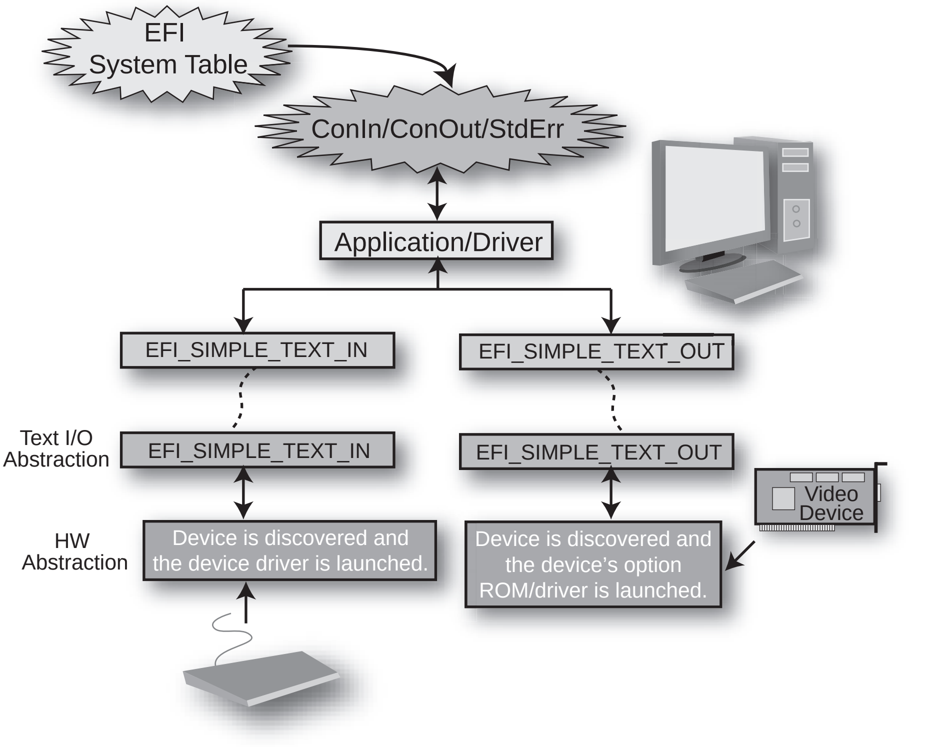


Figure 6.1 Initial Software Layering

To further describe these interactions it is necessary to delve a bit deeper into what these text I/O interfaces really look like and what they are effectively responsible for.

Simple Text Input Protocol

The Simple Text Input Protocol defines the minimum input required to support a specific ConIn device. This interface provides two basic functions for the caller:

- Reset – This function resets the input device hardware. As part of the initialization process, the firmware/device makes a quick but reasonable attempt to verify that the device is functioning. This hardware verification process is implementation-specific and is left up to the firmware and/or EFI driver to implement.

- ReadKeyStroke – This function reads the next keystroke from the input device. If no keystroke is pending, the function returns an EFI Not Ready error. If a keystroke is pending, an EFI key is returned. An EFI key is composed of a scan code as well as a Unicode character. The Unicode character is the actual printable character or is zero if the key is not represented by a printable character, such as the control key or a function key.

When reading a key from the ReadKeyStroke() function, an EFI Input Key is retrieved. In traditional firmware, all PS/2 keys had a hardware specific scan code, which was the sole item firmware dealt with. In EFI, things have been changed a bit to facilitate the reasonable transaction of this data both with local and remote users. The data sent back has two primary components:

- Unicode Character – The Simple Text Input protocol defines an input stream that contains Unicode characters. This value represents the Unicode-encoded 16-bit value that corresponds to the key that was pressed by the user. A few Unicode characters have special meaning and are thus defined as supported Unicode control characters, as described in Table 6.1.

Table 6.1 EFI-supported Unicode Control Characters

Mnemonic	Unicode	Description
Null	U+0000	Null character ignored when received.
BS	U+0008	Backspace. Moves cursor left one column. If the cursor is at the left margin, no action is taken.
TAB	U+0x0009	Tab.
LF	U+000A	Linefeed. Moves cursor to the next line.
CR	U+000D	Carriage Return. Moves cursor to left margin of the current line.

- Scan Code - The input stream supports EFI scan codes in addition to Unicode characters. If the scan code is set to 0x00 then the Unicode character is valid and should be used. If the EFI scan code is set to a value other than 0x00, it represents a special key as defined in Table 6.2.

Table 6.2 EFI-supported Scan Codes

EFI Scan Code	Description
0x00	Null scan code.
0x01	Move cursor up 1 row.
0x02	Move cursor down 1 row.
0x03	Move cursor right 1 column.
0x04	Move cursor left 1 column.
0x05	Home.
0x06	End.
0x07	Insert.
0x08	Delete.
0x09	Page Up.
0x0a	Page Down.
0x0b	Function 1.
0x0c	Function 2.
0x0d	Function 3.
0x0e	Function 4.
0x0f	Function 5.
0x10	Function 6.
0x11	Function 7.

EFI Scan Code	Description
0x12	Function 8.
0x13	Function 9.
0x14	Function 10.
0x17	Escape.

The ReadKeyStroke function provides the additional capability to signal an EFI event when a key has been received. To leverage this capability, one must use either the WaitForEvent or CheckEvent services. The event to pass into these services is the following:

- WaitForKey – The event to use when calling WaitForEvent() to wait for a key to be available.

The activity being handled by the Simple Text Input protocol is very similar to the INT 16h services that were available in legacy firmware. Some of the primary differences are that the legacy firmware service returned only the ASCII equivalent 8-bit value for the key that was pressed along with the hardware-specific (such as PS/2) scan codes.

Simple Text Output Protocol

The Simple Text Output protocol is used to control text-based output devices. It is the minimum required protocol for any handle supplied as the ConOut or StdOut device. In addition, the minimum supported text mode of such devices is at least 80 ×25 characters.

A video device that supports only graphics mode is required to emulate text mode functionality. Output strings themselves are not allowed to contain any control codes other than those defined in Table 6.1. Positional cursor placement is done only via the SetCursorPosition() function. It is highly recommended that text output to the StdErr device be limited to sequential string outputs. That is, it is not recommended to use ClearScreen() or SetCursorPosition() on ouput messages to StdErr, so that this data can be clearly captured or viewed.

The Simple Text Output protocol also has a pointer to some mode data, as shown in Figure 6.2. This mode data is used to determine what the current text settings are for the given device. Much of this information is used to determine what the current cursor position is as well as the given foreground and background color. In addition, one can stipulate whether a cursor should be visible or not.

```
typedef struct {
    INT32                               MaxMode;
    // current settings
    INT32                               Mode;
    INT32                               Attribute;
    INT32                               CursorColumn;
    INT32                               CursorRow;
    BOOLEAN                             CursorVisible;
} SIMPLE_TEXT_OUTPUT_MODE;
```

Figure 6.2 Mode Structure for EFI Simple Text Output Protocol

The Simple Text Output protocol also has a variety of text output related functions; however, this chapter focuses on some of the most commonly used ones:

- OutputString – Provides the ability to write a NULL-terminated Unicode string to the output device and have it displayed. All output devices must also support some of the basic Unicode drawing characters listed in the *EFI 1.1 Specification*. This is the most basic output mechanism on an output device. The string is displayed at the current cursor location on the output device(s) and the cursor is advanced according to the rules listed in Table 6.3.

Table 6.3 Cursor Advancement Rules

Mnemonic	Unicode	Description
Null	U+0000	Ignore the character, and do not move the cursor.
BS	U+0008	If the cursor is not at the left edge of the display, then move the cursor left one column.
LF	U+000A	If the cursor is at the bottom of the display, then scroll the display one row, and do not update the cursor position. Otherwise, move the cursor down one row.
CR	U+000D	Move the cursor to the beginning of the current row.
Other	U+XXXX	Print the character at the current cursor position and move the cursor right one column. If this moves the cursor past the right edge of the display, then the line should wrap to the beginning of the next line. This is equivalent to inserting a CR and an LF. Note that if the cursor is at the bottom of the display, and the line wraps, then the display will be scrolled one line.

By providing an abstraction that allows a console device, such as a video driver, to produce a text interface, this can be compared to legacy firmware support for INT 10h. The producer of the Simple Text Output interface is responsible for converting the Unicode text characters into the appropriate glyphs for that device. In the case where an unrecognized Unicode character has been sent to the OutputString() API, the result is typically a warning that indicates that these characters were skipped.

- SetAttribute – This function sets the background and foreground colors for both the OutputString() and ClearScreen() functions. A variety of foreground and background colors are defined by the *EFI 1.1 Specification*. The color mask can be set even if the device is in an invalid text mode. Devices that support a different number of text colors must emulate the specified colors to the best of the device's capabilities.
- ClearScreen – This function clears the output device(s) display to the currently selected background color. The cursor position is set to (0,0).
- SetCursorPosition – This function sets the current coordinates of the cursor position. The upper left corner of the screen is defined as coordinate (0,0).

Remote Console Support

The previous sections of this chapter described some of the text input and output protocols, and used some examples that were generated through local devices. EFI also supports many types of remote console. This support leverages the pre-existing local interfaces but enables the routing of this data to/from devices outside of the platform being executed.

When a remote console is instantiated, it typically results from EFI constructing an I/O abstraction that a console driver latches onto. In this case, the discussion initially concerns serial interface consoles. A variety of console transport protocols, such as PC-ANSI, VT-100, and so on, describe the format of the data that is sent to and from the machine.

The console driver responsible for producing the Text I/O interfaces acts as a filter for the I/O. For example, when a remote key is pressed, this might require a variety of pieces of data to be constructed and sent from the remote device and upon receipt, the console driver needs to interpret this information and convert it into the corresponding EFI semantics such as the EFI scan code and Unicode character. The same is true for any application running on the local machine that prints a message. This message is received by the console driver and translated to the remote terminal type semantics.

Table 6.4 gives examples of how an EFI scan code can be mapped to ANSI X3.64 terminal, PC-ANSI terminal, or an AT 101/102 keyboard. PC ANSI terminals support an escape sequence that begins with the ASCII character 0x1b and is followed by the ASCII character 0x5B, "[". ASCII characters that define the control sequence that should be taken follow the escape sequence. The escape sequence does not contain spaces, but spaces are used in Table 6.4 for ease of reading. For additional information on EFI terminal support, see the *EFI 1.1 Specification*.

Table 6.4 Sample Conversion Table for EFI Scan Codes to other Terminal Formats

EFI Scan Code	Description	ANSI X3.64 Codes	PC ANSI Codes	AT 101/102 Keyboard Scan Codes
0x00	Null scan code	N/A	N/A	N/A
0x01	Move cursor up 1 row	CSI A	ESC [A	0xe0, 0x48
0x02	Move cursor down 1 row	CSI B	ESC [B	0xe0, 0x50
0x03	Move cursor right 1 column	CSI C	ESC [C	0xe0, 0x4d
0x04	Move cursor left 1 column	CSI D	ESC [D	0xe0, 0x4b
0x05	Home	CSI H	ESC [H	0xe0, 0x47
0x06	End	CSI K	ESC [K	0xe0, 0x4f
0x07	Insert	CSI @	ESC [@	0xe0, 0x52
0x08	Delete	CSI P	ESC [P	0xe0, 0x53
0x09	Page Up	CSI ?	ESC [?	0xe0, 0x49
0x0a	Page Down	CSI /	ESC [/	0xe0, 0x51

Table 6.5 shows some of the PC-ANSI and ANSI X3.64 control sequences for adjusting display/text display attributes for text displays.

Table 6.5 Example Control Sequences that Can Be Used in Console Drivers

PC ANSI Codes	ANSI X3.64 Codes	Description
ESC [2 J	CSI 2 J	Clear Display Screen.
ESC [0 m	CSI 0 m	Normal Text.
ESC [1 m	CSI 1 m	Bright Text.
ESC [7 m	CSI 7 m	Reversed Text.
ESC [30 m	CSI 30 m	Black foreground, compliant with ISO Standard 6429.
ESC [31 m	CSI 31 m	Red foreground, compliant with ISO Standard 6429.
ESC [32 m	CSI 32 m	Green foreground, compliant with ISO Standard 6429.
ESC [33 m	CSI 33 m	Yellow foreground, compliant with ISO Standard 6429.
ESC [34 m	CSI 34 m	Blue foreground, compliant with ISO Standard 6429.

Figure 6.3 illustrates the software layering for a remote serial interface with Text I/O abstractions. The primary difference between this illustration and one that exhibits the same Text I/O abstractions on local devices is that this one has one additional layer of software drivers. In the former examples, the local device was discovered by an agent launched, and it in turn would establish a set of Text I/O abstractions. In the remote case, the local device is a serial device, which has a console driver that is layered onto it, and it in turn would establish a set of Text I/O abstractions.

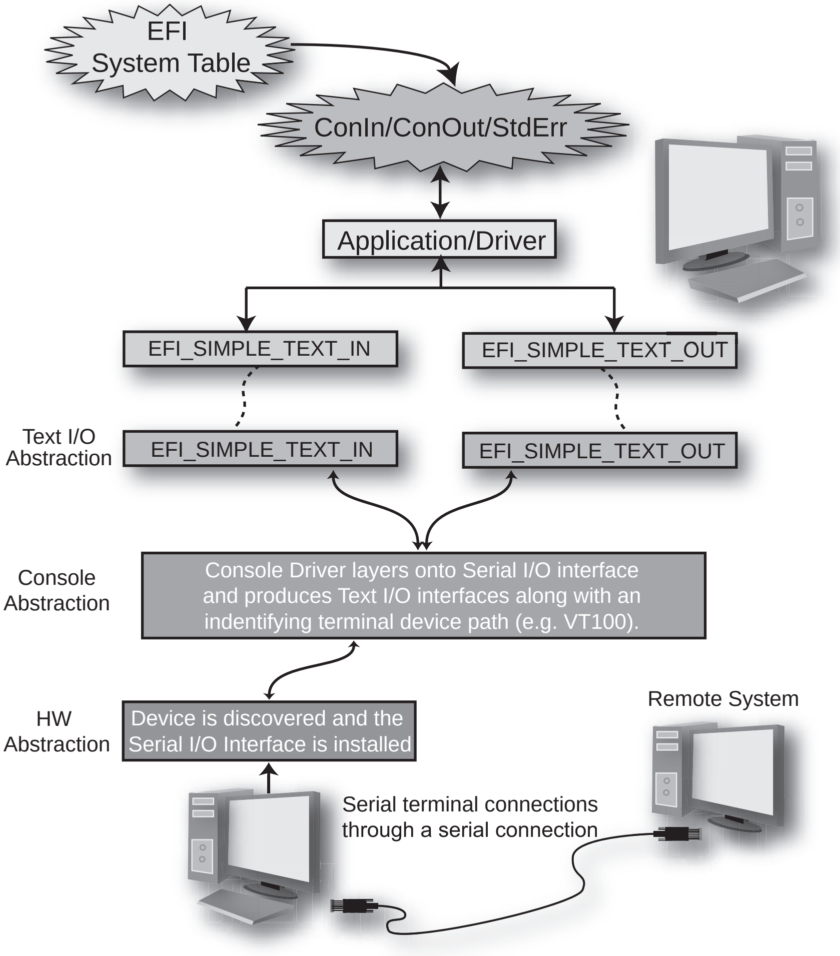

EFI
System Table
ConIn/ConOut/StdErr
Application/Driver
EFI_SIMPLE_TEXT_IN
EFI_SIMPLE_TEXT_OUT
Text I/O
Abstraction
EFI_SIMPLE_TEXT_IN
EFI_SIMPLE_TEXT_OUT
Console
Abstraction
Console Driver layers onto Serial I/O interface
and produces Text I/O interfaces along with an
indentifying terminal device path (e.g. VT100).
HW
Abstraction
Device is discovered and the
Serial I/O Interface is installed
Remote System
Serial terminal connections
through a serial connection

Figure 6.3 Remote Console Software Layering

Console Splitter

The ability to describe a variety of console devices poses interesting new possibilities. In previous generations of firmware, one had a single means by which one could describe what the Text I/O sources and targets were. Now the EFI variables that specify which are the active consoles are specified by a device path. In this case, these device paths are multi-instance, meaning that more than one target device could be the active input or output. For instance, if one wanted to be able to have an application print text to the local screen as well as to the screen of a remote terminal, it would be highly impractical for anyone to customize their software to accommodate that particular scenario. In the solution that EFI provides with its console splitting/merging capability, an application can simply use the standard text interfaces that EFI provides and the Console Splitter routes the text requests to the appropriate target or targets. This works for both input as well as output streams.

The way this works is that when the EFI-compliant platform initializes, the console splitter installs itself in the EFI System Table as the primary active console. In doing so, it can then proceed to monitor the platform as other EFI text interfaces get installed as protocols and the console splitter keeps a running tally of the user selected devices for a given console variable, such as ConOut, ConIn, or ErrOut.

Figure 6.4 illustrates a scenario where an application is calling EFI text interfaces, which in turn calls the EFI System Table console interfaces. These interfaces belong to the console splitter, and the console splitter then sends the text I/O requests from the application to the platform-configured consoles.

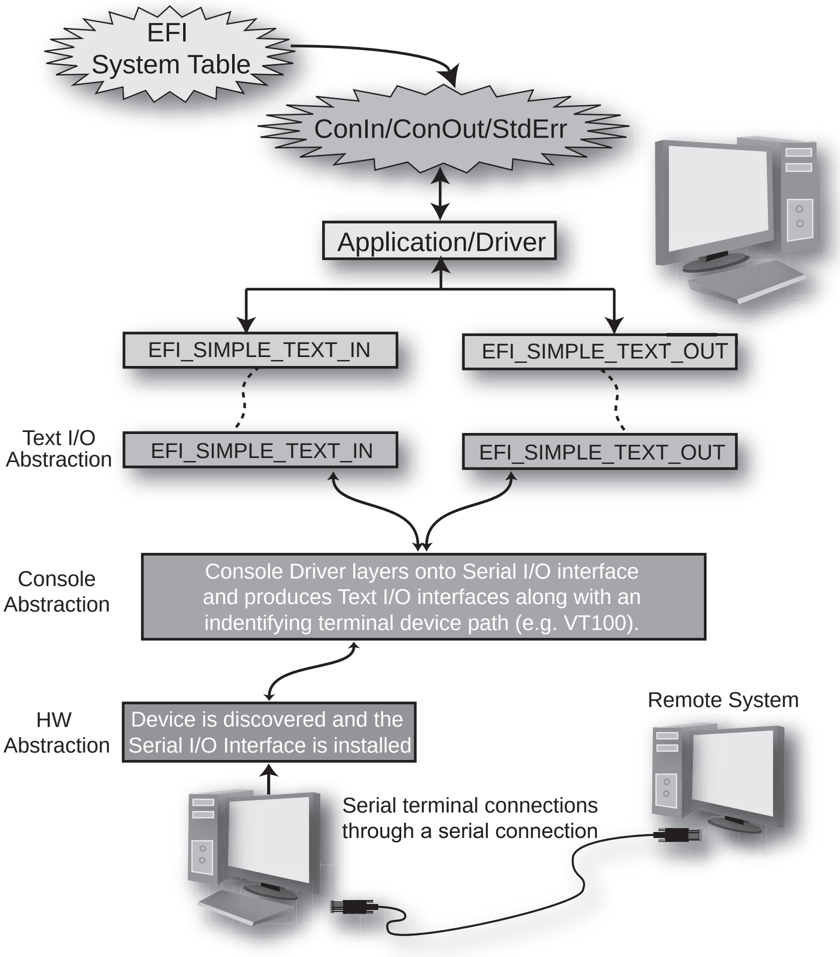

EFI
System Table
ConIn/ConOut/StdErr
Application/Driver
EFI_SIMPLE_TEXT_IN
EFI_SIMPLE_TEXT_OUT
Text I/O
Abstraction
EFI_SIMPLE_TEXT_IN
EFI_SIMPLE_TEXT_OUT
Console
Abstraction
Console Driver layers onto Serial I/O interface
and produces Text I/O interfaces along with an
indentifying terminal device path (e.g. VT100).
HW
Abstraction
Device is discovered and the
Serial I/O Interface is installed
Remote System
Serial terminal connections
through a serial connection

Figure 6.4 Software Layering Description of the EFI Console Splitter

Network Consoles

EFI also provides the ability to establish data connections with remote platforms across a network. Given the appropriate installed drivers, one could also enable an EFI-compliant platform to support a text I/O set of abstractions. Similar to previously discussed concepts where the hardware interface (for example, serial device, keyboard, video, network interface card) has an abstraction, other components build on top of this hardware abstraction to provide a working software stack.

Some network components that EFI might include are as follows:

- Network Interface Identifier – This is an optional protocol that is produced by the Universal Network Driver Interface (UNDI) and is used to produce the Simple Network Protocol. This protocol is only required if the underlying network interface is a 16-bit UNDI, 32/64-bit S/W UNDI, or H/W UNDI. It is used to obtain type and revision information about the underlying network interface.
- Simple Network Protocol – This protocol provides a packet level interface to a network adapter. It in addition provides services to initialize a network interface, transmit packets, receive packets, and close a network interface.

To illustrate what a common network console might look like, you could describe an initial hardware abstraction that talks directly to the network interface controller (NIC) produced by an UNDI driver. This in turn has a Simple Network Protocol that layers on top of UNDI. It provides basic network abstraction interfaces such as Send and Receive. On top of this a transport protocol might be installed such as a TCP/IP stack. As with most systems, once an established transport mechanism is provided, one can build all sorts of extensions into the platform such as a Telnet Daemon to allow remote users to log into the system through a network connection. Ultimately, this daemon would produce and be responsible for handling the normal Text I/O interfaces already described in this chapter.

Figure 6.5 illustrates an example where a remote machine is able to access the EFI-compliant platform through a network connection. Providing the top layer of the software stack (EFI_SIMPLE_TEXT_IN and EFI_SIMPLE_TEXT_OUT) as the interoperable surface area that applications talk to allows for all standard EFI applications to seamlessly leverage the console support in a platform. Couple this with console splitting and merging as inherent capabilities and you have the ability to interact with the platform in a much more robust manner without requiring a lot of specially tuned software to enable it.

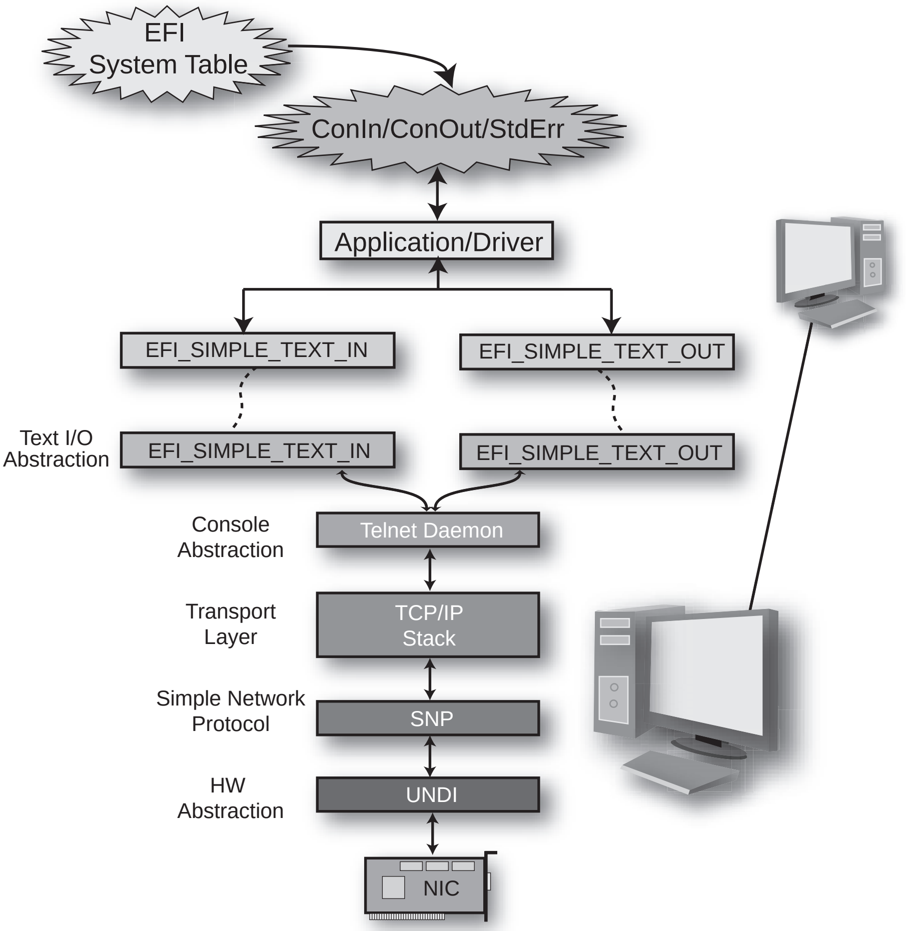


Figure 6.5 Example of Network Console Software Layering

Chapter 7

Different Types of Platforms

Variety's the very spice of life, That gives it all its flavor.

—William Cowper

This chapter describes different platform types and instantiations of the Framework, such as embedded system, PDA, desktop, and server. In addition to providing a "BIOS replacement" for platforms that are commonly referred to as the Personal Computer, the Framework infrastructure can be used to construct a boot and initialization environment for servers, handheld devices, televisions, and so on. These sundry devices may include the more common IA-32 processors in the PC, but also feature the lower-power Intel XScale® processor, or the mainframe-class processors such as the Itanium®-based systems. This chapter examines the PEI modules and DXE drivers that are necessary to construct a standard PC platform. Then a subset of these modules used for emulation and Intel XScale-based PDAs is described.

Figure 7.1 is a block diagram of a typical system, showing the various components, integrating the north bridge, south bridge, and super I/O, beyond other possible components. These blocks represent components manufactured on the system board. Each silicon and platform component will have an associated module or driver to handle the respective initialization. In addition to the components on the system board, the initial memory map of the platform has specific region allocations. Figure 7.2 shows the memory map of the PC platform. The system flash in this platform configuration is 512 kilobytes in size. The system flash appears at the upper end of the 32-bit address space in order to allow the Pentium®

4 processor to fetch the first opcodes from flash upon reset. The reset vector lies 16 bytes from the end of the address space. In the SEC, the initial opcodes of the SEC file allow for initial control flow of the Framework-based platform firmware. From the SEC a collection of additional modules are executed.

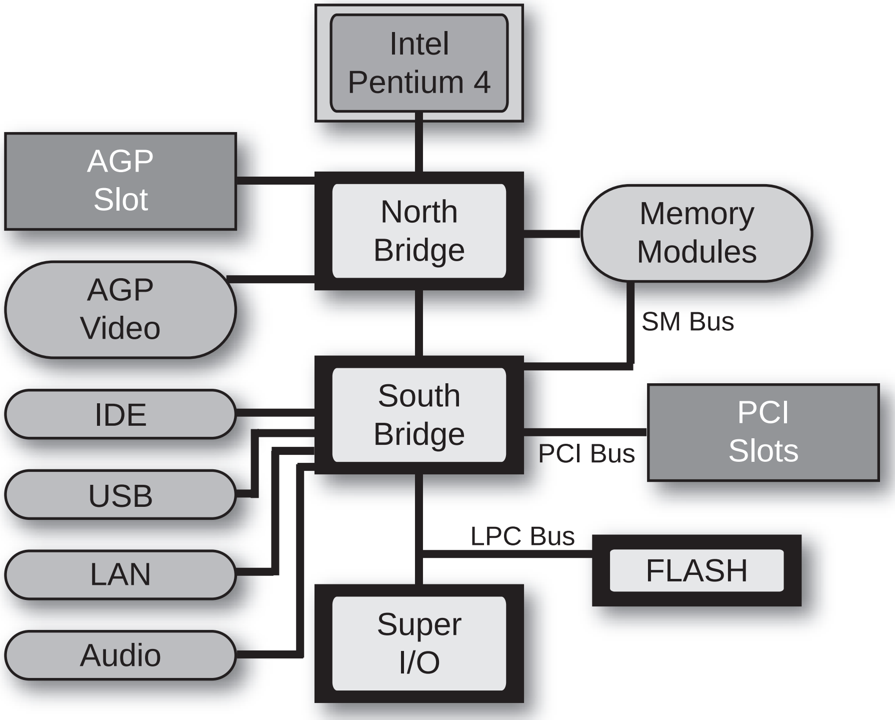


Figure 7.1 Typical PC System

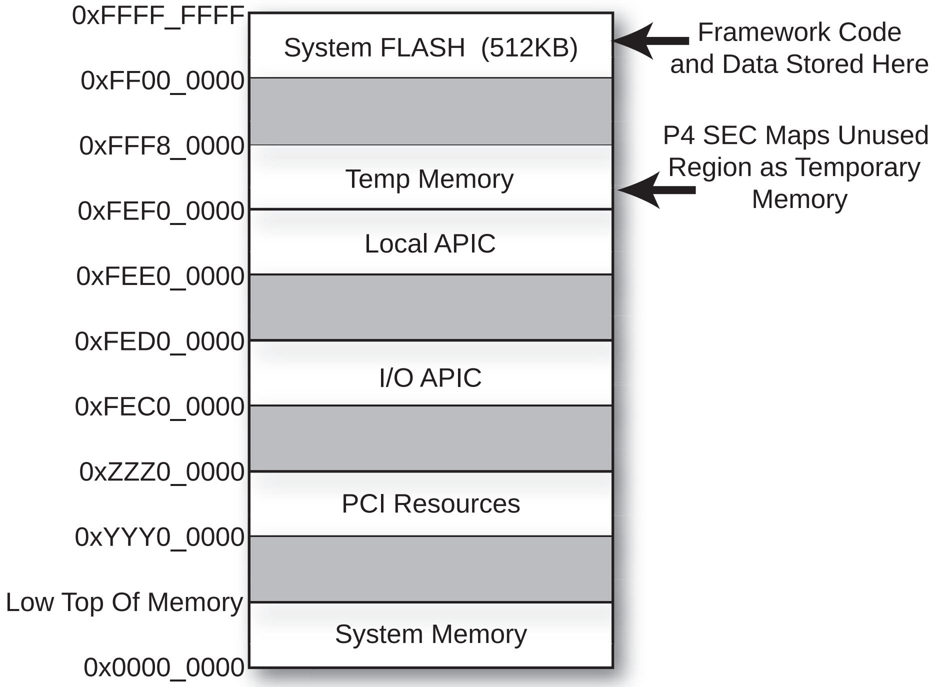


Figure 7.2 Address Map

Before going through the various components of the PC Firmware load, a few other platforms will be reviewed. These include the wireless personal digital assistant, which can be a low-power IA32 CPU or an Intel XScale processor. The platforms then scale up to a server. This is shown in Figure 7.3.

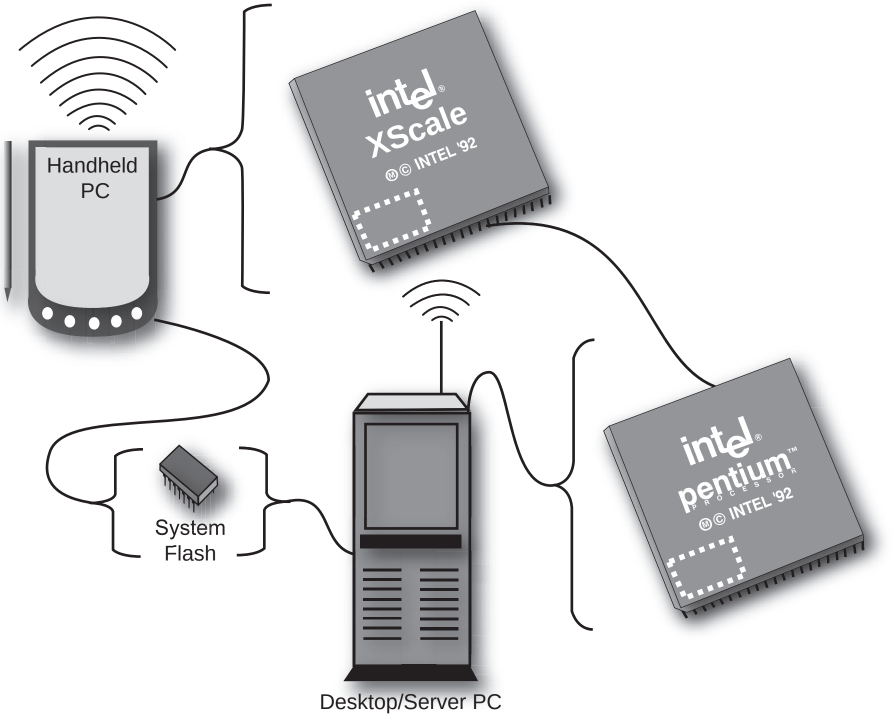

Handheld PC
intel® XScale ©© INTEL '92
System Flash
intel® pentium™ PROCESSOR ©© INTEL '92
Desktop/Server PC

Figure 7.3 Span of Systems

Figure 7.4 shows an intelligent I/O board, namely a storage accelerator. The I/O card attaches to a server via a PCI bus slot. When the server restarts, the I/O card restarts independently. Within the I/O board, an Intel XScale-specific collection of PEI modules (PEIMs) and DXE drivers execute to initialize the local SCSI controller and SDRAM components. Then the I/O board would boot an EFI-aware version of an embedded operating system, such as BSD Unix, or continue to use the DXE components as the I/O runtime to service read/write requests from the server host CPU complex. This demonstrates how the platform concept can span many different topologies. These topologies include the classical, open-architecture PC and the headless, closed embedded system of an I/O board.

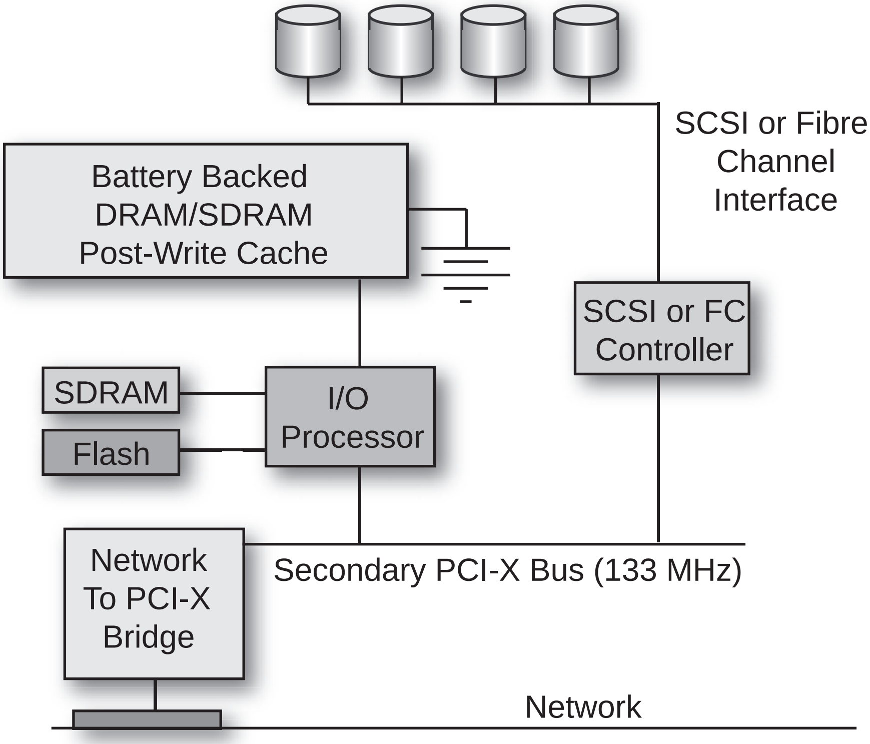
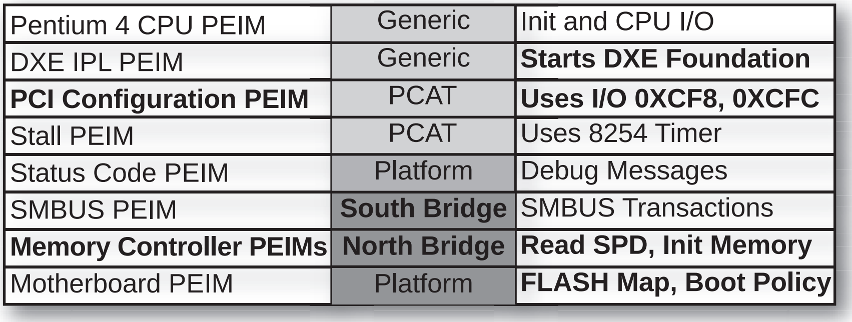

SCSI or Fibre Channel Interface
Battery Backed DRAM/SDRAM Post-Write Cache
SCSI or FC Controller
SDRAM
Flash
I/O Processor
Network To PCI-X Bridge
Secondary PCI-X Bus (133 MHz)
Network

Figure 7.4 An Intel XScale®-based System

Now let's examine the components for the PC in Figure 7.1 in greater detail. The PEI phase of execution runs immediately after a restart event, such as a power-on reset, resume from hibernate, and so on. The PEI modules execute-in-place from the flash store, at least until the main memory complex (such as DRAM) has been initialized.

Figure 7.5 displays the collection of PEIMs for the PC platform. Different business interests would supply the modules. For example, in the "Lakeport" platform, Intel would provide the 945 GMCH (Graphics Memory Controller Hub) Memory Controller PEIM and the ICH6 (I/O Controller Hub) PEIM. In addition, for the SMBUS (System management bus) attached to the ICH, there would be an ICH-specific SMBUS PEIM. The status code PEIM would describe a platform-specific means by which to emit debug information, such as an 8-bit code emitted to I/O port 80-hex.



Pentium 4 CPU PEIM	Generic	Init and CPU I/O
DXE IPL PEIM	Generic	**Starts DXE Foundation**
PCI Configuration PEIM	PCAT	**Uses I/O 0XCF8, 0XCFC**
Stall PEIM	PCAT	Uses 8254 Timer
Status Code PEIM	Platform	Debug Messages
SMBUS PEIM	**South Bridge**	SMBUS Transactions
Memory Controller PEIMs	**North Bridge**	**Read SPD, Init Memory**
Motherboard PEIM	Platform	**FLASH Map, Boot Policy**

Figure 7.5 Components of PEI on PC

The SMBUS PEIM for the ICH listed in Figure 7.5 provides a standard interface, or PEIM-to-PEIM interface (PPI), as shown in Figure 7.6. This allows the memory controller PEIM to use the SMBUS read command in order to get information regarding the DIMM (Dual-Inline Memory Module) SPD (Serial Presence Detect) data on the memory. The SPD data includes the size, timing, and other details about the memory modules. The memory initialization PEIM will use the **EFI_PEI_SMBUS_PPI** so that the GMCH-specific memory initialization module does not need to know which component provides the SMBUS capability. In fact, many Integrated Super I/O (SIO) components also provide an SMBUS controller, so this platform could have replaced the ICH SMBUS PEIM with an SIO SMBUS PEIM without having to modify the memory controller PEIM.

```
typedef
EFI_STATUS
(EFIAPI *PEI_SMBUS_PPI_EXECUTE_OPERATION) (
  IN      EFI_PEI_SERVICE            **PeiServices,
  IN      struct EFI_PEI_SMBUS_PPI   *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS   SlaveAddress,
  IN      EFI_SMBUS_DEVICE_COMMAND   Command,
  IN      EFI_SMBUS_OPERATION        Operation,
  IN      BOOLEAN                    PecCheck,
  IN OUT  UINTN                      *Length,
  IN OUT  VOID                       *Buffer
  );

typedef struct {
  PEI_SMBUS_PPI_EXECUTE_OPERATION  Execute;
  PEI_SMBUS_PPI_ARP_DEVICE         ArpDevice;
} EFI_PEI_SMBUS_PPI;
```

Figure 7.6 Code fragment for a PEIM PPI

Many implementations are possible beyond the **EFI_PEI_SMBUS_PPI** shown earlier. Figure 7.7 shows a code fragment that implements the SMBUS read operation for the ICH component listed earlier. Note the use of the CPU I/O abstraction for performing the I/O operations against the ICH component. The fact that the logic is written in C means that this same ICH6 on an Intel XScale or Itanium-based system could reuse the same source code through a simple compilation for the target microarchitecture.

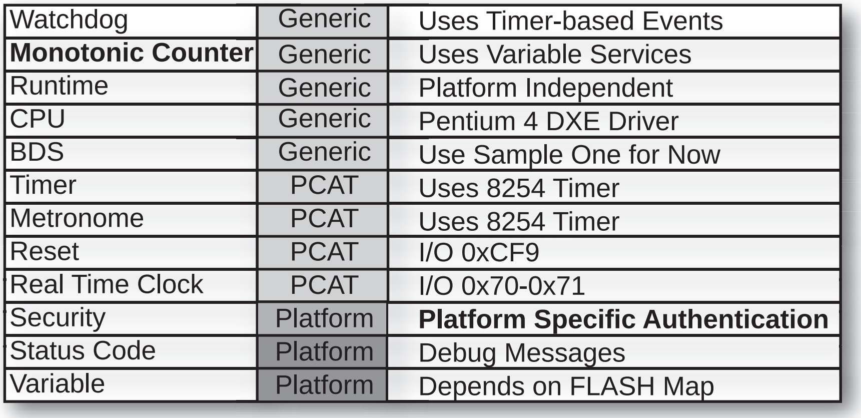
```
#define SMBUS_R_HD0  0xEFA5
#define SMBUS_R_HBD  0xEFA7

EFI_PEI_SERVICES          *PeiServices;
SMBUS_PRIVATE_DATA        *Private;?
UINT8  Index, BlockCount  *Length;
UINT8                     *Buffer;

BlockCount = Private->CpuIo.IoRead8 (
               *PeiServices,Private->CpuIo,SMBUS_R_HD0);
if (*Length < BlockCount) {
  return EFI_BUFFER_TOO_SMALL;
} else {
  for (Index = 0; Index < BlockCount; Index++) {
    Buffer[Index] = Private->CpuIo.IoRead8 (
                      *PeiServices,Private->CpuIo,SMBUS_R_HBD);
  }
}
```

Figure 7.7 Code Fragment of PEIM Implementation

Beyond the PEI phase, the DXE core requires a series of platform, CPU, and chipset specific drivers in order to provide a fully-instantiated set of DXE/EFI services. Figure 7.8 lists the collection of architectural protocols that are necessary for the PC platform under study.



Watchdog	Generic	Uses Timer-based Events
Monotonic Counter	Generic	Uses Variable Services
Runtime	Generic	Platform Independent
CPU	Generic	Pentium 4 DXE Driver
BDS	Generic	Use Sample One for Now
Timer	PCAT	Uses 8254 Timer
Metronome	PCAT	Uses 8254 Timer
Reset	PCAT	I/O 0xCF9
Real Time Clock	PCAT	I/O 0x70-0x71
Security	Platform	**Platform Specific Authentication**
Status Code	Platform	Debug Messages
Variable	Platform	Depends on FLASH Map

Figure 7.8 Architectural Protocols

The fact that the DXE Foundation does not presume anything about the time-keeping logic, interrupt controller, instruction set, and so on, means that the DXE Foundation C code can be retargeted for a large class of platforms without reengineering the Foundation code itself. Instead, a different collection of the architectural protocols (APs) can affect the Foundation port.

One aspect of the system that needs to be abstracted is the management of time. The time-keeping hardware on a PC/AT compatible chipset, such as the 8254 timer, differs from the CPU-integrated timer-counter (ITC) on the Itanium processor or the Intel XScale processor specific time-keeping logic. As such, in order to have a single implementation of the DXE Foundation watchdog-timer logic, the access to CPU/chipset specific timing hardware is implemented via the Timer Architectural Protocol. There are a series of services in this AP, such as getting and setting the time period. The setting of the time period will be reviewed across our reference class of platforms.

To begin, Figure 7.9 provides an instance of the set timer service for the NT32 platform. NT32 is a virtual Framework platform that executes upon a 32-bit Microsoft Windows system as a user-mode process. It is a “soft” platform in that the platform capabilities are abstracted through Win32 services. As such, the implementation of this AP service doesn’t access an I/O controller or chipset control/status registers. Instead, the

AP invokes a series of Win32 services to provide mutual exclusion and an operating system thread to emulate the timer action.

```
EFI_STATUS
TimerDriverSetTimerPeriod (
  IN EFI_TIMER_ARCH_PROTOCOL  *This,
  IN UINT64                   TimerPeriod
  )
{
. . .
  gWinNt->EnterCriticalSection (&mNtCriticalSection);
  mTimerPeriod = TimerPeriod;
  mCancelTimerThread = FALSE;
  gWinNt->LeaveCriticalSection (&mNtCriticalSection);
  mNtLastTick = gWinNt->GetTickCount ();
  mNtTimerThreadHandle = gWinNt->CreateThread (
                                  NULL,
                                  0,
                                  NtTimerThread,
                                  &mTimer,
                                  0,
                                  &NtThreadId);
. . .
}
```

Figure 7.9 NT32 Architectural Protocol

The NT32 implementation is radically different from a bare-metal Framework implementation. An instance of a hardware implementation can be found in Figure 7.10. Herein the memory-mapped registers of an Intel XScale system-on-a-chip (SOC) are accessed by the same AP set timer interface. The DXE Foundation cannot discern the difference between the virtual NT32 platform service and the actual hardware instance for an Intel XScale processor.

```
EFI_STATUS
TimerDriverSetTimerPeriod (
  IN EFI_TIMER_ARCH_PROTOCOL  *This,
  IN UINT64                   TimerPeriod
  )
{
 UINT64  Count;
 UINT32  Data;
. . .
 Count = DivU64x32 (MultU64x32 (TimerPeriod, OST_CRYSTAL_FREQ) + 5000000,
                    10000000, NULL);
 mCpuIo->Mem.Read  (mCpuIo,EfiWidthUint32,OSCR_BASE_PHYSICAL,1,&Data);
 Data += (UINT32)Count;
 mCpuIo->Mem.Write (mCpuIo,EfiWidthUint32,OSMR0_BASE_PHYSICAL,1,&Data);
 mCpuIo->Mem.Read  (mCpuIo,EfiWidthUint32,OIER_BASE_PHYSICAL,1,&Data);
 Data |= (UINT32)1;
 mCpuIo->Mem.Write (mCpuIo,EfiWidthUint32,OIER_BASE_PHYSICAL,1,&Data);
 mCpuIo->Mem.Read  (mCpuIo,EfiWidthUint32,ICMR_PHYSICAL,1,&Data);
 Data |= (UINT32)(1 << SA_OST0_IRQ_No);
 mCpuIo->Mem.Write (mCpuIo,EfiWidthUint32,ICMR_PHYSICAL,1,&Data);
. . .
}
```

Figure 7.10 AP from Intel XScale®

Finally, for the PC/AT and the circa mid 1980s ISA I/O hardware, there is an additional implementation of the AP service. Figure 7.11 shows the same set timer service when accessing the 8254 timer-counter and then registering an interrupt with the 8259 Programmable Interrupt Controller (PIC). This is often referred to as a PC/AT version of the AP since all PCs since the PC-XT have supported these hardware interfaces. For the PC example in this chapter, these ISA I/O resources are supported by the ICH component, versus discrete components in the original PC.

```
EFI_STATUS
TimerDriverSetTimerPeriod (
  IN EFI_TIMER_ARCH_PROTOCOL  *This,
  IN UINT64                   TimerPeriod
  )
{
 UINT64  Count;
 UINT8   Data;
. . .
 Count = DivU64x32 (MultU64x32(119318, (UINTN) TimerPeriod) + 500000,
                   1000000, NULL);
 Data = 0x36;
 mCpuIo->Io.Write(mCpuIo,EfiCpuIoWidthUint8,TIMER_CONTROL_PORT, 1, &Data);
 mCpuIo->Io.Write(mCpuIo,EfiCpuIoWidthFifoUint8,TIMER0_COUNT_PORT,2,&Count);
 mLegacy8259->EnableIrq (mLegacy8259, Efi8259Irq0, FALSE);
. . .
}
```

Figure 7.11 AP for PC/AT

Beyond the many implementation options for an AP to provide the breadth of platform porting, additional capabilities in DXE support various platform targets. In EFI, the interaction with the platform occurs through the input and output console services. The console input for a PC is typically a PS/2 or USB keyboard, and the output is a VGA or enhanced video display. But the I/O card studied earlier has no traditional “head” or display. These deeply embedded platforms may only have a simple serial interface to the system. Interestingly, the same PC hardware can also run without a traditional display and interact with the user via a simple serial interface. Figure 7.12 displays a console stack for an EFI system built upon a serial interface.

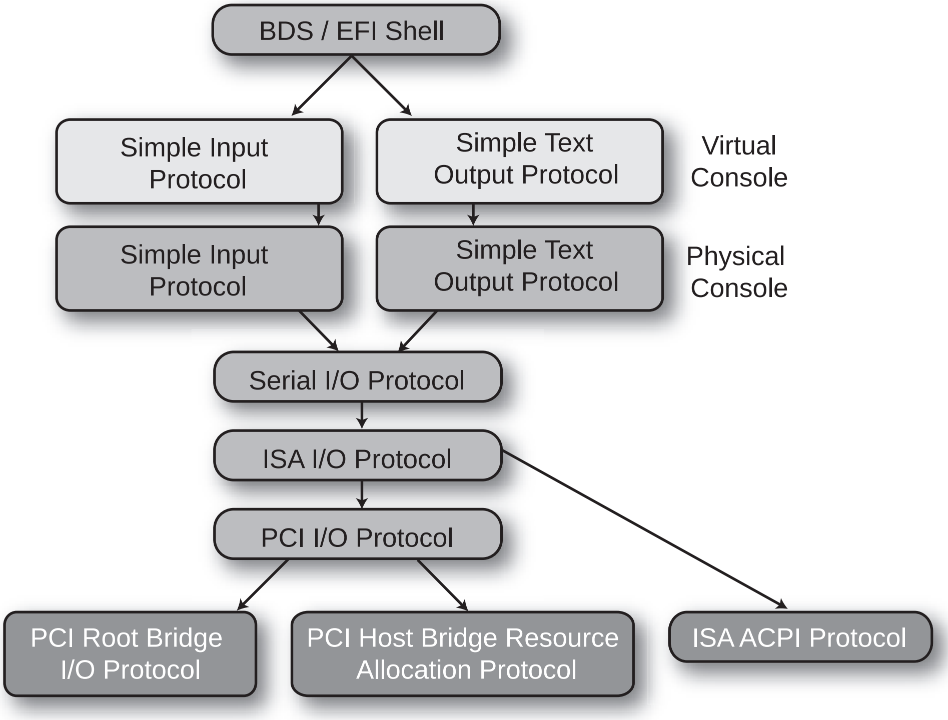
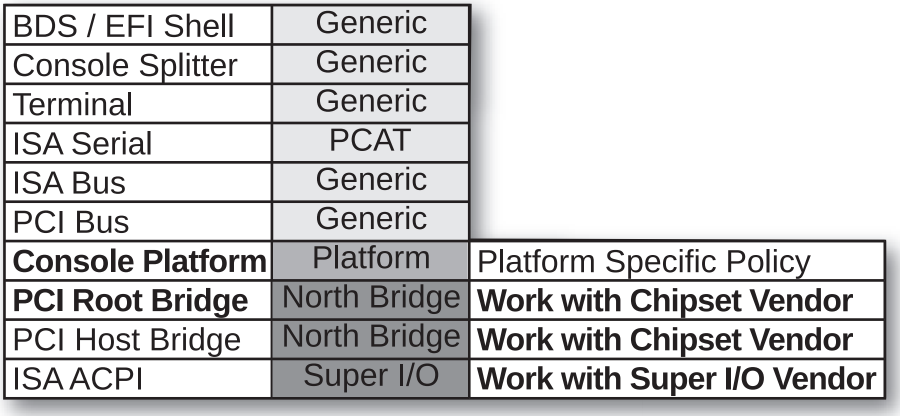


Figure 7.12 Console Stack on a PC

In order to build out this stack, the boot-device selection (BDS) or the EFI shell provides an application or command line interface (CLI) to the user. The Simple Input and output protocols are published via a console driver that layers upon the Serial I/O protocol. For the PCI-based PC, a PCI root bridge protocol allows access to the serial port control and status registers; for the Intel XScale platform with an internally-integrated UART/serial port, an alternate low-level protocol may exist to access these same registers.

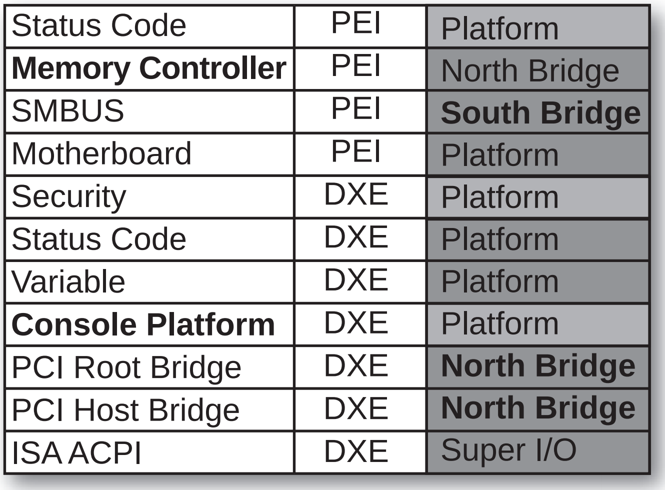
For this platform layering, the components listed in Figure 7.13 describe the DXE and EFI components needed to build out this console stack. Just as in the case of the PEI modules, different interests can deliver the DXE and EFI drivers. For example, the Super I/O vendor may deliver the ISA ACPI driver, the silicon vendor PCI root bridge (such as the GMCH in this PC), a platform console driver, and then a set of reusable components based upon the PC/AT ISA hardware.



BDS / EFI Shell	Generic	
Console Splitter	Generic	
Terminal	Generic	
ISA Serial	PCAT	
ISA Bus	Generic	
PCI Bus	Generic	
Console Platform	Platform	Platform Specific Policy
PCI Root Bridge	North Bridge	**Work with Chipset Vendor**
PCI Host Bridge	North Bridge	**Work with Chipset Vendor**
ISA ACPI	Super I/O	**Work with Super I/O Vendor**

Figure 7.13 Components for Console Stack

Beyond the console components, several other PEI modules and DXE components need to be included into the firmware volume. These other components, listed in Figure 7.14, provide for other capabilities. These include the platform-specific means by which to store EFI variables, platform policy for security, and configuration.



Status Code	PEI	Platform
Memory Controller	PEI	North Bridge
SMBUS	PEI	**South Bridge**
Motherboard	PEI	Platform
Security	DXE	Platform
Status Code	DXE	Platform
Variable	DXE	Platform
Console Platform	DXE	Platform
PCI Root Bridge	DXE	**North Bridge**
PCI Host Bridge	DXE	**North Bridge**
ISA ACPI	DXE	Super I/O

Figure 7.14 DXE Drivers on a PC

The EFI variables can be stored in various regions of the flash part (or a service processor on a server), so a driver needs to abstract this store. For security, the vendor may demand that field component updates be signed or that modules dispatched be hash-extended into a Trusted Platform Module (TPM). The security driver will abstract these security capabilities.

A final feature to describe the component layering of DXE drivers is support for the disk subsystem, namely the Integrated Device Electronics (IDE) and an EFI file system. The protocol layering for the disk subsystem up to the file system instance are shown in Figure 7.15.

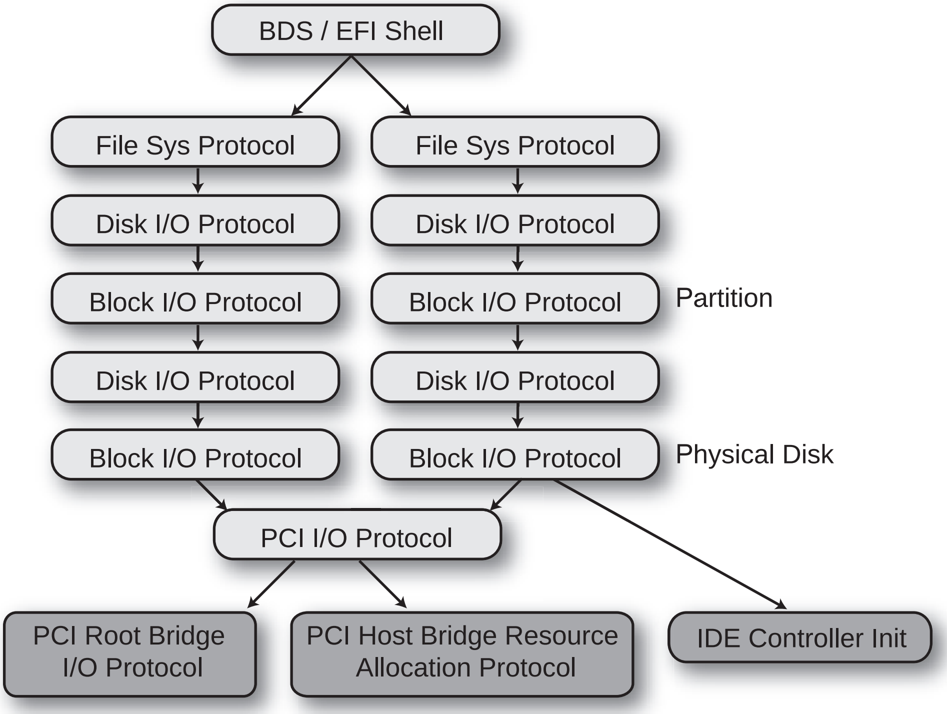


Figure 7.15 IDE Stack

The same EFI shell or BDS resides at the top of the protocol layering. Instances of the simple file system (FS) protocol provide the read/write/open/close capability to applications. The FS protocols layer atop disk I/O protocol. A disk I/O provides byte-level access to a sector-oriented block device. As such, disk I/O is a software-only driver that provides this mapping from hardware-specific block I/O abstractions. The disk I/O layer binds to a series of block I/O instances. The block I/O protocol is published by the block device interest, such as the ICH driver in DXE that abstracts the Serial AT-Attachment (SATA) disk controller in the ICH. The disk driver uses the PCI Block I/O protocol to access the control and status registers in the ICH component.

The components that provide these capabilities in the file system stack can be found in Figure 7.16. The file system components include the File Allocation Table (FAT) driver, a driver that provides FAT12/16/32 support. FAT is the original file system for MS-DOS on the original PC that has been extended over time, culminating in the 32-bit evolution of FAT in Windows95 as FAT32. In addition, providing different performance options of the storage channel can be abstracted via the IDE Controller Initialization component. This provides an API so that a platform setup/configuration program or diagnostic can change the ICH settings of this feature.

BDS / EFI Shell	Generic	
FAT	Generic	
Partition	Generic	
Disk I/O	Generic	
IDE Bus	PCAT	
PCI Bus	Generic	
PCI Root Bridge	North Bridge	
PCI Host Bridge	North Bridge	
IDE Controller Init	**South Bridge**	IDE Channel Attributes

Figure 7.16 Components for IDE Init

This same console stack for the serial port and file system stack for the SATA controller only depends upon the ICH components, a PCI abstraction, and appropriate support components. As such, putting this same ICH, or a logically-equivalent version of this chip integrated into another application-specific integrated circuit (ASIC), will admit reusage of these same binaries on other like systems (such as an IA32 desktop to an IA32 server). Beyond this binary reuse across IA32 platform classes, the C code allows for reuse. The use of this ICH, whether the literal component or the aforementioned logical integration, on the Itanium Processor , can occur via a recompilation of the component C code with the Itanium Processor as the target for the binary.

Chapter 8

Volumes, Files, and Sections

Number 6: *What do you want?*
Number 2: *We want information.*
Number 6: *Whose side are you on?*
Number 2: *That would be telling. We want information... information... information.*
Number 6: *You won't get it.*
Number 2: *By hook or by crook, we will.*

—Patrick McGoohan, *The Prisoner*

The *file* is one of the most basic concepts in computing. The concept of a named array of data stored on some media is basic to most users' and indeed most programmers' concept of how computers work.

Files are so fundamental that most programmers really don't give them much thought. If they do, they think in terms of creating, reading, writing, and deleting files. They know that you can store data in files, usually treat them as linear vectors of data, and that they are nonvolatile.

Users and programmers also know that files are stored on hard drives, floppies, thumb drives, and other media. Firmware programmers are rarely granted such traditional devices to store their programs and data. So, it is not commonly the case that firmware stores its data using the traditional concepts of files. In the Framework, however, files for program and data storage are a fundamental construct. It is important to note that this usage of files and file systems is *not* the same as that used in EFI for managing the boot partition.

The Framework defines the basic mechanisms for the files it uses based on operating system principals. The targeted environment, however, molds the typical concepts of files into some unusual ways.

Terminology

There seem to be as many terms for the various concepts surrounding files as there are operating systems. The following terms are used in the Framework. It is important to note that, while the name may be the same as in your favorite (or not-so-favorite) operating system, the functionality is probably not the same.

Firmware Device (FD): Any physical device or set of devices that can store firmware. In the case of multiple devices, the devices are abstracted to act as a single device.

Firmware Volume (FV): A logically linear partition of a single Firmware Device formatted as a volume. This specifically prohibits an FV from spanning FDs (although remember, an FD can be made up of multiple devices). Unlike most operating systems, it is valid for all FVs in an FD to take up only part of the available space in that firmware device. It is also valid for the location of FVs inside an FD to be predefined at build time and then described by drivers. Put another way, it isn't a requirement that volumes be discoverable—their location may simply be known ahead of time and be told to the driver stack by other drivers. For example, drivers in FV0 can describe other FVs. FV0 is, by definition, known since it is the target of the boot vector. It is always important to know where your home is. The major operation that occurs with a firmware volume is that it is discovered (typically by a driver or by a PEIM) and the support infrastructure for it is built up.

Firmware File (File): A named collection of information stored inside firmware volumes. The file names in the Framework are (almost inevitably) GUIDs. The *File Name GUIDs* must be unique to the FV but different FVs may have files with the same name. As such, it is required to refer to a file by an (FV, File) pair. This requirement is quite similar to the rules for files in most operating systems. Files are typed with the types serving much the same function as extensions (such as .TXT, .H) in operating systems, although the requirement is more strict in the Framework in many cases—the DXE dispatcher will only dispatch files of a certain type for example. Files appear at a single level in FVs—there are no subdirectories. Files may, however, contain FVs in them, which serves a similar purpose. This recursive mechanism also provides a cleaner way to deliver encapsulated modules consisting of possibly a large number of files as a single unit. Operations commonly accomplished on files include scanning an FV for certain file types, opening files, reading files, and closing files.

Section: A sub-partition of a file. Sections are typed but not named. There are two kinds of sections: container sections (also known as encapsulation sections) and leaf-node sections (or just leaf sections). Containers contain other sections whereas leaf-node-sections do not. Container sections support features such as compression and security. Typical operations for sections include scanning a file for sections of a particular type and reading those sections.

These pieces form a strict hierarchy. As such they can be viewed as nodes in a tree, as shown in Figure 8.1 (A). What seems to be a more useful analogy for most people is to think of the components as a nested series of containers and it is this analogy that will be predominant in the discussions that follow, as shown in Figure 8.1 (B). The rest of this chapter is a discussion of how these pieces interact and how they are used, accessed, and implemented.

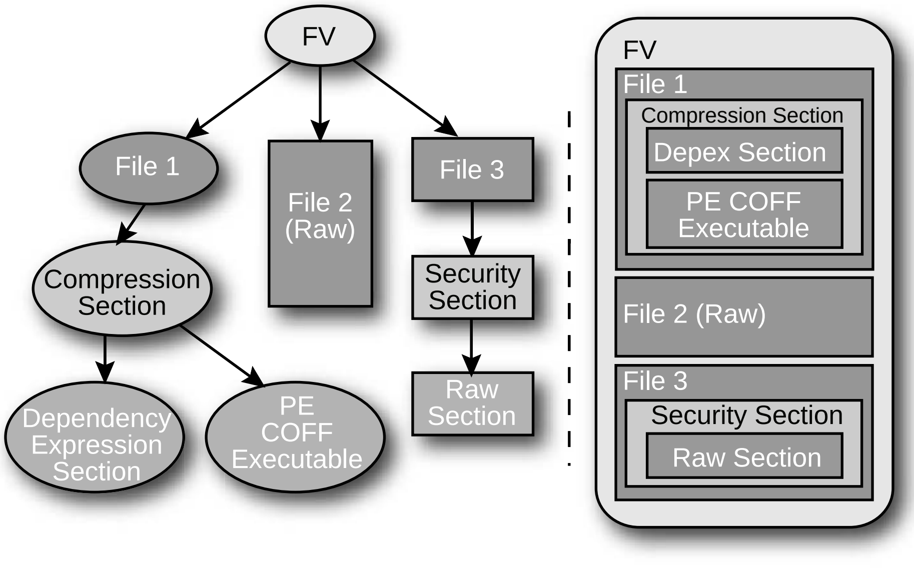


Figure 8.1 Hierarchical Views of FVs

Design Constraints

The mechanisms by which firmware is stored and accessed are strongly affected by the way firmware is used and some of the media in which firmware is likely to be stored.

The Boot Vector

The boot vector points to a firmware device at a fixed address. The file system must address the ability to keep a file or other construct at a fixed address to ensure that the system has some place to boot from. More complex processors do not use a traditional boot vector but still assume at least one fixed location, or more probably, several fixed locations in the boot firmware device (known as FD0).

Interestingly, investigations during design and experience subsequently, have shown that the boot vector is the only address required externally to be known. Initial design efforts targeted at allowing an arbitrary number of fixed address files proved unnecessary and added extreme complexity. Some unique cases require separate handling.

Alignment

While there are limited requirements for the exact position of files in volumes, there is ample requirement for ensuring that *the data in a file* appear on a particular address boundary. For example, some computer architectures either require, or operate more efficiently, when execution starts on a 16-byte boundary.

Easy Access

Files must be located and read starting in the PEI phase, a phase that is designed with the expectation of only limited RAM and other resource availability. This means the internal file storage format must be simple enough for PEI to scan through without complex calculations or building complex data structures.

Rare Writes

Writes to firmware devices are much less likely than their operating system counterparts and tend to be much more isolated. In the case of EFI, most writes during power-on or runtime are focused on variables. In fact, as detailed later, there are good reasons to consider storing variables outside the normal file system.

Flash Headaches

The devices in which firmware is stored pose their own set of complexities. Initially, boot firmware was stored in Read-Only-Memory chips (ROMs). These could not be written and generally had no partition scheme.

Today, most firmware is stored in flash devices. These devices are random access read devices but may only be generally written as sectors called blocks although they may be read randomly. At first glance, the write limitation may seem no more onerous than that imposed on hard drives, where only sectors may be written.

Unfortunately, compared to the size of sectors in hard disks (or even floppies), blocks are typically quite large relative to the size of the devices. For example, some 512 kilobyte parts have only 8 blocks. (Imagine managing a 1.44MB floppy with 24 sectors rather than around 2900.) More typical are 1MB parts with 4KB blocks. This comparatively large block size relative to part size leads to problems using flash efficiently. In particular, the traditional operating system practice of storing files as integral numbers of sectors becomes impractical. First, the number of files becomes limited by the number of blocks and, more importantly, the internal fragmentation suffered by incomplete use of blocks leads to inefficient use of the flash, as shown in Figure 8.2.

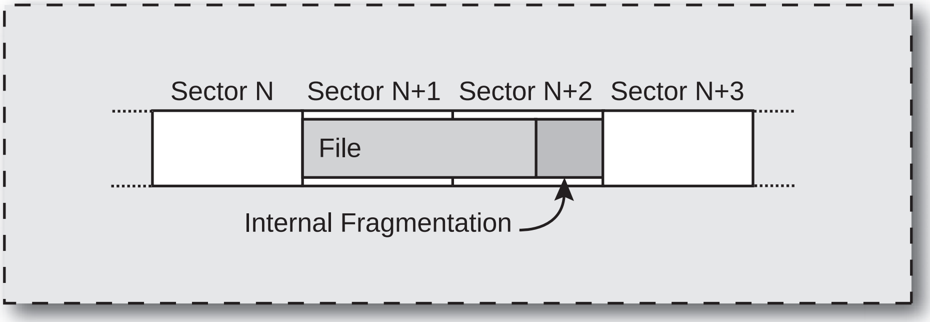


Figure 8.2 Internal Fragmentation

The pressures to use flash efficiently are higher than might otherwise be expected due to flash's high per-bit cost. The smaller flash parts are, per bit, some of the highest cost memory available today. The incentive is therefore to apply otherwise unusual techniques for using flash efficiently. Perhaps the most significant difference between an operating system's file system and the Framework's is that the Framework's file system's lack of any alignment between files and blocks.

With EFI's rich execution environment, it is more likely that drivers and applications will arrive from nontraditional firmware locations including via the network and from "hidden" partitions on hard drives. Files from these locations, however, are not managed via the firmware file protocols and are not further discussed in this chapter.

Bank Switching

Flash does not always appear linearly in memory. It is quite common in servers and other systems with large amounts of flash for at least some of the flash to be bank switched, as shown in Figure 8.3. An I/O port is used to select which piece of the flash appears (is decoded) in a window in the memory space. It is not easily possible to execute out of any but the default bank in bank switched parts so PEI abstractions do not support this. In DXE, devices are abstracted to the point that switching takes place behind the scenes.

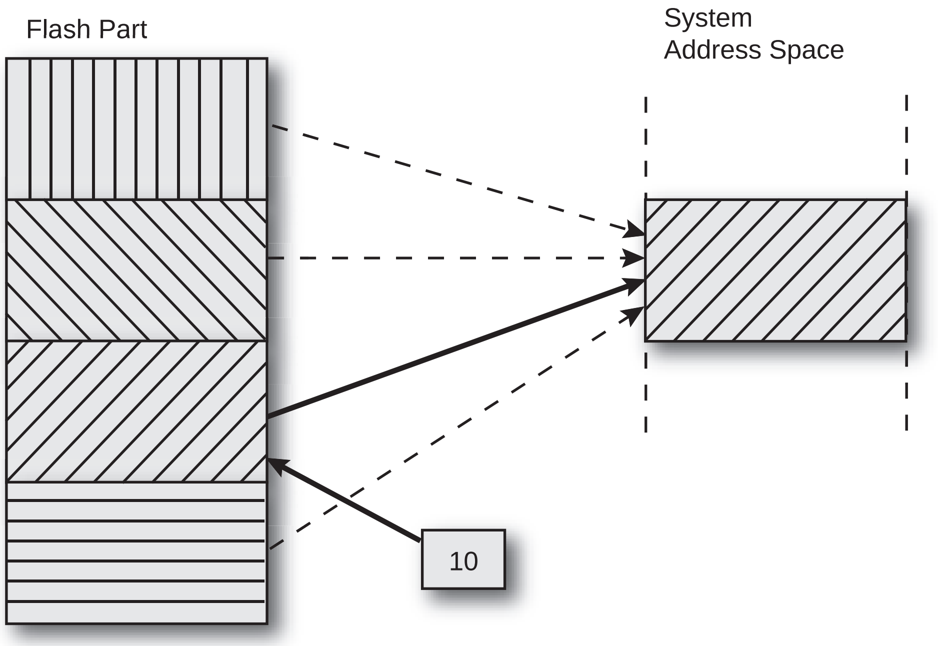


Figure 8.3 Bank Switching

Fault Tolerance

If certain parts of the firmware become corrupted, the system will not boot until the flash part is unsoldered and a new programmed part soldered in its place. Flash can break like any other component in a computer.

The most common point for corruption to occur, however, is during update. When FV0 requires update, the flash (or other media) blocks it is stored in must first be erased and then written with the new data. If the power is lost during this process, the system is left in a dead state. Note that it is generally the case that losing any of the files in FV0 kills the system. File-level fault tolerance is therefore not sufficient—volume-level fault tolerance is required.

When we refer to fault tolerance for firmware volumes, we refer to the ability for the system to at least recover from corruption during update caused by external sources, usually loss of AC power.

There are a number of algorithms for fault tolerant update. It is possible to duplicate the contents of the FVs and alternate between the two copies. The algorithm is thus to update the unused half of the flash and then select that as the now used side, as illustrated in Figure 8.4.

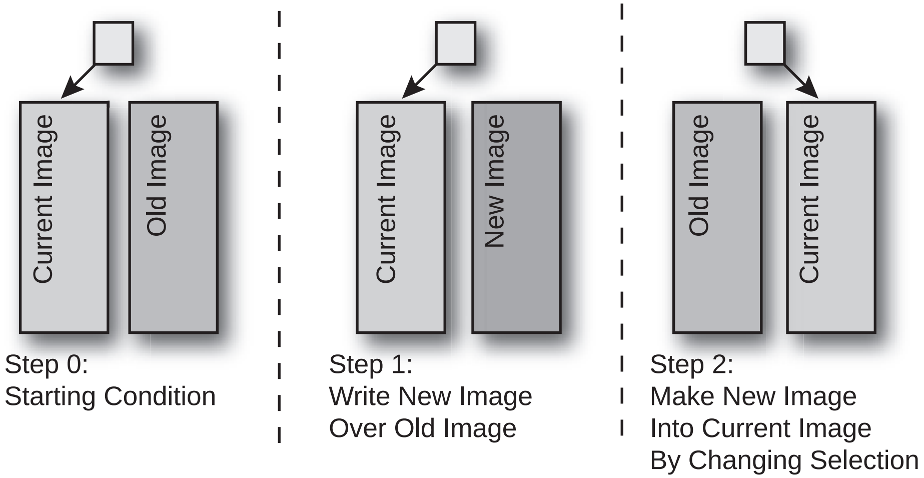


Figure 8.4 Dual Image Update

For a less expensive solution, certain chipsets, such as the Intel ICH7, support a feature known as TOP_SWAP. Upon selection, bit 16 of the address line to the top 128KB of memory is inverted. This has the effect of making the top 64KB the next lower 64KB and decoding the next to top 64KB as the top 64KB, as shown in Figure 8.5.

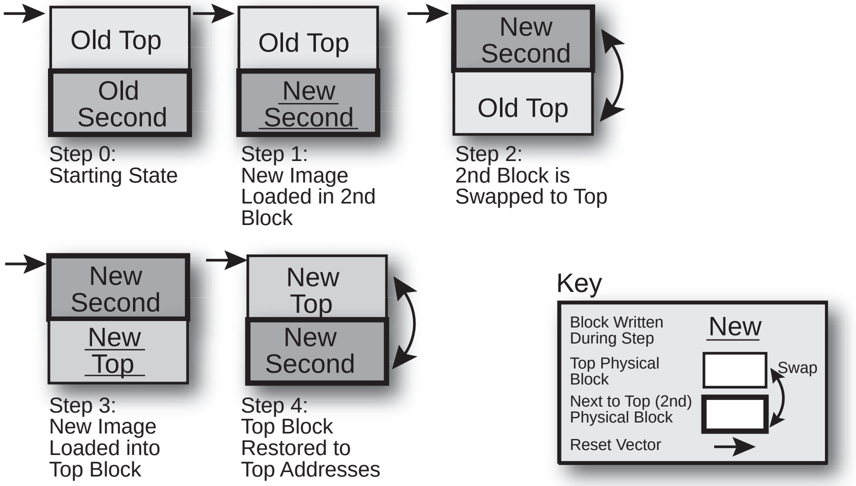


Figure 8.5 Top Swap Update

To File or Not to File

One of the most uncharacteristic adjustments to the traditional operating system-based concept of file management is that all of the space managed in the media is not required to be managed in volumes. It is valid, for example, to support EFI Variables in the same FD as the file systems without encompassing the variable storage in a firmware volume.

This is not a weakness of the FV system. It is intentional. Instead of trying to make a single mechanism fit all needs, it makes more sense to support alternatives where possible. Variables have attributes quite different from most of the data in FDs: They are required to be (basically) nonvolatile, they are written relatively frequently, and the writes are comparatively of quite small size (tens to hundreds of bytes).

It is not impossible to use the file system to support variables and several successful implementations have supported variables this way. It is however, not a requirement. It is up to the system designer to determine the proper way to support the system's requirements.

The one area where the use of the file system as described here is required is for the root firmware volume, also called FV0 and for other firmware volumes accessed during the PEI phase. This is because the PEI dispatcher accesses the files directly and therefore needs a consistent storage layout.

File Types

File types are used in the Framework for much the same reason they are used in operating systems: to divide an unordered assortment of files into… several unordered assortments of files.

Several files are special to the framework itself and so have reserved types, such as the PEI "core," DXE loader, and DXE core. There are types to indicate executable files, PEIMs, DXE drivers, and EFI applications.

As noted previously, the Framework's concept of volumes doesn't allow for subdirectories. Instead, it uses files that can contain volumes. This enables some features that a hierarchical file system doesn't comprehend. See the section "Capsules" later in the chapter for an example.

Files which do not conform to any architecturally defined set of requirements may be of two types: freeform and raw. Freeform files are files with sections inside them where as raw files are just binary blobs of data. The format and use of both of these is left to the definition of the intended users.

The Firmware Volume Protocol GetNextFile allows for scanning through a volume for files of particular types (or any type).

Section Types

Most Framework files contain sections. A large number of section types define the various elements that may be contained in files. Analogous to GetNextFile, the Firmware Volume Protocol provides GetSection to scan through sections.

Sections fall into two categories, encapsulation sections and leaf sections. Encapsulation sections have other sections inside of them, leaf sections don't.

In implementation, sections are designed to be quite inexpensive, with only a type (byte) and length being required. Some sections have an extended header (particularly those with GUIDs).

Common Encapsulation Section Types

Volumes cannot be intrinsically compressed in the FFS nor can files. Instead, a compression section is provided allowing the encompassed sections inside to be stored compressed. The compression section header describes how to find a corresponding protocol with defined interfaces to decompress the contents. A common way of putting together a volume intended for DXE is to wrap the volume with a compression section and then store the results in a volume file.

An alternative section is a GUIDed section, which may be used generically to associate a section with an arbitrary protocol. The section header includes an application-specific GUID. A protocol with a defined interface (known as a GUIDed Section Extraction Protocol) and named with the same application-specific GUID as the section, must be defined prior to encountering the section. The protocol is then invoked when the section with the corresponding GUID is encountered.

As an (already implemented real-world) example, the GUIDed type may be used to provide digital signature validation of the sections inside it. The GUID corresponds to the GUID of a protocol, which allows the DXE dispatcher to determine if any executables contained there-in should be queued for dispatch.

Common Leaf Section Types

At last count, over ten leaf section types have been defined. They fall into several categories:

- One leaf type exists for each of the supported executable formats: PE32+ image, Position-independent code (PIC) image, and Terse Executable (TE). PIC code is already relocated code that may be executed anywhere (subject to alignment restrictions). A special section type is reserved for Compatibility16 (CSM) executables.
- Executable file types may also have DXE dependency expression and PEI dependency expression sections.
- Although not required, version and file name sections are defined allowing files to be managed in a more user friendly manner.
- Firmware volume image is the section type to store a volume inside a file.
- Free-form subtype GUIDed and raw sections are analogous to their equivalents in files.

Operations on Files

As well as the file and section location functions already discussed, the Firmware Volume Protocol provides services one might expect from a file interface: Get Attributes (to read a volume's current status), Set Attributes, Read File, and Write File. Each instance of the protocol corresponds to a firmware volume. The normal protocol services may thus be used to scan for firmware volumes.

How the Dispatcher Uses Volumes, Files, and Sections

The major job of the DXE dispatcher is to find and dispatch drivers. The dispatcher uses the Firmware File infrastructure to accomplish these activities. In doing so, it provides one of the more complex examples of how FVs, files, and sections are used.

Each FV known to the system is represented by a protocol. The dispatcher uses this fact to key its search for new volumes by simply waiting for new versions of the protocol (via the EFI `RequestProtocolNotify` service). Volumes discovered in PEI are described via HOB and are processed when the dispatcher initializes itself.

When the Dispatcher is notified of the appearance of a new volume, it uses the Firmware Volume Protocol to scan the volume for files with the type `DXE Driver`. When it finds these files, it processes each by searching for a `Dependency Expression` section. It uses these sections to apply a dispatch order to the drivers.

When the Dispatcher decides to dispatch a driver, it reads the `PE/COFF Image` section from the file corresponding to the dependency expression. It can then dispatch the driver.

Internal Format

Although the Framework is designed to support an arbitrary number of file systems, it must rely on a single file system early in boot so that it may find the modules that tell it how to access the other file systems. This is known as the Framework File System format, or FFS.

Basic Format

A firmware volume is defined as a linear number of blocks containing a collection of files. Files have a header, a body of data, and a "tail." The firmware volume starts at its lowest address with a header, which contains a GUID and attributes that define the size of the volume and other pertinent data. The volume header starts at the lowest location so that it may be easily and unambiguously located.

Files are stored contiguously in the volume starting just after the volume header. The file header contains the name (GUID), size, type, attributes, integrity check, and state. The size indicates the size of the file.

The integrity check allows the file system to determine if the file has been corrupted. Attributes contain various flags associated with the file. Most important of these is the alignment of the data part of the file (not the header). This in essence indicates the number of bytes to skip beyond the header so the data body starts on a boundary. This is required for certain types of files that map directly to hardware. Eleven alignments are available encompassing most powers of 2 between 1 byte and 64K bytes.

The state is used as a part of the update operations. An update operation is essentially a delete and a write. The goal of the state operation is to provide these operations atomically. That is, at any state during the update, either the old file or the new file is valid. This implementation assumes that the media allows bits to be flipped from one value to the other (such as from 1 to 0) easily, that is, without erasing the entire block, which is correct on known parts today. The erase state of the FD is known as its *polarity*. An FD's polarity is stored in its header.

Managing the FV.

Files are allocated starting at the bottom (lowest address) in the volume and are concatenated one to the next (allowing for alignment) to the topmost file. New files are then written in the empty space above the most recently written file.

Deleted files do not immediately cause the files above to be moved down. Files are marked for deletion by changing bits in the file header from the erase state to the non-erase state.

A *coalesce* operation occurs when there is not enough space left for a write request. This is analogous to early (1960s vintage) file systems and is used for much the same reasons: simplicity, expected size constraints, and low block to volume ratio.

No single method is specified for coalescing FVs. In choosing a mechanism, the designer must determine available resources and requirements for fault tolerance, among other factors. A comparatively simple model is to copy the non-deleted files to a memory buffer creating a new image of the FV. The FV is then erased and the image written. This is not a fault-tolerant solution but is quite acceptable for many applications.

Another mechanism is to keep two copies of the firmware volume, ping-ponging between the two. This is fault-tolerant (assuming care is taken so drivers can tell which volume is newer) but expensive. Developer imagination can define a number of other schemes.

The FFS state data may be used for a safer update/write mechanism.

Below the File System

The drivers that support operating system file systems typically map devices whose media is represented as blocks into the files themselves. The Framework is no different.

The Driver Stack

A firmware volume is manifested within the system as a Firmware Volume Protocol. Any file system may manifest this protocol and thus create a new firmware volume.

The Framework FFS defines a set of interfaces that allow a driver stack, shown in Figure 8.6, to be created so that the driver(s) that manifest the firmware volume stack do not have to be carried for each new firmware volume.

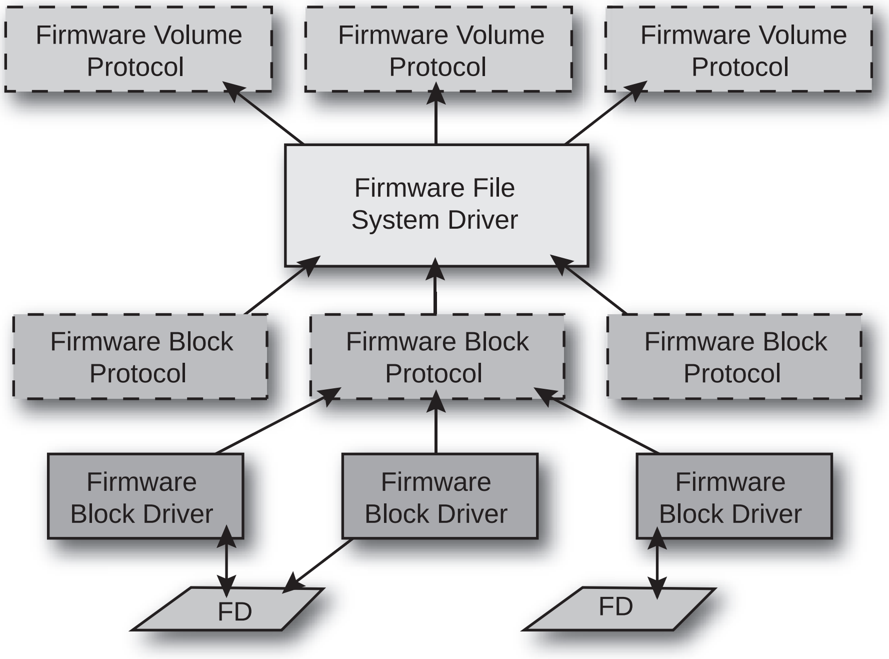


Figure 8.6 FFS Driver Stack

The architecture does not have any predefined mechanism to discover a new FD. Instead, a DXE driver familiar with the hardware must either be written to "know" about a firmware device or have some mechanism to discover it.

Once it is discovered, the driver must provide an interface called a Firmware Block Protocol. The FFS driver monitors for the creation of this protocol. Triggered by its creation, the FFS driver creates the Firmware Volume Protocol. Triggered by that the Firmware Volume Protocol, the dispatchers (PEI and DXE) search for modules to execute.

The Framework may be told about new devices in PEI or DXE. The PEI method is simpler. The devices are memory mapped and must be read-only. As such, the interface is essentially providing the base address of the device.

Firmware Volume Block Interfaces

Continuing the theme in this chapter of Framework interfaces being similar to operating system interfaces, the FVB interfaces are fairly similar to those one might expect for an operating system disk driver except for the substitution of *block* for *sector*.

The FVB interface defines seven interfaces:

- **Attributes**: The GetAttributes and SetAttributes functions perform analogous actions to their FV counterparts. The attributes in question manage read and write protection and locking and alignment. GetBlockSize gets the sizes of the blocks within the volume. GetPhysicalAddress allows drivers to determine the physical location of the volume in the address space, if one exists.
- **I/O Operations**: Read and Write are the basic functions to allow the devices to do their jobs.
- **Erase**: EraseBlocks erases blocks as a part of the coalesce operation.

Variables

The EFI specification defines required support for Variables, a construct intended to be similar in access and function to the environment variables that gained popularity in Unix and now appear in most operating systems. Environment variables are shell-level named data stores that are mainly used to set defaults for the system and applications.

EFI's variables are named (inevitably) with a GUID and a UNICODE name (both are significant) and may be used to store an arbitrary amount of data. The EFI specification does not define the amount to be stored.

The essential difference between Variables (using capitalization to disambiguate from C variables, for example) is that most FVs are written rarely whereas Variables may be expected to be written more often (in some sense, during each boot, since a monotonic count is required and is expected to be stored as a variable).

Implementation of variables can take two basic forms, as noted above. Either the variables can be stored in an FV or they can be stored outside of it.

Even if variables are stored inside FVs, the FV may be managed completely differently from other FVs in the system. It may indicate that it has no files and is full, using the FV mechanisms simply for location and management. The FV may be backed up in ways the other blocks aren't since the EFI specification requires a level of fault tolerance on variables but is silent about the executables supporting that data storage.

Variables can be implemented in what must be close to an infinite number of variations. Two of the simpler involve using a block for variable storage and allocating another block for backup. In the first case, the two blocks alternate being the "active" and "backup" block. In the other, a single block is designated as the "active" block and the other is used to backup the variable data while the active block is being updated. If the active block is found to be in the middle of update, the backup block's contents are copied to the active block.

Capsules

A *capsule* is a mechanism that allows the operating system to provide data to be used by the next pre-boot. The motivation for developing capsules was the need to create a mechanism to update firmware, but nothing in the infrastructure makes this a requirement. In fact, the UEFI 2.0 specification envisions that capsules be used to send data from an operating system that believes that it is no longer stable back through a reboot to a hopefully more stable version of itself for crash reporting and analysis.

The Structure of a Capsule

A capsule consists of one or more headers and a firmware volume. Only the firmware volume makes the journey through the reboot. The headers are intended to enable the creation of generic capsule processing programs. As such, they include entries such as short and long UNICODE descriptions of the contents of the capsule (which can be localized into an arbitrary number of languages) and a mechanism that allows for the capsule to be delivered in several pieces and then reconstituted.

Importantly for use with add-in devices, the capsule includes a device description, which is the counterpoint to a device path: a description of the device that does not include the path. This is necessary since the developer of the update cannot rely on knowing the slot location of the card carrying the driver option ROM to be updated. The device descriptor identifies the device so that the update program can associate it with the one-or-more applicable devices in the system being updated.

The update application is responsible for describing the devices to be updated via a file stored in the capsule firmware volume (CFV) in a known format.

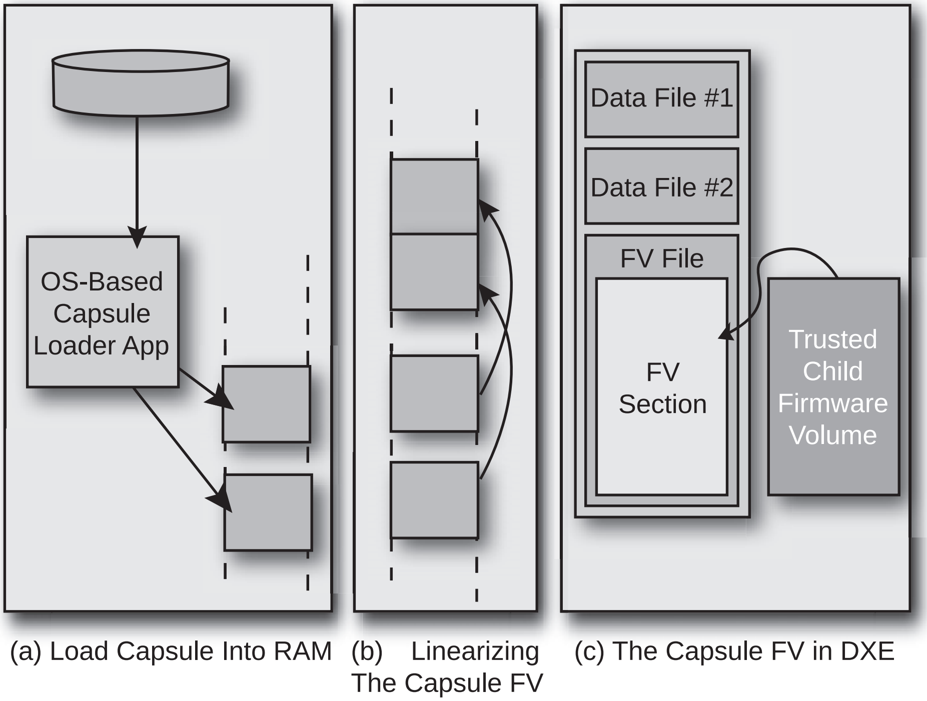

OS-Based Capsule Loader App
Data File #1
Data File #2
FV File
FV Section
Trusted Child Firmware Volume
(a) Load Capsule Into RAM
(b) Linearizing The Capsule FV
(c) The Capsule FV in DXE

Figure 8.7 Capsule Loading

Passing through Reset

Once stripped of its header, the CFV is stored in non-swapped memory (to ensure that it will exist in RAM at reboot), as shown in Figure 8.7 (b). Non-swapped memory is readily available in most operating systems for use as DMA memory from hardware such as LANs, USB controllers, and disk host adapters. If the operating system is secure enough, it must also be told not to erase this RAM before reboot. Since it may not be possible to allocate enough contiguous non-swappable memory, capsules define a mechanism for storing the capsule as a list of noncontiguous chunks. The same mechanism is also used to support passing more than one capsule to the pre-boot space at a time.

The Framework defines an EFI Reset type to allow passing of the base physical address of the capsule data structure. UEFI 2.0 defines a set of runtime services to manage capsules. Both methods work and can easily be made not to interfere with one-another.

When the request is made to do the reset, control of the entire system is passed from the operating system back to the EFI firmware. For this reason, the operating system is expected to perform operations equivalent to what it would do in an S0 to S5 ACPI transition (a soft reset with no RAM data retained). The firmware now owns the entire system. It must only preserve the capsule memory through a reset. Note that in extremely high-end high-availability systems, a full reset may not be required. Implementation generally requires operating system cooperation and is beyond the scope of this book.

The method by which the reset occurs is not architectural. In systems that support the ACPI S3 ("save to RAM") state, this is an obvious method, particularly if there is a mechanism to preset a wake event so the reset is virtually instantaneous. It is certainly not the only one.

The method by which the address of the capsule data structure is passed is also not architectural. Since the Framework generally obviates the need for the real-time clock CMOS, examples here assume the address is stored there. For example, a value of less than 0xFF would mean that no capsule was present, and that instead the value contained status data. Any other value is treated as an address. Again, there are many other mechanisms.

On the Other Side of Reset

Assuming that the S3 mechanism is being used for reset and the CMOS method to store the address of the capsule, the reboot continues until after RAM has been restored exactly as if it were a normal S3. If the address is less than 0xFF, the reset is treated as a normal S3.

Assuming an address is found, the structure addressed is validated. If it is invalid, a status is reported and an S5 equivalent reset is instituted.

The most complex part of the implementation then proceeds during PEI. This is reconstituting the possibly scattered pieces of the capsule into a linear whole. An implementation starts with the tricky job of finding an area of memory large enough for the capsule that is not occupied by any of the capsule. The pieces of the capsule are then copied into the free RAM. This RAM is then reserved and reported to DXE via a HOB.

As the capsule firmware volume resides only in RAM, accessing it requires a stunted FV Block driver that can provide the interfaces to the FFS to create the FV protocol. It also requires

The Capsule in DXE

On first glance, capsules seem to be a method for the operating system to infect the pre-boot space with viruses. Without precautions, this would indeed be the case. Some of the features of the firmware volume hierarchy and the Framework's security hierarchy are used to resolve this, as shown in Figure 8.7 (c).

The DXE driver responsible for managing the capsule marks the volume as untrusted, meaning that the DXE dispatcher will not execute from it. By clever design, one (or more) of the files in the capsule is a firmware volume file. The firmware volume file is wrapped by a GUIDed section. The driver that publishes the corresponding GUIDed section extraction protocol then validates the contents of the volume. If valid, it sets the attributes of the smaller volume to be trusted. Executables can then run.

When the executables run, they can find the volume they came from and, in this case importantly, its parent. The parent volume is the CFV itself and contains the data files possibly written by the update application.

Validation is not required on data-only capsules, such as those used by the operating system to communicate with the next reboot of itself since data-only capsules have no executable code. They are just shuffled up the chain.

It is interesting to note how little extra code is required to enable capsules inside the Framework. Implementation includes a small amount of runtime reset processing, special casing during the S3, the code required to coalesce the capsule, and the driver to make the FVB protocol for the capsule. The verification code rounds out the requirements. All of the heavy lifting is then done by the firmware volume services as they would for any internal volume. Experience indicates this at around 4KB uncompressed when compiled for a 32-bit Intel® Pentium® 4. That amount is saved by the fact that the update code can be carried inside the capsule rather than having to live in the flash.

Chapter 9

DXE Basics: Foundation, Dispatching, and Drivers

I do not fear computers. I fear the lack of them.

—Isaac Asimov

This chapter describes the makeup of the Driver Execution Environment (DXE) and how it operates during the platform evolution. In addition, it should describe some of the fundamental concepts of how information is handed off between phases of the platform boot process and how the underlying components are launched. The launching description should also provide some insight into how launch orders are constructed, since they do deviate from what is commonly referred to as POST tables in legacy firmware.

The DXE phase contains an implementation of EFI that is compliant with the *EFI 1.1 Specification*. As a result, both the DXE Foundation and DXE drivers share many of the attributes of EFI images. The DXE phase is the phase where most of the system initialization is performed. The Pre-EFI Initialization (PEI) phase is responsible for initializing permanent memory in the platform so the DXE phase can be loaded and executed. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position-independent data structures called Hand-Off Blocks (HOBs). The DXE phase consists of several components:

- DXE Foundation
- DXE Dispatcher
- DXE Drivers

The DXE Foundation produces a set of Boot Services, Runtime Services, and DXE Services. The DXE Dispatcher is responsible for discovering and executing DXE drivers in the correct order. The DXE drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for console and boot devices. These components work together to initialize the platform and provide the services required to boot an OS. The DXE and Boot Device Selection (BDS) phases work together to establish consoles and attempt the booting of operating systems. The DXE phase is terminated when an OS successfully begins its boot process—that is, when the BDS phase starts. Only the runtime services provided by the DXE Foundation and services provided by runtime DXE drivers are allowed to persist into the OS runtime environment. The result of DXE is the presentation of a fully formed EFI interface.

Figure 9.1 shows the phases that a platform with Framework firmware goes through on a cold boot. This chapter covers the following:

- Transition from the PEI to the DXE phase
- The DXE phase
- The DXE phase's interaction with the BDS phase

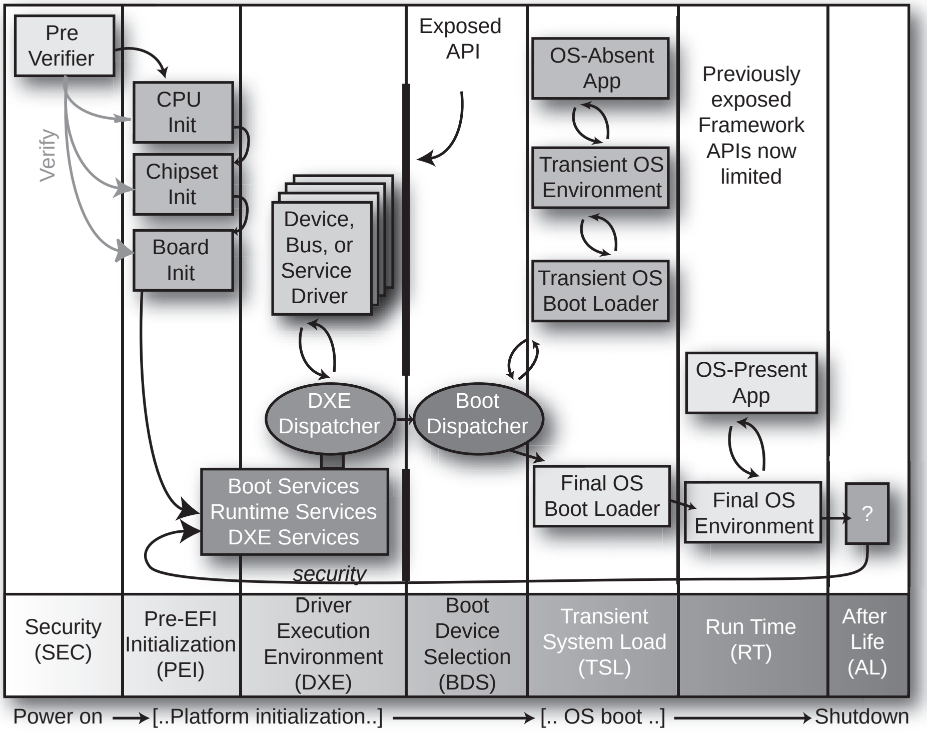


Figure 9.1 Framework Firmware Phases

DXE Foundation

The DXE Foundation is designed to be completely portable with no processor, chipset, or platform dependencies. This portability is accomplished by incorporating several features:

- The DXE Foundation depends only upon a HOB list for its initial state. This single dependency means that the DXE Foundation does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE Foundation.
- The DXE Foundation does not contain any hard-coded addresses. As a result, the DXE Foundation can be loaded anywhere in

physical memory, and it can function correctly no matter where physical memory or where firmware volumes are located in the processor's physical address space.

- The DXE Foundation does not contain any processor-specific, chipset-specific, or platform-specific information. Instead, the DXE Foundation is abstracted from the system hardware through a set of architectural protocol interfaces. These architectural protocol interfaces are produced by a set of DXE drivers that are invoked by the DXE Dispatcher.

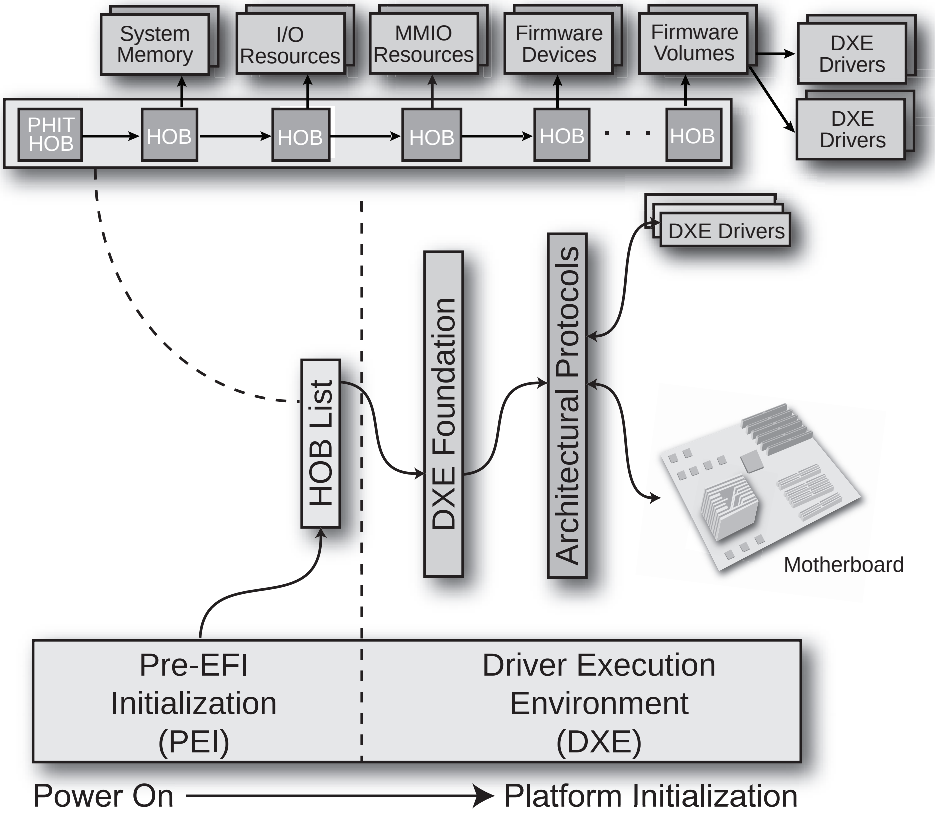


Figure 9.2 Early Initialization Illustrating a Handoff between PEI and DXE

The DXE Foundation produces the EFI System Table and its associated set of EFI Boot Services and EFI Runtime Services. The DXE Foundation also contains the DXE Dispatcher, whose main purpose is to discover and execute DXE drivers stored in firmware volumes. The order in which DXE drivers are executed is determined by a combination of the optional a priori file (see the section on the DXE dispatcher) and the set of dependency expressions that are associated with the DXE drivers. The firmware volume file format allows dependency expressions to be packaged with the executable DXE driver image. DXE drivers utilize a PE/COFF image format, so the DXE Dispatcher must also contain a PE/COFF loader to load and execute DXE drivers.

The DXE Foundation must also maintain a handle database. A handle database is a list of one or more handles, and a handle is a list of one or more unique protocol GUIDs. A protocol is a software abstraction for a set of services. Some protocols abstract I/O devices, and other protocols abstract a common set of system services. A protocol typically contains a set of APIs and some number of data fields. Every protocol is named by a GUID, and the DXE Foundation produces services that allow protocols to be registered in the handle database. As the DXE Dispatcher executes DXE drivers, additional protocols are added to the handle database including the DXE Architectural Protocols that are used to abstract the DXE Foundation from platform-specific details.

Hand-Off Block (HOB) List

The HOB list contains all the information that the DXE Foundation requires to produce its memory-based services. The HOB list contains information on the boot mode, the processor's instruction set, and the memory that was discovered in the PEI phase. It also contains a description of the system memory that was initialized by the PEI phase, along with information about the firmware devices that were discovered in the PEI phase. The firmware device information includes the system memory locations of the firmware devices and of the firmware volumes that are contained within those firmware devices. The firmware volumes may contain DXE drivers, and the DXE Dispatcher is responsible for loading and executing the DXE drivers that are discovered in those firmware volumes. Finally, the HOB list may contain the I/O resources and memory-mapped I/O resources that were discovered in the PEI phase.

Figure 9.3 shows an example HOB list. The first entry in the HOB list is always the Phase Handoff Information Table (PHIT) HOB that contains the boot mode. The rest of the HOB list entries can appear in any order.

This example shows the different types of system resources that can be described in a HOB list. The most important ones to the DXE Foundation are the HOBs that describe system memory and the HOBs that describe firmware volumes. A HOB list is always terminated by an end-of-list HOB. The one additional HOB type that is not shown in Figure 9.3 is the GUID extension HOB that allows a PEIM to pass private data to a DXE driver. Only the DXE driver that recognizes the GUID value in the GUID extension HOB can understand the data in that HOB. The HOB entries are all designed to be position-independent. This independence allows the DXE Foundation to relocate the HOB list to a different location if it is not suitable to the DXE Foundation.

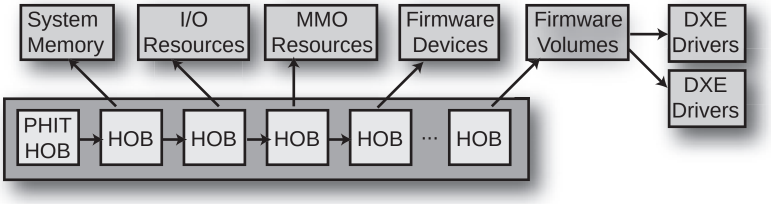


Figure 9.3 HOB List

DXE Architectural Protocols

The DXE Foundation is abstracted from the platform hardware through a set of DXE Architectural Protocols. The DXE Foundation consumes these protocols to produce the EFI Boot Services and EFI Runtime Services. DXE drivers that are loaded from firmware volumes produce the DXE Architectural Protocols. This design means that the DXE Foundation must have enough services to load and start DXE drivers before even a single DXE driver is executed.

The DXE Foundation is passed a HOB list that must contain a description of some amount of system memory and at least one firmware volume. The system memory descriptors in the HOB list are used to initialize the EFI services that require only memory to function correctly. The system is also guaranteed to be running on only one processor in flat physical mode with interrupts disabled. The firmware volume is passed to the DXE Dispatcher, which must contain a read-only FFS driver to search for the a priori file and any DXE drivers in the firmware volumes. When a driver is discovered that needs to be loaded and executed, the DXE Dispatcher uses a PE/COFF loader to load and invoke the DXE driver. The early DXE drivers produce the DXE Architectural Protocols, so the DXE Foundation can produce the full complement of EFI Boot Services and EFI Runtime Services. Figure 9.4 shows the HOB list being passed to the DXE Foundation. The DXE Foundation consumes the services of the DXE Architectural Protocols shown in the figure and then produces the EFI System Table, EFI Boot Services Table, and the EFI Runtime Services Table.

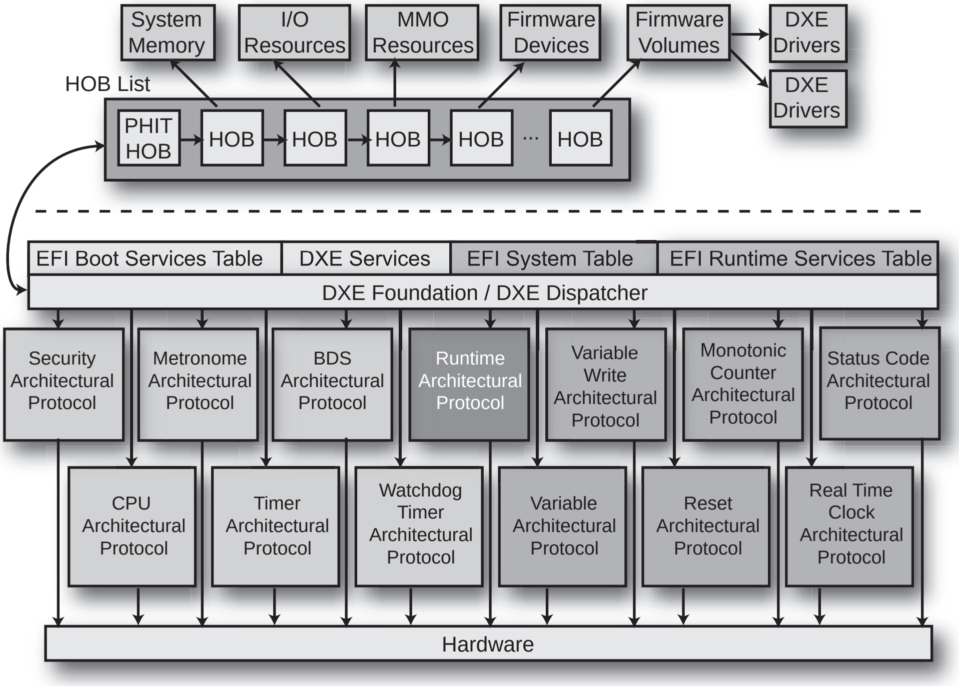

System Memory
I/O Resources
MMO Resources
Firmware Devices
Firmware Volumes
DXE Drivers
DXE Drivers
HOB List
PHIT HOB
HOB
HOB
HOB
HOB
...
HOB
EFI Boot Services Table
DXE Services
EFI System Table
EFI Runtime Services Table
DXE Foundation / DXE Dispatcher
Security Architectural Protocol
Metronome Architectural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

Figure 9.4 DXE Architectural Protocols

Figure 9.4 shows all the major components present in the DXE phase. The EFI Boot Services Table and DXE Services Table shown on the left are allocated from EFI boot services memory. This allocation means that the EFI Boot Services Table and DXE Services Table are freed when the OS runtime phase is entered. The EFI System Table and EFI Runtime Services Table on the right are allocated from EFI Runtime Services memory, and they do persist into the OS runtime phase.

The DXE Architectural Protocols shown on the left in Figure 9.4 are used to produce the EFI Boot Services. The DXE Foundation, DXE Dispatcher, and the protocols shown on the left are freed when the system transitions to the OS runtime phase. The DXE Architectural Protocols shown on the right are used to produce the EFI Runtime Services. These services persist in the OS runtime phase. The Runtime Architectural Protocol in the middle is special. This protocol provides the services that are required to transition the runtime services from physical mode to virtual mode under the direction of an OS. Once this transition is complete, these services can no longer be used.

The following is a brief summary of the DXE Architectural Protocols:

- Security Architectural Protocol: Allows the DXE Foundation to authenticate files stored in firmware volumes before they are used.
- CPU Architectural Protocol: Provides services to manage caches, manage interrupts, retrieve the processor's frequency, and query any processor-based timers.
- Metronome Architectural Protocol: Provides the services required to perform very short calibrated stalls.
- Timer Architectural Protocol: Provides the services required to install and enable the heartbeat timer interrupt required by the timer services in the DXE Foundation.
- BDS Architectural Protocol: Provides an entry point that the DXE Foundation calls once after all of the DXE drivers have been dispatched from all of the firmware volumes. This entry point is the transition from the DXE phase to the BDS phase, and it is responsible for establishing consoles and enabling the boot devices required to boot an OS.
- Watchdog Timer Architectural Protocol: Provides the services required to enable and disable a watchdog timer in the platform.
- Runtime Architectural Protocol: Provides the services required to convert all runtime services and runtime drivers from physical mappings to virtual mappings.
- Variable Architectural Protocol: Provides the services to retrieve environment variables and set volatile environment variables. .
- Variable Write Architectural Protocol: Provides the services to set nonvolatile environment variables.

- Monotonic Counter Architectural Protocol: Provides the services required by the DXE Foundation to manage a 64-bit monotonic counter.
- Reset Architectural Protocol: Provides the services required to reset or shut down the platform.
- Status Code Architectural Protocol: Provides the services to send status codes from the DXE Foundation or DXE drivers to a log or device.
- Real Time Clock Architectural Protocol: Provides the services to retrieve and set the current time and date as well as the time and date of an optional wakeup timer.

EFI System Table

The DXE Foundation produces the EFI System Table, which is consumed by every DXE driver and executable image invoked by BDS. It contains all the information that is required for these components to use the services provided by the DXE Foundation and any previously loaded DXE driver. Figure 9.5 shows the various components that are available through the EFI System Table.

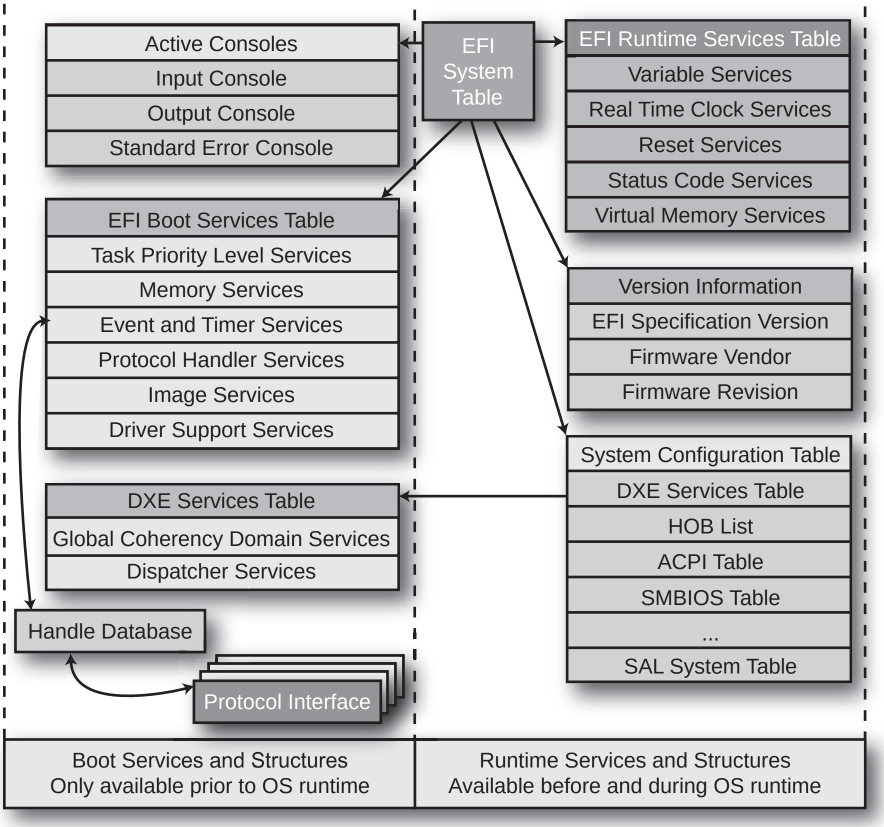


Figure 9.5 EFI System Table and Related Components

The DXE Foundation produces the EFI Boot Services, EFI Runtime Services, and DXE Services with the aid of the DXE Architectural Protocols. The EFI System Table provides access to all the active console devices in the platform and the set of EFI Configuration Tables. The EFI Configuration Tables are an extensible list of tables that describe the configuration of the platform including pointers to tables such as DXE Services, the HOB list, ACPI, System Management BIOS (SMBIOS), and the SAL System Table. This list may be expanded in the future as new table types are defined. Also, through the use of the Protocol Handle Services in the EFI Boot Services Table, any executable image can access the handle database and any of the protocol interfaces that have been registered by DXE drivers.

When the transition to the OS runtime is performed, the handle database, active consoles, EFI Boot Services, and services provided by boot service DXE drivers are terminated. This termination frees more memory for use by the OS and leaves the EFI System Table, EFI Runtime Services Table, and the system configuration tables available in the OS runtime environment. You also have the option of converting all of the EFI Runtime Services from a physical address space to an operating system specific virtual address space. This address space conversion may only be performed once.

EFI Boot Services Table

The following is a brief summary of the services that are available through the EFI Boot Services Table:

- Task Priority Services: Provides services to increase or decrease the current task priority level. This priority mechanism can be used to implement simple locks and to disable the timer interrupt for short periods of time. These services depend on the CPU Architectural Protocol.

- Memory Services: Provides services to allocate and free pages in 4 KB increments and allocate and free pool on byte boundaries. It also provides a service to retrieve a map of all the current physical memory usage in the platform.

- Event and Timer Services: Provides services to create events, signal events, check the status of events, wait for events, and close events. One class of events is timer events, which supports periodic timers with variable frequencies and one-shot timers with variable durations. These services depend on the CPU Architectural Protocol, Timer Architectural Protocol, Metronome Architectural Protocol, and Watchdog Timer Architectural Protocol.

- Protocol Handler Services: Provides services to add and remove handles from the handle database. It also provides services to add and remove protocols from the handles in the handle database. Additional services are available that allow any component to look up handles in the handle database and open and close protocols in the handle database.

- Image Services: Provides services to load, start, exit, and unload images using the PE/COFF image format. These services depend on the Security Architectural Protocol.

- Driver Support Services: Provides services to connect and disconnect drivers to devices in the platform. These services are used by the BDS phase to either connect all drivers to all devices, or to connect only the minimum number of drivers to devices required to establish the consoles and boot an OS. The minimal connect strategy is how a fast boot mechanism is provided.

EFI Runtime Services Table

The following is a brief summary of the services that are available through the EFI Runtime Services Table:

- Variable Services: Provides services to lookup, add, and remove environment variables from nonvolatile storage. These services depend on the Variable Architectural Protocol and the Variable Write Architectural Protocol.
- Real Time Clock Services: Provides services to get and set the current time and date. It also provides services to get and set the time and date of an optional wakeup timer. These services depend on the Real Time Clock Architectural Protocol.
- Reset Services: Provides services to shutdown or reset the platform. These services depend on the Reset Architectural Protocol.
- Status Code Services: Provides services to send status codes to a system log or a status code reporting device. These services depend on the Status Code Architectural Protocol.
- Virtual Memory Services: Provides services that allow the runtime DXE components to be converted from a physical memory map to a virtual memory map. These services can only be called once in physical mode. Once the physical to virtual conversion has been performed, these services cannot be called again. These services depend on the Runtime Architectural Protocol.

DXE Services Table

The following is a brief summary of the services that are available through the DXE Services Table:

- Global Coherency Domain Services: Provides services to manage I/O resources, memory-mapped I/O resources, and system memory resources in the platform. These services are used to dynamically add and remove these resources from the processor's Global Coherency Domain (GCD).

- DXE Dispatcher Services: Provides services to manage DXE drivers that are being dispatched by the DXE Dispatcher.

Global Coherency Domain Services

The Global Coherency Domain (GCD) Services are used to manage the memory and I/O resources visible to the boot processor. These resources are managed in two different maps:

- GCD memory space map
- GCD I/O space map

If memory or I/O resources are added, removed, allocated, or freed, then the GCD memory space map and GCD I/O space map are updated. GCD Services are also provided to retrieve the contents of these two resource maps.

The GCD Services can be broken up into two groups. The first manages the memory resources visible to the boot processor, and the second manages the I/O resources visible to the boot processor. Not all processor types support I/O resources, so the management of I/O resources may not be required. However, since system memory resources and memory-mapped I/O resources are required to execute the DXE environment, the management of memory resources is always required.

GCD Memory Resources

The Global Coherency Domain (GCD) Services used to manage memory resources include the following:

- AddMemorySpace()
- AllocateMemorySpace()
- FreeMemorySpace()
- RemoveMemorySpace()
- SetMemorySpaceAttributes()

The GCD Services used to retrieve the GCD memory space map include the following:

- GetMemorySpaceDescriptor()

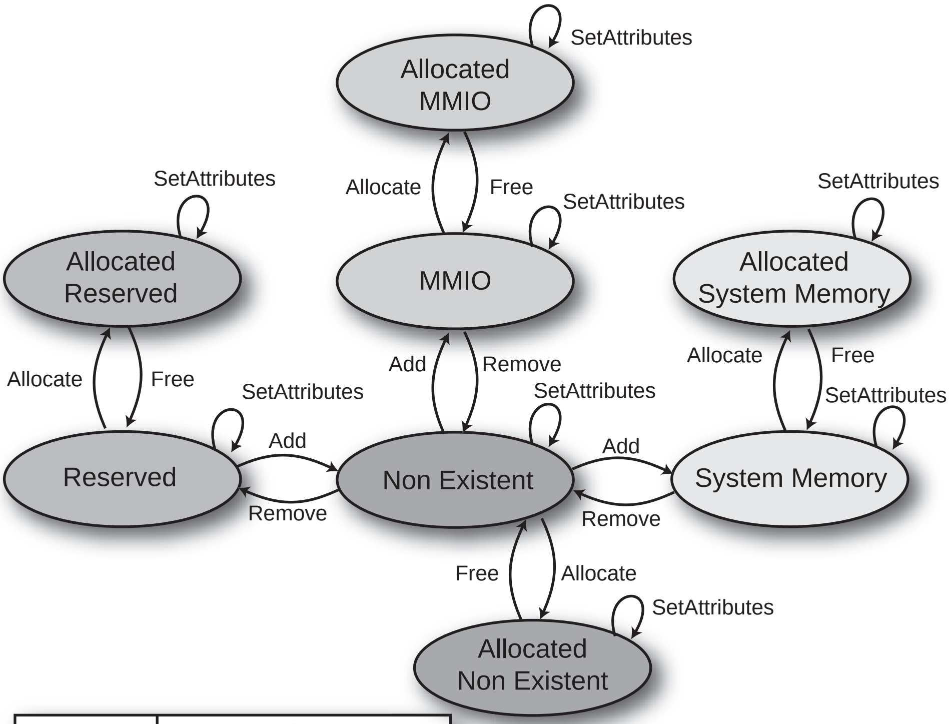
- GetMemorySpaceMap()

The GCD memory space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access memory resources. This information is used to initialize the state of the GCD memory space map. Any memory regions outside this initial region are unavailable to any of the GCD Services that are used to manage memory resources. The GCD memory space map is designed to describe the memory address space with as many as 64 address lines. Each region in the GCD memory space map can begin and end on a byte boundary. Additional HOB types describe the location of system memory, the location memory mapped I/O, the location of firmware devices, the location of firmware volumes, the location of reserved regions, and the location of system memory regions that were allocated prior to the execution of the DXE Foundation. The DXE Foundation must parse the contents of the HOB list to guarantee that memory regions reserved prior to the execution of the DXE Foundation are honored. As a result, the GCD memory space map must reflect the memory regions described in the HOB list. The GCD memory space map provides the DXE Foundation with the information required to initialize the memory services such as AllocatePages(), FreePages(), AllocatePool(), FreePool(), and GetMemoryMap().

A memory region described by the GCD memory space map can be in one of several different states:

- Nonexistent memory
- System memory
- Memory-mapped I/O
- Reserved memory

These memory regions can be allocated and freed by DXE drivers executing in the DXE environment. In addition, a DXE driver can attempt to adjust the caching attributes of a memory region. Figure 9.6 shows the possible state transitions for each byte of memory in the GCD memory space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD services are required to merge similar memory regions that are adjacent to each other into a single memory descriptor, which reduces the number of entries in the GCD memory space map.



Operation	GCD Service
Add	AddMemorySpace()
Remove	RemoveMemorySpace()
Allocate	AllocateMemorySpace()
Free	FreeMemorySpace()
SetAttributes	SetMemorySpaceAttributes()

Figure 9.6 GCD Memory State Transitions

GCD I/O Resources

The Global Coherency Domain (GCD) Services used to manage I/O resources include the following:

- AddIoSpace()
- AllocateIoSpace()
- FreeIoSpace()

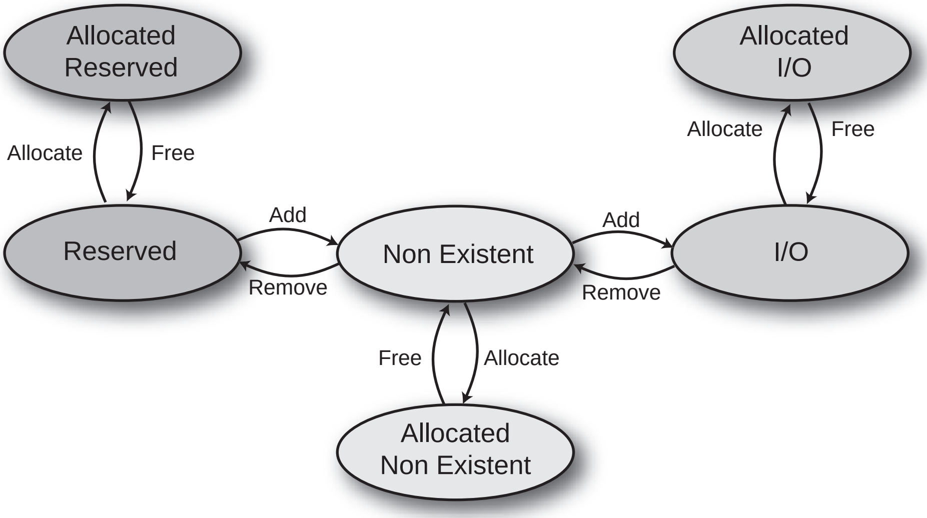
- RemoveIoSpace()

The GCD Services used to retrieve the GCD I/O space map include the following:

- GetIoSpaceDescriptor()
- GetIoSpaceMap()

The GCD I/O space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access I/O resources. This information is used to initialize the state of the GCD I/O space map. Any I/O regions outside this initial region are not available to any of the GCD Services that are used to manage I/O resources. The GCD I/O space map is designed to describe the I/O address space with as many as 64 address lines. Each region in the GCD I/O space map can being and end on a byte boundary.

An I/O region described by the GCD I/O space map can be in several different states. These include nonexistent I/O, I/O, and reserved I/O. These I/O regions can be allocated and freed by DXE drivers executing in the DXE environment. Figure 9.7 shows the possible state transitions for each byte of I/O in the GCD I/O space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD Services are required to merge similar I/O regions that are adjacent to each other into a single I/O descriptor, which reduces the number of entries in the GCD I/O space map.



Operation	GCD Service
Add	AddIoSpace()
Remove	RemoveIoSpace()
Allocate	AllocateIoSpace()
Free	FreeIoSpace()

Figure 9.7 GCD I/O State Transitions

DXE Dispatcher

After the DXE Foundation is initialized, control is handed to the DXE Dispatcher. The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes. The DXE Dispatcher searches for drivers in the firmware volumes described by the HOB list. As execution continues, other firmware volumes might be located. When they are, the DXE Dispatcher searches them for drivers as well.

When a new firmware volume is discovered, a search is made for its a priori file. The a priori file has a fixed file name and contains the list of DXE drivers that should be loaded and executed first. There can be at most one a priori file per firmware volume, although it is acceptable to have no a priori file at all. Once the DXE drivers from the a priori file have been loaded and executed, the dependency expressions of the remaining DXE drivers in the firmware volumes are evaluated to determine the order in which they will be loaded and executed. The a priori file provides a strongly ordered list of DXE drivers that are not required to use dependency expressions. The dependency expressions provide a weakly ordered execution of the remaining DXE drivers. Before each DXE driver is executed, it must be authenticated with the Security Architectural Protocol. This authentication prevents DXE drivers with unknown origins from being executed.

Control is transferred from the DXE Dispatcher to the BDS Architectural Protocol after the DXE drivers in the a priori file and all the DXE drivers whose dependency expressions evaluate to TRUE have been loaded and executed. The BDS Architectural Protocol is responsible for establishing the console devices and attempting the boot of operating systems. As the console devices are established and access to boot devices is established, additional firmware volumes may be discovered. If the BDS Architectural Protocol is unable to start a console device or gain access to a boot device, it reinvokes the DXE Dispatcher. This invocation allows the DXE Dispatcher to load and execute DXE drivers from firmware volumes that have been discovered since the last time the DXE Dispatcher was invoked. Once the DXE Dispatcher has loaded and executed all the DXE drivers it can, control is once again returned to the BDS Architectural Protocol to continue the OS boot process. Figure 9.8 illustrates this basic flow between the Dispatcher, its launched drivers, and the BDS.

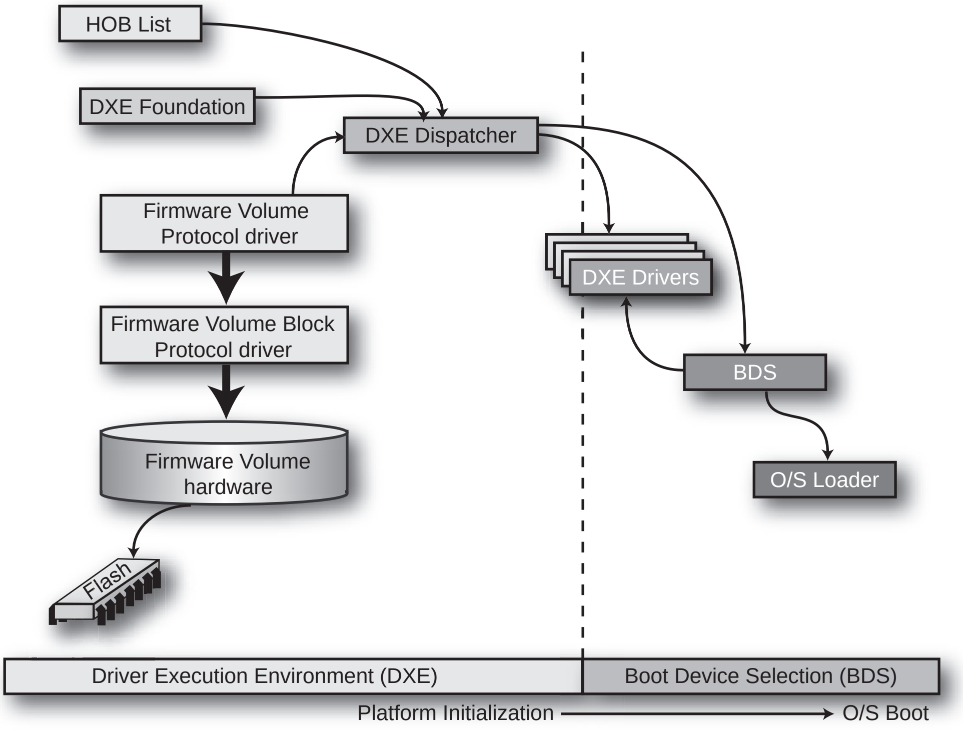

HOB List
DXE Foundation
DXE Dispatcher
Firmware Volume Protocol driver
Firmware Volume Block Protocol driver
Firmware Volume hardware
Flash
DXE Drivers
BDS
O/S Loader
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Platform Initialization
O/S Boot

Figure 9.8 The Handshake between the Dispatcher and Other Components.

The *a priori* File

The a priori file is a special file that may be present in a firmware volume. The rule is that there may be at most one a priori file per firmware volume present in a platform. The a priori file has a known GUID file name, so the DXE Dispatcher can always find the a priori file. Every time the DXE Dispatcher discovers a firmware volume, it first looks for the a priori file. The a priori file contains the list of DXE drivers that should be loaded and executed before any other DXE drivers are discovered. The DXE drivers listed in the a priori file are executed in the order that they appear. If any of those DXE drivers have an associated dependency expression, then those dependency expressions are ignored.

The purpose of the a priori file is to provide a deterministic execution order of DXE drivers. DXE drivers that are executed solely based on their dependency expression are weakly ordered, which means that the execution order is not completely deterministic between boots or between platforms. Some cases, however, require a deterministic execution order. One example would be to list the DXE drivers that are required to debug the rest of the DXE phase in the a priori file. These DXE drivers that provide debug services might have been loaded much later if only their dependency expressions were considered. By loading them earlier, more of the DXE Foundation and DXE drivers can be debugged. Another example is to use the a priori file to eliminate the need for dependency expressions. Some embedded platforms may require only a few DXE drivers with a highly deterministic execution order. The a priori file can provide this ordering, and none of the DXE drivers would require dependency expressions. The dependency expressions do have some amount of firmware device overhead, so this method might actually conserve firmware space. The main purpose of the a priori file is to provide a greater degree of flexibility in the firmware design of a platform.

Dependency Grammar

A DXE driver is stored in a firmware volume as a file with one or more sections. One of the sections must be a PE/COFF image. If a DXE driver has a dependency expression, then it is stored in a dependency section. A DXE driver may contain additional sections for compression and security wrappers. The DXE Dispatcher can identify the DXE drivers by their file type. In addition, the DXE Dispatcher can look up the dependency expression for a DXE driver by looking for a dependency section in a DXE driver file. The dependency section contains a section header followed by the actual dependency expression that is composed of a packed byte stream of opcodes and operands.

Dependency expressions stored in dependency sections are designed to be small to conserve space. In addition, they are designed to be simple and quick to evaluate to reduce execution overhead. These two goals are met by designing a small, stack-based instruction set to encode the dependency expressions. The DXE Dispatcher must implement an interpreter for this instruction set to evaluate dependency expressions. Table 9.1 gives a summary of the supported opcodes in the dependency expression instruction set.

Table 9.1 Supported Opcodes in the Dependency Expression Instruction Set

Opcode	Description
0x00	BEFORE <File Name GUID>
0x01	AFTER <File Name GUID>
0x02	PUSH <Protocol GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END
0x09	SOR

Because multiple dependency expressions may evaluate to TRUE at the same time, the order in which the DXE drivers are loaded and executed may vary between boots and between platforms even though the contents of their firmware volumes are identical. This variation is why the ordering is weak for the execution of DXE drivers in a platform when dependency expressions are used.

DXE Drivers

DXE drivers have two subclasses:

- DXE drivers that execute very early in the DXE phase
- DXE drivers that comply with the EFI 1.1 Driver Model

The execution order of the first subclass, the early DXE drivers, depends on the presence and contents of an a priori file and the evaluation of dependency expressions. These early DXE drivers typically contain processor, chipset, and platform initialization code. They also typically produce the DXE Architectural Protocols that are required for the DXE Foundation to produce its full complement of EFI Boot Services and EFI Runtime Services. To support the fastest possible boot time, as much initialization as possible should be deferred to the second subclass of DXE drivers, those that comply with the EFI 1.10 Driver Model.

The DXE drivers that comply with the EFI 1.10 Driver Model do not perform any hardware initialization when they are executed by the DXE Dispatcher. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the BDS phase to connect the drivers to the devices required to establish consoles and provide access to boot devices. The DXE Drivers that comply with the EFI 1.10 Driver Model ultimately provide software abstractions for console devices and boot devices but only when they are explicitly asked to do so.

All DXE drivers may consume the EFI Boot Services and EFI Runtime Services to perform their functions. However, the early DXE drivers need to be aware that not all of these services may be available when they execute because not all of the DXE Architectural Protocols might have been registered yet. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed.

The DXE drivers that comply with the EFI 1.10 Driver Model do not need to be concerned with this possibility. These drivers simply register the Driver Binding Protocol in the handle database when they are executed. This operation can be performed without the use of any DXE Architectural Protocols. The BDS phase will not be entered until all of the DXE Architectural Protocols are registered. If the DXE Dispatcher does not have any more DXE drivers to execute but not all of the DXE Architectural Protocols have been registered, then a fatal error has occurred and the system will be halted.

Boot Device Selection (BDS) Phase

The Boot Device Selection (BDS) Architectural Protocol executes during the BDS phase. The BDS Architectural Protocol is discovered in the DXE phase, and it is executed when two conditions are met:

- All of the DXE Architectural Protocols have been registered in the handle database. This condition is required for the DXE Foundation to produce the full complement of EFI Boot Services and EFI Runtime Services.
- The DXE Dispatcher does not have any more DXE drivers to load and execute. This condition occurs only when all the a priori files from all the firmware volumes have been processed and all the DXE drivers whose dependency expression have evaluated to TRUE have been loaded and executed.

The BDS Architectural Protocol locates and loads various applications that execute in the pre-boot services environment. Such applications might represent a traditional OS boot loader or extended services that might run instead of or prior to loading the final OS. Such extended preboot services might include setup configuration, extended diagnostics, flash update support, OEM services, or the OS boot code.

Vendors such as IBVs, OEMs, and ISVs may choose to use a reference implementation, develop their own implementation based on the reference, or develop an implementation from scratch.

The BDS phase performs a well-defined set of tasks. The user interface and user interaction that occurs on different boots and different platforms may vary, but the boot policy that the BDS phase follows is very rigid. This boot policy is required so OS installations will behave predictably from platform to platform. The tasks include the following:

- Initialize console devices based on the ConIn, ConOut, and StdErr environment variables.
- Attempt to load all drivers listed in the Driver#### and DriverOrder environment variables.
- Attempt to boot from the boot selections listed in the Boot#### and BootOrder environment variables.

If the BDS phase is unable to connect a console device, load a driver, or boot a boot selection, it is required to reinvoke the DXE Dispatcher. This invocation is required because additional firmware volumes may have been discovered while attempting to perform these operations. These additional firmware volumes may contain the DXE drivers required to manage the console devices or boot devices. Once all of the DXE drivers have been dispatched from any newly discovered firmware volumes, control is returned to the BDS phase. If the BDS phase is unable to make any additional forward progress in connecting the console device or the boot device, then the connection of that console device or boot selection fails. When a failure occurs, the BDS phase moves on to the next console device, driver load, or boot selection.

Console Devices

Console devices are abstracted through the Simple Text Output and Simple Input Protocols. Any device that produces one or both of these protocols may be used as a console device on a Framework-based platform. Several types of devices are capable of producing these protocols, including the following:

- VGA Adapters: These adapters can produce a text-based display that is abstracted with the Simple Text Output Protocol.
- Universal Graphics Adapters (UGA): These adapters produce a graphical interface that supports Block Transfer (BLT) operations. A text-based display that produces the Simple Text Output Protocol can be simulated on top of a UGA display by using BLT operations to send Unicode glyphs into the frame buffer.
- Serial Terminal: A serial terminal device can produce both the Simple Input and Simple Text Output Protocols. Serial terminals are very flexible, and they can support a variety of wire protocols such as PC-ANSI, VT-100, VT-100+, and VTUTF8.
- Telnet: A telnet session can produce both the Simple Input and Simple Text Output Protocols. Like the serial terminal, a variety of wire protocols can be supported including PC-ANSI, VT-100, VT-100+, and VTUTF8. .
- Remote Graphical Displays (HTTP): A remote graphical display can produce both the Simple Input and Simple Text Output Protocols. One possible implementation could use HTTP, so standard Internet browsers could be used to manage a Framework-based platform.

Boot Devices

Several types of boot devices are supported in the Framework:

- Devices that produce the Block I/O Protocol and are formatted with a FAT file system
- Devices that directly produce the File System Protocol
- Devices that directly produce the Load File Protocol

Disk devices typically produce the Block I/O Protocol, and network devices typically produce the Load File Protocol.

A Framework implementation may also choose to include legacy compatibility drivers. These drivers provide the services required to boot a traditional OS, and the BDS phase must also support booting a traditional OS.

Boot Services Terminate

The BDS phase is terminated when an OS loader is executed and an OS is successfully booted. An OS loader or an OS kernel may call a single service called ExitBootServices() to terminate the BDS phase. Once this call is made, all of the boot service components are freed and their resources are available for use by the OS. When the call to ExitBootServices() returns, the Runtime (RT) phase has been entered.

Chapter 10

Some Common EFI Functions

> *Never let the future disturb you. You will meet it, if you have to, with the same weapons of reason which today arm you against the present.*
>
> —Marcus Aurelius Antoninus

EFI provides a variety of functions that are used for drivers and applications to communication with the underlying EFI components. Many of the designs for interfaces have historically been short-sighted due to their inability to predict changes in technology. An example of such shortsightedness might be where a disk interface assumed that a disk might never have more than 8 gigabytes of space available. It is often hard to predict what changes technology might provide. Many famous statements have been made that fret about how a personal computer might never be practical, or assure readers that 640 kilobytes of memory would be more than anyone would ever need. With these poor past predictions in mind, one can attempt to learn from such mistakes and design interfaces that are robust enough for common practices today, and make the best attempt at predicting how one might use these interfaces years from today.

This chapter describes a selection of common interfaces that show up in EFI as well as the Framework:

- *Architectural Protocols*: These are a set of protocols that abstract the platform hardware from the EFI drivers and applications. They are unusual only in that they are the protocols that are going to be used by the DXE Foundation and are the basis for a Framework-based implementation. These protocols in their current form were introduced into the Framework and are not defined in the current EFI specifications.

- *PCI Protocols*: These protocols abstract all aspects of interaction with the underlying PCI bus, enumeration of said bus, as well as resource allocation. These interfaces were introduced for EFI, and would be present in both EFI and Framework implementations.
- *Block I/O*: This protocol is used to abstract mass storage devices to allow code running in the EFI boot services environment to access them without specific knowledge of the type of device or controller that manages the device. This interface was introduced for EFI, and would be present in both EFI and Framework implementations.
- *Disk I/O*: This protocol is used to abstract the block accesses of the Block I/O protocol to a more general offset-length protocol. The firmware is responsible for adding this protocol to any Block I/O interface that appears in the system that does not already have a Disk I/O protocol. File systems and other disk access code utilize the Disk I/O protocol. This interface was introduced for EFI, and would be present in both EFI and Framework implementations.
- *Simple File System*: This protocol allows code running in the EFI boot services environment to obtain file-based access to a device. The Simple File System protocol is used to open a device volume and return an EFI_FILE handle that provides interfaces to access files on a device volume. This interface was introduced for EFI, and would be present in both EFI and Framework implementations.

Architectural Protocol Examples

A variety of architectural protocols exist in the platform. These protocols function just like other protocols in every way. The only difference is that these protocols are consumed by the platform's core services and the remainder of the drivers and applications in turn call these core services to act on the platform in various ways. Generally, the only users of the architectural protocols are the core services themselves. The architectural protocols abstract the hardware and are the only agents in the system that would typically talk directly to the hardware in the pre-boot environment. Everything else in the system would communicate with a core service to communicate any sort of requests to the hardware. Figure 10.1 illustrates this high level software handshake.

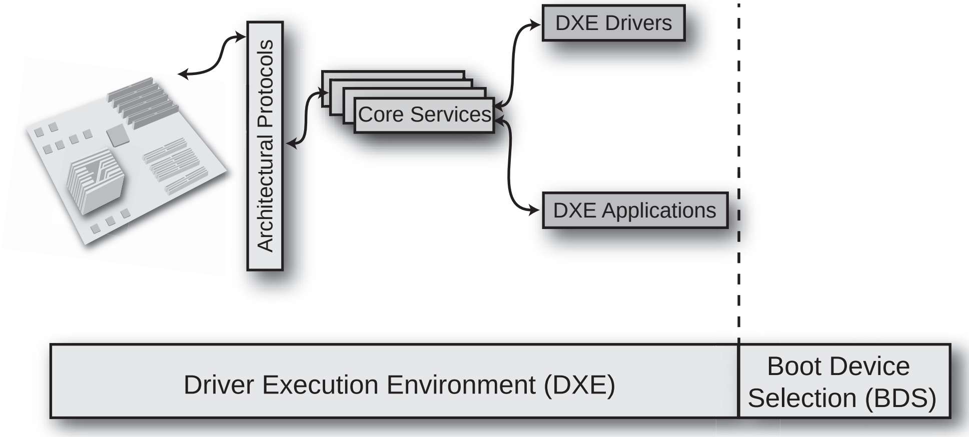


Figure 10.1 Platform Software Flow Diagram

To show more clearly how some of these architectural protocols are designed and how they operate, several key examples will be examined in further detail. Note that the following examples are not the full set of architectural protocols but are used to illustrate some of their functionality. For the full set, please refer to the appropriate DXE specifications.

CPU Architectural Protocol

The CPU Architectural Protocol is used to abstract processor-specific functions from the DXE Foundation. This includes flushing caches, enabling and disabling interrupts, hooking interrupt vectors and exception vectors, reading internal processor timers, resetting the processor, and determining the processor frequency. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation and DXE drivers that produce architectural protocols. By allowing this protocol to be produced by a boot service driver, it is evident that this abstraction will not persist when the platform has the boot services terminated by launching a boot target such as an operating system.

The GCD memory space map is initialized by the DXE Foundation based on the contents of the HOB list. The HOB list contains the capabilities of the different memory regions, but it does not contain their current attributes. The DXE driver that produces the CPU Architectural Protocol is responsible for maintaining the current attributes of the memory regions visible to the processor.

This means that the DXE driver that produces the CPU Architectural Protocol must seed the GCD memory space map with the initial state of the attributes for all the memory regions visible to the processor. The DXE Service SetMemorySpaceAttributes() allows the attributes of a memory range to be modified. The SetMemorySpaceAttributes() DXE Service is implemented using the SetMemoryAttributes() service of the CPU Architectural Protocol.

To initialize the state of the attributes in the GCD memory space map, the DXE driver that produces the CPU Architectural Protocol must call the DXE Service SetMemorySpaceAttributes() for all the different memory regions visible to the processor passing in the current attributes. This, in turn, will call back to the SetMemoryAttributes() service of the CPU Architectural Protocol, and all of these calls must return EFI_SUCCESS, since the DXE Foundation is only requesting that the attributes of the memory region be set to their current settings. This forces the current attributes in the GCD memory space map to be set to these current settings. After this initialization is complete, the next call to the DXE Service GetMemorySpaceMap() will correctly show the current attributes of all the memory regions. In addition, any future calls to the DXE Service SetMemorySpaceAttributes() will in turn call the CPU Architectural Protocol so see of those attributes can be modified, and if they can, the GCD memory space map will be updated accordingly.

The CPU Architectural Protocol uses the following protocol definition:

```
Protocol Interface Structure
typedef struct _EFI_CPU_ARCH_PROTOCOL {
  EFI_CPU_FLUSH_DATA_CACHE                FlushDataCache;
  EFI_CPU_ENABLE_INTERRUPT                EnableInterrupt;
  EFI_CPU_DISABLE_INTERRUPT
DisableInterrupt;
  EFI_CPU_GET_INTERRUPT_STATE
GetInterruptState;
  EFI_CPU_INIT                            Init;
  EFI_CPU_REGISTER_INTERRUPT_HANDLER
RegisterInterruptHandler;
```

```
  EFI_CPU_GET_TIMER_VALUE                GetTimerValue;
  EFI_CPU_SET_MEMORY_ATTRIBUTES
SetMemoryAttributes;
  UINT32                                 NumberOfTimers;
  UINT32
DmaBufferAlignment;
} EFI_CPU_ARCH_PROTOCOL;
```

- *FlushDataCache* - Flushes a range of the processor's data cache. If the processor does not contain a data cache, or the data cache is fully coherent, then this function can just return EFI_SUCCESS. If the processor does not support flushing a range of addresses from the data cache, then the entire data cache must be flushed. This function is used by the root bridge I/O abstractions to flush data caches for DMA operations.
- *EnableInterrupt* - Enables interrupt processing by the processor. See the EnableInterrupt() function description. This function is used by the Boot Service RaiseTPL() and RestoreTPL().
- *DisableInterrupt* - Disables interrupt processing by the processor. See the DisableInterrupt() function description. This function is used by the Boot Service RaiseTPL() andRestoreTPL().
- *GetInterruptState* - Retrieves the processor's current interrupt state.
- *Init* - Generates an INIT on the processor. This function may be used by the Reset Architectural Protocol depending upon a specified boot path. If a processor cannot programmatically generate an INIT without help from external hardware, then this function returns EFI_UNSUPPORTED.
- *RegisterInterruptHandler* - Associates an interrupt service routine with one of the processor's interrupt vectors. This function is typically used by the EFI_TIMER_ARCH_PROTOCOL to hook the timer interrupt in a system. It can also be used by the debugger to hook exception vectors.
- *GetTimerValue* - Returns the value of one of the processor's internal timers.
- *SetMemoryAttributes* - Attempts to set the attributes of a memory region.

- *NumberOfTimers* – Gives the number of timers that are available in a processor. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.
- *DmaBufferAlignment* – Gives the size, in bytes, of the alignment required for DMA buffer allocations. This is typically the size of the largest data cache line in the platform. This value can be determined by looking at the data cache line sizes of all the caches present in the platform, and returning the largest. This is used by the root bridge I/O abstraction protocols to guarantee that no two DMA buffers ever share the same cache line. The value in this field is a constant that must not be modified after the CPU Architectural Protocol is installed. All consumers must treat this as a read-only field.

Real Time Clock Architectural Protocol

The Real Time Clock Architectural Protocol provides the services required to access a system's real time clock hardware. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation.

The DXE driver that produces this protocol must be a runtime driver. This driver is responsible for initializing the GetTime(), SetTime(), GetWakeupTime(), and SetWakeupTime() fields of the EFI Runtime Services Table. See the section "Time Services" in Chapter 5 for details on these services. After the four fields of the EFI Runtime Services Table have been initialized, the driver must install the Real Time Clock Architectural Protocol on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the real time clock-related services are now available and that the DXE Foundation must update the 32-bit CRC of the EFI Runtime Services Table.

Timer Architectural Protocol

The Timer Architectural Protocol provides the services to initialize a periodic timer interrupt and to register a handler that is called each time the timer interrupt fires. It may also provide a service to adjust the rate of the periodic timer interrupt. When a timer interrupt occurs, the handler is passed the amount of time that has passed since the previous timer interrupt. This protocol enables the use of the SetTimer() Boot

Service. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols. By allowing this protocol to be produced by a boot service driver, it is evident that this abstraction will not persist when the platform has the boot services terminated by launching a boot target, such as an operating system.

```
Protocol Interface Structure

typedef struct _EFI_TIMER_ARCH_PROTOCOL {
  EFI_TIMER_REGISTER_HANDLER          RegisterHandler;
  EFI_TIMER_SET_TIMER_PERIOD          SetTimerPeriod;
  EFI_TIMER_GET_TIMER_PERIOD          GetTimerPeriod;
  EFI_TIMER_GENERATE_SOFT_INTERRUPT
GenerateSoftInterrupt;
} EFI_TIMER_ARCH_PROTOCOL;
```

- *RegisterHandler* - Registers a handler that is called each time the timer interrupt fires. TimerPeriod defines the minimum time between timer interrupts, so TimerPeriod is also the minimum time between calls to the registered handler.
- *SetTimerPeriod* - Sets the period of the timer interrupt in 100 nanosecond units. This function is optional and may return EFI_UNSUPPORTED. If this function is supported, then the timer period is rounded up to the nearest supported timer period.
- *GetTimerPeriod* - Retrieves the period of the timer interrupt in 100 nanosecond units.
- *GenerateSoftInterrupt* - Generates a soft timer interrupt that simulates the firing of the timer interrupt. This service can be used to invoke the registered handler if the timer interrupt has been masked for a period of time.

Reset Architectural Protocol

The Reset Architectural Protocol provides the service required to reset a platform. This protocol must be produced by a runtime DXE driver and may only be consumed by the DXE Foundation. This driver is responsible for initializing the ResetSystem() field of the EFI Runtime Services Table. After this field of the EFI Runtime Services Table has been initialized, the driver must install the Reset Architectural Protocol on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the reset system service is now available and that the DXE Foundation must update the 32-bit CRC of the EFI Runtime Services Table.

Boot Device Selection Architectural Protocol

The Boot Device Selection (BDS) Architectural Protocol transfers control from DXE to an operating system or a system utility, as illustrated in Figure 10.2. This protocol must be produced by a boot service or runtime DXE driver and may only be consumed by the DXE Foundation. By allowing this protocol to be produced by a boot service driver, it is evident that this abstraction will not persist when the platform has the boot services terminated by launching a boot target such as an operating system.

If not enough drivers have been initialized when this protocol is used to access the required boot device(s), then this protocol should add drivers to the dispatch queue and return control back to the dispatcher. Once the required boot devices are available, then the boot device can be used to load and invoke an OS or a system utility.

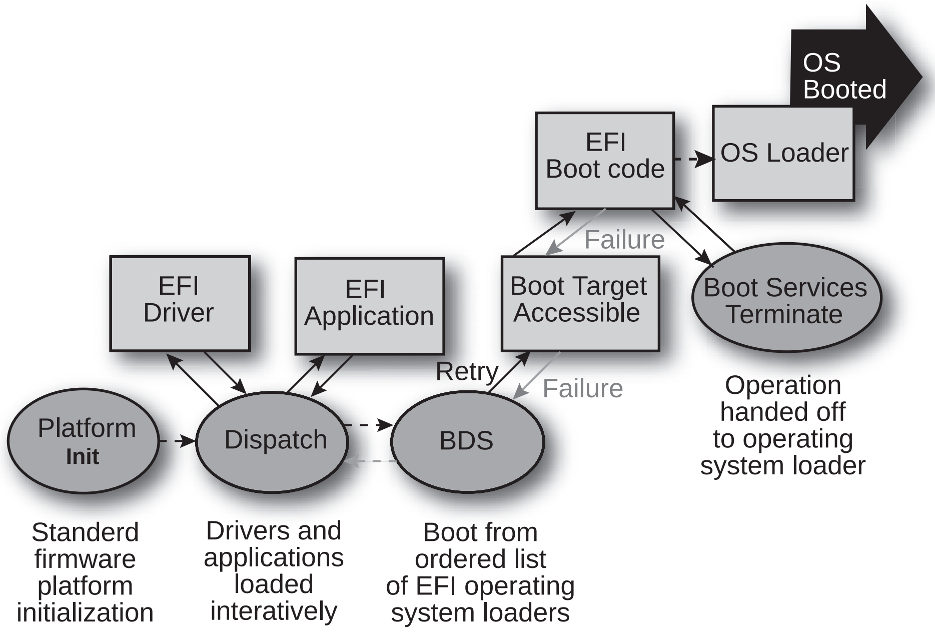


Figure 10.2 Basic Dispatch and BDS Software Flow

```
Protocol Interface Structure

typedef struct _EFI_BDS_ARCH_PROTOCOL {
  EFI_BDS_ENTRY               Entry;
} EFI_BDS_ARCH_PROTOCOL;
```

- *Entry* - The entry point to BDS. See the Entry() function description. This call does not take any parameters, and the return value can be ignored. If it returns, then the dispatcher must be invoked again, if it never returns, then an operating system or a system utility have been invoked.

Variable Architectural Protocol

The Variable Architectural Protocol provides the services required to get and set environment variables. This protocol must be produced by a runtime DXE driver and may be consumed only by the DXE Foundation. This driver is responsible for initializing the GetVariable(), GetNextVariableName(), and SetVariable() fields of the EFI Runtime Services Table. See the section "Variable Services" in Chapter 5 for details on these services. After the three fields of the EFI Runtime Services Table have been initialized, the driver must install the Variable Architectural Protocol on a new handle with a NULL interface pointer. The installation of this protocol informs the DXE Foundation that the read-only and the volatile environment variable related services are now available and that the DXE Foundation must update the 32-bit CRC of the EFI Runtime Services Table. The full complement of environment variable services is not available until both this protocol and Variable Write Architectural Protocol are installed. DXE drivers that require read-only access or read/write access to volatile environment variables must have this architectural protocol in their dependency expressions. DXE drivers that require write access to nonvolatile environment variables must have the Variable Write Architectural Protocol in their dependency expressions.

Watchdog Timer Architectural Protocol

The Watchdog Timer Architectural Protocol is used to program the watchdog timer and optionally register a handler when the watchdog timer fires. This protocol must be produced by a boot service or runtime DXE driver and may be consumed only by the DXE Foundation or DXE drivers that produce other DXE Architectural Protocols. If a platform wishes to perform a platform-specific action when the watchdog timer expires, then the DXE driver containing the implementation of the BDS Architectural Protocol should use this protocol's RegisterHandler() service.

This protocol provides the services required to implement the Boot Service SetWatchdogTimer(). It provides a service to set the amount of time to wait before firing the watchdog timer, and it also provides a service to register a handler that is invoked when the watchdog timer fires. This protocol can implement the watchdog timer by using the event and timer Boot Services, or it can make use of custom hardware. When the watchdog timer fires, control will be passed to a handler if a handler has been registered. If no handler has been registered, or the registered handler returns, then the system will be reset by calling the Runtime Service ResetSystem().

```
Protocol Interface Structure

typedef struct _EFI_WATCHDOG_TIMER_ARCH_PROTOCOL {
  EFI_WATCHDOG_TIMER_REGISTER_HANDLER
RegisterHandler;
  EFI_WATCHDOG_TIMER_SET_TIMER_PERIOD
SetTimerPeriod;
  EFI_WATCHDOG_TIMER_GET_TIMER_PERIOD
GetTimerPeriod;
} EFI_WATCHDOG_TIMER_ARCH_PROTOCOL;
```

- RegisterHandler - Registers a handler that is invoked when the watchdog timer fires.
- SetTimerPeriod - Sets the amount of time in 100 nanosecond units to wait before the watchdog timer is fired. If this function is supported, then the watchdog timer period is rounded up to the nearest supported watchdog timer period.
- GetTimerPeriod - Retrieves the amount of time in 100 nanosecond units that the system will wait before the watchdog timer is fired.

PCI Protocols

This section describes a series of protocols that are all related to abstracting various aspects of PCI related interaction such as resource allocation and I/O.

PCI Host Bridge Resource Allocation Protocol

The PCI Host Bridge Resource Allocation Protocol is used by a PCI bus driver to program a PCI host bridge. The registers inside a PCI host bridge that control configuration of PCI root buses are not governed by the PCI specification and vary from chipset to chipset. The PCI Host Bridge Resource Allocation Protocol implementation is therefore specific to a particular chipset.

Each PCI host bridge is composed of one or more PCI root bridges, and hardware registers are associated with each PCI root bridge. These registers control the bus, I/O, and memory resources that are decoded by the PCI root bus that the PCI root bridge produces and all the PCI buses that are children of that PCI root bus.

The PCI Host Bridge Resource Allocate Protocol allows for future innovation of the chipsets. It abstracts the PCI bus driver from the chipset details. This design allows system designers to make changes to the host bridge hardware without impacting a platform independent PCI bus driver.

Figure 10.3 shows a platform with a set of processors (CPUs) and a set of core chipset components that produce *n* host bridges. Most systems with one PCI host bus controller contain a single instance of the PCI Host Bridge Allocation Protocol. More complex systems may contain multiple instances of this protocol.

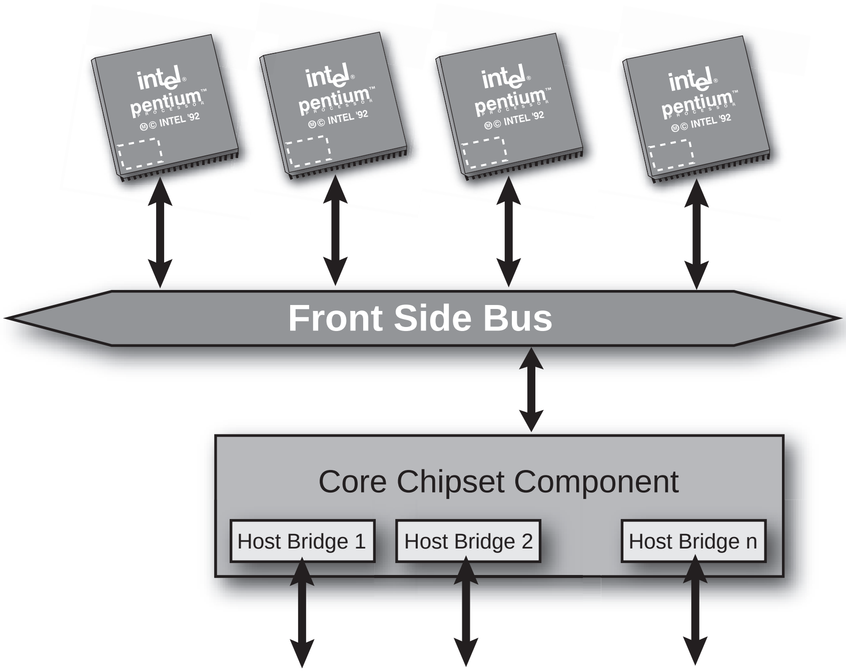

intel pentium PROCESSOR ©© INTEL '92
intel pentium PROCESSOR ©© INTEL '92
intel pentium PROCESSOR ©© INTEL '92
intel pentium PROCESSOR ©© INTEL '92
Front Side Bus
Core Chipset Component
Host Bridge 1
Host Bridge 2
Host Bridge n

Figure 10.3 Example Host Bus Controllers

Figure 10.4 shows how the PCI Host Bridge Resource Allocation Protocol is used to identify the associated PCI root bridges. After the steps shown in Figure 10.4 are completed, the PCI Host Bridge Resource Allocation Protocol can then be queried to identify the device handles of the associated PCI root bridges.

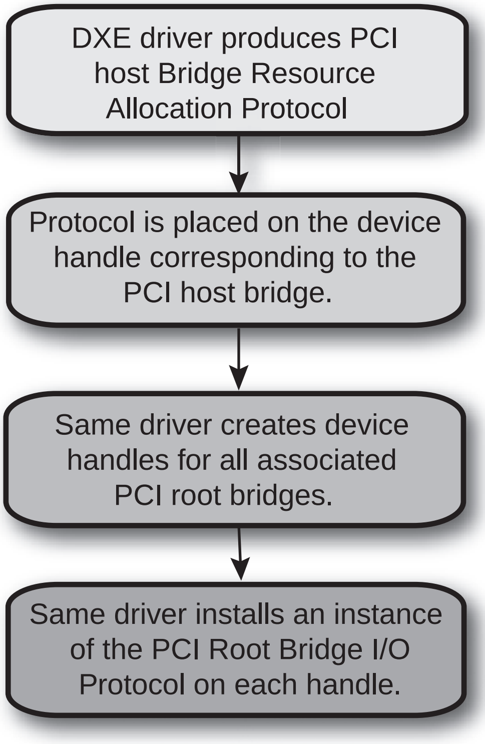

DXE driver produces PCI host Bridge Resource Allocation Protocol
Protocol is placed on the device handle corresponding to the PCI host bridge.
Same driver creates device handles for all associated PCI root bridges.
Same driver installs an instance of the PCI Root Bridge I/O Protocol on each handle.

Figure 10.4 Producing the PCI Host Bridge Resource Allocation Protocol

Sample Desktop System with One PCI Root Bridge

Figure 10.5 shows an example of a PCI host bus with one PCI root bridge. This PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard and/or PCI slots. This setup would be typical of a desktop system. In this system, the PCI root bridge needs minimal setup. Typically, the PCI root bridge decodes the following:

- The entire bus range on Segment 0
- The entire I/O space of the processor
- All the memory above the top of system memory

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol

- One instance of PCI Root Bridge I/O Protocol

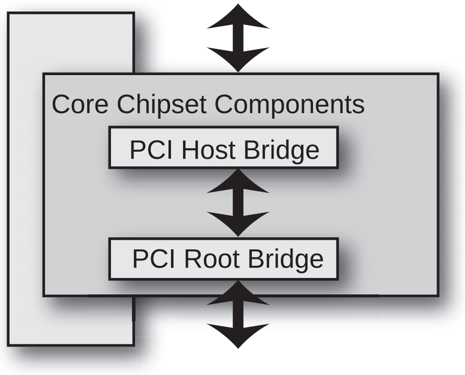


Figure 10.5 Desktop System with One PCI Root Bridge

Sample Server System with Four PCI Root Bridges

Figure 10.6 shows an example of a larger server with one PCI host Bus with four PCI root bridges (RBs). The PCI devices that are attached to the PCI root bridges are all part of the same coherency domain, which means they share the following:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

As a result, each PCI root bridge must get resources out of a common pool. Each PCI root bridge produces one PCI local bus that can contain PCI devices on the motherboard or PCI slots. The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Four instances of the PCI Root Bridge I/O Protocol

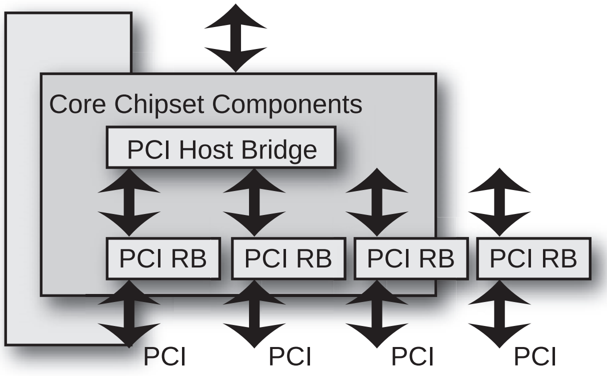


Figure 10.6 Server System with Four PCI Root Bridges

Sample Server System with 2 PCI Segments

Figure 10.7 shows an example of a server with one PCI host bus and two PCI root bridges (RBs). Each of these PCI root bridges is on a different PCI segment, which allows the system to have up to 512 PCI buses. A single PCI segment is limited to 256 PCI buses. These two segments do not share the same PCI configuration space, but they do share the following, which is why they can be described with a single PCI host bus:

- A common PCI I/O space
- A common PCI memory space
- A common PCI prefetchable memory space

The firmware for this platform would produce the following:

- One instance of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

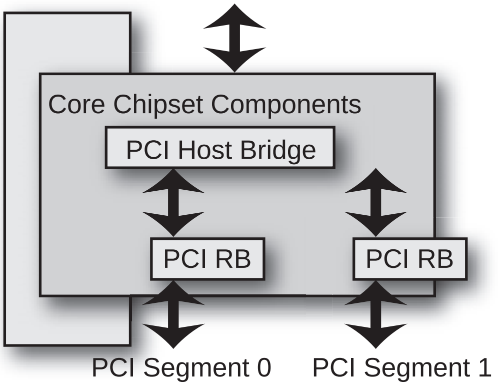


Figure 10.7 Server System with 2 PCI Segments

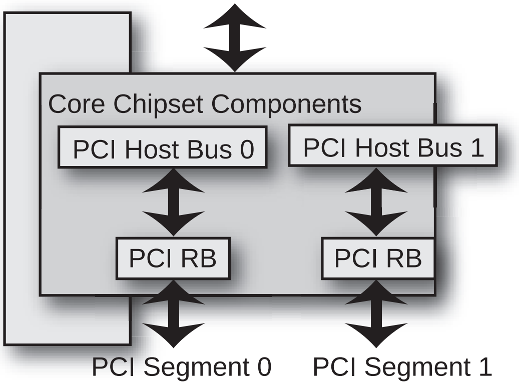


Figure 10.8 Sample Server System with Two PCI Host Buses

Figure 10.8 shows a server system with two PCI host buses and one PCI root bridge (RB) per PCI host bus. Like the server system with 2 PCI segments, this system supports up to 512 PCI buses, but the following resources are not shared between the two PCI root bridges:

- PCI I/O space
- PCI memory space
- PCI prefetchable memory space

The firmware for this platform would produce the following:

- Two instances of the PCI Host Bridge Resource Allocation Protocol
- Two instances of the PCI Root Bridge I/O Protocol

PCI Root Bridge I/O

The interfaces provided in the PCI Root Bridge I/O Protocol are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources.

The PCI Root Bridge I/O Protocol allows for future innovation of the platform. It abstracts device-specific code from the system memory map. This allows system designers to make changes to the system memory map without impacting platform-independent code that is consuming basic system resources.

PCI Root Bridge I/O Protocol instances are either produced by the system firmware or by an EFI driver. When a PCI Root Bridge I/O Protocol is produced, it is placed on a device handle along with an EFI Device Path Protocol instance. The PCI Root Bridge I/O Protocol does not abstract access to the chipset-specific registers that are used to manage a PCI Root Bridge. This functionality is hidden within the system firmware or the EFI driver that produces the handles that represent the PCI Root Bridges.

```
Protocol Interface Structure

typedef struct _EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL {
  EFI_HANDLE
ParentHandle;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM
PollMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_POLL_IO_MEM
PollIo;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS           Mem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS           Io;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ACCESS           Pci;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_COPY_MEM
CopyMem;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_MAP              Map;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_UNMAP
Unmap;
```

```
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_ALLOCATE_BUFFER
AllocateBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FREE_BUFFER
FreeBuffer;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_FLUSH
Flush;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_GET_ATTRIBUTES
GetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_SET_ATTRIBUTES
SetAttributes;
  EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL_CONFIGURATION
Configuration;
  UINT32
SegmentNumber;
} EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL;
```

- *ParentHandle* – Gives the EFI_HANDLE of the PCI Host Bridge of which this PCI Root Bridge is a member.
- *PollMem* - Polls an address in memory mapped I/O space until an exit condition is met, or a timeout occurs.
- *PollIo* - Polls an address in I/O space until an exit condition is met, or a timeout occurs.
- *Mem* - Allows reads and writes for memory mapped I/O space.
- *Io* - Allows reads and writes for I/O space.
- *Pci* - Allows reads and writes for PCI configuration space.
- *CopyMem* - Allows one region of PCI root bridge memory space to be copied to another region of PCI root bridge memory space.
- *Map* - Provides the PCI controller–specific addresses needed to access system memory for DMA.
- *Unmap* - Releases any resources allocated by Map().
- *AllocateBuffer* - Allocates pages that are suitable for a common buffer mapping.
- *FreeBuffer* – Frees pages that were allocated with AllocateBuffer().
- *Flush* - Flushes all PCI posted write transactions to system memory.

- *GetAttributes* - Gets the attributes that a PCI root bridge supports setting with SetAttributes(), and the attributes that a PCI root bridge is currently using.
- *SetAttributes* - Sets attributes for a resource range on a PCI root bridge.
- *Configuration* - Gets the current resource settings for this PCI root bridge.
- *SegmentNumber* - The segment number that this PCI root bridge resides.

PCI I/O

The interfaces provided in the PCI I/O Protocol are for performing basic operations to memory, I/O, and PCI configuration space. The system provides abstracted access to basic system resources to allow a driver to have a programmatic method to access these basic system resources. The main goal of this protocol is to provide an abstraction that simplifies the writing of device drivers for PCI devices. This goal is accomplished by providing the following features:

- A driver model that does not require the driver to search the PCI busses for devices to manage. Instead, drivers are provided the location of the device to manage or have the capability to be notified when a PCI controller is discovered.
- A device driver model that abstracts the I/O addresses, Memory addresses, and PCI Configuration addresses from the PCI device driver. Instead, BAR (Base Address Register) relative addressing is used for I/O and Memory accesses, and device relative addressing is used for PCI Configuration accesses. The BAR relative addressing is specified in the PCI I/O services as a BAR index. A PCI controller may contain a combination of 32-bit and 64-bit BARs. The BAR index represents the logical BAR number in the standard PCI configuration header starting from the first BAR. The BAR index does not represent an offset into the standard PCI Configuration Header because those offsets will vary depending on the combination and order of 32-bit and 64-bit BARs.
- The Device Path for the PCI device can be obtained from the same device handle that the PCI I/O Protocol resides.

- The PCI Segment, PCI Bus Number, PCI Device Number, and PCI Function Number of the PCI device if they are required. The general idea is to abstract these details away from the PCI device driver. However, if these details are required, then they are available.
- Details on any nonstandard address decoding that are not covered by the PCI device's Base Address Registers.
- Access to the PCI Root Bridge I/O Protocol for the PCI Host Bus for which the PCI device is a member.
- A copy of the PCI Option ROM if it is present in system memory.
- Functions to perform bus mastering DMA. This includes both packet based DMA and common buffer DMA.

```
Protocol Interface Structure

typedef struct _EFI_PCI_IO_PROTOCOL {
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM         PollMem;
  EFI_PCI_IO_PROTOCOL_POLL_IO_MEM         PollIo;
  EFI_PCI_IO_PROTOCOL_ACCESS              Mem;
  EFI_PCI_IO_PROTOCOL_ACCESS              Io;
  EFI_PCI_IO_PROTOCOL_CONFIG_ACCESS       Pci;
  EFI_PCI_IO_PROTOCOL_COPY_MEM            CopyMem;
  EFI_PCI_IO_PROTOCOL_MAP                 Map;
  EFI_PCI_IO_PROTOCOL_UNMAP               Unmap;
  EFI_PCI_IO_PROTOCOL_ALLOCATE_BUFFER     AllocateBuffer;
  EFI_PCI_IO_PROTOCOL_FREE_BUFFER         FreeBuffer;
  EFI_PCI_IO_PROTOCOL_FLUSH               Flush;
  EFI_PCI_IO_PROTOCOL_GET_LOCATION        GetLocation;
  EFI_PCI_IO_PROTOCOL_ATTRIBUTES          Attributes;
  EFI_PCI_IO_PROTOCOL_GET_BAR_ATTRIBUTES
GetBarAttributes ;
  EFI_PCI_IO_PROTOCOL_SET_BAR_ATTRIBUTES
SetBarAttributes;
  UINT64                                  RomSize;
  VOID                                    *RomImage;
} EFI_PCI_IO_PROTOCOL;
```

- *PollMem* - Polls an address in PCI memory space until an exit condition is met, or a timeout occurs.
- *PollIo* - Polls an address in PCI I/O space until an exit condition is met, or a timeout occurs.
- *Mem* - Allows BAR relative reads and writes for PCI memory space.

- *Io* - Allows BAR relative reads and writes for PCI I/O space.
- *Pci* - Allows PCI controller relative reads and writes for PCI configuration space.
- *CopyMem* - Allows one region of PCI memory space to be copied to another region of PCI memory space.
- *Map* - Provides the PCI controller–specific address needed to access system memory for DMA.
- *Unmap* - Releases any resources allocated by Map().
- *AllocateBuffer* - Allocates pages that are suitable for a common buffer mapping.
- *FreeBuffer* - Frees pages that were allocated with AllocateBuffer().
- *Flush* - Flushes all PCI posted write transactions to system memory.
- *GetLocation* - Retrieves this PCI controller's current PCI bus number, device number, and function number.
- *Attributes* - Performs an operation on the attributes that this PCI controller supports. The operations include getting the set of supported attributes, retrieving the current attributes, setting the current attributes, enabling attributes, and disabling attributes.
- *GetBarAttributes* - Gets the attributes that this PCI controller supports setting on a BAR using SetBarAttributes(), and retrieves the list of resource descriptors for a BAR.
- *SetBarAttributes* - Sets the attributes for a range of a BAR on a PCI controller.
- *RomSize* – Gives the size, in bytes, of the ROM image.
- *RomImage* – Returns a pointer to the in memory copy of the ROM image. The PCI Bus Driver is responsible for allocating memory for the ROM image, and copying the contents of the ROM to memory. The contents of this buffer are either from the PCI option ROM that can be accessed through the ROM BAR of the PCI controller, or from a platform-specific location. The Attributes() function can be used to determine from which of these two sources the RomImage buffer was initialized.

Block I/O

The Block I/O Protocol is used to abstract mass storage devices to allow code running in the EFI boot services environment to access them without specific knowledge of the type of device or controller that manages the device. Functions are defined to read and write data at a block level from mass storage devices as well as to manage such devices in the EFI boot services environment.

The Block interface constructs a logical abstraction of the storage device. Figure 10.9 shows how a typical device that has multiple partitions will have a variety of Block interfaces constructed on it. For example, a partition that is a logical designation of how a disk might be apportioned will have a block interface for it. It should be noted that a particular storage device will have a block interface that has a scope that spans the entire storage device, and the logical partitions will have a scope that is a subset of the device. For instance, in the example shown in Figure 10.8, Block I/O #1 has access to the entire disk, while Block I/O #2 has its first LBA starting at the physical location of the partition it is associated with.

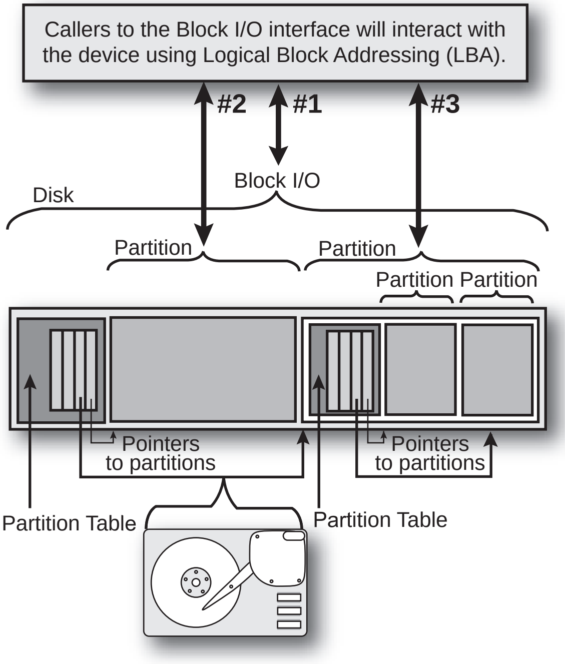


Figure 10.9 Software Layering of the Storage Device

```
Protocol Interface Structure

typedef struct _EFI_BLOCK_IO_PROTOCOL {
  UINT64                    Revision;

  EFI_BLOCK_IO_MEDIA        *Media;

  EFI_BLOCK_RESET           Reset;
  EFI_BLOCK_READ            ReadBlocks;
  EFI_BLOCK_WRITE           WriteBlocks;
  EFI_BLOCK_FLUSH           FlushBlocks;

} EFI_BLOCK_IO_PROTOCOL;
```

- *Revision* - The revision to which the block IO interface adheres. All future revisions must be backward compatible. If a future version is not backward compatible it is not the same GUID.

- *Media* - A pointer to the EFI_BLOCK_IO_MEDIA data for this device. Type EFI_BLOCK_IO_MEDIA is defined in the next code sample.
- *Reset* - Resets the block device hardware.
- *ReadBlocks* - Reads the requested number of blocks from the device.
- *WriteBlocks* - Writes the requested number of blocks to the device.
- *FlushBlocks* - Flushes and cache blocks. This function is optional and only needs to be supported on block devices that cache writes.

```
Protocol Interface Structure

typedef struct {
  UINT32              MediaId;
  BOOLEAN             RemovableMedia;
  BOOLEAN             MediaPresent;

  BOOLEAN             LogicalPartition;
  BOOLEAN             ReadOnly;
  BOOLEAN             WriteCaching;

  UINT32              BlockSize;
  UINT32              IoAlign;

  EFI_LBA             LastBlock;
} EFI_BLOCK_IO_MEDIA;
```

Disk I/O

The Disk I/O protocol is used to abstract the block accesses of the Block I/O protocol to a more general offset-length protocol. The firmware is responsible for adding this protocol to any Block I/O interface that appears in the system that does not already have a Disk I/O protocol. File systems and other disk access code utilize the Disk I/O protocol.

The disk I/O functions allow I/O operations that need not be on the underlying device's block boundaries or alignment requirements. This is done by copying the data to/from internal buffers as needed to provide the proper requests to the block I/O device. Outstanding write buffer data is flushed by using the Flush() function of the Block I/O protocol on the device handle.

The firmware automatically adds a Disk I/O interface to any Block I/O interface that is produced. It also adds file system, or logical block I/O, interfaces to any Disk I/O interface that contains any recognized file system or logical block I/O devices. EFI compliant firmware must automatically support the following required formats:

- The EFI FAT12, FAT16, and FAT32 file system type.
- The legacy master boot record partition block. (The presence of this on any block I/O device is optional, but if it is present the firmware is responsible for allocating a logical device for each partition).
- The extended partition record partition block.
- The El Torito logical block devices.

The Disk I/O interface provides a very simple interface that allows for a more general offset-length abstraction of the underlying Block I/O protocol.

```
Protocol Interface Structure

typedef struct _EFI_DISK_IO_PROTOCOL {
  UINT64              Revision;
  EFI_DISK_READ       ReadDisk;
  EFI_DISK_WRITE      WriteDisk;
} EFI_DISK_IO_PROTOCOL;
```

- *Revision* - The revision to which the disk I/O interface adheres. All future revisions must be backwards compatible. If a future version is not backwards compatible, it is not the same GUID.
- *ReadDisk* - Reads data from the disk.
- *WriteDisk* - Writes data to the disk.

Simple File System

The Simple File System protocol allows code running in the EFI boot services environment to obtain file based access to a device. The Simple File System protocol is used to open a device volume and return an EFI File Handle that provides interfaces to access files on a device volume.

This protocol is a bit different from most, since its use exposes a secondary protocol that will directly act on the device on top of which the Simple File System was layered. Figure 10.10 illustrates this concept.

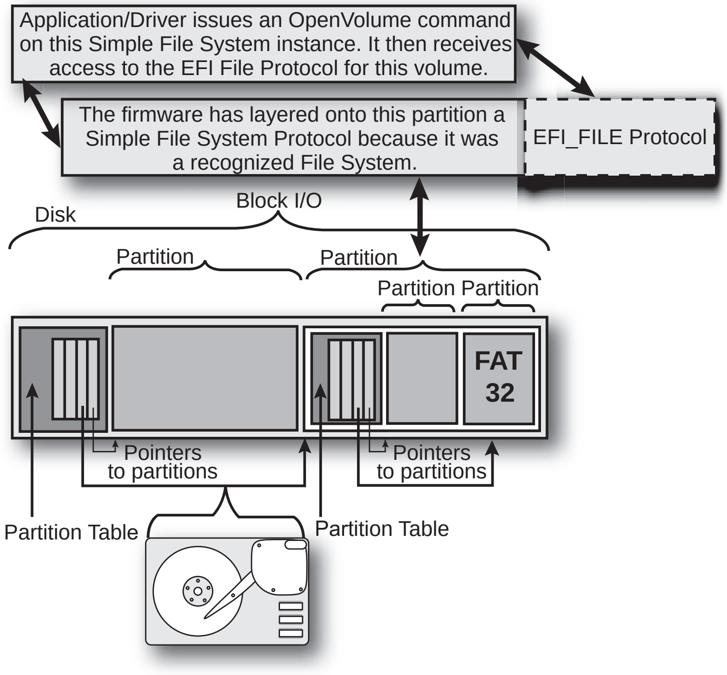

Application/Driver issues an OpenVolume command on this Simple File System instance. It then receives access to the EFI File Protocol for this volume.
The firmware has layered onto this partition a Simple File System Protocol because it was a recognized File System.
EFI_FILE Protocol
Block I/O
Disk
Partition
Partition
Partition
Partition
FAT 32
Pointers to partitions
Pointers to partitions
Partition Table
Partition Table

Figure 10.10 Simple File System Software Layering

```
Protocol Interface Structure

typedef struct {
     UINT64              Revision;
     EFI_VOLUME_OPEN     OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

- *Revision* - The version of the EFI Simple File System Protocol. The version specified by this specification is 0x00010000. All future revisions must be backward compatible. If a future version is not backward compatible, it is not the same GUID.
- *OpenVolume* - Opens the volume for file I/O access.

EFI File Protocol

On requesting the file system protocol on a device, the caller gets the instance of the Simple File System protocol to the volume. This interface is used to open the root directory of the file system when needed. The caller must Close() the file handle to the root directory and any other opened file handles before exiting. While open files are on the device, usage of underlying device protocol(s) that the file system is abstracting must be avoided. For example, when a file system is layered on a DISK_IO / BLOCK_IO protocol, direct block access to the device for the blocks that comprise the file system must be avoided while open file handles to the same device exist.

A file system driver may cache data relating to an open file. A Flush() function is provided that flushes all dirty data in the file system, relative to the requested file, to the physical medium. If the underlying device may cache data, the file system must inform the device to flush as well.

```
Protocol Interface Structure

typedef struct _EFI_FILE {
  UINT64                      Revision;
  EFI_FILE_OPEN               Open;
  EFI_FILE_CLOSE              Close;
  EFI_FILE_DELETE             Delete;
  EFI_FILE_READ               Read;
  EFI_FILE_WRITE              Write;
  EFI_FILE_GET_POSITION       GetPosition;
  EFI_FILE_SET_POSITION       SetPosition;
  EFI_FILE_GET_INFO           GetInfo;
  EFI_FILE_SET_INFO           SetInfo;
  EFI_FILE_FLUSH              Flush;
} EFI_FILE;
```

- Revision - The version of the EFI_FILE interface. The version specified by this specification is 0x00010000. Future versions are required to be backward compatible to version 1.0.
- Open - Opens or creates a new file.
- Close - Closes the current file handle.
- Delete - Deletes a file.
- Read - Reads bytes from a file.
- Write - Writes bytes to a file.

- GetPosition - Returns the current file position.
- SetPosition - Sets the current file position.
- GetInfo - Gets the requested file or volume information.
- SetInfo - Sets the requested file information.
- Flush - Flushes all modified data associated with the file to the device.

Chapter 11

Information Passing

> *In the end the seed took, and from the soil of Australia came forth straight lines and rational curves. Rectilinear offices and hotels. The curve of freeways and the smooth arc of theatre and opera house. Solids of the most intricate and sophisticated geometry.*
>
> —Michael Frayn

This is a chapter about infrastructure. It is about the Framework's equivalent of the plumbing, electrical, waste, and information systems that entwine the modern world. Unlike the other chapters in this book, the different constructs are united by concept rather than function. They also share a family resemblance brought on by the requirements of the environment. The differences between them are brought upon by the requirements being addressed and the data being processed: most houses receive both electricity and water but mixing the two is not beneficial.

The constructs discussed in this chapter include:

- **Variables,** which serve as the non-volatile repository for system data and as a communication channel with operating systems.
- **Hand-off blocks (HOBs),** which are the main conduit across the PEI to DXE transition.
- **The data hub**, which serves as a repository to accumulate and then disperse data mainly intended for external consumption.
- **The Human Interface Infrastructure** (HII), which centralizes management of the configuration of modules by users and others.

Similarities and Differences

The constructs described here share some qualities:

- They are conduits for data. The callable interfaces (protocol member functions, for example) themselves serve only to enable data transfer and management between modules. They do little or no editing of the data during the transmission. Depending on the construct, they may provide support for others to do editing. The electrical infrastructure may change voltages for efficiency on the trip from power station to house too.
- They have well defined sets of producers and consumers for the data. This means for example that relevant routing information is provided so the consumers can determine the applicability of the data packets unambiguously.
- They serve to loosen the binding between modules. As an example, consider the case of a memory initialization PEIM. During the course of its efforts, it discovers valuable information concerning the available memory slots and types of memory present in the system. No other module has this data available, yet it is particularly valuable to system managers as a part of inventory. Using a HOB, the PEIM can bridge the gap to DXE. From DXE a consumer can pass the data to the data hub. Consumers of the data hub packets can then provide the data via the required OS or pre-OS interfaces.
- They serve to loosen the binding between producers and the standards the consumers are required to follow. In the memory initialization PEIM example, the data could be consumed by any of the several competing system management standards. The memory PEIM doesn't have to know that the final consumer is IPMI or SMBIOS or some new interface.
- They function without requiring services based on architectural protocols. They function early enough that any module in PEI and beyond can make use of the interfaces defined for that phase. In many cases, it would be hard to invent services that *did* use architectural protocols so this attribute falls out rather easily. On the other hand, it is still important. An early DXE driver can provide its configuration information data to the HII database. For example, a driver can provide that data prior to receiving a start request. Once the driver *does* receive a start request and enumer-

ates the items downstream from its associated hardware, it can then update the configuration information to describe that. This does require some forethought on the part of the designer to allow both the generic and more specific versions to make sense to the end user.

- They are position-independent. The data provided may be moved from location to location in memory. This, in turn, affects the design of the data structures passed around and some of the requirements on that data that the programmatic interfaces and validation tests can't otherwise enforce. In particular, pointers of the form

  ```
  MY_STRUCT *MyPointer;
  ```

  from one point in the data structure to another point in that structure are prohibited. The pointer becomes invalid if the structure is moved to another location. Instead, the structures should use self-relative offsets and length fields, which can be copied. This requirement can be subtle. One HOB entry describes the top of RAM, which is a pointer. This sort of pointer is allowed because it points to an unchangeable location outside the data structure itself.

If these constructs share so many qualities, why aren't they unified into a single infrastructure? There are several reasons roughly analogous to why there isn't a single type of physical infrastructure in a large city.

There are historical reasons. Variables have been a part of the EFI specification since its initial drafts. The other pieces were invented for the Framework and manifest themselves indirectly or not at all in EFI. Most commonly used variables are nonvolatile (they stay valid when power is removed) whereas the other instances are recreated at each boot.

The pieces perform different functions. HOBs leap over a single phase boundary. The limitations imposed by PEI space are unnecessarily constraining in DXE. The data hub is the most generic piece. Its generic nature is its benefit but also makes it unsuitable for HII data, which provides support for a more constrained more editable type of data.

Variables

Variables are intended as a message-passing facility between EFI and the operating system. They may also be used within the Framework itself. Unlike the other mechanisms in this chapter, variables are accessible both before and after ExitBootServices.

To understand how variables are intended to be used, it is helpful to understand why they are called variables in the first place. Unix provides its users with what it calls *environment variables,* which are arbitrary strings associated with other arbitrary (ASCII) labels. Applications can then look up a label and find the string. In their lineage, environment variables are friendlier versions of the Fortran logical unit construct (where 5 is always standard input, 6 always standard output, for example).

EFI variables serve the same purpose as environment variables and operate in an analogous fashion. Their names (keys) are both a GUID and a Unicode string. Both are part of the name and both must be provided. The GUID serves the same purpose as elsewhere in EFI and the Framework: to provide uniqueness and avoid collisions. It is valid, however, to have several variables with the same name. For example, a platform might choose to have its setup configuration data stored in a variable with a given GUID and the name "Setup" and store its default values in another variable with the same GUID and the name "Defaults".

The structure of the variable data (value) is defined by the producers and consumers of the data.

Using Variables

Variables may be created or updated via `SetVariable`, retrieved via `GetVariable`, and discovered via `GetNextVariableName`. All are runtime (`gRT`) functions.

Variable Volatility

Variables are available in two types: volatile and nonvolatile. The nonvolatile variables survive system resets whereas the volatile ones don't. It is up to the system designers to ensure that the nonvolatile variables are updated in a manner sufficiently fault tolerant so that, once SetVariable returns successfully, the variable is ensconced in the nonvolatile storage and will be available on subsequent boots.

Size Concerns

Most variables are nonvolatile. This presents a problem to the system designer. In all but the most complex systems, the space that the system has to store nonvolatile data is fixed when the system is built. On the other hand, applications can write arbitrarily large numbers of arbitrarily large variables. The only solution is to reject requests that overflow the variable storage space, thus effectively putting the onus for using the limited resource on the callers.

The EFI specification does not define the actual amount of variable space required for a system. More importantly, the specification refrains from defining the amount that must be available when the system is shipped. This is intentional since it is likely that the amount required for certain classes of systems may be larger than for others. Also, the amount that is considered "huge" one year may be "fairly small" a few years later.

One area that is also not clear in the documentation is who gets to use variables. In particular, it would be tempting for option ROMs to using variables but they have compelling reasons to store their data in their own configuration areas.

First, if the card containing the option ROM is removed, any data left by that option ROM in the variable store is now stale. There is no way to ascertain that the data in a variable is *only* used by a now-departed card.

Secondly, if the configuration information in the card now resides in the variable space, and the card is moved to a new system, the card loses all of its configuration, something not particularly friendly to the user.

Furthermore, option ROMs storing their data in variables present the system designer with another question with no obvious answer: how much space to allocate for option ROMs. The usual solution is to assume that they require no space. Now, two negative outcomes become likely. Either the option ROM cannot store its configuration because the variable space is full or the option ROM can store its data successfully and the operating system can't.

Common Variables

Sprinkled throughout the EFI specification are definitions of particular variables used for communication between the operating system and EFI. Most of these enable the operating system to pass configuration that it has discovered or requires back to EFI for use in subsequent boots. This includes boot order and language data.

Summary

Use variables sparingly.

Bridging the Phase Gap: Hand-Off Blocks (HOBs) –

The Framework defines several boot phases. Some of the boundaries between the phases are more solid than others. The most distinct chasm between two boundaries occurs between the PEI and DXE phases. PEI is a RAM-limited environment whereas DXE is one that may use a "reasonable" amount of memory. The concept is to do what needs to be done in PEI to get memory and then leave for the greener pastures of DXE.

PEI, before and during memory initialization, accumulates data that DXE Framework requires as a part of its initialization and the DXE drivers may require as a part of their execution. Data needs to be transferred from PEI to DXE.

PEI, however, is constrained by its low memory environment. It needs a suitably memory-conserving and primitive data-passing mechanism. It is provided with one via the Hand Off Blocks (HOBs).

Keeping PHIT

The HOB data structure is organized as a contiguous range of memory with a known header called the Phase Handoff Information Table (PHIT) and with other HOBs following. Each HOB has a common header that includes a length allowing for easy HOB scanning. The PHIT and all current HOBs are together known as the *HOB List*.

The location of the PHIT itself is available via a PEI Framework call (`GetHobList`). The PHIT indicates the maximum address of memory available to store HOBs and the current end of the HOB table. Adding a HOB is accomplished via the `CreateHob` call. This service handles the job of adjusting the PHIT end-of-list pointer and copying the new HOB data in.

Except for the pointers in the PHIT, no HOBs should contain pointers to other HOBs. This is because part of the process of handing the HOBs off from PEI to DXE involves copying the HOB list to a new location in "real" RAM, which is almost certainly at different addresses. No pointers between HOBs are updated during this copy process, so the pointers are rendered invalid. Pointers to areas outside HOBs are generally valid. For example, the address of the top of RAM discovered during PEI is a pointer. Copying the HOB from place to place does not make this address invalid.

Using HOBs

HOBs can be used for two types of communication. The most obvious is between producers in the PEI phase and consumers in DXE. The second is between producers and consumers both in the PEI phase. HOBs thus end up being a memory allocation scheme for the limited RAM assumed in PEI space. It is not generally safe to deallocate HOBs used for this purpose as this would involve moving other HOBs. Moving other HOBs would invalidate pointers to the moved HOBs held by other PEIMs.

As a part of their more traditional job (PEI to DXE communication) several types of HOBs are specified as a part of the Framework including: Memory, CPU, Resource, and FV. Some of these HOBs are used by the DXE initialization code to initially populate the GCD (Global Coherency Domain) table.

HOBs may also be used for PEIM to DXE driver communication. This allows the PEIM to be smaller and may avoid duplicating code. The GUID identifying the HOB need only be known and agreed upon between (the authors of) the PEIM and driver.

HOBs in DXE

The DXE loader consumes the HOB list. It uses CPU, Firmware Volume, and Resource HOBs to populate the initial GCD table and the PHIT table to ensure that it doesn't destroy the HOB list in the process of using it. Other HOBs are available via the EFI Configuration Tables, which are available from the EFI System Table. HOBs may only be consumed in DXE, not produced.

Summary

Use HOBs sparingly.

The Human Interface Infrastructure (HII)

The time when it was acceptable for computers to speak only English—and that not particularly well—is over. Computers are supposed to be user-friendly or generate a good end-user experience. They are supposed to be (even more) manageable and dependable too.

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 ground that HII covers has been studied and analyzed by our operating system resident brethren and their valuable discoveries have formed the basis for the Framework's support.

The Framework divides configuration support into four categories:

- Keyboards: Full operating-system–equivalent keyboard abstraction is well beyond the scope of firmware. On the other hand, the Framework offers the abstractions required to provide localized keyboard support on the level required for firmware.
- Fonts: Unlike BIOS, the EFI video interfaces themselves do not carry mechanisms to generate characters.
- Strings: Localization methods for strings are well known and common in languages today. Referring to strings via token is quite common. The exceptions (such as date, time, and currency localization) are also well known.
- Forms: With the popularity of HTML and then XML and its progeny, markup languages have become familiar to many. These languages are, however, not particularly space-efficient nor do they lend themselves to localization. The Framework provides the Internal Forms Representation, a language that can be transformed into common markup languages but retains its sensitivities to its firmware environment.

The HII infrastructure defines the structures of the data describing each of these and mechanisms for contributing and accessing that data via a dynamically created *HII Database* and associated protocols.

These combine to provide a surprising rich infrastructure that addresses both end-user and management needs.

Terminology

Internationalization: The use of icons and other non–locale-specific constructs to attempt to make an interface usable by all languages. Icons such as stop signs, the use of blue for information, and the use of mice, are international. Suitable for simple interfaces (such as to select a language or time zone).

Localization: The use of constructs to adapt an interface to a particular language and/or place.

Logographic: Non-phonetic written languages such as Chinese, Japanese, and Korean ("CJK") which use one symbol per word rather than a symbol per sound or syllable.

The Configuration Model

Configuration, as illustrated in Figure 11.1, is modeled as circular. Starting with boot, drivers read their nonvolatile data to configure their hardware. They also provide the package of data to the HII Database. If setup is run, it accesses the database much as a browser would access a Web site. It then drives the resulting configuration back to the drivers' NV RAM.

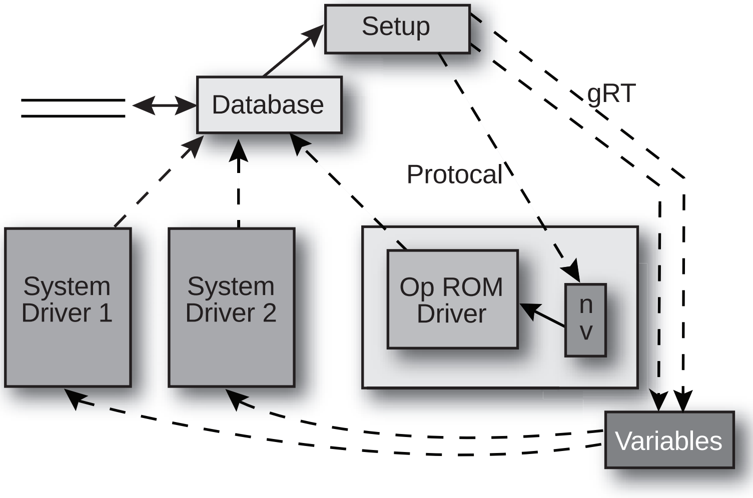


Figure 11.1 Configuration Flow

The drivers can have some of their data pre-built. Most drivers, however, need to report dynamically derived data. The HII Database enables this as well. In the end, some data may be so dynamic that it changes from second to second. (The most obvious candidate, the real-time clock, is actually treated as a special case by HII. Consider system temperature as an example.) In this case, IFR supports call-backs to the drivers to provide real-time information. Although available, use is discouraged where possible.

The reason call-backs are discouraged is important. This model allows for other customers of the HII database. Drivers that interface with remote pre-boot applications can also send across the HII Database (an "extract" command gets the database in a standard form) and receive the modified configuration data back.

As illustrated in Figure 11.2, the extracted data can be left around for OS-based applications to use, either via an browser application present within the operating system or via a scripting language for management.

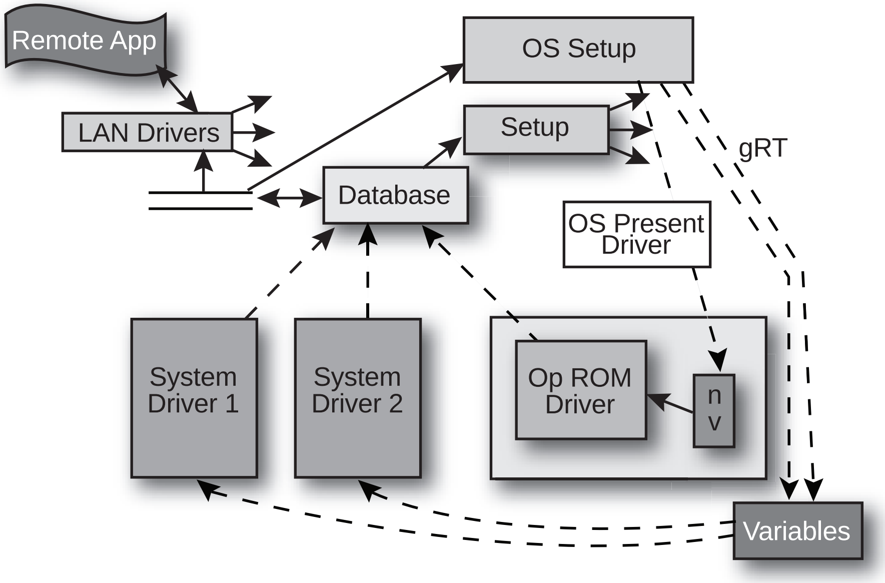


Figure 11.2 More Configuration Flow

The loop gets more complex but not for the driver. The driver and storage are completely abstracted from this level of support.

Keyboards

The Framework strikes a balance between the complexity of full keyboard localization and the requirements of user friendly interfaces.

The Issue

The default keyboard for the Framework is the U. S. English keyboard. Most end-users would are not aware of this as the input of most firmware setup engines is internationalized enough (cursor keys, Fn keys, Enter, Tab, numbers) to abstract away the requirement.

Input of hexadecimal numbers, passwords, pass-keys, and the like make internationalization impossible. Examining the U. S. English and French keyboards, we see that the Q, A, and Z keys are in different locations. Both the legacy ("PS/2"-style) and USB (HID) keyboards return keystroke data based on key location, not key cap. That is, a "4" from a PS/2 keyboard or a "31" from a USB keyboard both mean "a" on an English keyboard but "q" on a French one. Entering a hexadecimal number with the wrong translation table is nearly impossible.

This complexity is magnified many times with logographic languages. In Japanese, for example, Kana alphabets are phonetic but Kanji, the preferred form, is not. Complex tools called IMEs are used to watch the input Kana stream for Kanji equivalents and, upon user agreement, change to Kanji.

The Infrastructure

The Framework solution falls somewhere between assuming everyone speaks U. S. English and IME support (which requires advanced features elsewhere in the I/O stream to support).

Four translations may be associated with each key: the unmodified key and the key with the Shift, Control, or Alt (Gr) pressed. Each keystroke is mapped to a Unicode character. The modifiers are stored as deltas from the U. S. English keyboard so Latin-1 languages are much smaller and so that internationalized keys (Fn, numeric, cursor control, and so on) may remain unmodified.

Fonts

The protocols proposed to interface with video in EFI (UGA, GOP) do not provide for character generation support. That is, they don't support fonts. On the other hand, EFI states that it provides display support for LATIN-1 (generally that means Western European) languages.

EFI and the Framework need common character size and representation support both so that calling programs can know what is available and so extensions for Cyrillic, CJK, Arabic, and Hebrew, for example, can be created.

HII defines standards based on fairly simple calculations. A minimum height in pixels of 480 divided by 25 lines on a standard CRT gives a little over 19 pixels per character. For efficiency of storage, 8 wide gives a fairly pleasing ratio. Logographic characters are not readable in 8 pixels so are extended by doubling their width to 16 pixels. Characters that are 8 pixels wide are known as *narrow,* while characters that are 16 pixels side are *wide*. It turns out the characters are also quite readable at 800 × 600, a required screen resolution for EFI's proposed graphics support, providing 31 lines of 100 narrow characters.

The data structures store 16 ×19 characters roughly as two 8 ×19 characters for simplicity in dividing the characters for overlapping windows. Associated with each character, regardless of size, is its Unicode UTF-16 weight (for example: 0x0037 for a "7") and a flag word. The flag byte is mainly used to indicate if the character is non-spacing, a particularly clever but insidious Unicode invention.

A non-spacing character is one whose bit patterned is to be effectively OR'd with the following pattern. This is used to enable unusual accent patterns as found in some languages such as Vietnamese. The concept is clever. The implementation is disappointing: the non-spacing characters are sprinkled unpredictably throughout Unicode's weights. The solution here is to let the character describe itself.

These data structures are provided in a narrow list and then a wide list. Each list must be sorted by Unicode weight from lowest to highest, enabling binary search on the fixed size (22 and 44 byte) structures.

Note that each driver using fonts that are not Latin-1 is required to provide those fonts. The font database, however, is only required to keep one example of each font weight and size (narrow and wide) so a driver may end up using some of its glyphs (the HII name for an actual representation of a single weight in a single size) and some of other drivers.

The UEFI committee is contemplating extensions and updates to this format to enable support for other heights and multiple font styles.

Strings

Fonts and keyboards are shared by all drivers in the system. Strings and forms are local to the driver that owns them.

The common way to support localization is to tokenize string references and then have translations created for each snippet of text. In HII, the tokens are 16 bits and the snippets of text are called *strings*.

Tokens are intended to be allocated monotonically increasing starting from one (1, 2, 3, 4...). Token zero is reserved for a non-present string.

The string data structure consists of a header, an array of indexes, and the strings themselves, null (Unicode weight 0x0000) terminated.

The header describes the language in both ISO-639-2 format (or 3166 depending on UEFI whim) and text in that language ("Cymraeg," not "Welsh," for example)

The array is indexed via 2 times the token value, thus allowing for 65,535 strings per driver—more strings than there are sentences in this book. The offsets are 16 bits allowing for over 65,535 characters total.

Although most strings will probably be known when the driver is built, it is quite common for some strings to be only derived during a driver's initialization. The HII Database protocols have functions to update strings and create new ones.

Again, the UEFI working groups and their sub-teams are contemplating extensions and updates mainly to enhance compression.

Forms

Forms are generally considered that subset of markup languages that deals with user input. Internet forms are encountered when purchasing items, filling out surveys, and the like.

A goal of EFI has been to simplify access to the firmware user interface. In BIOS, the user often had to know any of several hot-keys to gain access to the various setup programs for various parts of the system (the system BIOS and the option ROMs). The goal was to have a single mechanism to access a boot manager that could present all configuration in an easier to access easier to use format. As well, the option ROMs wouldn't have to carry their own setup engines that duplicated the basic functionality of the system's setup engine. Forms are a way to address this.

Forms are built upon the other pieces of the HII: keyboards, fonts, and strings. In particular, HII forms use string tokens as opposed to embedded strings.

General Format

Forms are represented by a language known as the Internal Forms Representation (IFR), whose closest analog is the binary language of Complex Instruction Set (CISC) processors: an ordered sequence of variable length binary structures known as *instructions*. A separate language, Visual Forms Representation (VFR) has been defined as a human-readable equivalent to IFR that is compiled/assembled into IFR. It is technically not a part of the Framework. Examples below are in a pidgin VFR, which allows complicating parts to be ignored.

The first byte in each structure is an opcode and the second is the length in bytes of the instruction *not including the opcode or length*. The shortest instruction is, thus, two bytes long containing an opcode and a byte of binary zero. The instructions and all fields in them are byte aligned and follow the EFI coding guidelines (little endian, and so on). It is up to the browser to realign any fields it has to.

String references are represented via HII string tokens.

Storage references are via two methods. The first, known as the offset-width style provides a zero-based byte offset and a width in bytes into a standard EFI variable, as described earlier in this chapter. The variable GUIDs' names are provided via separate opcodes to conserve space. The second, a name value, provides a name or value index for forming into the standard HTML / XML name=value&name=value&… pairs. The offset of the offset-width style is reused as the string token, since both are 16 bits, to describe names. Values may only be set for nonnumeric items. Numeric items are returned in decimal format.

HTML and IFR

Those with experience with HTML who are new to HII forms are likely to have a fairly quick ramp but feel somewhat constrained by their new surroundings.

IFR supports many HTML forms constructs but at a subset level. For example, instead of 8 types of headers, there is only one. Instead of several different types of enumerated (choose one among several possibilities) input types, IFR offers only "one of".

HTML offers the escape to scripting languages such as JavaScript if things get too tough. IFR must stand alone.

The decisions as to which features of markup languages to support, which to discard, and which to modify or reinvent were motivated by the projected environments systems using the Framework might find themselves.

The use of string tokens rather than embedded strings (as in HTML) was motivated by the need to conserve space while localizing.

The reduction in number of constructs such as headers was based on experience with forms of the targeted complexity, which actually used only a few. The reduction and the combination of a number of tag types into a much smaller number of opcodes also is to shrink the size of the browser. Just as importantly, it enables the browser to form more of the presentation duties by taking control from the forms themselves. This is critical for future extensibility. Also, the forms may be presented under quite unusual conditions such as via scripts or 2-line by 16-character server front panels.

Form Hierarchy

Forms are made up of pages. Pages are then made up of instructions.

Instructions are categorized into several types:

- Textual: Titles, Subtitles, and Text. These instructions allow text output.
- Reference: Hypertext jumps are allowed from pages to the start of any other page.
- Questions: These are the instructions that actually request data and describe how to store it. All questions support definitions of default state and bounds checking. All questions support prompt and context help string tokens. All questions support call-backs. The following question types are supported:
  - Check Box: Analogous to a Boolean, this instruction requests a single binary value: 1 if checked, zero if not.
  - One-of: Analogous to an enumerated type, this requests a single value out of a collection of items known as "options". One-of is the only multi-opcode question: One for the one of and one for each option.
  - Numeric: Analogous to an integer type, this requests a numeric field (limited to 16 bits). The instruction contains the range and step and default values.
  - Date, Time: These are special-cased because they change while being displayed and are common.
  - Passwords: A primitive password scheme is handled by default. Call-backs are required for complex encoding schemes.
- Boolean Expressions: Post-fix ("Reverse Polish") expressions allow the results of questions to be compared against each other and against constants. Magnitude comparisons (such as less than) and Boolean operations (AND, OR, NOT) are supported.
- Conditionals: Three types of instructions consume Boolean expressions. One, gray out, starts a block (terminated with an End opcode) which tells the browser to gray-out the intervening text including questions. Another, Suppress, starts an End terminated

block which tells the browser to hide all of the encompassed text including questions if the Boolean expression is true. The third allows the form to do complex inter-question validation, known as consistency checking and provide a diagnostic message if the comparison fails.

Example

Consider the classic setup sequence: legacy Serial port configuration.

```
...
OneOf SerialPort1, SerialPort1Store, SerialPort1Prompt
   SerialPort1Help
Option SerialPort1Off, SerialPort1OffText, 0
Option SerialPort13F8, SerialPort3F8Text, 1
Option SerialPort12F8, SerialPort2F8Text, 0

OneOf SerialPort2, SerialPort2Store, SerialPort2Prompt
   SerialPort2Help
Option SerialPort2Off, SerialPort1OffText, 0
Option SerialPort13F8, SerialPort3F8Text, 0
Option SerialPort12F8, SerialPort2F8Text, 1

InconsistentIf SerialPortConflict,
   SerialPort1 == SerialPort2 and SerialPort1 != 0
```

This might be displayed as:

Serial Port 1 Address
○ Off
⊙ At 3F8
○ At 2F8
Serial Port 2 Address
○ Off
○ 3F8
⊙ 2F8

The pidgin-VFR defines two simple enumerated entries, which here are represented as radio buttons. Each OneOf is followed immediately by its options. The only other valid instructions in the OneOf / Options sequence are Gray Out and Suppress.

The Option line, for example, defines the equated value corresponding to the option, the text (the "At 3F8"), and flags, including 1 for default value.

The InconsistentIf line defines the warning string. The expression is to be compiled into RPN.

The HII Database Interface

There is only (supposed to be) one instance of the HII Database Protocol. It is typically supported by a driver (or possibly a series of supporting drivers). These drivers should be written so as to only require memory. The protocol functions are designed to facilitate this effort.

The basic functions allow for submission and extraction of HII data. When submitted, the packages are assigned handles by which they are referred for the rest of their time in the database. These handles define the extent of scope for such things as string tokens, so a string token is unique to a (handle, string) pair.

The database has been designed to avoid call-backs to the extent possible. For example, the protocol makes it possible to edit strings after the package has been submitted to the database. To see how this might be used, consider a driver that is responsible for configuring a single SCSI host controller. The SCSI driver knows much of its configuration during build but cannot know, for example, the types of devices attached to it. The SCSI driver can submit a common package and then, as it discovers the devices attached to its host controller, update the relevant strings to describe the attached peripherals.

Strings and Scripting

Given that forms reference strings and strings can be localized to different languages, it is not hard to make the intellectual leap to localize them to non-human languages such as languages built for scripting. This requires some support on the consumer side to translate the HII database into the local syntax. It is probable that two languages would have to be supported: one per “standard” scripting language supported and one local to the producing organization for its own manufacturing and test automation.

The alternative is to build support in each firmware to have separate mechanisms to support each scripting configuration language and separate descriptions of the configurable data, which increases size and increases the likelihood of misalignment between the various drivers trying to present the same information differently.

Driver Design Notes

HII Strings and Forms are local to a single driver. This does not, however, mean that each driver must provide a separate HII package. For example, it is the expected that a single driver (or perhaps a few) provide the HII data for a motherboard. These drivers are expected to be motherboard-specific and are expected, as one of their duties, to disburse configuration data to the other drivers.

There are several reasons compelling this sort of design. In particular, having the platform provide the user interface allows the motherboard designer to control which questions the user actual gets to answer and which ones are kept constants or are otherwise derived. A designer of a peripheral driver must provide a superset of the questions generally required in order to keep the driver generic. The platform drivers can then edit these as they see fit.

As well as abstracting the questions, this methodology allows the platform designer to maintain a consistency in text style and consistency. It also conserves on font space.

The Data Hub

Boot firmware derives a large amount of data as an incidental part of doing its job. Various OS-based consumers are interested in some of this data. The data hub is the mechanism by which the drivers that create the data are weakly bound to those who create the interfaces to provide to the consumers. The data hub resides in DXE and is recreated at every boot, although some of the data provided to it may be nonvolatile. Its design is a classic example of a producer/consumer model. This interface survives until ExitBootServices.

If you are starting to think this sounds like an event log, you aren't far off. There are so many industry specifications that define what *they* mean by event log that the Framework uses a different name.

The assertion of weak binding is important. The goal of the data hub is to abstract the producers of the data from their consumers. Using this model the producer need not know what management specifications the system is required to comply with and does not have to carry every such required interface and the size impact that goes along with that. Consumers can more easily edit the data to suit their needs as well.

Message Ordering

It is quite possible that not all the producers run before all consumers. As shown in Figure 11.3, the data hub's data is stored as a journal of *data records*—the data blobs sent from the producers to the data hub itself. These are required to be kept such that the receipt order (from oldest to newest) of the records is retained and are, in fact, time and monotonically stamped.

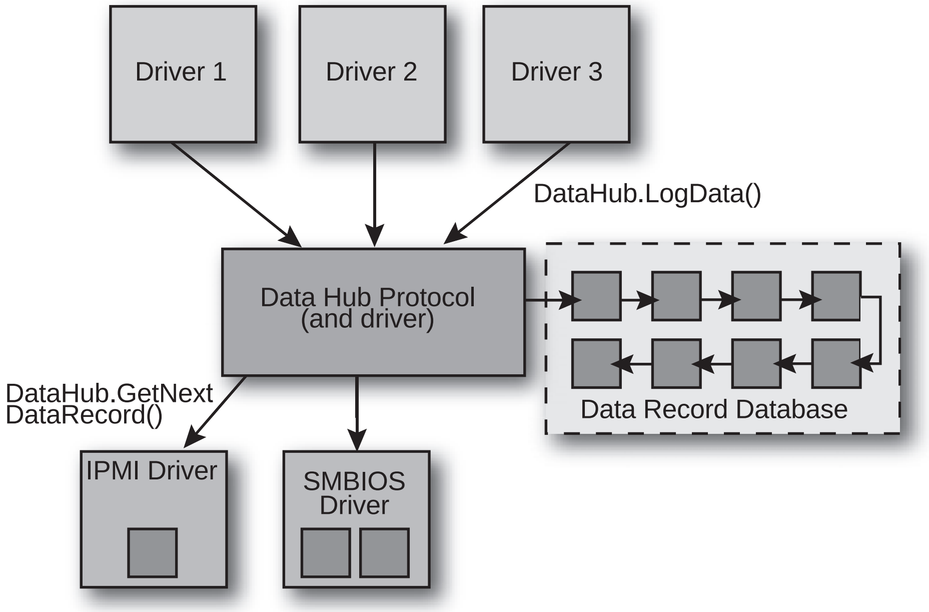


Figure 11.3 Data Hub

Records are provided to and retrieved from the data hub via the Data Hub protocol. Consumers may define themselves as "filter drivers" to receive notification (and hence control) when new records are presented to the data hub.

When a consumer signs up to receive data from the data hub, it is responsible first for fetching all of the records currently retained by the data hub. It then receives data records as they are generated. Since the data is by definition not intended to be timing critical, this means that the consumer cannot tell the difference between signing up before any data it is interested in is produced or after all of it has been produced or somewhere in between.

It is valid, although discouraged, for a producing driver to provide data regarding a certain item or event and later update that data as it gains more detail. As such, the data hub does not edit older messages of the same class but hands all requested messages through. Although not implemented via the data hub, this evolution of a representation of the system is analogous to the memory HOB passed from PEI to DXE. DXE drivers may update the memory usage map (in the Global Coherency Domain) as they initialize their peripherals. The picture of memory usage in the system becomes more and more real as the boot continues.

Classes for Editing

A major issue for the producers of the data hub's data is to filter the data into manageable chunks as required by industry standards and internal use. Different combinations of the same data may be consumed by code supporting manageability interfaces such as SMBIOS and IPMI as well as internal consumers including evaluation and manufacturing support. Each consumer would ideally see only the data it was interested in.

The data hub design addresses this goal by requiring the producer to label the data by *class*. Certain commonly used classes—such as processor, memory, and peripheral—are defined as a part of the Framework specifications. Headroom is left in the class definitions to support data that falls outside the defined classes. The consumer may sign up to receive messages by class or do all editing itself. The data provided to the consumer may need to be reformatted (as from Megahertz to Gigahertz, for example) before use based on the specification the consuming driver is complying with.

Progress Codes

One less obvious use of the data hub is as an abstraction for progress codes. These are more advanced versions of the POST Codes (or Port 80h codes, from the standard address of the card). Progress codes are provided using the same data hub mechanism as other data hub data records and, in fact, form one class.

Driving progress codes via the data hub resolves a couple of otherwise tough issues. First, if different drivers are developed by different organizations, how are progress codes (the Port 80 cards are limited to 256 values: 00 to FF) allocated among the drivers? During debug, a driver can do as many status codes as it wishes. For production drivers, however, all drivers should provide their progress codes to the data hub. A status driver can then be written which maps those into the status reports of interest to that platform. Second, the log may also be retained and provided as part of the system table, for example, by the driver for post-boot analysis.

Chapter 12

Differences between DXE Drivers and EFI Drivers

Mixing one's wines may be a mistake, but old and new wisdom mix admirably.

—Bertrolt Brecht

The Extensible Firmware Interface (EFI) can be implemented in many ways. One way is to implement via the Driver Execution Environment (DXE) Foundation portion of the Framework. As such, DXE represents a special type of driver that can be combined with EFI drivers in a given firmware volume.

There are two basic classes of DXE drivers. The first class is DXE drivers that execute very early in the DXE phase. The execution order of these DXE drivers depends on the evaluation of dependency expressions. These early DXE drivers typically contain basic services, processor initialization code, chipset initialization code, and platform initialization code. These early drivers also typically produce the architectural protocols that are required for the DXE Core to produces its full complement of EFI Boot Services and EFI Runtime Services. In order to support that fastest possible boot time, as much initialization should be deferred to the DXE drivers that follow EFI Driver Model described in the *EFI 1.10 Specification*. Most of the platform and chipset drivers belong to this category. These drivers need to be aware that not all of these services may be available when they execute and use dependency expressions to make sure the protocols and services that they need are available.

The second class of DXE drivers consists of those which follow the EFI Driver Model. These drivers do not touch any hardware resources when they initialize. Instead, they register a Driver Binding Protocol interface in the handle database. The Driver Binding Protocols are used by the BDS phase to connect the drivers to the devices required to establish consoles and provide access to boot devices. The DXE Drivers that follow the EFI Driver Model ultimately provide software abstractions for console devices and boot devices, but only when they are explicitly asked to do so. All the Bus Drivers (including PCI bus enumerator) and EFI ROMs belong to this category.

Beyond these types of drivers, the DXE drivers use two services that that an EFI driver would never use, specifically the Global Coherency Domain (GCD) and the dispatcher based on a dependency expression, or *depex*. For an EFI driver, the former is not relevant because an EFI memory map is fully-instantiated. For the latter, all of the necessary services are available and an EFI driver effectively has a non-existence depex (that is, assume EFI drivers are dispatched after any DXE driver that has a non-NULL depex). The lack of a depex on EFI drivers allows mixing of EFI drivers that may antedate depexes or are not aware of the Framework implementation with DXE drivers in a given firmware container.

The use of the Global Coherency Domain (GCD) and the depex-based dispatcher are of such importance to distinguish an EFI driver from a DXE driver that they will be discussed in more detail in the following sections.

Global Coherency Domain Services

Global Coherency Domain (GCD) Services Overview

The Global Coherency Domain (GCD) Services are used to manage the memory and I/O resources visible to the boot processor. These resources are managed in two different maps:

- GCD memory space map
- GCD I/O space map

If memory or I/O resources are added, removed, allocated, or freed, then the GCD memory space map and GCD I/O space map are updated. GCD

Services are also provided to retrieve the contents of these two resource maps.

The GCD Services can be broken up into two groups. The first manages the memory resources visible to the boot processor, and the second manages the I/O resources visible to the boot processor. Not all processor types support I/O resources, so the management of I/O resources may not be required. However, since system memory resources and memory-mapped I/O resources are required to execute the DXE environment, the management of memory resources is always required.

GCD Memory Resources

The Global Coherency Domain (GCD) Services used to manage memory resources include the following:

- `AddMemorySpace()`
- `AllocateMemorySpace()`
- `FreeMemorySpace()`
- `RemoveMemorySpace()`
- `SetMemorySpaceAttributes()`

The GCD Services used to retrieve the GCD memory space map include the following:

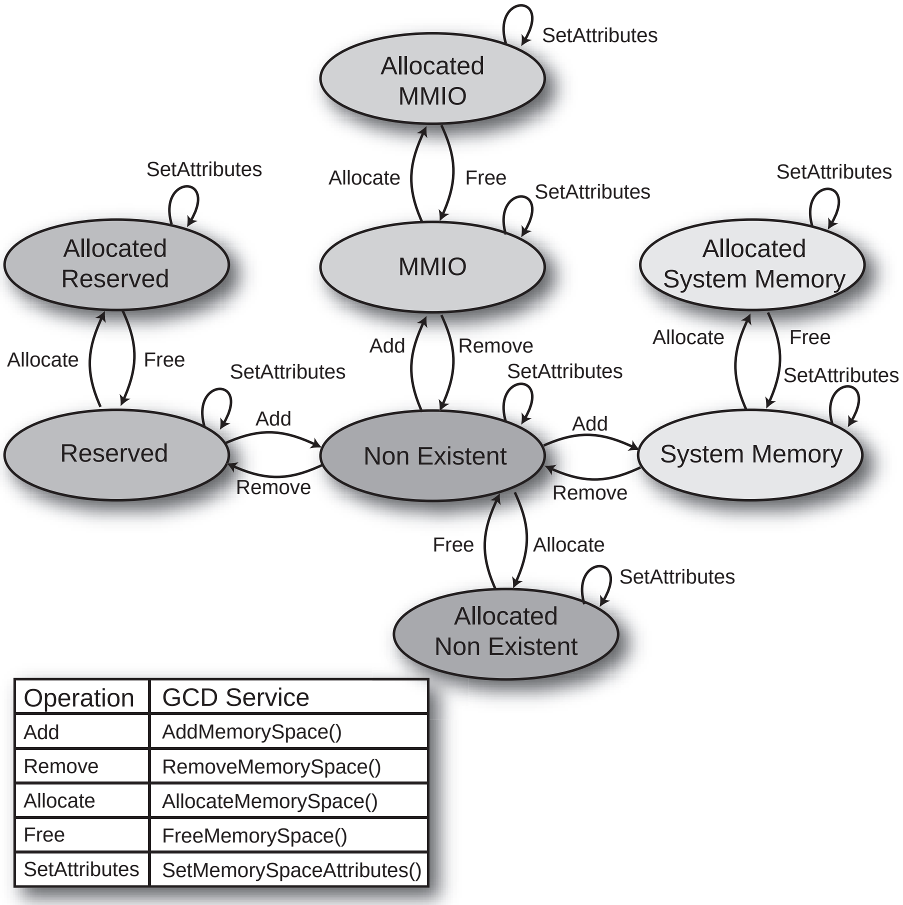
- `GetMemorySpaceDescriptor()`
- `GetMemorySpaceMap()`

The GCD memory space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access memory resources. This information is used to initialize the state of the GCD memory space map. Any memory regions outside this initial region are unavailable to any of the GCD Services used to manage memory resources. The GCD memory space map is designed to describe the memory address space with as many as 64 address lines. Each region in the GCD memory space map can begin and end on a byte boundary. Additional HOB types describe the location of system memory, the location memory mapped I/O, the location of firmware devices, the location of firmware volumes, the location of reserved regions, and the location of system memory regions that were allocated prior to the execution of the DXE Foundation. The DXE Foundation must parse the contents of the HOB list to guarantee that memory regions reserved prior to the execution of the DXE Foundation are honored. As a result, the GCD memory space map must reflect the memory regions described in the HOB list. The GCD memory space map provides the DXE Foundation with the information required to initialize the memory services such as `AllocatePages()`, `FreePages()`, `AllocatePool()`, `FreePool()`, and `GetMemoryMap()`. See the *EFI 1.10 Specification* for definitions of these services.

A memory region described by the GCD memory space map can be in one of several different states:

- Nonexistent memory
- System memory
- Memory-mapped I/O
- Reserved memory

These memory regions can be allocated and freed by DXE drivers executing in the DXE environment. In addition, a DXE driver can attempt to adjust the caching attributes of a memory region. Figure 21.1 shows the possible state transitions for each byte of memory in the GCD memory space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD services are required to merge similar memory regions that are adjacent to each other into a single memory descriptor, which reduces the number of entries in the GCD memory space map.



Operation	GCD Service
Add	AddMemorySpace()
Remove	RemoveMemorySpace()
Allocate	AllocateMemorySpace()
Free	FreeMemorySpace()
SetAttributes	SetMemorySpaceAttributes()

Figure 12.1 GCD Memory State Transitions

GCD I/O Resources

The Global Coherency Domain (GCD) Services used to manage I/O resources include the following:

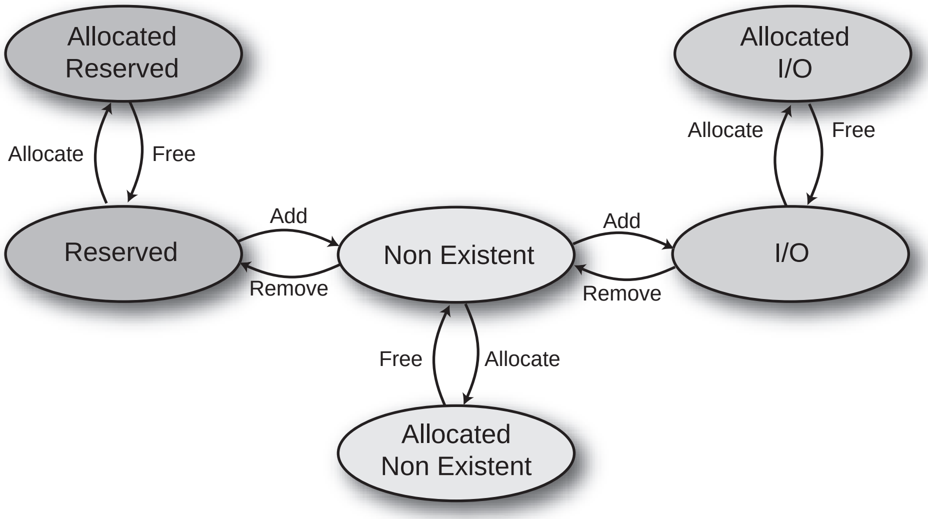
- `AddIoSpace()`
- `AllocateIoSpace()`
- `FreeIoSpace()`
- `RemoveIoSpace()`

The GCD Services used to retrieve the GCD I/O space map include the following:

- `GetIoSpaceDescriptor()`
- `GetIoSpaceMap()`

The GCD I/O space map is initialized from the HOB list that is passed to the entry point of the DXE Foundation. One HOB type describes the number of address lines that are used to access I/O resources. This information is used to initialize the state of the GCD I/O space map. Any I/O regions outside this initial region are not available to any of the GCD Services that are used to manage I/O resources. The GCD I/O space map is designed to describe the I/O address space with as many as 64 address lines. Each region in the GCD I/O space map can being and end on a byte boundary.

An I/O region described by the GCD I/O space map can be in several different states. These include nonexistent I/O, I/O, and reserved I/O. These I/O regions can be allocated and freed by DXE drivers executing in the DXE environment. Figure 12.2 shows the possible state transitions for each byte of I/O in the GCD I/O space map. The transitions are labeled with the GCD Service that can move the byte from one state to another. The GCD Services are required to merge similar I/O regions that are adjacent to each other into a single I/O descriptor, which reduces the number of entries in the GCD I/O space map.



Operation	GCD Service
Add	AddIoSpace()
Remove	RemoveIoSpace()
Allocate	AllocateIoSpace()
Free	FreeIoSpace()

Figure 12.2 GCD I/O State Transitions

The functions that make up Global Coherency Domain (GCD) Services are used during preboot to add, remove, allocate, free, and provide maps of the system memory, memory-mapped I/O, and I/O resources in a platform. These services, used in conjunction with the Memory Allocation Services, provide the ability to manage all the memory and I/O resources in a platform. Table 12.1 lists the Global Coherency Domain Services.

Table 12.1 Global Coherency Domain Services

Name	Type	Description
AddMemorySpace	Boot	Adds reserved memory, system memory, or memory-mapped I/O resources to the global coherency domain of the processor.
AllocateMemorySpace	Boot	Allocates nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
FreeMemorySpace	Boot	Frees nonexistent memory, reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
RemoveMemorySpace	Boot	Removes reserved memory, system memory, or memory-mapped I/O resources from the global coherency domain of the processor.
GetMemorySpaceDescriptor	Boot	Retrieves the descriptor for a memory region containing a specified address.
SetMemorySpaceAttributes	Boot	Modifies the attributes for a memory region in the global coherency domain of the processor.
GetMemorySpaceMap	Boot	Returns a map of the memory resources in the global coherency domain of the processor.
AddIoSpace	Boot	Adds reserved I/O, or I/O resources to the global coherency domain of the processor.
AllocateIoSpace	Boot	Allocates nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
FreeIoSpace	Boot	Frees nonexistent I/O, reserved I/O, or I/O resources from the global coherency domain of the processor.
RemoveIoSpace	Boot	Removes reserved I/O, or I/O resources from the global coherency domain of the processor.
GetIoSpaceDescriptor	Boot	Retrieves the descriptor for an I/O region containing a specified address.
GetIoSpaceMap	Boot	Returns a map of the I/O resources in the global coherency domain of the processor.

Dispatcher Services

The functions that make up the Dispatcher Services are used during preboot to schedule drivers for execution. A driver may optionally have the Schedule On Request (SOR) flag set in the driver's dependency expression. Drivers with this bit set will not be loaded and invoked until explicitly requested. Files loaded from firmware volumes may be placed in the untrusted state by the Security Architectural Protocol. The services in this section provide this ability to clear the SOR flag in a DXE driver's dependency expression and the ability to promote a file from a firmware volume from the untrusted to the trusted state. Table 12.2 lists the Dispatcher Services.

Table 12.2 Dispatcher Services

Name	Type	Description
Dispatch	Boot	Loads and executed DXE drivers from firmware volumes.
Schedule	Boot	Clears the Schedule on Request (SOR) flag for a component that is stored in a firmware volume.
Trust	Boot	Changes the state of a file stored in a firmware volume from the untrusted state to the trusted state.
ProcessFirmwareVolume	Boot	Creates a firmware volume handle for a firmware volume that is present in system memory.

Dependency Expression Reverse Polish Notation (RPN)

The actual equations will be presented by the DXE driver in a simple-to-evaluate form, namely postfix.

The following is a BNF encoding of this grammar. See Dependency Expression Instruction Set for definitions of the dependency expressions.

```
<statement> ::= SOR <expression> END |
                BEFORE <guid> END |
                AFTER <guid> END |
                <expression> END

<expression> ::= PUSH <guid> |
                 TRUE |
                 FALSE |
                 <expression> NOT |
                 <expression> <expression> OR |
                 <expression> <expression> AND
```

DXE Dispatcher State Machine

The DXE Dispatcher is responsible for tracking the state of a DXE driver from the time that the DXE driver is discovered in a firmware volume until the DXE Foundation is terminated with a call to `ExitBootServices()`. During this time, each DXE driver may be in one of several different states. The state machine that the DXE Dispatcher must use to track a DXE driver is shown in Figure 12.3.

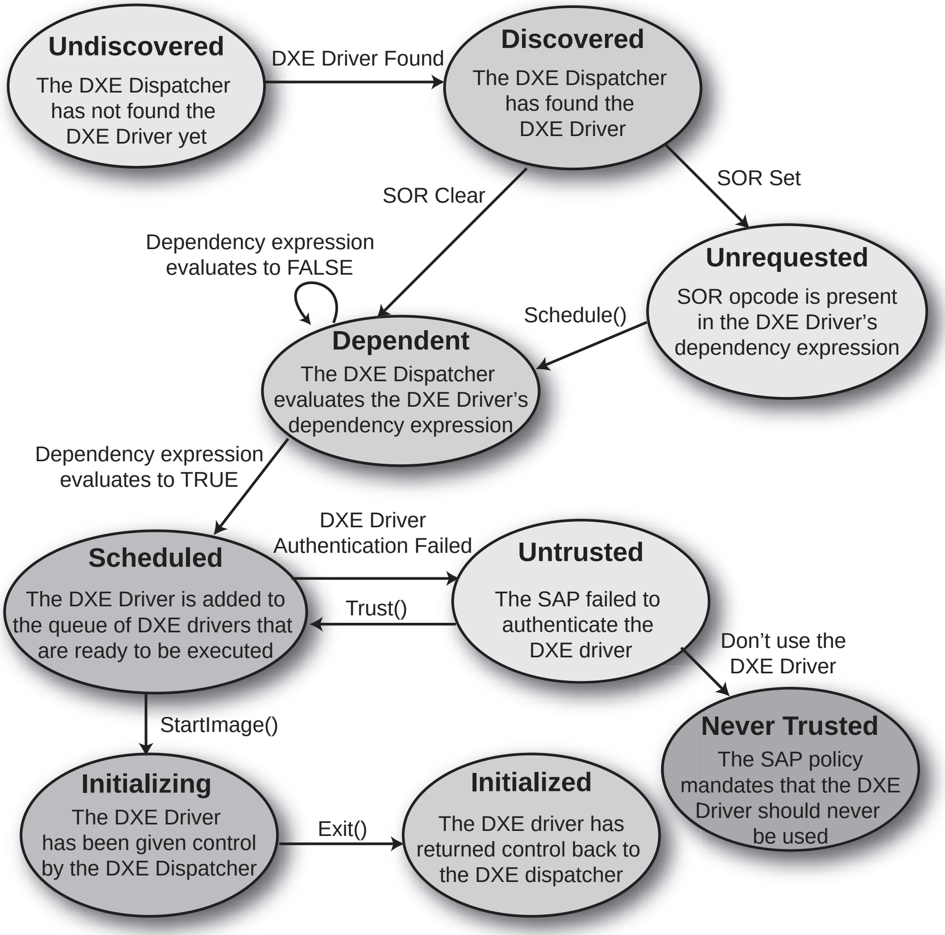

Undiscovered
The DXE Dispatcher has not found the DXE Driver yet
DXE Driver Found
Discovered
The DXE Dispatcher has found the DXE Driver
SOR Clear
SOR Set
Dependency expression evaluates to FALSE
Unrequested
SOR opcode is present in the DXE Driver's dependency expression
Schedule()
Dependent
The DXE Dispatcher evaluates the DXE Driver's dependency expression
Dependency expression evaluates to TRUE
DXE Driver Authentication Failed
Scheduled
The DXE Driver is added to the queue of DXE drivers that are ready to be executed
Untrusted
The SAP failed to authenticate the DXE driver
Trust()
Don't use the DXE Driver
StartImage()
Never Trusted
The SAP policy mandates that the DXE Driver should never be used
Initializing
The DXE Driver has been given control by the DXE Dispatcher
Exit()
Initialized
The DXE driver has returned control back to the DXE dispatcher

Figure 12.3 DXE Driver States

A DXE driver starts in the Undiscovered state, which means that the DXE driver is in a firmware volume that the DXE Dispatcher does not know about yet. When the DXE Dispatcher discovers a new firmware volume, any DXE drivers from that firmware volume listed in the *a priori* file are immediately loaded and executed. DXE drivers listed in the *a priori* file are immediately promoted to the Scheduled state. The DXE Dispatcher then searches the firmware volume for DXE drivers that are not listed in the *a priori* file. Any DXE drivers found are promoted from the Undiscovered to the Discovered state. The dependency expression for each DXE driver is evaluated. If the SOR opcode is present in a DXE driver's dependency expression, then the DXE driver is placed in the Unrequested state. If the SOR opcode is not present in the DXE driver's dependency expression, then the DXE driver is placed in the Dependent state. Once a DXE driver is in the Unrequested state, it may only be promoted to the Dependent state with a call to the DXE Service `Schedule()`.

Once a DXE Driver is in the Dependent state, the DXE Dispatcher evaluates the DXE driver's dependency expression. If the DXE driver does not have a dependency expression, then a dependency expression of all the architectural protocols ANDed together is assumed for that DXE driver. If the dependency expression evaluates to `FALSE`, then the DXE driver stays in the Dependent state. If the dependency expression never evaluates to `TRUE`, then it will never leave the Dependent state. If the dependency expression evaluates to `TRUE`, then the DXE driver is promoted to the Scheduled state.

A DXE driver that is promoted to the Scheduled state is added to the end of the queue of other DXE drivers that have been promoted to the Scheduled state. When the DXE driver has reached the head of the queue, the DXE Dispatcher must use the services of the Security Authentication Protocol (SAP) to check the authentication status of the DXE Driver. If the Security Authentication Protocol deems that the DXE Driver violates the security policy of the platform, then the DXE Driver is placed in the Untrusted state. The Security Authentication Protocol can also tell the DXE Dispatcher that the DXE driver should never be executed and be placed in the Never Trusted state. If a DXE driver is placed in the Untrusted state, it can only be promoted back to the Scheduled state with a call to the DXE Service `Trust()`.

Once a DXE driver has reached the head of the scheduled queue, and the DXE driver has passed the authentication checks of the Security Authentication Protocol, the DXE driver is loaded into memory with the Boot Service `LoadImage()`. Control is then passed from the DXE Dispatcher to the DXE driver with the Boot Service `StartImage()`. When `StartImage()` is called for a DXE driver, that DXE driver is promoted to the Initializing state. The DXE driver returns control to the DXE Dispatcher through the Boot Service `Exit()`. When a DXE driver has returned control to the DXE Dispatcher, the DXE driver is in the terminal state called Initialized.

The DXE Dispatcher is responsible for draining the queue of DXE drivers in the Scheduled state until the queue is empty. Once the queue is empty, then DXE Dispatcher must evaluate all the DXE drivers in the Dependent state to see if any of them need to be promoted to the Scheduled state. These evaluations need to be performed every time one or more DXE drivers have been promoted to the Initialized state, because those DXE drivers may have produced protocol interfaces for which the DXE drivers in the Dependent state are waiting.

Example Orderings

The order that DXE drivers are loaded and executed by the DXE Dispatcher is a mix of strong and weak orderings. The strong orderings are specified through *a priori* files, and the weak orderings are specified by dependency expressions in DXE drivers. Figure 12.4 shows the contents of a sample firmware volume that contains the following:

- DXE Foundation image
- DXE driver images
- An *a priori* file

The order that these images appear in the firmware volume is arbitrary. The DXE Foundation and the DXE Dispatcher must not make any assumptions about the locations of files in firmware volumes. The *a priori* file contains the GUID file names of the DXE drivers that are to be loaded and executed first. The dependency expressions and the protocols that each DXE driver produces are shown next to each DXE driver image in the firmware volume.

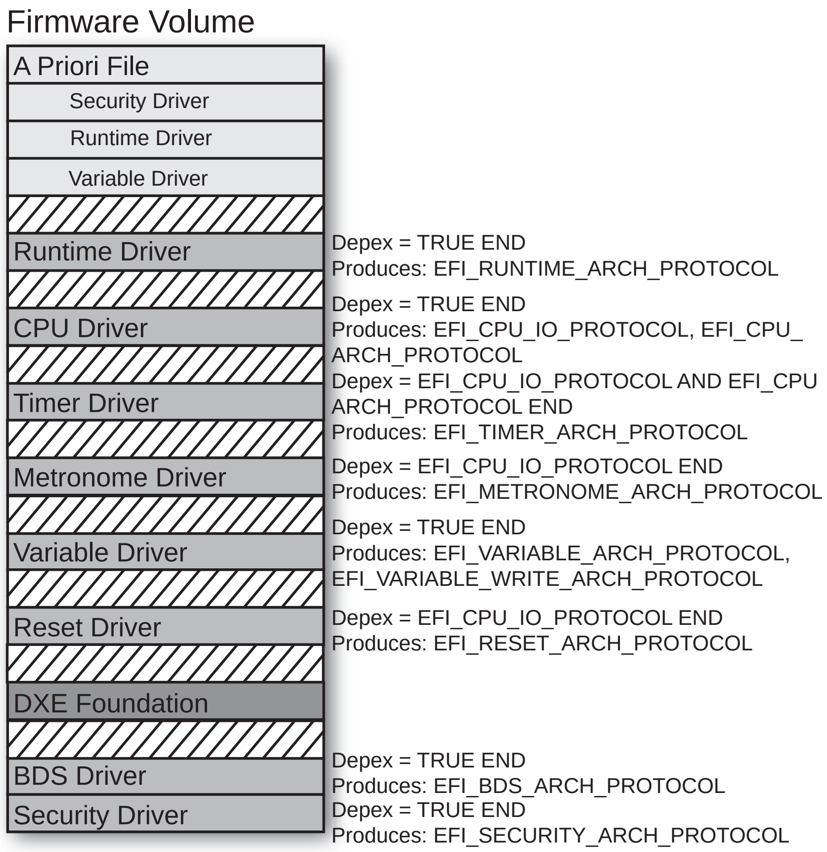

Firmware Volume
A Priori File
Security Driver
Runtime Driver
Variable Driver
Runtime Driver
Depex = TRUE END
Produces: EFI_RUNTIME_ARCH_PROTOCOL
CPU Driver
Depex = TRUE END
Produces: EFI_CPU_IO_PROTOCOL, EFI_CPU_ARCH_PROTOCOL
Timer Driver
Depex = EFI_CPU_IO_PROTOCOL AND EFI_CPU ARCH_PROTOCOL END
Produces: EFI_TIMER_ARCH_PROTOCOL
Metronome Driver
Depex = EFI_CPU_IO_PROTOCOL END
Produces: EFI_METRONOME_ARCH_PROTOCOL
Variable Driver
Depex = TRUE END
Produces: EFI_VARIABLE_ARCH_PROTOCOL, EFI_VARIABLE_WRITE_ARCH_PROTOCOL
Reset Driver
Depex = EFI_CPU_IO_PROTOCOL END
Produces: EFI_RESET_ARCH_PROTOCOL
DXE Foundation
BDS Driver
Depex = TRUE END
Produces: EFI_BDS_ARCH_PROTOCOL
Security Driver
Depex = TRUE END
Produces: EFI_SECURITY_ARCH_PROTOCOL

Figure 12.4 Sample Firmware Volume

Based on the contents of the firmware volume in Figure 12.4, the Security Driver, Runtime Driver, and Variable Driver will always be executed first. This is an example of a strongly ordered dispatch due to the *a priori* file. The DXE Dispatcher will then evaluate the dependency expressions of the remaining DXE drivers to determine the order that they will be executed. Based on the dependency expressions and the protocols that each DXE driver produces there are 30 valid orderings from which the DXE Dispatcher may choose. The BDS Driver and CPU Driver tie for the next drivers to be scheduled, because their dependency expressions are simply TRUE. A dependency expression of TRUE means that the DXE driver does not require any other protocol interfaces to be executed. The DXE Dispatcher may choose either one of these drivers to be scheduled first. The Timer Driver, Metronome Driver, and Reset Driver all depend on the protocols produced by the CPU Driver. Once the CPU Driver has been loaded and executed, the Timer Driver, Metronome Driver, and Reset Driver may be scheduled in any order. A reasonable implementation of a DXE Dispatcher would consistently produce the same ordering for a given system configuration. If the configuration of the system is changed in any way (including an order of files stored in a firmware volume), then a different dispatch ordering may be generated, but this new ordering should be consistent until the next system configuration change.

Chapter 13

Boot Device Selection

I just invent, then wait until man comes around to needing what I invented.

—R. Buckminster Fuller

EFI has over time evolved a very basic paradigm for establishing a firmware policy engine. The concept was developed prior to the inception of the framework (EFI 1.1 or earlier) of a single boot manager whose sole purpose was exercising the policy established by some architecturally defined global NVRAM variables. As the framework evolved, and several distinct boot phases such as SEC, PEI, DXE, BDS, Runtime, and Afterlife were defined, the BDS (Boot Device Selection) phase became a distinct Boot Manager-like phase. In this chapter, the architectural components that steer the policy of the Boot Manager are reviewed. This content forms the architectural basis for what eventually became the BDS phase. Selection

In fact, the differences between what is known as the boot manager in pre-Framework solutions and what is known as the BDS in Framework solutions is easy to illustrate. Figure 13.1 shows the software flow in an EFI 1.1 compatible (pre-Framework) environment, and Figure 13.2 shows one that is Framework compatible.

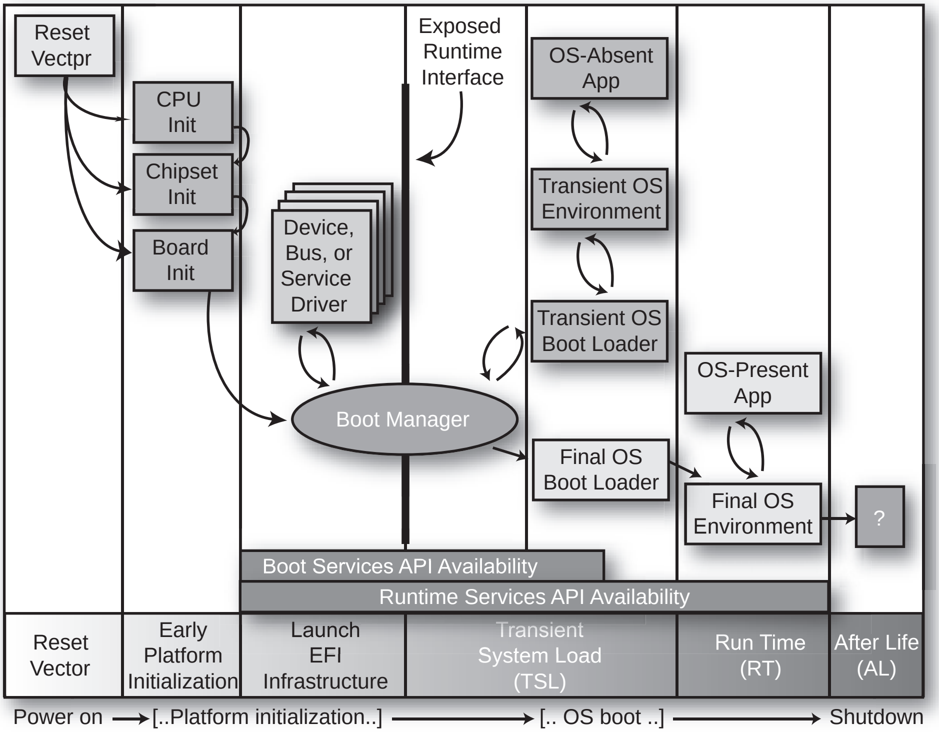


Figure 13.1 EFI with Boot Manager Component

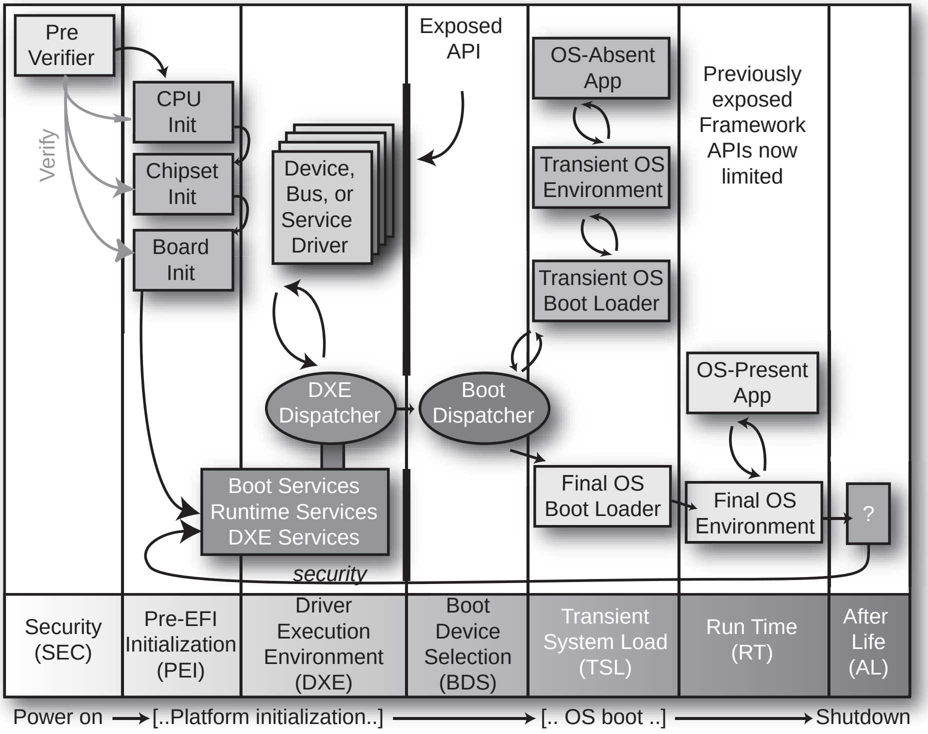

Pre Verifier
Verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
DXE Dispatcher
Boot Services Runtime Services DXE Services
security
Exposed API
Boot Dispatcher
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
Previously exposed Framework APIs now limited
OS-Present App
Final OS Environment
?
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on → [..Platform initialization..] → [.. OS boot ..] → Shutdown

Figure 13.2 Framework with BDS Component

As you can see from comparing the two figures, there is much overlap. The BDS phase subsumes the direction described in this chapter and is further explained in Chapter 9.

The EFI boot manager is a firmware policy engine that can be configured by modifying architecturally defined global NVRAM variables. The boot manager attempts to load EFI drivers and EFI applications (including EFI OS boot loaders) in an order defined by the global NVRAM variables. The platform firmware must use the boot order specified in the global NVRAM variables for normal boot. The platform firmware may add extra boot options or remove invalid boot options from the boot order list.

The platform firmware may also implement value-added features in the boot manager if an exceptional condition is discovered in the firmware boot process. One example of a value-added feature would be not loading an EFI driver if booting failed the first time the driver was loaded.

Another example would be booting to an OEM-defined diagnostic environment if a critical error was discovered during the boot process.

The boot sequence for EFI consists of the following:

- The platform firmware reads the boot order list from a globally defined NVRAM variable. The boot order list defines a list of NVRAM variables that contain information about what is to be booted. Each NVRAM variable defines a Unicode name for the boot option that can be displayed to a user.
- The variable also contains a pointer to the hardware device and to a file on that hardware device that contains the EFI image to be loaded.
- The variable might also contain paths to the OS partition and directory along with other configuration-specific directories.

The NVRAM can also contain load options that are passed directly to the EFI image. The platform firmware has no knowledge of what is contained in the load options. The load options are set by higher level software when it writes to a global NVRAM variable to set the platform firmware boot policy. This information could be used to define the location of the OS kernel if it was different than the location of the EFI OS loader.

Firmware Boot Manager

The boot manager is a component in the EFI firmware that determines which EFI drivers and EFI applications should be explicitly loaded and when. Once the EFI firmware is initialized, it passes control to the boot manager. The boot manager is then responsible for determining what to load and any interactions with the user that may be required to make such a decision. Much of the behavior of the boot manager is left up to the firmware developer to decide, and details of boot manager implementation are outside the scope of this specification. In particular, likely implementation options might include any console interface concerning boot, integrated platform management of boot selections, possible knowledge of other internal applications or recovery drivers that may be integrated into the system through the boot manager.

Programmatic interaction with the boot manager is accomplished through globally defined variables. On initialization the boot manager reads the values which comprise all of the published load options among the EFI environment variables. By using the SetVariable() function the data that contain these environment variables can be modified.

Each load option entry resides in a Boot#### variable or a Driver#### variable where the #### is replaced by a unique option number in printable hexadecimal representation using the digits 0–9, and the uppercase versions of the characters A–F (0000–FFFF). The #### must always be four digits, so small numbers must use leading zeros. The load options are then logically ordered by an array of option numbers listed in the desired order. There are two such option ordering lists. The first is DriverOrder that orders the Driver#### load option variables into their load order. The second is BootOrder that orders the Boot#### load options variables into their load order.

For example, to add a new boot option, a new Boot#### variable would be added. Then the option number of the new Boot#### variable would be added to the BootOrder ordered list and the BootOrder variable would be rewritten. To change boot option on an existing Boot####, only the Boot#### variable would need to be rewritten. A similar operation would be done to add, remove, or modify the driver load list.

If the boot via Boot#### returns with a status of EFI_SUCCESS the boot manager stops processing the BootOrder variable and presents a boot manager menu to the user. If a boot via Boot#### returns a status other than EFI_SUCCESS, the boot has failed and the next Boot#### in the BootOrder variable will be tried until all possibilities are exhausted.

The boot manager may perform automatic maintenance of the database variables. For example, it may remove unreferenced load option variables, any unparseable or unloadable load option variables, and rewrite any ordered list to remove any load options that do not have corresponding load option variables. In addition, the boot manager may automatically update any ordered list to place any of its own load options where it desires. The boot manager can also, based on its platform-specific behavior, provide for manual maintenance operations as well. Examples include choosing the order of any or all load options, activating or deactivating load options, and so on.

The boot manager is required to process the Driver load option entries before the Boot load option entries. The boot manager is also required to initiate a boot of the boot option specified by the BootNext variable as the first boot option on the next boot, and only on the next boot. The boot manager removes the BootNext variable before transferring control to the BootNext boot option. If the boot from the BootNext boot option fails the boot sequence continues utilizing the BootOrder variable. If the boot from the BootNext boot option succeeds by returning EFI_SUCCESS the boot manager will not continue to boot utilizing the BootOrder variable.

The boot manager must call LoadImage(), which supports at least SIMPLE_FILE_PROTOCOL and LOAD_FILE_PROTOCOL for resolving load options. If LoadImage() succeeds, the boot manager must enable the watchdog timer for 5 minutes by using the SetWatchdogTimer()boot service prior to calling StartImage(). If a boot option returns control to the boot manager, the boot manager must disable the watchdog timer with an additional call to the SetWatchdogTimer() boot service.

If the boot image is not loaded via LoadImage(), the boot manager is required to check for a default application to boot. Searching for a default application to boot happens on both removable and fixed media types. This search occurs when the device path of the boot image listed in any boot option points directly to a SIMPLE_FILE_SYSTEM device and does not specify the exact file to load. The file discovery method is explained in the section "Default Behavior for Boot Option Variables" later in this chapter. The default media boot case of a protocol other than SIMPLE_FILE_SYSTEM is handled by the LOAD_FILE_PROTOCOL for the target device path and does not need to be handled by the boot manager.

The boot manager must also support booting from a short-form device path that starts with the first element being a hard drive media device path. The boot manager must use the GUID or signature and partition number in the hard drive device path to match it to a device in the system. If the drive supports the GPT partitioning scheme the GUID in the hard drive media device path is compared with the UniquePartitionGuid field of the GUID Partition Entry. If the drive supports the PC-AT MBR scheme the signature in the hard drive media device path is compared with the UniqueMBRSignature in the Legacy Master Boot Record. If a signature match is made, then the partition number must also be matched. The hard drive device path can be appended to the matching hardware device path and normal boot behavior can then be used. If more than one device matches the hard drive device path, the boot manager will pick one arbitrarily. Thus the operating system must ensure the uniqueness of the signatures on hard drives to guarantee deterministic boot behavior.

Each load option variable contains an EFI_LOAD_OPTION descriptor that is a byte-packed buffer of variable-length fields. Since some of the fields are of variable length, an EFI_LOAD_OPTION cannot be described as a standard C data structure. Instead, the fields are listed here in the order that they appear in an EFI_LOAD_OPTION descriptor:

```
UINT32              Attributes;
UINT16              FilePathListLength;
CHAR16              Description[];
```

```
EFI_DEVICE_PATH     FilePathList[];
UINT8               OptionalData[];
```

- *Attributes* - The attributes for this load option entry. All unused bits must be zero and are reserved by the EFI specification for future growth. See "Related Definitions."
- *FilePathListLength* - Length in bytes of the FilePathList. OptionalData starts at offset sizeof(UINT32) + sizeof(UINT16) + StrSize(Description) + FilePathListLength of the EFI_LOAD_OPTION descriptor.
- *Description* - The user readable description for the load option. This field ends with a Null Unicode character.
- *FilePathList* - A packed array of EFI device paths. The first element of the array is an EFI device path that describes the device and location of the Image for this load option. The FilePathList[0] is specific to the device type. Other device paths may optionally exist in the FilePathList, but their usage is OSV specific. Each element in the array is variable length, and ends at the device path end structure. Because the size of Description is arbitrary, this data structure is not guaranteed to be aligned on a natural boundary. This data structure may have to be copied to an aligned natural boundary before it is used.
- *OptionalData* - The remaining bytes in the load option descriptor are a binary data buffer that is passed to the loaded image. If the field is zero bytes long, a Null pointer is passed to the loaded image. The number of bytes in OptionalData can be computed by subtracting the starting offset of OptionalData from total size in bytes of the EFI_LOAD_OPTION.

Related Definitions

The load option attributes are defined by the values below.

```
//
// Attributes
//
#define LOAD_OPTION_ACTIVE          0x00000001
#define LOAD_OPTION_FORCE_RECONNECT 0x00000002
```

Calling SetVariable() creates a load option. The size of the load option is the same as the size of the DataSize argument to the SetVariable() call that created the variable. When creating a new load option, all undefined attribute bits must be written as zero. When updating a load option, all undefined attribute bits must be preserved. If a load option is not marked as LOAD_OPTION_ACTIVE, the boot manager will not automatically load the option. This provides an easy way to disable or enable load options without needing to delete and re-add them. If any Driver#### load option is marked as LOAD_OPTION_FORCE_RECONNECT, then all of the EFI drivers in the system will be disconnected and reconnected after the last Driver#### load option is processed. This allows an EFI driver loaded with a Driver#### load option to override an EFI driver that was loaded prior to the execution of the EFI Boot Manager.

Globally-Defined Variables

This section defines a set of variables that have architecturally defined meanings. In addition to the defined data content, each such variable has an architecturally defined attribute that indicates when the data variable may be accessed. The variables with an attribute of NV are nonvolatile. This means that their values are persistent across resets and power cycles. The value of any environment variable that does not have this attribute will be lost when power is removed from the system and the state of firmware reserved memory is not otherwise preserved. The variables with an attribute of BS are only available before ExitBootServices() is called. This means that these environment variables can only be retrieved or modified in the preboot environment. They are not visible to an operating system. Environment variables with an attribute of RT are available before and after ExitBootServices() is called. Environment variables of this type can be retrieved and modified in the preboot environment, and from an operating system. All architecturally defined variables use the EFI_GLOBAL_VARIABLE VendorGuid:

```
#define EFI_GLOBAL_VARIABLE \
        {8BE4DF61-93CA-11d2-AA0D-00E098032B8C}
```

To prevent name collisions with possible future globally defined variables, other internal firmware data variables that are not defined here must be saved with a unique VendorGuid other than EFI_GLOBAL_VARIABLE. Table 13.1 lists the global variables.

Table 13.1 Global Variables

Variable Name	Attribute	Description
LangCodes	BS, RT	The language codes that the firmware supports.
Lang	NV, BS, RT	The language code that the system is configured for.
Timeout	NV, BS, RT	The firmware's boot managers timeout, in seconds, before initiating the default boot selection.
ConIn	NV, BS, RT	The device path of the default input console.
ConOut	NV, BS, RT	The device path of the default output console.
ErrOut	NV, BS, RT	The device path of the default error output device.
ConInDev	BS, RT	The device path of all possible console input devices.
ConOutDev	BS, RT	The device path of all possible console output devices.
ErrOutDev	BS, RT	The device path of all possible error output devices.
Boot####	NV, BS, RT	A boot load option, where #### is a printed hex value. No 0x or h is included in the hex value.
BootOrder	NV, BS, RT	The ordered boot option load list.
BootNext	NV, BS, RT	The boot option for the next boot only.
BootCurrent	BS, RT	The boot option that was selected for the current boot.
Driver####	NV, BS, RT	A driver load option, where #### is a printed hex value.
DriverOrder	NV, BS, RT	The ordered driver load option list.

The LangCodes variable contains an array of 3-character (8-bit ASCII characters) ISO-639-2 language codes that the firmware can support. At initialization time the firmware computes the supported languages and creates this data variable. Since the firmware creates this value on each initialization, its contents are not stored in nonvolatile memory. This value is considered read-only.

The Lang variable contains the 3-character (8-bit ASCII characters) ISO-639-2 language code that the machine has been configured for. This value may be changed to any value supported by LangCodes; however, the change does not take effect until the next boot. If the language code is set to an unsupported value, the firmware chooses a supported default at initialization and sets Lang to a supported value.

The Timeout variable contains a binary UINT16 that supplies the number of seconds that the firmware will wait before initiating the original default boot selection. A value of 0 indicates that the default boot selection is to be initiated immediately on boot. If the value is not present, or contains the value of 0xFFFF, then firmware will wait for user input before booting. This means the default boot selection is not automatically started by the firmware.

The ConIn, ConOut, and ErrOut variables each contain an EFI_DEVICE_PATH descriptor that defines the default device to use on boot. Changes to these values do not take effect until the next boot. If the firmware cannot resolve the device path, it is allowed to automatically replace the value(s) as needed to provide a console for the system.

The ConInDev, ConOutDev, and ErrOutDev variables each contain an EFI_DEVICE_PATH descriptor that defines all the possible default devices to use on boot. These variables are volatile, and are set dynamically on every boot. ConIn, ConOut, and ErrOut are always proper subsets of ConInDev, ConOutDev, and ErrOutDev.

Each Boot#### variable contains an EFI_LOAD_OPTION. Each Boot#### variable is the name "Boot" appended with a unique four digit hexadecimal number. For example, Boot0001, Boot0002, Boot0A02, etc.

The BootOrder variable contains an array of UINT16's that make up an ordered list of the Boot#### options. The first element in the array is the value for the first logical boot option, the second element is the value for the second logical boot option, etc. The BootOrder order list is used by the firmware's boot manager as the default boot order.

The BootNext variable is a single UINT16 that defines the Boot#### option that is to be tried first on the next boot. After the BootNext boot option is tried the normal BootOrder list is used. To prevent loops, the boot manager deletes this variable before transferring control to the preselected boot option.

The BootCurrent variable is a single UINT16 that defines the Boot#### option that was selected on the current boot.

Each Driver#### variable contains an EFI_LOAD_OPTION. Each load option variable is appended with a unique number, for example Driver0001, Driver0002, and so on.

The DriverOrder variable contains an array of UINT16s that make up an ordered list of the Driver#### variable. The first element in the array is the value for the first logical driver load option, the second element is the value for the second logical driver load option, and so on. The DriverOrder list is used by the firmware's boot manager as the default load order for EFI drivers that it should explicitly load.

Default Behavior for Boot Option Variables

The default state of globally-defined variables is firmware vendor specific. However the boot options require a standard default behavior in the exceptional case that valid boot options are not present on a platform. The default behavior must be invoked any time the BootOrder variable does not exist or only points to nonexistent boot options.

If no valid boot options exist, the boot manager will enumerate all removable EFI media devices followed by all fixed EFI media devices. The order within each group is undefined. These new default boot options are not saved to nonvolatile storage. The boot manger will then attempt to boot from each boot option. If the device supports the SIMPLE_FILE_SYSTEM protocol then the removable media boot behavior (see the section "Removable Media Boot Behavior") is executed. Otherwise the firmware will attempt to boot the device via the LOAD_FILE protocol.

It is expected that this default boot will load an operating system or a maintenance utility. If this is an operating system setup program it is then responsible for setting the requisite environment variables for subsequent boots. The platform firmware may also decide to recover or set to a known set of boot options.

Boot Mechanisms

EFI can boot from a device using the SIMPLE_FILE_SYSTEM protocol or the LOAD_FILE protocol. A device that supports the SIMPLE_FILE_SYSTEM protocol must materialize a file system protocol for that device to be bootable. If a device does not support a complete file system it may produce a LOAD_FILE protocol that allows it to materialize an image directly. The Boot Manager will attempt to boot using the SIMPLE_FILE_SYSTEM protocol first. If that fails, then the LOAD_FILE protocol will be used.

Boot via Simple File Protocol

When booting via the SIMPLE_FILE_SYSTEM protocol, the FilePath will start with a device path that points to the device that "speaks" the SIMPLE_FILE_SYSTEM protocol. The next part of the FilePath will point to the file name, including subdirectories that contain the bootable image. If the file name is a null device path, the file name must be discovered on the media using the rules defined for removable media devices with ambiguous file names (see the section "Removable Media Boot Behavior").

The format of the file system specified by EFI is contained in the EFI specification. While the firmware must produce a SIMPLE_FILE_SYSTEM protocol that understands the EFI file system, any file system can be abstracted with the SIMPLE_FILE_SYSTEM protocol interface.

Removable Media Boot Behavior

On a removable media device it is not possible for the FilePath to contain a file name, including subdirectories. The FilePath is stored in nonvolatile memory in the platform and cannot possibly be kept in sync with a media that can change at any time. A FilePath for a removable media device will point to a device that "speaks" the SIMPLE_FILE_SYSTEM protocol. The FilePath will not contain a file name or subdirectories.

The system firmware will attempt to boot from a removable media FilePath by adding a default file name in the form \EFI\BOOT\BOOT{machine type short-name}.EFI. Where machine type short-name defines a PE32+ image format architecture. Each file only contains one EFI image type, and a system may support booting from one or more images types. Table 13.2 lists the EFI image types.

Table 13.2 EFI Image Types

Architecture	File name convention	PE Executable machine type*
IA-32	BOOTIA32.EFI	0x14c
Itanium architecture	BOOTIA64.EFI	0x200

Note: The PE Executable machine type is contained in the machine field of the COFF file header as defined in the Microsoft Portable Executable and Common Object File Format Specification, Revision 6.0.

A media may support multiple architectures by simply having a \EFI\BOOT\BOOT{machine type short-name}.EFI file of each possible machine type.

Boot via LOAD_FILE Protocol

When booting via the LOAD_FILE protocol, the FilePath is a device path that points to a device that "speaks" the LOAD_FILE protocol. The image is loaded directly from the device that supports the LOAD_FILE protocol.

The remainder of the FilePath will contain information that is specific to the device. EFI firmware passes this device-specific data to the loaded image, but does not use it to load the image. If the remainder of the FilePath is a null device path it is the loaded image's responsibility to implement a policy to find the correct boot device.

The LOAD_FILE protocol is used for devices that do not directly support file systems. Network devices commonly boot in this model where the image is materialized without the need of a file system.

Network Booting

Network booting is described by the Preboot eXecution Environment (PXE) BIOS Support Specification that is part of the Wired for Management Baseline specification. PXE specifies UDP, DHCP, and TFTP network protocols that a booting platform can use to interact with an intelligent system load server. EFI defines special interfaces that are used to implement PXE. These interfaces are contained in the PXE_BASE_CODE protocol defined in the EFI specification.

Future Boot Media

Since EFI defines an abstraction between the platform and the operating system and its loader it should be possible to add new types of boot media as technology evolves. The OS loader will not necessarily have to change to support new types of boot. The implementation of the EFI platform services may change, but the interface will remain constant. The operating system will require a driver to support the new type of boot media so that it can make the transition from EFI boot services to operating system control of the boot media.

Chapter 14

Boot Flows

Two roads diverged in a wood….

—Robert Frost, "The Road Less Taken"

The restart of a system admits to many possibilities, or paths of execution. The restart of a CPU execution for a given CPU can have many causes and different environment states that impinge upon it. These can include requests to the firmware for an update of the flash store, resumption of a power management event, initial startup of the system, and other possible restarts. This chapter describes some of these possible flows and how the Framework handles the events.

To begin, the normal code flow in the Framework passes through a succession of phases, in the following order:

1. SEC
2. PEI
3. DXE
4. BDS
5. Runtime
6. Afterlife

This chapter describes alternatives to this ordering, which can also be seen in Figure 14.1.

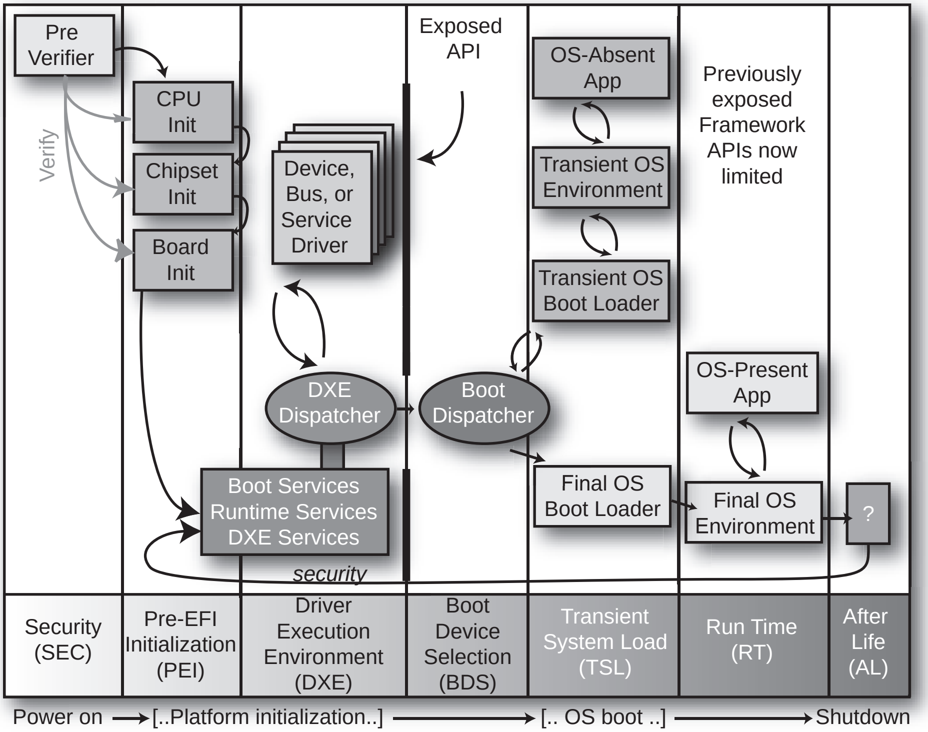

Pre Verifier
Verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
DXE Dispatcher
Boot Services Runtime Services DXE Services
security
Exposed API
Boot Dispatcher
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
Previously exposed Framework APIs now limited
OS-Present App
Final OS Environment
?
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on → [..Platform initialization..] → [.. OS boot ..] → Shutdown

Figure 14.1 Ordering of Framework Execution Phases

The PEI Foundation is unaware of the boot path required by the system. It relies on the PEIMs to determine the boot mode—R0, R1, S3, and so on—and to take appropriate action depending on the mode. To implement this determination of the boot mode, each PEIM has the ability to manipulate the boot mode using the PEI Service `SetBootMode()` described in Chapter 15. Note that the PEIM does not change the order in which PEIMs are dispatched depending on the boot mode.

Defined Boot Modes

The list of possible boot modes and their corresponding priorities is shown in the following section. Framework architecture avoids defining an upgrade path specifically, should new boot modes need be defined. This is necessary as the nature of those additional boot modes may work in conjunction with or may conflict with the previously defined boot modes.

Priority of Boot Paths

Within a given PEIM, a priority of the boot modes must be observed, as shown in Figure 14.2. The priority ordering of the sources of boot mode should be as follows (from highest priority to lowest):

1. `BOOT_IN_RECOVERY_MODE`
2. `BOOT_ON_FLASH_UPDATE`
3. `BOOT_ON_S3_RESUME`
4. `BOOT_WITH_MINIMAL_CONFIGURATION`
5. `BOOT_WITH_FULL_CONFIGURATION`
6. `BOOT_ASSUMING_NO_CONFIGURATION_CHANGES`
7. `BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS`
8. `BOOT_WITH_DEFAULT_SETTINGS`
9. `BOOT_ON_S4_RESUME`
10. `BOOT_ON_S5_RESUME`
11. `BOOT_ON_S2_RESUME`

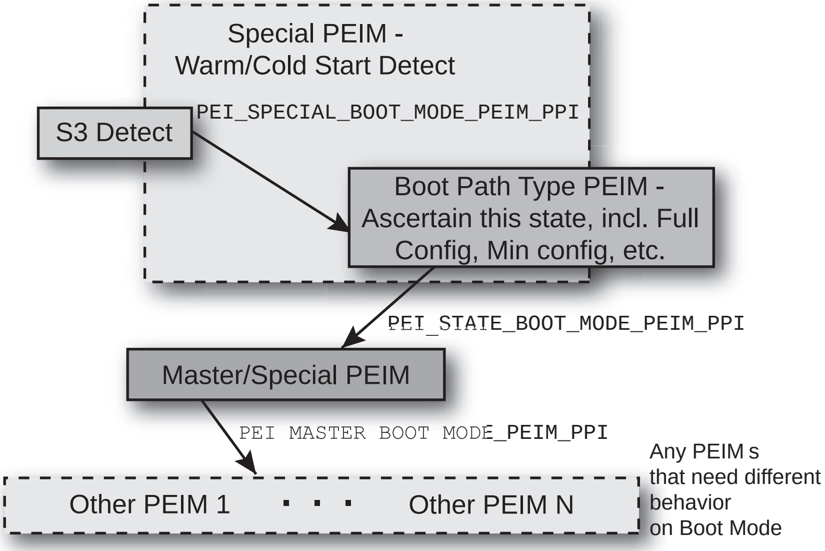


Figure 14.2 Priority of the boot modes

Table 14.1 lists the assumptions that can and cannot be made about the system for each sleep state.

Table 14.1 Boot Path Assumptions

System State	Description	Assumptions
R0	Cold Boot	Cannot assume that the previously stored configuration data is valid.
R1	Warm Boot	May assume that the previously stored configuration data is valid.

System State	Description	Assumptions
S3	ACPI Save to RAM Resume	The previously stored configuration data is valid and RAM is valid. RAM configuration must be restored from nonvolatile storage (NVS) before RAM may be used. The firmware may only modify previously reserved RAM. There are two types of reserved memory. One is the equivalent of the BIOS INT15h, E820 type-4 memory and indicates that the RAM is reserved for use by the firmware. The suggestion is to add another type of memory that allows the OS to corrupt the memory during runtime but that may be overwritten during resume.
S4, S5	Save to Disk Resume, “Soft Off”	S4 and S5 are identical from a PEIM's point of view. The two are distinguished to support follow-on phases. The entire system must be reinitialized but the PEIMs may assume that the previous configuration is still valid.
Boot on Flash Update		This boot mode can be either an INIT, S3, or other means by which to restart the machine. If it is an S3, for example, the flash update cause will supersede the S3 restart. It is incumbent upon platform code, such as the Memory Initialization PEIM, to determine the exact cause and perform correct behavior (that is, S3 state restoration versus INIT behavior).

Reset Boot Paths

The following sections describe the boot paths that are followed when a system encounters several different types of reset.

Intel® Itanium® Processor Reset

Intel Itanium architecture contains enough hooks to authenticate PAL-A and PAL-B code that is distributed by the processor vendor. The internal microcode on the processor silicon, which starts up on a PowerGood reset, finds the first layer of processor abstraction code (called PAL-A) that is located in the Boot Firmware Volume (BFV) using architecturally defined pointers in the BFV. It is the responsibility of this microcode to authenticate that the PAL-A code layer from the processor vendor has not been tampered with. If the authentication of the PAL-A layer passes, control then passes to the PAL-A layer, which then authenticates the next layer of processor abstraction code (called PAL-B) before passing control to it. In addition to this microarchitecture-specific authentication, the SEC phase of EFI is still responsible for locating the PEI Foundation and verifying its authenticity.

In an Itanium-based system, it is also imperative that the firmware modules in the BFV be organized such that at least the PAL-A is contained in the fault-tolerant regions. This processor-specific PAL-A authenticates the PAL-B code, which is usually contained in the regions of the firmware system that do not support fault-tolerant updates. The PAL-A and PAL-B binary components are always visible to all the processors in a node at the time of power-on; the system fabric should not need to be initialized.

Non-Power-on Resets

Non-power-on resets can occur for many reasons. Some PEI and DXE system services reset and reboot the entire platform, including all processors and devices. It is important to have a standard variant of this boot path for cases such as the following:

- Resetting the processor to change frequency settings
- Restarting hardware to complete chipset initialization
- Responding to an exception from a catastrophic error

This reset is also used for Configuration Values Driven through Reset (CVDR) configuration.

Normal Boot Paths

A traditional BIOS executes POST from a cold boot (G3 to S0 state), on resumes, or in special cases like INIT. EFI covers all those cases but provides a richer and more standardized operating environment

The basic code flow of the system needs to be changeable due to different circumstances. The boot path variable satisfies this need. The initial value of the boot mode is defined by some early PEIMs, but it can be altered by other, later PEIMs. All systems must support a basic S0 boot path. Typically a system has a richer set of boot paths, including S0 variations, S-state boot paths, and one or more special boot paths.

The architecture for multiple boot paths presented here has several benefits:

- The PEI Foundation is not required to be aware of system-specific requirements such as MP and various power states. This lack of awareness allows for scalability and headroom for future expansion.
- Supporting the various paths only minimally impacts the size of the PEI Foundation.
- The PEIMs required to support the paths scale with the complexity of the system.

Note that the Boot Mode Register becomes a variable upon transition to the DXE phase. The DXE phase can have additional modifiers that affect the boot path more than the PEI phase. These additional modifiers can indicate if the system is in manufacturing mode, chassis intrusion, or AC power loss or if silent boot is enabled.

In addition to the boot path types, modifier bits might be present. The recovery-needed modifier is set if any PEIM detects that it has become corrupted.

Basic G0-to-S0 and S0 Variation Boot Paths

The basic S0 boot path is *boot with full configuration.* This path setting informs all PEIMs to do a full configuration. The basic S0 boot path must be supported.

The Framework architecture also defines several optional variations to the basic S0 boot path. The variations that are supported depend on the following:

- Richness of supported features
- If the platform is open or closed
- Platform hardware

For example, a closed system or one that has detected a chassis intrusion could support a boot path that assumes no configuration changes from last boot option, thus allowing a very rapid boot time. Unsupported variations default to basic S0 operation. The following are the defined variations to the basic boot path:

- *Boot with minimal configuration:* This path is for configuring the minimal amount of hardware to boot the system.
- *Boot assuming no configuration changes:* This path uses the last configuration data.
- *Boot with full configuration plus diagnostics:* This path also causes any diagnostics to be executed.
- *Boot with default settings:* This path uses a known set of safe values for programming hardware.

S-State Boot Paths

The following optional boot paths allow for different operation for a resume from S3, S4, and S5:

- *S3 (Save to RAM Resume):* Platforms that support S3 resume must take special care to preserve/restore memory and critical hardware.
- *S4 (Save to Disk):* Some platforms may want to perform an abbreviated PEI and DXE phase on a S4 resume.
- *S5 (Soft Off):* Some platforms may want an S5 system state boot to be differentiated from a normal boot—for example, if buttons other than the power button can wake the system.

An S3 resume needs to be explained in more detail because it requires cooperation between a G0-to-S0 boot path and an S3 resume boot path. The G0-to-S0 boot path needs to save hardware programming information that the S3 resume path needs to retrieve. This information is saved in the Hardware Save Table using predefined data structures to perform

I/O or memory writes. The data is stored in an EFI equivalent of the INT15 E820 type 4 (firmware reserved memory) area or a firmware device area that is reserved for use by EFI. The S3 resume boot path code can access this region after memory has been restored.

Recovery Paths

All of the previously described boot paths can be modified or aborted if the system detects that recovery is needed. Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes on nonvolatile storage devices (flash, disk) are managed as blocks. If the system loses power while a block, or semantically bound blocks, are being updated, the storage might become invalid. On the other hand, the device might become corrupted by an errant program or by errant hardware. The system designers must determine the level of support for recovery based on their perceptions of the probabilities of these events occurring and their consequences.

The following are some reasons why system designers may choose not to support recovery:

- A system's firmware volume storage media might not support modification after being manufactured. It might be the functional equivalent of ROM.
- Most mechanisms of implementing recovery require additional firmware volume space, which might be too expensive for a particular application.
- A system may have enough firmware volume space and hardware features that the firmware volume can be made sufficiently fault tolerant to make recovery unnecessary.

Discovery

Discovering that recovery is required may be done using a PEIM (for example, by checking a "force recovery" jumper) or the PEI Foundation itself. The PEI Foundation might discover that a particular PEIM has not validated correctly or that an entire firmware has become corrupted.

General Recovery Architecture

The concept behind recovery is to preserve enough of the system firmware so that the system can boot to a point where it can do the following:

- Read a copy of the data that was lost from chosen peripherals.
- Reprogram the firmware volume with that data.

Preserving the recovery firmware is a function of the way the firmware volume store is managed, which is generally beyond the scope of this book. For the purpose of this description, it is expected that the PEIMs and other contents of the firmware volumes required for recovery are marked. The architecture of the firmware volume store must then preserve marked items, either by making them unalterable (possibly with hardware support) or must protect them using a fault-tolerant update process. Note that a PEIM is required to be in a fault-tolerant area if it indicates it is required for recovery or if a PEIM required for recovery depends on it. This architecture also assumes that it is fairly easy to determine that firmware volumes have become corrupted.

The PEI Dispatcher then proceeds as normal. If it encounters PEIMs that have been corrupted (for example, by receiving an incorrect hash value), it itself must change the boot mode to recovery. Once set to recovery, other PEIMs must not change it to one of the other states. After the PEI Dispatcher has discovered that the system is in recovery mode, it will restart itself, dispatching only those PEIMs that are required for recovery. A PEIM can also detect a catastrophic condition or a forced-recovery event and inform the PEI Dispatcher that it needs to proceed with a recovery dispatch. A PEIM can alert the PEI Foundation to start recovery by OR-ing the `BOOT_IN_RECOVERY_MODE_MASK` bit onto the present boot mode. The PEI Foundation then resets the boot mode to `BOOT_IN_RECOVERY_MODE` and starts the dispatch from the beginning with `BOOT_IN_RECOVERY_MODE` as the sole value for the mode.

It is possible that a PEIM could be built to handle the portion of the recovery that would initialize the recovery peripherals (and the buses they reside on) and then to read the new images from the peripherals and update the firmware volumes.

It is considered far more likely that the PEI will transition to DXE because DXE is designed to handle access to peripherals. This transition has the additional benefit that, if DXE then discovers that a device has become corrupted, it may institute recovery without transferring control back to the PEI.

If the PEI Foundation does not have a list of what it is to dispatch, how does it know whether an area of invalid space in a firmware volume should have contained a PEIM or not? It seems that the PEI Foundation may discover most corruption as an incidental result of its search for PEIMs. In this case, if the PEI Foundation completes its dispatch process without discovering enough static system memory to start DXE, then it should go into recovery mode.

Special Boot Path Topics

The remaining sections in this chapter discuss special boot paths that might be available to all processors or specific considerations that apply only for Intel Itanium processors.

Special Boot Paths

The following are special boot paths in the Framework architecture. Some of these paths are optional and others are processor-family specific.

- *Forced recovery boot:* A jumper or an equivalent mechanism indicates a forced recovery.
- *Intel Itanium architecture boot paths:* See the next section.
- *Capsule update:* This boot mode can be an INIT, S3, or some other means by which to restart the machine. If it is an S3, for example, the capsule cause will supersede the S3 restart. It is incumbent upon platform code, such as a memory initialization PEIM, to determine the exact cause and perform the correct behavior—that is, S3 state restoration versus INIT behavior.

Special Intel Itanium® Architecture Boot Paths

The architecture requires the following special boot paths:

- *Boot after INIT:* An INIT has occurred.
- *Boot after MCA:* A Machine Check Architecture (MCA) event has occurred.

Intel Itanium processors possess several unique boot paths that also invoke the dispatcher located at the System Abstraction Layer entry point SALE_ENTRY. The processor INIT and MCA are two asynchronous events that start up the SEC code/dispatcher in an Itanium-based system. The EFI security module is transparent during all the code paths except for the recovery check call that happens during a cold boot. The PEIMs or DXE drivers that handle these events are architecture-aware and do not return the control to the core dispatcher. They call their respective architectural handlers in the OS.

Intel Itanium® Architecture Access to the Boot Firmware Volume

Figure 14.3 shows the reset boot path that an Intel Itanium processor follows. Figure 14.4 shows the boot flow.

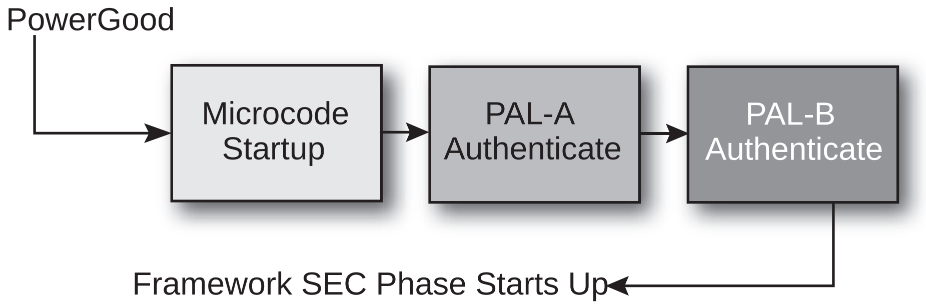


Figure 14.3 Intel® Itanium® Architecture Resets

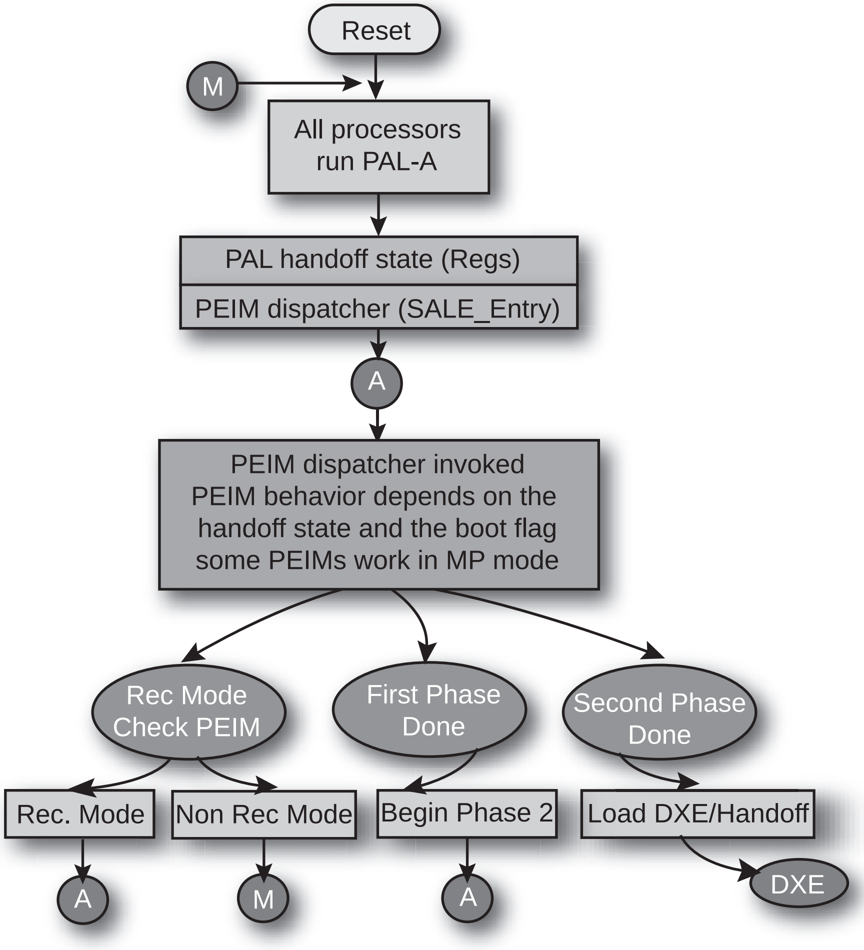


Figure 14.4 Intel® Itanium® Processor Boot Flow (MP versus UP on Other CPUs)

In Intel Itanium architecture, the microcode starts up the first layer of the PAL code, provided by the processor vendor, that resides in the Boot Firmware Volume (BFV). This code minimally initializes the processor and then finds and authenticates the second layer of PAL code (called PAL-B). The location of both PAL-A and PAL-B can be found by consulting either the architected pointers in the ROM near the 4-gigabyte region or by consulting the Firmware Interface Table (FIT) pointer in the ROM.

The PAL layer communicates with the OEM boot firmware using a single entry point called SALE_ENTRY.

The Intel Itanium architecture defines the initialization described above. In addition, however, Itanium-based systems that use the Framework architecture must do the following:

- *A "special" PEIM must be resident in the BFV to provide information about the location of the other firmware volumes.*

  The PEI Foundation will be located at the SALE_ENTRY point on the BFV. The Intel Itanium architecture PEIMs may reside in the BFV or other firmware volumes, but a special PEIM must be resident in the BFV to provide information about the location of the other firmware volumes.

- *The BFV of a particular node must be accessible by all the processors running in that node.*

  All the processors in each node start up and execute the PAL code and subsequently enter the PEI Foundation. The BFV of a particular node must be accessible by all the processors running in that node. This distinction also means that some of the PEIMs in the Intel Itanium architecture boot path will be multiprocessor-aware.

- *Firmware modules in a BFV must be organized such that PAL-A, PAL-B, and FIT binaries are always visible to all the processors in a node at the time of power-on.*

  These binaries must be visible without any initialization of the system fabric.

```
//*******************************************************
// EFI_BOOT_MODE
//*******************************************************
typedef UINT32     EFI_BOOT_MODE;

#define
BOOT_WITH_FULL_CONFIGURATION                        0x00
#define
BOOT_WITH_MINIMAL_CONFIGURATION                     0x01
#define
BOOT_ASSUMING_NO_CONFIGURATION_CHANGES              0x02
#define
BOOT_WITH_FULL_CONFIGURATION_PLUS_DIAGNOSTICS       0x03
#define
BOOT_WITH_DEFAULT_SETTINGS                          0x04
#define
BOOT_ON_S4_RESUME                                   0x05
```

```
#define
BOOT_ON_S5_RESUME                                     0x06
#define
BOOT_ON_S2_RESUME                                     0x10
#define
BOOT_ON_S3_RESUME                                     0x11
#define
BOOT_ON_FLASH_UPDATE                                  0x12
#define
BOOT_IN_RECOVERY_MODE                                 0x20

0x21 – 0xF..F Reserved Encodings
```

Table 14.2 lists the values and descriptions of the boot modes.

Table 14.2 Boot Mode Register

REGISTER BIT(S)	VALUES	DESCRIPTIONS
MSBit-0	000000b	Boot with full configuration
	000001b	Boot with minimal configuration
	000010b	Boot assuming no configuration changes from last boot
	000011b	Boot with full configuration plus diagnostics
	000100b	Boot with default settings
	000101b	Boot on S4 resume
	000110b	Boot in S5 resume
	000111b-001111b	Reserved for boot paths that configure memory
	010000b	Boot on S2 resume
	010001b	Boot on S3 resume
	010010b	Boot on flash update restart
	010011b-011111b	Reserved for boot paths that preserve memory context
	100000b	Boot in recovery mode
	100001b-111111b	Reserved for special boots

Architectural Boot Mode PPIs

In the PEI chapter the concept of an PEIM-to-PEIM interface (PPI) is introduced as the unit of interoperability in this phase of execution. PEI modules can ascerain the boot mode via the GetBootMode service once the module is dispatched, but a system designer may not want a PEIM to even run unless in a given boot mode. A possible hierarchy of boot mode PPIs abstracts the various producers of the boot mode. It is a hierarchy in that there should be an order of precedence in which each mode can be set. The PPIs and their respective GUIDs are described in Required Architectural PPIs for the PEI phase that can be found in the PEI Core Interface Specification and Optional Architectural PPIs. The hierarchy includes the master PPI, which publishes a PPI depended upon by the appropriate PEIMs, and some subsidiary PPI. For PEIMs that require that the boot mode is finally known, the Master Boot Mode PPI can be used as a dependency.

Table 14.3 lists the architectural boot mode PPIs.

Table 14.3 Architectural Boot Mode PPIs

PPI Name	Required or Optional?	PPI Definition in Section…
Master Boot Mode PPI	Required	Architectural PPIs: Required Architectural PPIs
Boot in Recovery Mode PPI	Optional	Architectural PPIs: Optional Architectural PPIs

Recovery

This section will talk about platform firmware recovery. "Recovery" is an optional to provide higher RASUM (Reliability, Availability, Usability, Managability) in the field. Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes (FVs) in nonvolatile storage (NVS) devices (flash or disk, for example) are managed as blocks. If the system loses power while a block, or semantically bound blocks, are being updated, the storage might become invalid. On the other hand, an errant program or hardware could corrupt the device. The system designers must determine the level of support for recovery based on their perceptions of the probabilities of these events occurring and the consequences.

The designers of a system may choose not to support recovery for the following reasons:

- A system's FV storage media might not support modification after being manufactured. It might be the functional equivalent of a ROM.
- Most mechanisms of implementing recovery require additional FV space that might be too expensive for a particular application.
- A system may have enough FV space and hardware features that the FV can be made sufficiently fault tolerant to make recovery unnecessary.

Discovery

Discovering that recovery is required may be done using a PEIM (for example, by checking a "force recovery" jumper) or the PEI Foundation itself. The PEI Foundation might discover that a particular PEIM has not validated correctly or that an entire firmware has become corrupted.

Note At this point a physical reset of the system has not occurred. The PEI Dispatcher has only cleared all state information and restarted itself.

It is possible that a PEIM could be built to handle the portion of the recovery that would initialize the recovery peripherals (and the buses they reside on) and then to read the new images from the peripherals and update the FVs.

It is considered far more likely that the PEI will transition to DXE because DXE is designed to handle access to peripherals. This has the additional benefit that, if DXE then discovers that a device has become corrupted, it may institute recovery without transferring control back to the PEI.

Since the PEI Foundation does not have a list of what to dispatch, how does it know if an area of invalid space in an FV should have contained a PEIM or not? The PEI Foundation should discover most corruption as an incidental result of its search for PEIMs. In this case, if the PEI Foundation completes its dispatch process without discovering enough static system memory to start DXE, then it should go into recovery mode.

Chapter 15

Pre-EFI Initialization (PEI)

Small is Beautiful

—E.F. Schumacher

The pre-EFI initialization (PEI) phase of execution has two primary roles in a platforms life: determine the source of the restart and provide a minimum amount of permanent memory for the ensuing DXE phase. "Small", "minimal", etc are terms often used for PEI code because of hardware resource constraints that limit the programming environment. Specifically, the Pre-EFI Initialization (PEI) phase provides a standardized method of loading and invoking specific initial configuration routines for the processor, chipset, and system board. The PEI phase occurs after the Security (SEC) phase. The primary purpose of code operating in this phase is to initialize enough of the system to allow instantiation of the Driver Execution Environment (DXE) phase. At a minimum, the PEI phase is responsible for determining the system boot path and initializing and describing a minimum amount of system RAM and firmware volume(s) that contain the DXE Foundation and DXE Architectural Protocols. As an application of Occam's Razor to the system design, the minimum amount of activity should be orchestrated and located in this phase of execution; no more, no less.

Scope

The PEI phase is responsible for initializing enough of the system to provide a stable base for subsequent phases. It is also responsible for detecting and recovering from corruption of the firmware storage space and providing the restart reason (boot-mode).

Today's PC generally starts execution in a very primitive state, from the perspective of the boot firmware, such as BIOS or the Framework. Processors might need updates to their internal microcode; the chipset (the chips that provide the interface between processors and the other major components of the system) require considerable initialization; and RAM requires sizing, location, and other initialization. The PEI phase is responsible for initializing these basic subsystems. The PEI phase is intended to provide a simple infrastructure by which a limited set of tasks can easily be accomplished to transition to the more advanced DXE phase. The PEI phase is intended to be responsible for only a very small subset of tasks that are required to boot the platform; in other words, it should perform only the minimal tasks that are required to start DXE. As improvements in the hardware occur, some of these tasks may migrate out of the PEI phase of execution.

Rationale

The design for PEI is essentially a miniature version of DXE that addresses many of the same issues. The PEI phase consists of several parts:

- A PEI Foundation
- One or more Pre-EFI Initialization Modules (PEIMs)

The goal is for the PEI Foundation to remain relatively constant for a particular processor architecture and to support add-in modules from various vendors for particular processors, chipsets, platforms, and other components. These modules usually cannot be coded without some interaction between one another and, even if they could, it would be inefficient to do so.

PEI is unlike DXE in that DXE assumes that reasonable amounts of permanent system RAM are present and available for use. PEI instead assumes that only a limited amount of temporary RAM exists and that it could be reconfigured for other uses during the PEI phase after permanent system RAM has been initialized. As such, PEI does not have the rich feature set that DXE does. The following are the most obvious examples of this difference:

- DXE has a rich database of loaded images and protocols bound to those images.
- PEI lacks a rich module hierarchy such as the DXE driver model.

Overview

The PEI phase consists of some Foundation code and specialized drivers known as PEIMs that customize the PEI phase operations to the platform. It is the responsibility of the Foundation code to dispatch the plug-ins in a sequenced order and provide basic services. The PEIMs are analogous to DXE drivers and generally correspond to the components being initialized. It is expected that common practice will be that the vendor of the component will provide the PEIM, possibly in source form so the customer can quickly debug integration problems.

The implementation of the PEI phase is more dependent on the processor architecture than any other Framework phase. In particular, the more resources that the processor provides at its initial or near initial state, the richer the PEI environment will be. As such, several parts of the following discussion note requirements for the architecture but are otherwise left less completely defined because they are specific to the processor architecture.

PEI can be viewed from both temporal and spatial perspectives. Figure 15.1 provides the overall Framework boot phase. The spatial view of PEI can be found in Figure 15.2. This picture describes the layering of the Framework components. This figure has often been referred to as the "H". PEI compromises the lower half of the "H". The temporal perspective entails "when" the PEI foundation and its associated modules execute. Figure 15.3 highlights the portions of Figure 15.1 that include PEI.

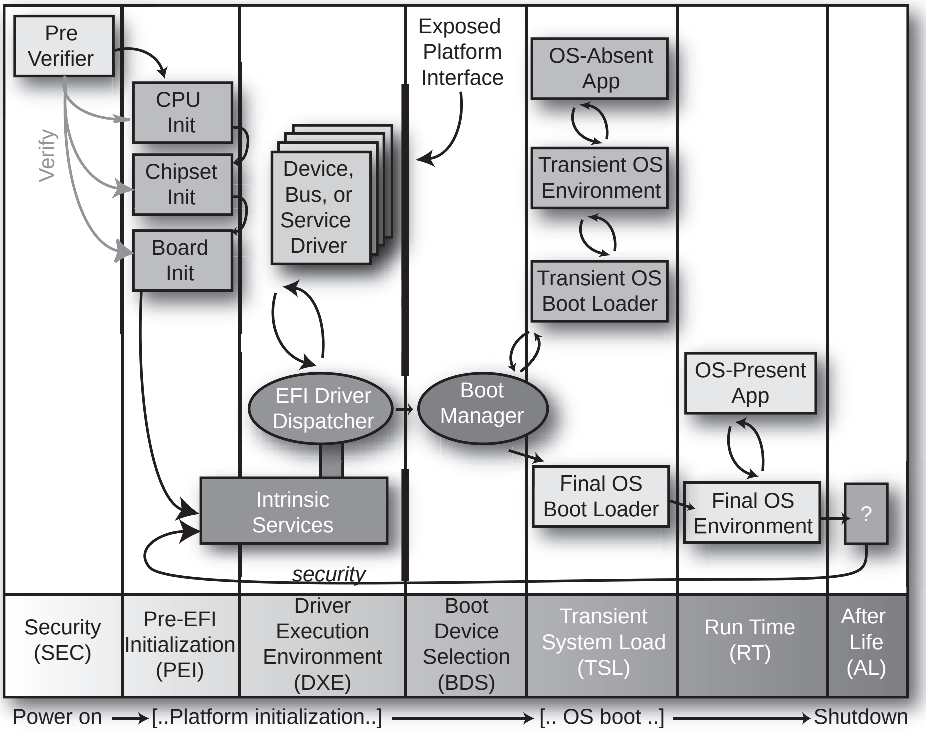


Figure 15.1 Overall Boot Flow

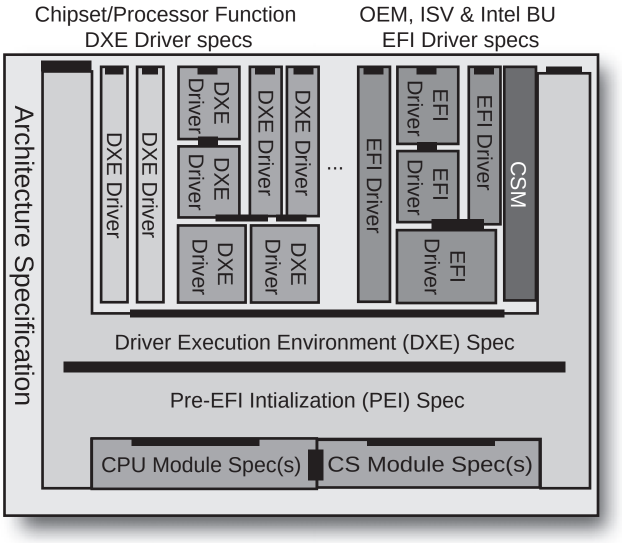


Figure 15.2 System Components

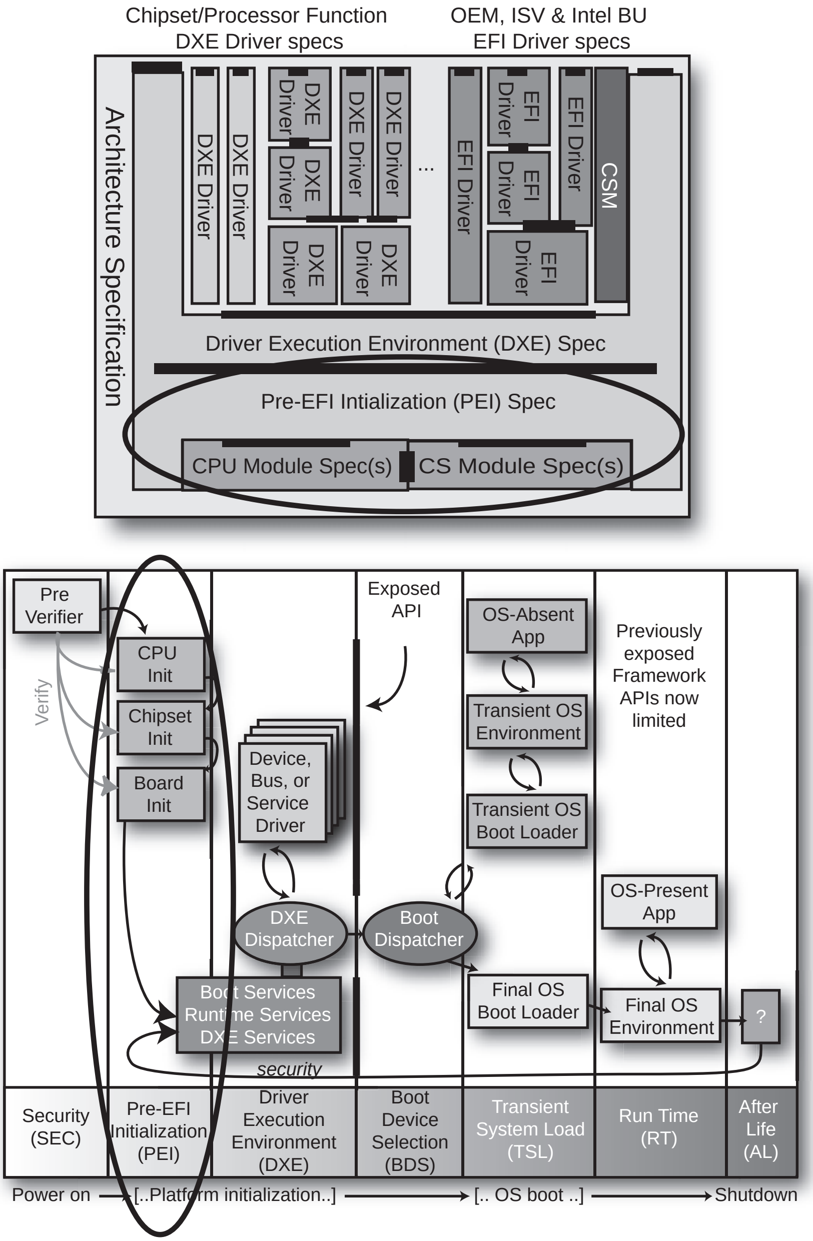


Figure 15.3 Portion of the Overall Boot Flow and Components for PEI

Phase Prerequisites

The following sections describe the prerequisites necessary for the successful completion of the PEI phase.

Temporary RAM

The PEI Foundation requires that the SEC phase initialize a minimum amount of scratch pad RAM that can be used by the PEI phase as a data store until system memory has been fully initialized. This scratch pad RAM should have access properties similar to normal system RAM—through memory cycles on the front side bus, for example. After system memory is fully initialized, the temporary RAM may be reconfigured for other uses. Typical provision for the temporary RAM is an architectural mode of the processor's internal caches.

Boot Firmware Volume

The Boot Firmware Volume (BFV) contains the PEI Foundation and PEIMs. It must appear in the memory address space of the system without prior firmware intervention and typically contains the reset vector for the processor architecture.

The contents of the BFV follow the format of the EFI flash file system. The PEI Foundation follows the EFI flash file system format to find PEIMs in the BFV. A platform-specific PEIM may inform the PEI Foundation of the location of other firmware volumes in the system, which allows the PEI Foundation to find PEIMs in other firmware volumes. The PEI Foundation and PEIMs are named by unique IDs in the EFI flash file system.

The PEI Foundation and some PEIMs required for recovery must either be locked into a nonupdateable BFV or be able to be updated using a fault-tolerant mechanism. The EFI flash file system provides error recovery; if the system halts at any point, either the old (preupdate) PEIM(s) or the newly updated PEIM(s) are entirely valid and the PEI Foundation can determine which is valid.

Security Primitives

The SEC phase provides an interface to the PEI Foundation to perform verification operations. To continue the root of trust, the PEI Foundation will use this mechanism to validate various PEIMs.

Concepts

The following sections describe the concepts in the PEI phase design.

PEI Foundation

The PEI Foundation is a single binary executable that is compiled to function with each processor architecture. It performs two main functions:

- Dispatching PEIMs
- Providing a set of common core services used by PEIMs

The PEI Dispatcher's job is to transfer control to the PEIMs in an orderly manner. The common core services are provided through a service table referred to as the PEI Services Table. These services do the following:

- Assist in PEIM-to-PEIM communication.
- Abstract management of the temporary RAM.
- Provide common functions to assist the PEIMs in the following:
  - Finding other files in the FFS
  - Reporting status codes
  - Preparing the handoff state for the next phase of the Framework

When the SEC phase is complete, SEC invokes the PEI Foundation and provides the PEI Foundation with several parameters:

- The location and size of the BFV so that the PEI Foundation knows where to look for the initial set of PEIMs.
- A minimum amount of temporary RAM that the PEI phase can use
- A verification service callback to allow the PEI Foundation to verify that PEIMs that it discovers are authenticated to run before the PEI Foundation dispatches them

The PEI Foundation assists PEIMs in communicating with each other. The PEI Foundation maintains a database of registered interfaces for the PEIMs, as shown in Figure 15.4. These interfaces are called PEIM-to-PEIM Interfaces (PPIs). The PEI Foundation provides the interfaces to allow PEIMs to register PPIs and to be notified (called back) when another PEIM installs a PPI.

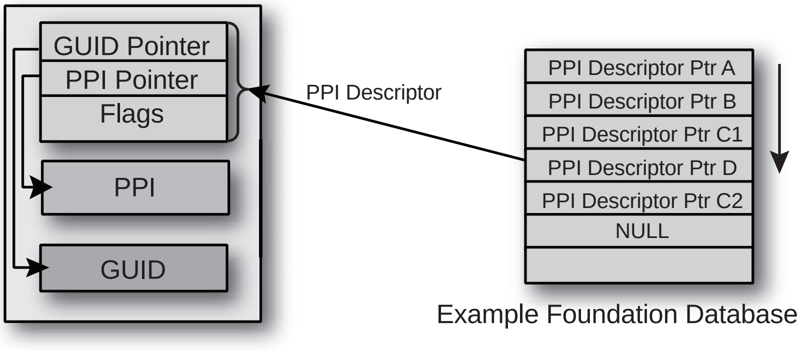


Figure 15.4 How a PPI Is Registered

The PEI Dispatcher consists of a single phase. It is during this phase that the PEI Foundation examines each file in the firmware volumes that contain files of type PEIM. It examines the dependency expression (depex) within each firmware file to decide if a PEIM can run. A dependency expression is code associated with each driver that describes the dependencies that must be satisfied for that driver to run. The binary encoding of dependency expressions for PEIMs is the same as that of dependency expressions associated with a DXE driver.

Pre-EFI Initialization Modules (PEIMs)

Pre-EFI Initialization Modules (PEIMs) are executable binaries that encapsulate processor, chipset, device, or other platform-specific functionality. PEIMs may provide interface(s) that allow other PEIMs or the PEI Foundation to communicate with the PEIM or the hardware for which the PEIM abstracts. PEIMs are separately built binary modules that typically reside in ROM and are therefore uncompressed. A small subset of PEIMs exist that may run from RAM for performance reasons. These PEIMs reside in ROM in a compressed format. PEIMs that reside in ROM are execute-in-place modules that may consist of either position-independent code or position-dependent code with relocation information.

PEI Services

The PEI Foundation establishes a system table named the PEI Services Table that is visible to all PEIMs in the system. A PEI service is defined as a function, command, or other capability that is manifested by the PEI Foundation when that service's initialization requirements are met. Because the PEI phase has no permanent memory available until nearly the end of the phase, the range of services created during the PEI phase cannot be as rich as those created during later phases. Because the location of the PEI Foundation and its temporary RAM is not known at build time, a pointer to the PEI Services Table is passed into each PEIM's entry point and also to part of each PPI. The PEI Foundation provides the following classes of services:

- *PPI Services:* Manages PPIs to facilitate intermodule calls between PEIMs. Interfaces are installed and tracked on a database maintained in temporary RAM.
- *Boot Mode Services:* Manages the boot mode (S3, S5, normal boot, diagnostics, and so on) of the system.
- *HOB Services:* Creates data structures called Hand-Off Blocks (HOBs) that are used to pass information to the next phase of the Framework.
- *Firmware Volume Services:* Scans the FFS in firmware volumes to find PEIMs and other firmware files in the flash device.
- *PEI Memory Services:* Provides a collection of memory management services for use both before and after permanent memory has been discovered.
- *Status Code Services:* Common progress and error code reporting services, that is, port 080h or a serial port for simple text output for debug.
- *Reset Services:* Provides a common means by which to initiate a restart of the system.

PEIM-to-PEIM Interfaces (PPIs)

PEIMs may invoke other PEIMs through interfaces named PEIM-to-PEIM Interfaces (PPIs). The interfaces themselves are named using Globally Unique Identifiers (GUIDs) to allow the independent development of modules and their defined interfaces without naming collision. A GUID is a 128-bit value used to differentiate services and structures in the boot services. The PPIs are defined as structures that may contain functions, data, or a combination of the two. PEIMs must register their PPIs with the PEI Foundation, which manages a database of registered PPIs. A PEIM that wants to use a specific PPI can then query the PEI Foundation to find the interface it needs. The two types of PPIs are:

- Services
- Notifications

PPI services allow a PEIM to provide functions or data for another PEIM to use. PPI notifications allow a PEIM to register for a callback when another PPI is registered with the PEI Foundation.

Simple Heap

The PEI Foundation uses temporary RAM to provide a simple heap store before permanent system memory is installed. PEIMs may request allocations from the heap, but no mechanism exists to free memory from the heap. Once permanent memory is installed, the heap is relocated to permanent system memory, but the PEI Foundation does not fix up existing data within the heap. Therefore, a PEIM cannot store pointers in the heap when the target is other data within the heap, such as linked lists.

Hand-Off Blocks (HOBs)

Hand-Off Blocks (HOBs) are the architectural mechanism for passing system state information from the PEI phase to the DXE phase in the Framework architecture. A HOB is simply a data structure (cell) in memory that contains a header and data section. The header definition is common for all HOBs and allows any code using this definition to know two items:

- The format of the data section
- The total size of the HOB

HOBs are allocated sequentially in the memory that is available to PEIMs after permanent memory has been installed. A series of core services facilitate This sequential list of HOBs in memory is referred to as the *HOB list.* The first HOB in the HOB list must be the Phase Handoff Information Table (PHIT) HOB that describes the physical memory used by the PEI phase and the boot mode discovered during the PEI phase.

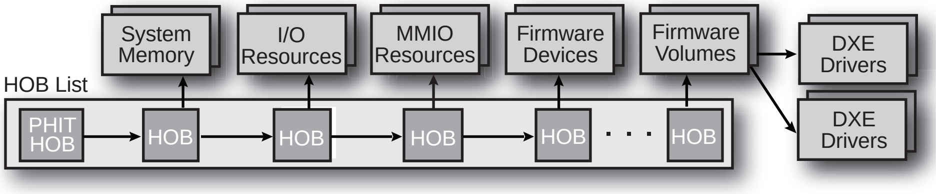


Figure 15.5 The HOB List

Only PEI components are allowed to make additions or changes to HOBs. Once the HOB list is passed into DXE, it is effectively read-only for DXE components. The ramifications of a read-only HOB list for DXE is that handoff information, such as boot mode, must be handled in a unique fashion; if DXE were to engender a recovery condition, it would not update the boot mode but instead would implement the action using a special type of reset call. The HOB list contains system state data at the time of PEI-to-DXE handoff and does not represent the current system state during DXE. DXE components should use services that are defined for DXE to get the current system state instead of parsing the HOB list.

As a guideline, it is expected that HOBs passed between PEI and DXE will follow a one producer–to–one consumer model. In other words, a PEIM will produce a HOB in PEI, and a DXE Driver will consume that HOB and pass information associated with that HOB to other DXE components that need the information. The methods that the DXE Driver uses to provide that information to other DXE components should follow mechanisms defined by the DXE architecture.

Operation

PEI phase operation consists of invoking the PEI Foundation, dispatching all PEIMs in an orderly manner, and discovering and invoking the next phase. During PEI Foundation initialization, the PEI Foundation initializes the internal data areas and functions that are needed to provide the common PEI services to PEIMs. During PEIM dispatch, the PEI Dispatcher traverses the firmware volume(s) and discovers PEIMs according to the flash file system definition. The PEI Dispatcher then dispatches PEIMs if the following criteria are met:

- The PEIM has not already been invoked.

- The PEIM file is correctly formatted.
- The PEIM is trustworthy.
- The PEIM's dependency requirements have been met.

After dispatching a PEIM, the PEI Dispatcher continues traversing the firmware volume(s) until either all discovered PEIMs have been invoked or no more PEIMs can be invoked because the requirements listed above cannot be met for any PEIMs. Once this condition has been reached, the PEI Dispatcher's job is complete and it invokes an architectural PPI for starting the next phase of the Framework, the DXE Initial Program Load (IPL) PPI.

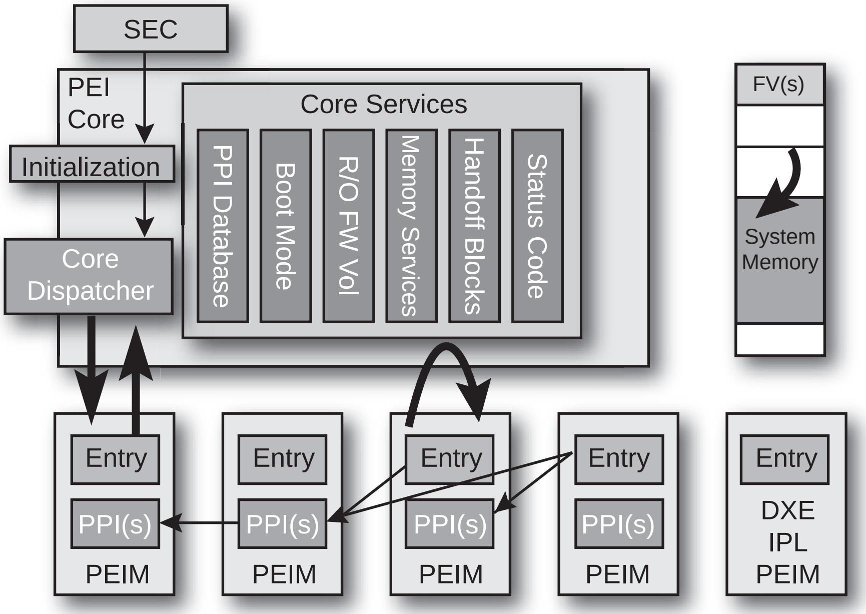


Figure 15.6 PEI Boot Flow

Dependency Expressions

The sequencing of PEIMs is determined by evaluating a *dependency expression* associated with each PEIM. This Boolean expression describes the requirements that are necessary for that PEIM to run, which imposes a weak ordering on the PEIMs. Within this weak ordering, the PEIMs may be initialized in any order. The GUIDs of PPIs and the GUIDs of file names are referenced in the dependency expression. The dependency expression is a representative syntax of operations that can be performed on a plurality of dependencies to determine whether the PEIM can be run. The PEI Foundation evaluates this dependency expression against an internal database of run PEIMs and registered PPIs. Operations that may be performed on dependencies are the logical operators `AND`, `OR`, and `NOT` and the sequencing operators `BEFORE` and `AFTER`.

Verification/Authentication

The PEI Foundation is stateless with respect to security. Instead, security decisions are assigned to platform-specific components. The two components of interest that abstract security include the Security PPI and a Verification PPI. The purpose of the Verification PPI is to check the authentication status of a given PEIM. The mechanism used therein may include digital signature verification, a simple checksum, or some other OEM-specific mechanism. The result of this verification is returned to the PEI Foundation, which in turn conveys the result to the Security PPI. The Security PPI decides whether to defer execution of the PEIM or to let the execution occur. In addition, the Security PPI provider may choose to generate an attestation log entry of the dispatched PEIM or provide some other security exception.

PEIM Execution

PEIMs run to completion when invoked by the PEI Foundation. Each PEIM is invoked only once and must perform its job with that invocation and install other PPIs to allow other PEIMs to call it as necessary. PEIMs may also register for a notification callback if it is necessary for the PEIM to get control again after another PEIM has run.

Memory Discovery

Memory discovery is an important architectural event during the PEI phase. When a PEIM has successfully discovered, initialized, and tested a contiguous range of system RAM, it reports this RAM to the PEI Foundation. When that PEIM exits, the PEI Foundation migrates PEI usage of the temporary RAM to real system RAM, which involves the following two tasks:

- The PEI Foundation must switch PEI stack usage from temporary RAM to permanent system memory.
- The PEI Foundation must migrate the simple heap allocated by PEIMs (including HOBs) to real system RAM.

Once this process is complete, the PEI Foundation installs an architectural PPI to notify any interested PEIMs that real system memory has been installed. This notification allows PEIMs that ran before memory was installed to be called back so that they can complete necessary tasks—such as building HOBs for the next phase of DXE—in real system memory.

Intel® Itanium® Processor MP Considerations

This section gives special consideration to the PEI phase operation in Intel Itanium processor family multiprocessor (MP) systems. In Itanium-based systems, all of the processors in the system start up simultaneously and execute the PAL initialization code that is provided by the processor vendor. Then all the processors call into the Framework start-up code with a request for recovery check. The start-up code allocates different chunks of temporary memory for each of the active processors and sets up stack and backing store pointers in the allocated temporary memory. The temporary memory could be a part of the processor cache (cache as RAM), which can be configured by invoking a PAL call. The start-up code then starts dispatching PEIMs on each of these processors. One of the early PEIMs that runs in MP mode is the PEIM that selects one of the processors as the boot-strap processor (BSP) for running the PEIM stage of the booting.

This BSP continues to run PEIMs until it finds permanent memory and installs the memory with the PEI Foundation. Then the BSP wakes up all the processors to determine their health and PAL compatibility status. If none of these checks warrants a recovery of the firmware, the processors are returned to the PAL for more processor initialization and a normal boot.

The Framework start-up code also gets triggered in an Itanium-based system whenever an INIT or a Machine Check Architecture (MCA) event occurs in the system. Under such conditions, the PAL code outputs status codes and a buffer called the *minimum state buffer*. A Framework-specific data pointer that points to the INIT and MCA code data area is attached to this minimum state buffer, which contains details of the code to be executed upon INIT and MCA events. The buffer also holds some important variables needed by the start-up code to make decisions during these special hardware events.

Recovery

Recovery is the process of reconstituting a system's firmware devices when they have become corrupted. The corruption can be caused by various mechanisms. Most firmware volumes on nonvolatile storage devices are managed as blocks. If the system loses power while a block or semantically bound blocks are being updated, the storage might become invalid. On the other hand, the device might become corrupted by an errant program or by errant hardware. Assuming PEI lives in a fault-tolerant block, it can support a recovery mode dispatch.

A PEIM or the PEI Foundation itself can discover the need to do recovery. A PEIM can check a "force recovery" jumper, for example, to detect a need for recovery. The PEI Foundation might discover that a particular PEIM does not validate correctly or that an entire firmware volume has become corrupted.

The concept behind recovery is that enough of the system firmware is preserved so that the system can boot to a point that it can read a copy of the data that was lost from chosen peripherals and then reprogram the firmware volume with that data.

Preservation of the recovery firmware is a function of the way the firmware volume store is managed. In the EFI flash file system, PEIMs required for recovery are marked as such. The firmware volume store architecture must then preserve marked items, either by making them unalterable (possibly with hardware support) or protect them using a fault-tolerant update process.

Until recovery mode has been discovered, the PEI Dispatcher proceeds as normal. If the PEI Dispatcher encounters PEIMs that have been corrupted (for example, by receiving an incorrect hash value), it must change the boot mode to recovery. Once set to recovery, other PEIMs must not change it to one of the other states. After the PEI Dispatcher has discovered that the system is in recovery mode, it will restart itself, dispatching only those PEIMs that are required for recovery. It is also possible for a PEIM to detect a catastrophic condition or to be a forced-recovery detect PEIM and to inform the PEI Dispatcher that it needs to proceed with a recovery dispatch. The recovery dispatch is completed when a PEIM finds a recovery firmware volume on a recovery media and the DXE Foundation is started from that firmware volume. Drivers within that DXE firmware volume can perform the recovery process.

S3 Resume

The PEI phase on S3 resume (save-to-RAM resume) differs in several fundamental ways from the PEI phase on a normal boot. The differences are as follows:

- The memory subsection is restored to its presleep state rather than initialized.
- System memory owned by the OS is not used by either the PEI Foundation or the PEIMs.
- The DXE phase is not dispatched on a resume because it would corrupt memory.
- The PEIM that would normally dispatch the DXE phase instead uses a special Hardware Save Table to restore fundamental hardware back to a boot configuration. After restoring the hardware, the PEIM passes control to the OS-supplied resume vector.
- The DXE and later phases during a normal boot save enough information in the Framework reserved memory or a firmware volume area for hardware to be restored to a state that the OS can use to restore devices. This saved information is located in the Hardware Save Table.

Example System

All of the concepts regarding PEI can be synthesized when reviewing a specific platform. The following list represents an 865 system with all of the associated system components. This same system is also shown in Figure 15.7, which includes the actual silicon components. Figure 15.8 provides an idealized version of this same system. The components in the latter figure have corresponding PEIM's to abstract both the initialization of and services by the components. For each of these components, one to several PEI Modules can be delivered that abstract the specific component's behavior. An example of these components can include:

- Pentium® 4 processor PEIM: Initialization and CPU I/O service
- PCI Configuration PEIM: PCI Configuration PPI
- ICH PEIM: ICH initialization and the SMBUS PPI
- Memory initialization PEIM: Reading SPD through the SMBUS PPI, initialization of the memory controller, and reporting memory available to the PEI core

- Platform PEIM: Creation of the flash mode, detection of boot mode
- DXE IPL: Generic services to launch DXE, invoke S3 or recovery flow

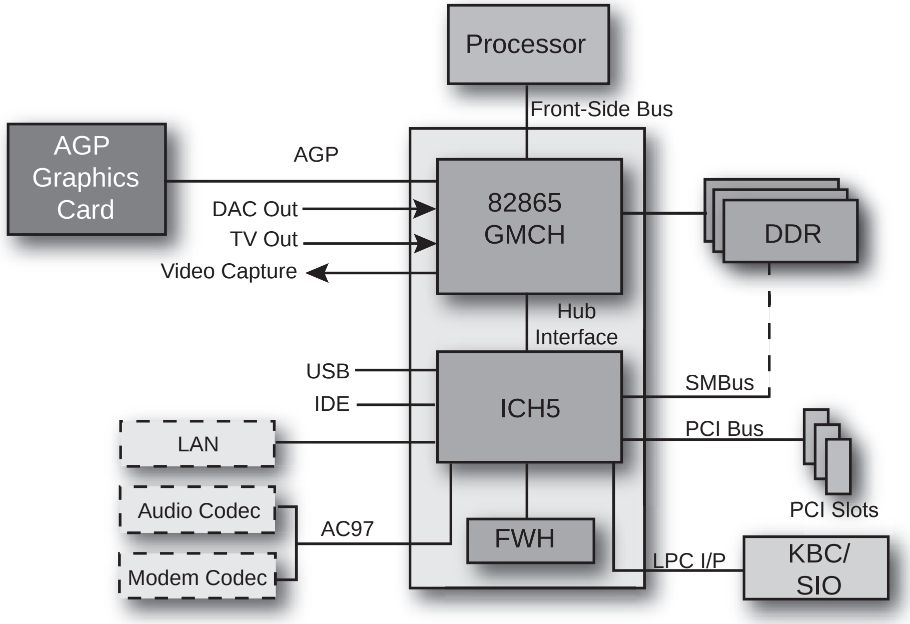


Figure 15.7 Specific System

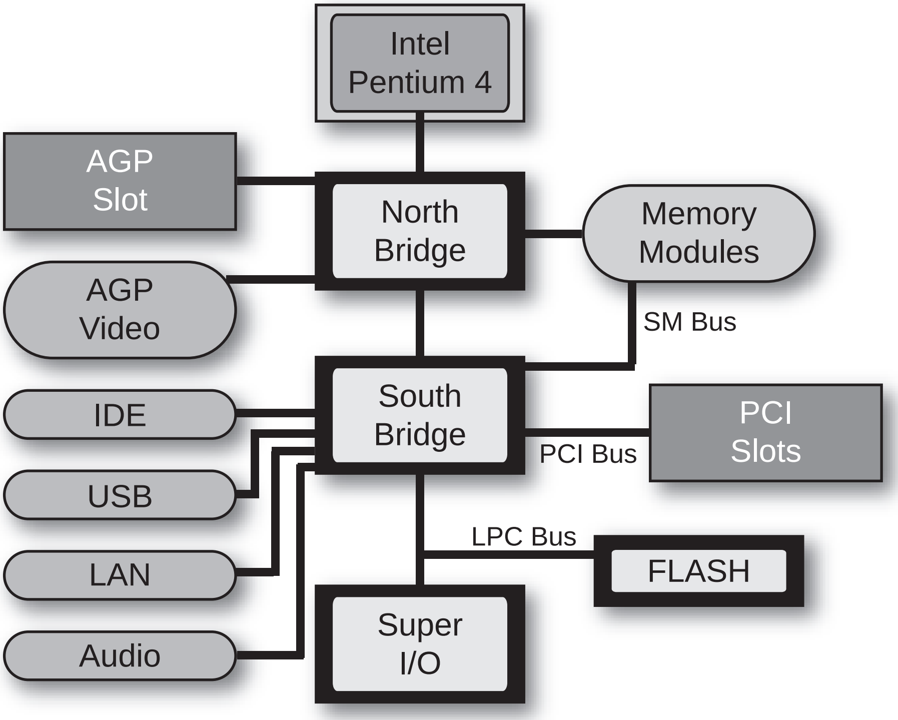


Figure 15.8 Idealization of Actual System

```
typedef
EFI_STATUS
(EFIAPI *PEI_SMBUS_PPI_EXECUTE_OPERATION) (
  IN      EFI_PEI_SERVICE            **PeiServices,
  IN      struct EFI_PEI_SMBUS_PPI   *This,
  IN      EFI_SMBUS_DEVICE_ADDRESS   SlaveAddress,
  IN      EFI_SMBUS_DEVICE_COMMAND   Command,
  IN      EFI_SMBUS_OPERATION        Operation,
  IN      BOOLEAN                    PecCheck,
  IN OUT  UINTN                      *Length,
  IN OUT  VOID                       *Buffer
  );
typedef struct {
  PEI_SMBUS_PPI_EXECUTE_OPERATION  Execute;
  PEI_SMBUS_PPI_ARP_DEVICE         ArpDevice;
} EFI_PEI_SMBUS_PPI;
```

Figure 15.9 Instance of a PPI

What is notable about a PPI is that it is like an EFI protocol in that it has member services and/or static data. The PPI is named by a GUID and can have several instances. The SMBUS PPI, for example, could be implemented for SMBUS controllers in the ICH, in another vendor's integrated Super I/O (SIO), or other component. Figure 15.10 illustrates an instance of an SMBUS PPI for an Intel ICH.

```
#define SMBUS_R_HD0  0xEFA5
#define SMBUS_R_HBD  0xEFA7

EFI_PEI_SERVICES          *PeiServices;
SMBUS_PRIVATE_DATA        *Private;
UINT8  Index, BlockCount  *Length;
UINT8                     *Buffer;

BlockCount = Private->CpuIo.IoRead8 (
               *PeiServices,Private->CpuIo,SMBUS_R_HD0);
if (*Length < BlockCount) {
  return EFI_BUFFER_TOO_SMALL;
} else {
  for (Index = 0; Index < BlockCount; Index++) {
    Buffer[Index] = Private->CpuIo.IoRead8 (
                      *PeiServices,Private-
>CpuIo,SMBUS_R_HBD);
  }
}
```

Figure 15.10 Code that Supports a PPI Service

Chapter 16

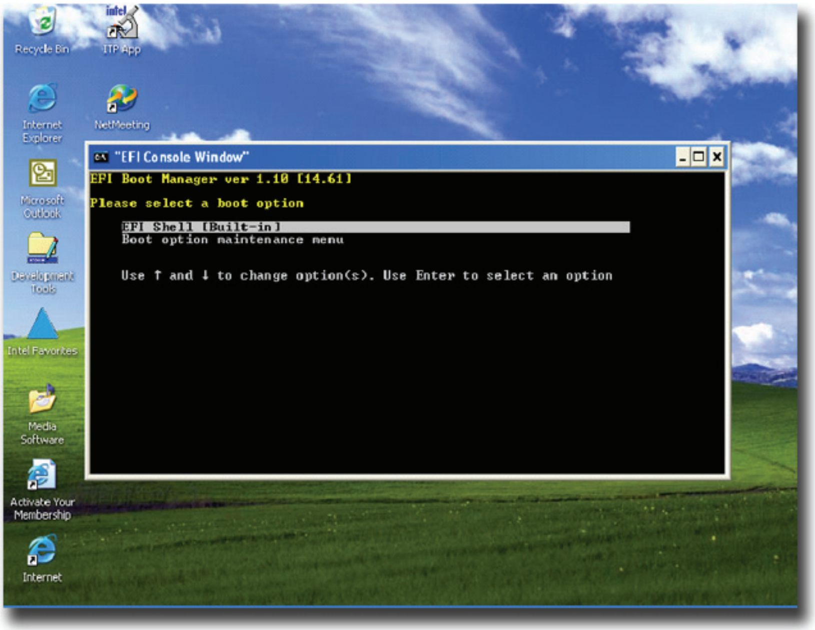
Putting It All Together—Firmware Emulation

An expert is a man who has made all the mistakes which can be made in a very narrow field.

—Niels Bohr

In the preceding chapters, various stages of the firmware initialization process have been described. In addition, various possible usage models have been described that can be implemented on a target hardware platform. In addition, by now it should have become evident that many of the EFI firmware interfaces do not in and of themselves directly talk to hardware; instead they actually talk to underlying components that are responsible for talking to hardware. Traditionally, firmware development has not been an activity that could be performed without an in-circuit emulator (ICE) or other hardware debug facility. Taking into consideration EFI's design and the fact that very few components in the firmware actually have direct interaction with hardware devices, it is possible to introduce a mechanism that allows the emulation of vast amounts of the firmware in a standard deployment operation system environment.

In the EFI sample implementation, a new target platform was introduced called "NT32". This environment features the ability to run much of the firmware code as an application running from the operating system, and provides the ability to establish a robust development and debug environment. Much of the firmware codebase was developed initially using the emulation environment with off-the-shell compilers and debuggers, and without the need of a real hardware debugger. Of course, this emulation has its limitations, since some components of the firmware must talk to hardware. It is much more difficult to emulate such components, though later in this chapter, some possibilities will be discussed to alleviate some of this issue. Figure 16.1 shows an example of a firmware emulation environment running the EFI shell within an operating system context.



```
EFI Boot Manager ver 1.10 [14.61]

Please select a boot option

    EFI Shell [Built-in]
    Boot option maintenance menu

    Use ↑ and ↓ to change option(s). Use Enter to select an option
```

Figure 16.1 An Emulation Environment Contained within an Operating System Environment

Virtual Platform

This NT32 platform can be described as a hardware-agnostic platform in that it uses operating system APIs for its primary hardware abstractions. Figure 16.2 shows how the firmware emulation environment gets launched. It is part of a normal boot process, and will essentially launch a firmware emulation environment as an application running from the operating system. For most developers, this simply means launching a standard platform, loading an operating system, and then building and executing the NT32 emulation environment as a native operating system application. This application will effectively execute the firmware that was built, and emulate the launch of a new system.

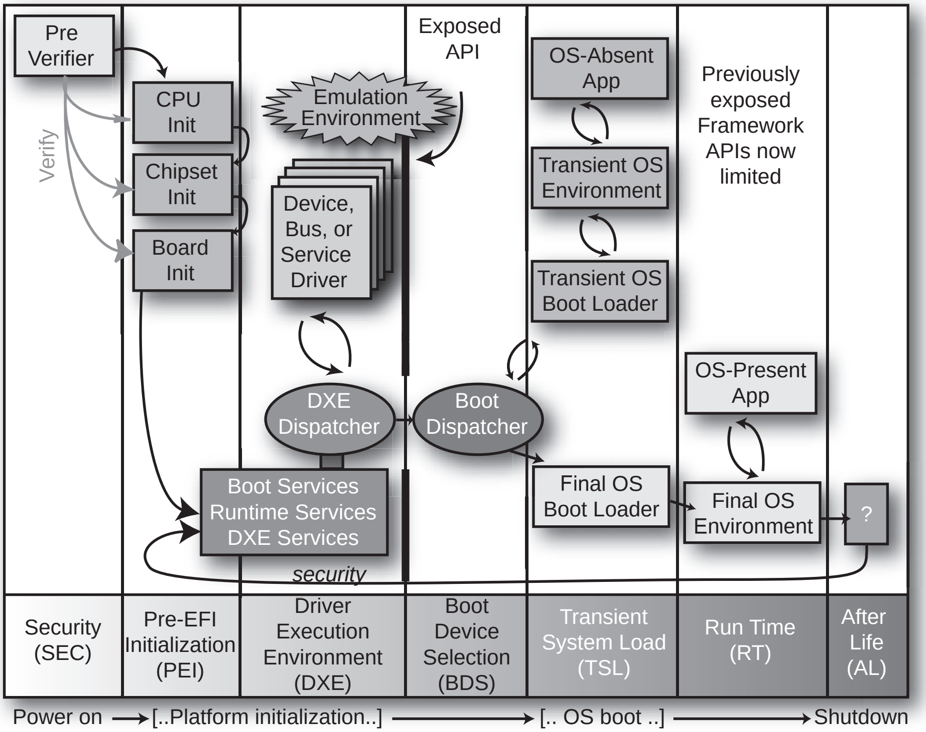

Pre Verifier
Verify
CPU Init
Chipset Init
Board Init
Emulation Environment
Device, Bus, or Service Driver
Exposed API
DXE Dispatcher
Boot Dispatcher
Boot Services Runtime Services DXE Services
security
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
Previously exposed Framework APIs now limited
OS-Present App
Final OS Environment
?
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on → [..Platform initialization..] → [.. OS boot ..] → Shutdown

Figure 16.2 The Normal Boot Process Launching an Operating System that Will Launch the Emulation Environment

In Figure 16.3, the timeline is actually intended to illustrate the emulated firmware timeline. It has the capability of processing all of the firmware evolution stages, yet of course certain operations will be emulated due to lack of direct hardware initialization. An example would be the direct initialization of memory, which would be somewhat different in this environment, whereas in a real platform, this process would be much more involved.

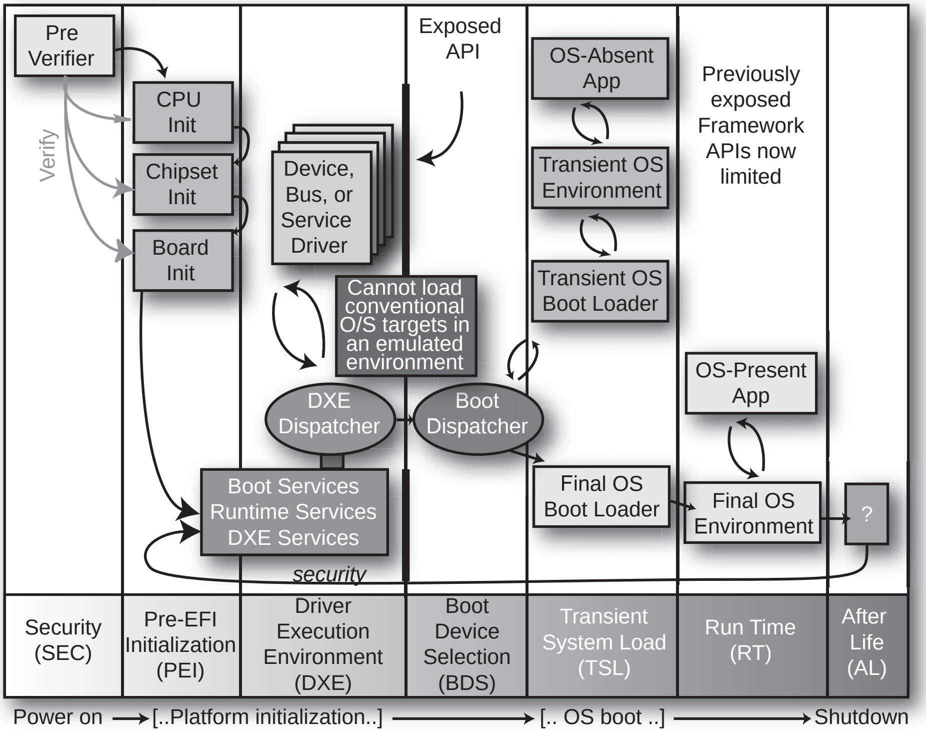

Pre Verifier
Verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
Exposed API
Cannot load conventional O/S targets in an emulated environment
DXE Dispatcher
Boot Dispatcher
Boot Services Runtime Services DXE Services
security
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Previously exposed Framework APIs now limited
OS-Present App
Final OS Boot Loader
Final OS Environment
?
Security (SEC)
Pre-EFI Initialization (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Run Time (RT)
After Life (AL)
Power on
[..Platform initialization..]
[.. OS boot ..]
Shutdown

Figure 16.3 The Firmware Emulation Environment Itself

Emulation Firmware Phases

It should be noted that the emulation environment has several distinct phases:

- Establishing a WinNtThunk capability for the emulation environment.

  This phase constructs a means by which firmware components can make reference to some "hardware" components. This is done by associating firmware-visible constructs that will then be associated with operating system native API calls.

  Figure 16.4 is an example where several firmware constructs are being associated with operating system native APIs. For example, to create a file, we establish a firmware calling mechanism (such as

WinNtCreateFile) to call an operating system API known as "CreateFile". The following examples illustrate a mechanism of associating firmware calls to Windows APIs, but this could just as easily happen for any underlying operation system.

```
typedef struct {
  UINT64                              Signature;

  //
  // Win32 Process APIs
  //
  WinNtGetProcAddress                 GetProcAddress;
  WinNtGetTickCount                   GetTickCount;
  WinNtLoadLibraryEx                  LoadLibraryEx;
  WinNtFreeLibrary                    FreeLibrary;
  WinNtSetPriorityClass               SetPriorityClass;
  WinNtSetThreadPriority              SetThreadPriority;
  WinNtSleep                          Sleep;
  WinNtSuspendThread                  SuspendThread;
  WinNtGetCurrentThread               GetCurrentThread;
  WinNtGetCurrentThreadId             GetCurrentThreadId;
  WinNtGetCurrentProcess              GetCurrentProcess;
  WinNtCreateThread                   CreateThread;
  WinNtTerminateThread                TerminateThread;
  WinNtSendMessage                    SendMessage;
  WinNtExitThread                     ExitThread;
  WinNtResumeThread                   ResumeThread;
  WinNtDuplicateHandle                DuplicateHandle;

  //
  // Wint32 Mutex primitive
  //
  WinNtInitializeCriticalSection      InitializeCriticalSection;
  WinNtEnterCriticalSection           EnterCriticalSection;
  WinNtLeaveCriticalSection           LeaveCriticalSection;
  WinNtDeleteCriticalSection          DeleteCriticalSection;
  WinNtTlsAlloc                       TlsAlloc;

  WinNtTlsFree                        TlsFree;
  WinNtTlsSetValue                    TlsSetValue;
  WinNtTlsGetValue                    TlsGetValue;
  WinNtCreateSemaphore                CreateSemaphore;
  WinNtWaitForSingleObject            WaitForSingleObject;
  WinNtReleaseSemaphore               ReleaseSemaphore;

  //
  // Win32 Console APIs
  //
  WinNtCreateConsoleScreenBuffer      CreateConsoleScreenBuffer;
  WinNtFillConsoleOutputAttribute     FillConsoleOutputAttribute;
  WinNtFillConsoleOutputCharacter     FillConsoleOutputCharacter;
  WinNtGetConsoleCursorInfo           GetConsoleCursorInfo;
```

```
WinNtGetNumberOfConsoleInputEvents  GetNumberOfConsoleInputEvents;
WinNtPeekConsoleInput               PeekConsoleInput;
WinNtScrollConsoleScreenBuffer      ScrollConsoleScreenBuffer;
WinNtReadConsoleInput               ReadConsoleInput;
WinNtSetConsoleActiveScreenBuffer   SetConsoleActiveScreenBuffer;
WinNtSetConsoleCursorInfo           SetConsoleCursorInfo;
WinNtSetConsoleCursorPosition       SetConsoleCursorPosition;
WinNtSetConsoleScreenBufferSize     SetConsoleScreenBufferSize;
WinNtSetConsoleTitleW               SetConsoleTitleW;
WinNtWriteConsoleInput              WriteConsoleInput;
WinNtWriteConsoleOutput             WriteConsoleOutput;

//
// Win32 File APIs
//
WinNtCreateFile                     CreateFile;
WinNtDeviceIoControl                DeviceIoControl;
WinNtCreateDirectory                CreateDirectory;
WinNtRemoveDirectory                RemoveDirectory;
WinNtGetFileAttributes              GetFileAttributes;
WinNtSetFileAttributes              SetFileAttributes;
WinNtCreateFileMapping              CreateFileMapping;
WinNtCloseHandle                    CloseHandle;
WinNtDeleteFile                     DeleteFile;
WinNtFindFirstFile                  FindFirstFile;
WinNtFindNextFile                   FindNextFile;
WinNtFindClose                      FindClose;
WinNtFlushFileBuffers               FlushFileBuffers;
WinNtGetEnvironmentVariable         GetEnvironmentVariable;
WinNtGetLastError                   GetLastError;
WinNtSetErrorMode                   SetErrorMode;
WinNtGetStdHandle                   GetStdHandle;
WinNtMapViewOfFileEx                MapViewOfFileEx;
WinNtReadFile                       ReadFile;
WinNtSetEndOfFile                   SetEndOfFile;
WinNtSetFilePointer                 SetFilePointer;
WinNtWriteFile                      WriteFile;
WinNtGetFileInformationByHandle     GetFileInformationByHandle;
WinNtGetDiskFreeSpace               GetDiskFreeSpace;
WinNtGetDiskFreeSpaceEx             GetDiskFreeSpaceEx;
WinNtMoveFile                       MoveFile;
WinNtSetFileTime                    SetFileTime;
WinNtSystemTimeToFileTime           SystemTimeToFileTime;

//
// Win32 Time APIs
//
WinNtFileTimeToLocalFileTime        FileTimeToLocalFileTime;
WinNtFileTimeToSystemTime           FileTimeToSystemTime;
WinNtGetSystemTime                  GetSystemTime;
WinNtSetSystemTime                  SetSystemTime;
WinNtGetLocalTime                   GetLocalTime;
WinNtSetLocalTime                   SetLocalTime;
WinNtGetTimeZoneInformation         GetTimeZoneInformation;
WinNtSetTimeZoneInformation         SetTimeZoneInformation;
WinNttimeSetEvent                   timeSetEvent;
```

```
WinNttimeKillEvent                  timeKillEvent;

//
// Win32 Serial APIs
//
WinNtClearCommError                 ClearCommError;
WinNtEscapeCommFunction             EscapeCommFunction;
WinNtGetCommModemStatus             GetCommModemStatus;
WinNtGetCommState                   GetCommState;
WinNtSetCommState                   SetCommState;
WinNtPurgeComm                      PurgeComm;
WinNtSetCommTimeouts                SetCommTimeouts;

WinNtExitProcess                    ExitProcess;
WinNtSprintf                        SPrintf;
WinNtGetDesktopWindow               GetDesktopWindow;
WinNtGetForegroundWindow            GetForegroundWindow;
WinNtCreateWindowEx                 CreateWindowEx;
WinNtShowWindow                     ShowWindow;
WinNtUpdateWindow                   UpdateWindow;
WinNtDestroyWindow                  DestroyWindow;
WinNtInvalidateRect                 InvalidateRect;
WinNtGetWindowDC                    GetWindowDC;
WinNtGetClientRect                  GetClientRect;
WinNtAdjustWindowRect               AdjustWindowRect;
WinNtSetDIBitsToDevice              SetDIBitsToDevice;
WinNtBitBlt                         BitBlt;
WinNtGetDC                          GetDC;
WinNtReleaseDC                      ReleaseDC;
WinNtRegisterClassEx                RegisterClassEx;
WinNtUnregisterClass                UnregisterClass;

WinNtBeginPaint                     BeginPaint;
WinNtEndPaint                       EndPaint;
WinNtPostQuitMessage                PostQuitMessage;
WinNtDefWindowProc                  DefWindowProc;
WinNtLoadIcon                       LoadIcon;
WinNtLoadCursor                     LoadCursor;
WinNtGetStockObject                 GetStockObject;
WinNtSetViewportOrgEx               SetViewportOrgEx;
WinNtSetWindowOrgEx                 SetWindowOrgEx;
WinNtMoveWindow                     MoveWindow;
WinNtGetWindowRect                  GetWindowRect;
WinNtGetMessage                     GetMessage;
WinNtTranslateMessage               TranslateMessage;
WinNtDispatchMessage                DispatchMessage;
WinNtGetProcessHeap                 GetProcessHeap;
WinNtHeapAlloc                      HeapAlloc;
WinNtHeapFree                       HeapFree;
} EFI_WIN_NT_THUNK_PROTOCOL;
```

Figure 16.4 Thunk Protocol that Associates Some Firmware Names with Operating System APIs

- Construct an EFI hardware API handler that will be specific to the emulation platform.

  In Figure 16.5, the EFI_SERIAL_IO_PROTOCOL interface is being seeded with a variety of information associated with platform specific function data. In this case, these platform-specific functions are tuned to the emulation environment.

```
SerialIo.Revision      = SERIAL_IO_INTERFACE_REVISION;
SerialIo.Reset         = WinNtSerialIoReset;
SerialIo.SetAttributes = WinNtSerialIoSetAttributes;
SerialIo.SetControl    = WinNtSerialIoSetControl;
SerialIo.GetControl    = WinNtSerialIoGetControl;
SerialIo.Write         = WinNtSerialIoWrite;
SerialIo.Read          = WinNtSerialIoRead;
SerialIo.Mode          = SerialIoMode;
```

Figure 16.5 Establishing an EFI API to Call Platform Specific Operations

- Platform specific functions (such as emulation platform) that are handling the calls to EFI interfaces and in turn will call the established WinNtThunk APIs that will end up making operating specific API calls.

  In Figure 16.6 are illustrated several calls that could occur from within an API handler to accomplish several tasks.

```
//
// Example of reading from a file
//
Result = WinNtThunk->ReadFile (
                     NtHandle,
                     Buffer,
                     (DWORD)*BufferSize,
                     &BytesRead,
                     NULL
                     );


//
// Example of resetting a serial device
//
WinNtThunk->PurgeComm (
              NtHandle,
              PURGE_TXCLEAR | PURGE_RXCLEAR
              );

//
// Example of getting local time components
//
```

```
WinNtThunk->GetLocalTime (&SystemTime);
WinNtThunk->GetTimeZoneInformation (&TimeZone);
```

Figure 16.6 Example Calls to the WinNtThunk Protocol

In summary, Figure 16.7 shows the software logic contained within the operating system, firmware emulation component, and their associated interaction logic. It should be noted that this logical software flow has three primary components:

- Firmware component under development
- Basic firmware codebase
- Firmware-to-Operating System thunk code

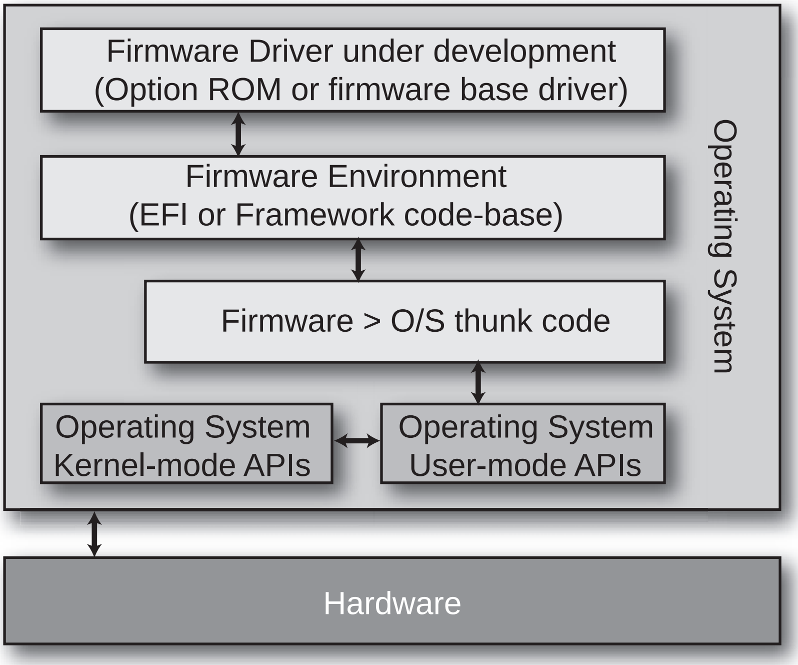

Firmware Driver under development
(Option ROM or firmware base driver)
Firmware Environment
(EFI or Framework code-base)
Firmware > O/S thunk code
Operating System
Kernel-mode APIs
Operating System
User-mode APIs
Operating System
Hardware

Figure 16.7 Firmware Emulation Software Logic Flow

Hardware Passthrough

As is evident through the previous examples, the underlying firmware can enable calling to several operating system APIs. However, since the firmware emulation environment is essentially an operating system application, certain functions are not going to be available. This is true since most operating systems have the concept of separating a user space from a more privileged kernel space to prevent applications from inadvertently crashing the entire operating system. Using this type of separation allows for the operating system to detect an error, and simply kill the user session without perturbing the remaining portions of the operating system.

It is possible to introduce several extensions to what is currently defined in the sample implementations that enable even further capabilities. An operating system kernel driver could be constructed to facilitate access to even more functions than would otherwise be available. This of course circumvents some of the inherent safety of the operating system and can introduce inadvertent crashes when care is not taken. By constructing a kernel driver that can reserve certain hardware resources and is able to advertise an interface that the emulation environment can call, the emulation environment can allow for an enhanced penetration into the hardware.

Figure 16.8 shows the logic flow associated with the various components and how they interact.

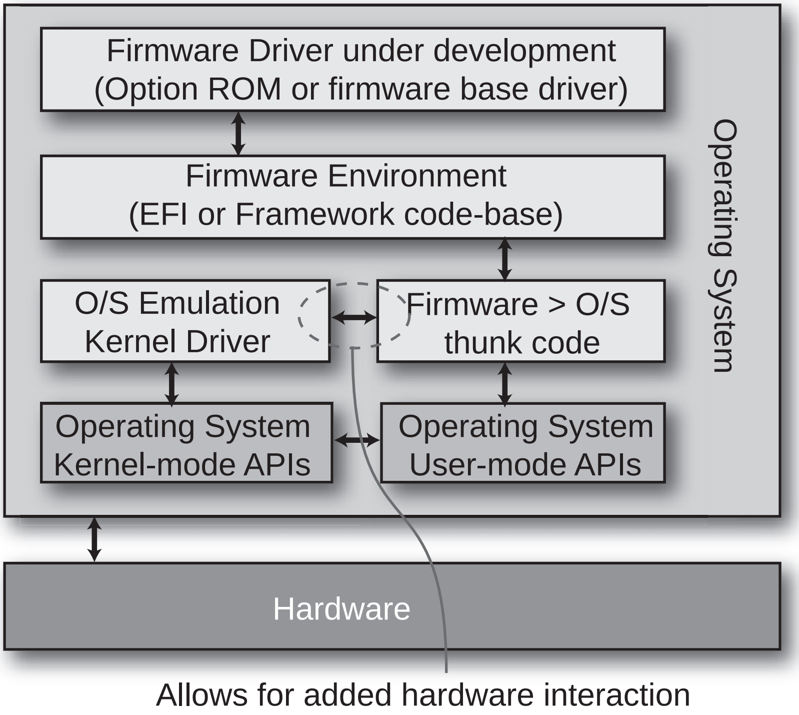


Figure 16.8 Software Flow for Hardware Enhanced Firmware Emulation

Chapter 17

Compatibility Support Module

The old order changeth, yielding place to new

—Alfred Tennyson

The Framework defines a new firmware model and achieves many advantages ranging from modularity, ease of firmware development even by multiple organizations, to reduced time to market for a product. However, the Framework does not define the runtime interfaces used by traditional operating systems. Thus, the Framework faces the immense challenge of ensuring the functionality of an enormous amount of legacy software.

The designers agree that all existing software must work flawlessly under the Framework. The legacy software requires traditional BIOS runtime services to function. The efforts of making legacy BIOS runtime services available under the Framework result in the birth of the Compatibility Support Module (hereafter referred to as CSM).

The CSM translates the information generated under the EFI environment into the information required by the legacy environment and makes the legacy BIOS services available for booting to the operating system and for use in runtime. The CSM also facilitates EFI to use the traditional Option ROMs. Thus, the CSM empowers a system with the ability to support EFI, legacy BIOS, or both under the Framework.

CSM: A Bridge between Innovation and Tradition

The computer industry usually experiences a transition period between the introduction and adoption of a new technology. The Framework also expects a natural transition phase, brief or prolonged. The CSM eases this transition phase by acting as a bridge between EFI and legacy BIOS interfaces. The CSM allows EFI to use legacy BIOS interfaces to boot to a traditional operating system such as DOS, Windows 2000, Windows XP, and so on. The CSM also allows EFI to boot to an EFI-aware operating system from a device controlled by a traditional Option ROM. With the advent of CSM, a system vendor can participate seamlessly in the innovation of new technology EFI without sacrificing any of its traditional functionalities. Therefore, directly and indirectly, CSM plays a vital role in ensuring the existing functionality while the industry pursues the transition from legacy BIOS to EFI.

The CSM uses the Framework for platform and hardware initialization, enumeration, and boot path selected by EFI. Additionally, the CSM provides the EfiCompatibility functionality to EFI in supporting traditional operating systems and traditional Option ROMs. The EfiCompatibility module along with the IBV supplied Compatibility16BIOS and Compatibility16SMM modules constitute the CSM. During the transition from legacy BIOS to EFI, the traditional Option ROM support may be required longer than traditional operating system support. The CSM is modular and hence can be removed if not required. Figure 17.1 represents a block diagram of how a legacy system operates using the CSM under the Framework.

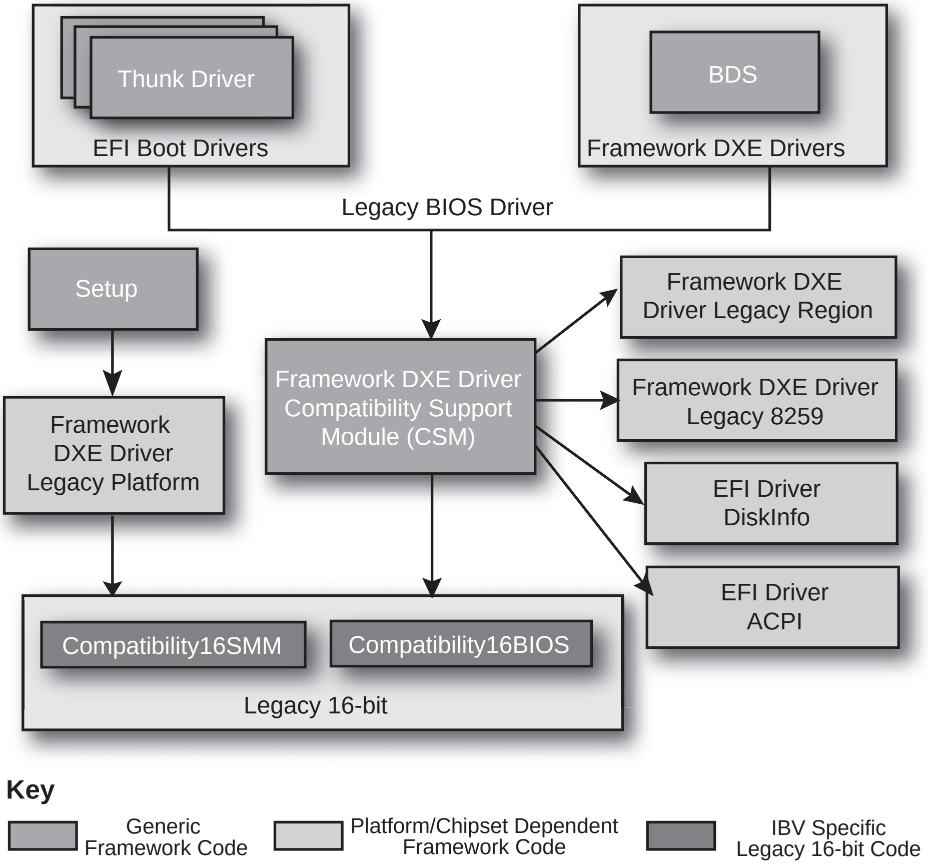

Thunk Driver
EFI Boot Drivers
BDS
Framework DXE Drivers
Legacy BIOS Driver
Setup
Framework DXE Driver Legacy Region
Framework DXE Driver Compatibility Support Module (CSM)
Framework DXE Driver Legacy 8259
Framework DXE Driver Legacy Platform
EFI Driver DiskInfo
EFI Driver ACPI
Compatibility16SMM
Compatibility16BIOS
Legacy 16-bit
Key
Generic Framework Code
Platform/Chipset Dependent Framework Code
IBV Specific Legacy 16-bit Code

Figure 17.1 Legacy System under the Framework

During the execution in EFI environment, only minimal system configuration takes place until the Boot Device Selection (BDS) phase. Neither display nor dispatching of other Option ROMs is necessary until the BDS phase. Setup is entered, if needed, from the BDS phase. The BDS decides whether or not to invoke CSM based primarily on three types of information:

- Target operating system
- Chosen boot device
- Device initialization policy

A legacy (non–EFI-aware) operating system requires traditional BIOS services. The CSM needs to be invoked if the system is booting to a legacy operating system. Additionally, any Option ROM involved in the process of booting to a legacy operating system must be a traditional Option ROM and the CSM needs to be invoked to support the traditional Option ROM.

If a traditional Option ROM controls the chosen boot device, the CSM needs to be invoked to support the boot device irrespective of whether the target operating system is legacy or EFI-aware.

If the policy is to initialize all devices and if a non-boot device only has a traditional Option ROM associated with it, the CSM needs to be invoked to support the non-boot device.

CSM Architecture

The CSM empowers EFI with the ability of loading a traditional operating system and using a traditional Option ROM. The CSM consists of five main components:

- EfiCompatibility (CSM32)
- Compatibility16BIOS (CSM16)
- Compatibility16SMM
- Thunk and ReverseThunk
- Legacy Option ROM

The EfiCompatibility, Compatibility16BIOS, Compatibility16SMM, and Thunk and ReverseThunk are integral parts of the EFI firmware image. The legacy Option ROM may or may not be a part of EFI firmware image.

The CSM operates primarily in two different environments, namely, booting a legacy operating system and loading an EFI-aware operating system from a device controlled by a legacy Option ROM. Booting a legacy operating system is the traditional environment. Industry trends may require the legacy Option ROM support even after the legacy operating system has been replaced by EFI-aware operating system. The functionality to load an EFI-aware operating system using a legacy Option ROM is a subset of the functionality to boot a legacy operating system. The Framework architecture exhibits this split functionality and allows removal of functionality, if necessary, in the future.

EfiCompatibility (CSM32)

EfiCompatibility is a 32-bit module and interfaces with the EFI environment to invoke Compatibility16 functions and legacy BIOS services. The Framework provides this module. This module consists mainly of four different types of EFI drivers:

- Platform- and hardware-independent drivers that form the CSM foundation
- Legacy component drivers that are also platform-independent; for example, a driver for a legacy 8259 Programmable Interrupt Controller
- Chipset-specific drivers such as the driver for ICH7
- Platform-specific drivers such as the driver for PCI IRQ routing

Compatibility16BIOS (CSM16)

Compatibility16BIOS is a 16-bit real-mode module providing legacy BIOS runtime services. This module mainly consists of the traditional interrupt service handlers available in a legacy BIOS such as INT13, INT19, INT15, and so on. This module also contains the Compatibility16 functions required for communication between EfiCompatibility and Compatibility16BIOS. The IBV provides this module.

Compatibility16SMM

Compatibility16SMM is an optional module to provide the traditional SMM functionalities that are not covered by EFI SMM. The IBV or OEM provides this module.

Thunk and ReverseThunk

Thunk and ReverseThunk modules provide the capability to transition from a 32-bit to a 16-bit environment and vice-versa. The EFICompatibility module uses Thunk to switch from EFI, a 32-bit mode to legacy, a 16-bit mode. Compatibility16BIOS may use ReverseThunk to switch from legacy 16-bit mode to EFI 32-bit mode. The Framework provides these modules.

Legacy Option ROM

The legacy Option ROM, though not internal to CSM, is still a vital component of a system, and works in conjunction with the CSM via the legacy Option ROM interface. The Option ROM vendors provide legacy Option ROMs that may or may not be integrated into the EFI firmware image.

EfiCompatibility (CSM32)

EfiCompatibility consists of the Legacy BIOS Driver with various protocols:

- Legacy BIOS Protocol
- Legacy BIOS Platform Protocol
- Legacy Region Protocol
- Legacy 8259 Protocol
- Legacy Interrupt Protocol

Legacy BIOS Protocol is the primary protocol of the CSM and is hardware-independent. Legacy BIOS Platform Protocol is platform-specific and differentiates a platform from other platforms using the same chipset. Legacy Region Protocol is chipset-specific and controls the read-write capability of the memory region 0xC0000–0xFFFFF. Legacy Region Protocol may also optionally manage the hardware to prevent a write to the region from propagating to any aliased memory region. Legacy 8259 Protocol is independent of platform and chipset. Legacy 8259 Protocol controls the Programmable Interrupt Controller 8259 in 32-bit protected mode as well as in 16-bit real mode and manages the interrupt masks and edge/level mode programming. Legacy Interrupt Protocol is chipset-specific and controls the assignment of IRQs to PCI devices.

EfiCompatibility also contains drivers to emulate various traditional software interrupts such as UGA emulation of INT10, Keyboard emulation of INT16, and block I/O emulation.

The driver for UGA emulation of INT10 is needed when traditional Option ROMs are invoked. The UGA controller is in VGA emulation mode and a VGA Option ROM is invoked. This driver translates EFI console-out data into their VGA equivalent. This driver assumes that all INT10 functions are supported, that the Option ROM may access directly the VGA registers and video memory buffers, and that UGA hardware supports a VGA mode that can be switched between UGA and VGA modes as and when required.

The driver for keyboard emulation of INT16 is needed because the Compatibility16BIOS does not take over USB keyboard emulation until the final stages of a legacy boot path. During this time, this driver converts the INT16 requests into EFI requests, receives the data, and converts back into INT16 format.

The block I/O emulation driver is needed for EFI to access a BIOS-Aware IPL Device (BAID) such as a floppy or hard disk. This driver translates EFI block I/O requests into the equivalent legacy BIOS INT13 requests and returns the result to EFI. A separate block I/O emulation driver may be needed to support EFI access to devices controlled by legacy Option ROM; for example, a SCSI drive controlled by SCSI Option ROM.

Legacy Bios Driver

The Legacy BIOS Driver is the main module in the CSM while others are support modules. The main functions of Legacy BIOS Driver are as follows:

- Initialize itself and load the Compatibility16BIOS
- Determine the boot device
- Load other EfiCompatibility drivers
- Load Thunk and ReverseThunk
- Prepare the interface with Compatibility16 functions
- Find, load, and invoke off-board and on-board traditional Option ROMs
- Load an operating system

The Legacy BIOS Driver analyzes the data hub and/or invokes EFI APIs to collect the information required by Compatibility16BIOS and converts the information into different Compatibility16 data structures. Compatibility16BIOS uses these data structures to initialize the legacy BIOS data area in 0x400-0x500 memory region (BDA), Extended BIOS Data Area (EBDA), and CMOS. The Legacy BIOS Protocol also uses these data structures to reprogram legacy devices to traditional resources.

EFI performs enumeration and configuration of low-level device hardware. EFI does not assign IRQ to legacy devices such as serial ports and parallel ports, but Compatibility16BIOS expects these legacy devices to be configured with the proper IRQ. The Legacy BIOS Driver reconfigures any legacy device that is configured by EFI into a valid legacy configuration.

A particular implementation may choose between a light and full version of the Legacy BIOS Driver. The light version can be used in environments where the target operating system is always an EFI-aware operating system that may have associated legacy Option ROMs. A full version is needed in environments where the target operating system can be a legacy operating system.

Legacy BIOS Protocol

The Legacy BIOS Protocol with the initialization of Legacy BIOS Driver forms the CSM foundation. The Legacy BIOS Protocol consists of the following functions required for legacy support.

BootUnconventionalDevice() provides the support to boot from an unconventional device in legacy environment such as PARTIES.

CheckPciRom()checks whether a device has an associated legacy Option ROM and whether a legacy Option ROM exists for a device without an EFI Option ROM. It also determines the valid legacy operating system boot devices.

CopyLegacyRegion() is used by EFI to copy data to the area specified by GetLegacyRegion().

FarCall86() allows the 32-bit protected mode to perform a FAR CALL to a 16-bit real mode. This function uses Thunk to switch to 16-bit real-mode environment, loads the IA-32 processor registers with the data area supplied as input to this function, and invokes the FAR CALL. On return from the FAR CALL, this function updates the data area with the value from the IA-32 processor registers, switches back to 32-bit protected mode, and returns to the caller.

GetBbsInfo() allows external drivers to access BBS data structures in the EfiCompatibility module. Setup mainly uses this function.

GetLegacyRegion() allows EFI to allocate an area in 0xE0000–0xFFFFF memory region.

Int86() allows the execution of a 16-bit software interrupt like INT16 from 32-bit protected mode environment. This function uses Thunk to switch to 16-bit real-mode environment, loads the IA-32 processor registers from the data area supplied to this function, and invokes the requested software interrupt. On return from the software interrupt handler, this function updates the data area with the value from the IA-32 processor registers, switches back to 32-bit protected mode, and returns to the caller.

InstallPciRom() installs a legacy Option ROM in the 0xC0000–0xFFFFF memory region.

LegacyBoot() initiates booting to a legacy operating system. This function implements most of the CSM functionalities. It may turn off the EFI functionality because of the commitment of booting to a legacy operating system. This function eventually calls the Compatibility16BIOS module to invoke traditional INT19. If INT19 fails to boot to the operating system from the selected boot devices, the Compatibility16BIOS module may return control to this function. In this case, this function may return to the caller only if the EFI environment is preserved or can be restored for interleave boot between EFI and legacy.

PrepareToBootEfi() allows external agents to prepare the environment to boot to an EFI-aware operating system. It performs a subset of the actions of the LegacyBoot() function including the drive number assignment to legacy devices.

ShadowAllLegacyOproms() allows external agents to force the loading of legacy Option ROMs. This function disconnects all EFI drivers, so EFI drivers must be reconnected after this function is invoked, if needed.

UpdateKeyboardLedStatus()is used to synchronize the legacy BDA with the EFI programmed information of keyboard LED status. This function does not program the keyboard and invokes the Compatibility16 function with keyboard LED status to allow any proprietary information to be maintained.

Legacy BIOS Platform Protocol

The Legacy BIOS Platform Protocol provides the ability to customize the CSM for both platform configuration and OEM requirements. The Legacy BIOS Platform Protocol also contains various functions to support CSM customizations.

GetPlatformHandle() finds all handles for the supplied entity and returns a list of handles of the specified type sorted by priority. The different types of handles that can be requested include VGA devices, IDE controllers, ISA Bus controllers, and USB devices.

GetPlatformInfo() returns platform-specific binary objects or data that may include MP table, OEM-specific 16-bit or 32-bit data image, TPM binary image, and so on.

GetRoutingTable() returns the platform-specific PCI IRQ routing information for the $PIR table.

PlatformHooks() performs any platform-specific action such as any processing before scanning Option ROMs, shadowing legacy Option ROM not associated with a physical device (PXE base code and BIS), and any processing after Option ROM initialization.

PrepareToBoot() attempts to boot to a legacy operating system.

SmmInit() checks whether any Compatibility16SMM module is present in EFI firmware volumes. If a Compatibility16SMM module is found, this function initializes and registers the Compatibility16SMM module with the EFI SMM driver. More than one or no Compatibility16SMM module may be present in an EFI firmware image.

TranslatePirq() translates the PCI INT line information into the corresponding PIRQ register of the chipset and returns the possible assignable IRQ for the specified PCI device.

Legacy Region Protocol

The Legacy Region Protocol is chipset-specific and controls the read-write capability for the memory region 0xC0000–0xFFFFF.

Decode() initializes the chipset to enable or disable decoding of the requested region.

Lock() sets the requested region to be read-only (write-protected).

BootLock() is called just before booting to a legacy operating system. This function write-protects the requested region and prevents writing to the requested region from propagating to any aliased address.

Unlock() sets the requested region to be read-write–enabled. This function also prevents write to the requested region from propagating to any aliased address.

Legacy 8259 Protocol

The Legacy 8259 Protocol is independent of chipset. It controls the programming of the 8259 PIC in both the EFI 32-bit protected mode environment and legacy 16-bit real mode environment and consists of the following functions.

SetVectorBase() initializes the vector base of the master and slave 8259 PIC. The vector base is set to 0x1A0 (INT68) for the master PIC and 0x1C0 (INT70) for the slave PIC in EFI environment. In legacy 16-bit environment, the vector base is set to 0x20 (INT08) for the master PIC and 0x1C0 (INT70) for slave PIC. A different vector base for the master PIC in EFI environment helps prevent overlaying of interrupts and processor exceptions.

GetMask() returns the current setting of master/slave interrupt mask and the edge/level programming information in EFI and legacy environments.

SetMask() sets the master and slave interrupt mask and the edge/level programming information for EFI and legacy environments.

SetMode() sets the current mask for the master and slave PIC and the edge/level programming for the requested 16-bit or 32-bit mode.

GetVector() translates the given IRQ into an INT. For example, it translates IRQ0 to INT68 in an EFI environment.

EnableIrq() enables an IRQ by unmasking the interrupt in an EFI environment and is used by non-CSM drivers.

DisableIrq() disables an IRQ by masking the interrupt in an EFI environment and is used by non-CSM drivers.

GetInterruptLine() reads the configuration space of the specified PCI device and returns the IRQ assigned to the PCI device.

EndOfInterrupt() issues an End Of Interrupt (EOI) command to PIC.

Legacy Interrupt Protocol

The Legacy Interrupt Protocol is chipset-specific and controls the programming of PCI interrupts.

GetNumberPirqs() returns the number of PIRQs supported by the chipset.

GetLocation() returns the PCI location of the device supporting this protocol.

ReadPirq() reads the specified PIRQ register and returns the current content of the PIRQ register.

WritePirq() programs the specified PIRQ with the given data.

Compatibility16BIOS (CSM16)

The Compatibility16BIOS module can be thought of as the runtime image of the legacy BIOS without the POST and Setup. The design goal is to make the Compatibility16BIOS module universal for all platforms. That is, the same Compatibility16BIOS should work in desktop, server, and mobile platforms. Thus, the Compatibility16BIOS is independent of chipset and platform. The Compatibility16BIOS does not configure any low-level hardware. However, it may control the legacy hardware through traditional hardware interfaces like serial ports, parallel ports, keyboard, and PS2 mouse. The chipset programming is done by EFI and/or the EfiCompatibility module. The chipset- and platform-independent design maximizes reusability and minimizes maintenance of the Compatibility16BIOS module.

The Compatibility16BIOS contains all runtime services and software and hardware interrupt service routines required in legacy BIOS runtime.

This module must also support the legacy-compatible data area in BDA, EBDA, and fixed entry points of different interrupt service handlers in the F000 segment. Additionally, this module contains the Compatibility16 functions used for communication between EfiCompatibility and Compatibility16BIOS. Table 17.1 describes the legacy interrupt services provided by the Compatibility16BIOS module.

Table 17.1 Legacy Interrupt Services Provided by Compatibility16BIOS

Interrupt	Description
0x02	NMI (Non Maskable Interrupt)
0x05	Print Screen Service
0x08	System Timer Interrupt (IRQ0)
0x09	Keyboard Interrupt (IRQ1)
0x10	Video Services
0x11	Equipment Determination Service
0x12	Base Memory Size Determination Service
0x13	Hard Disk and Floppy Diskette Services including all physical and emulated devices accessible through INT13 such as ATA, ATAPI, ARMD, and Magneto Optical devices.
0x14	Serial Communication Services
0x15	System Services
0x16	Keyboard Services
0x17	Parallel Printer Services
0x18	ROM BASIC or other Boot Loader Services
0x19	Bootstrap Loader
0x1A	Time-of-Day Services including PCI BIOS and TPM extensions.
0x1B	<Ctrl><Break> Service
0x1C	User Timer Tick Service
0x1D	Video Parameters
0x1E	Floppy Diskette Drive Parameters
0x1F	Video Graphics Characters
0x40	Floppy Diskette Services
0x41	Hard Disk C: Parameters
0x46	Hard Disk D: Parameters
0x70	Real-Time Clock – Periodic and Alarm Interrupt (IRQ8)
0x74	PS/2 Mouse Interrupt (IRQ12)
0x75	Math Coprocessor Interrupt (IRQ13)
0x76	Hard Disk Interrupt (IRQ14)
0x77	Hard Disk Interrupt (IRQ15)

Communication between CSM32 and CSM16

The communication between EfiCompatibility (CSM32) and Compatibility16BIOS (CSM16) is achieved through the three mechanisms of Compatibility16 Table, Compatibility16 functions, and Compatibility16 data structures. Figure 17.2 represents the communication mechanism between CSM32 and CSM16.

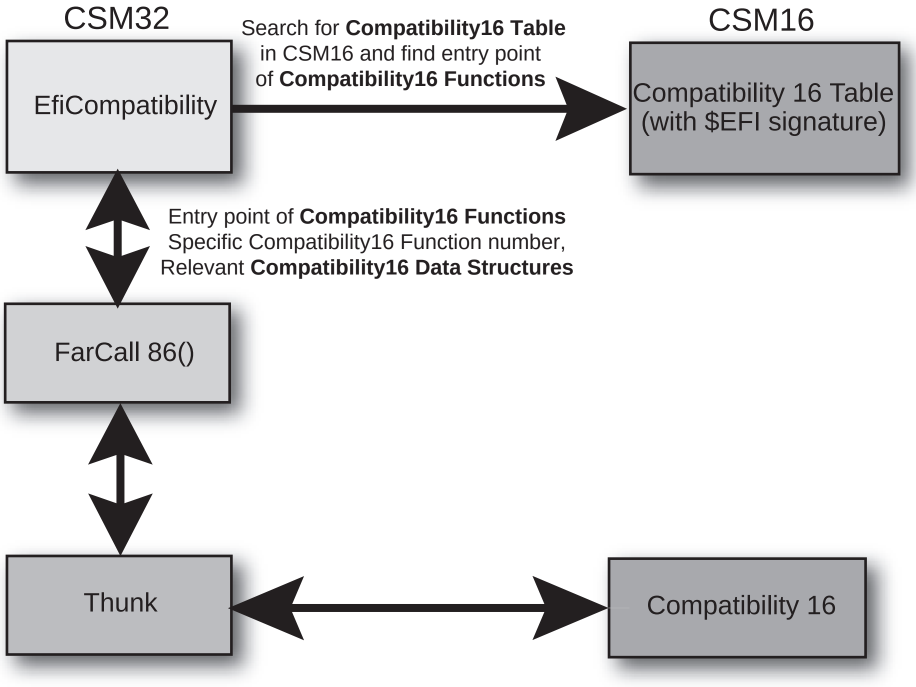

CSM32
Search for **Compatibility16 Table** in CSM16 and find entry point of **Compatibility16 Functions**
CSM16
EfiCompatibility
Compatibility 16 Table (with $EFI signature)
Entry point of **Compatibility16 Functions**
Specific Compatibility16 Function number,
Relevant **Compatibility16 Data Structures**
FarCall 86()
Thunk
Compatibility 16

Figure 17.2 Communication between CSM32 and CSM16

Compatibility16 Table

The Compatibility16 Table is by far the most important part of the Compatibility16BIOS module and resides on a 16-byte boundary in the memory region 0xE0000–0xFFFFF. This table contains some static information fixed during the build process of Compatibility16BIOS module. It also contains dynamic information filled by both EfiCompatibility and Compatibility16BIOS during CSM operation. The Compatibility16BIOS generates a default table at build time. The first field of this table contains the "$EFI" signature and as such the Compatibility16 Table is also known as the $EFI Table. The most important fields are the Compatibility16CallSegment and Compatibility16CallOffset that provide the entry point of Compatiblity16 functions within Compatibility16BIOS module. EfiCompatibility uses this entry point to invoke Compatibility16 functions as a FAR CALL via Thunk using the FarCall86() function of Legacy BIOS Protocol. The Compatibility16 Table in a particular CSM implementation can be of the EFI_COMPATIBILITY16_TABLE format. The CSM design allows the addition of new fields in the Compatibility16 Table depending on future requirements, without deleting any of the existing fields. Table 17.2 describes the different fields in EFI_COMPATIBILITY16_TABLE.

Table 17.2 Different Fields in Compatibility16 Table

Field	Offset	Description
Signature	0x00	The signature "$EFI" indicates the start of the EfiCompatibility table. Byte 0 is "I," byte 1 is "F," byte 2 is "E," and byte 3 is "$".
TableChecksum	0x04	The checksum is calculated such that byte addition of all bytes in this table equals zero. The length of this table is given by the *TableLength* field.
TableLength	0x05	The length of this table in bytes.
EfiMajorRevision	0x06	EFI major revision number.
EfiMinorRevision	0x07	EFI minor revision number.
TableMajorRevision	0x08	Major revision number of this table.
TableMinorRevision	0x09	Minor revision number of this table.
Reserved	0x0A	Reserved for future use.
Compatibility16CallSegment	0x0C	The segment of the entry point of Compatibility16 functions within the Compatibility16BIOS.
Compatibility16CallOffset	0x0E	The offset of the entry point of Compatibility16 functions within the Compatibility16BIOS.
PnPInstallationCheckSegment	0x10	The segment of PnP installation check structure within the Compatibility16BIOS.
PnPInstallationCheckOffset	0x12	The offset of PnP installation check structure within the Compatibility16BIOS.
EfiSystemTable	0x14	The 32-bit physical address of EFI system resources table.
OemIdStringPointer	0x18	The 32-bit physical address of an OEM provided identifier string. This string is null terminated.

Field	Offset	Description
AcpiRsdPtrPointer	0x1C	The 32-bit physical address where ACPI RSD PTR is stored within the legacy BIOS. The remainder of the ACPI tables are located at their EFI addresses.
OemRevision	0x20	Can be used for OEM-specific module.
E820Pointer	0x22	The 32-bit physical address where INT15 E820 data is stored within the legacy BIOS. EfiCompatibility fills this field and copies the data to the indicated area.
E820Length	0x26	The length of the E820 data is filled by EfiCompatibility.
IrqRoutingTablePointer	0x2A	The 32-bit physical address of the $PIR table within the legacy BIOS. EfiCompatibility fills this field value and copies the data to the indicated area.
IrqRoutingTableLength	0x2E	The length of the $PIR table is filled by EfiCompatibility.
MpTablePtr	0x32	The 32-bit physical address of the MP table within the legacy BIOS. EfiCompatibility fills this field and copy the data to the indicated area.
MpTableLength	0x36	The length of the MP table is filled in by EfiCompatibility.
OemIntSegment	0x3A	The segment of OEM-specific INT handler or table.
OemIntOffset	0x3C	The offset of OEM-specific INT handler or table.
Oem32Segment	0x3E	The segment of OEM-specific 32-bit handler or table.
Oem32Offset	0x40	The offset of OEM-specific 32-bit handler or table.
Oem16Segment	0x42	The segment of OEM-specific 16-bit handler or table.
Oem16Offset	0x44	The offset of OEM-specific 16-bit handler or table.
TpmSegment	0x46	The segment of TPM binary module passed to Compatibility16BIOS.
TpmOffset	0x48	The offset of TPM binary module passed to Compatibility16BIOS.
IbvPointer	0x4A	Independent BIOS Vendor identifier string.
PciExpressBase	0x4E	The base value of the start of the PCI Express memory-mapped configuration registers. It is NULL for all systems not supporting PCI Express. - EfiCompatibility fills this field prior to invoking the Compatibility16InitializeYourself() function.
LastPciBus	0x52	The value of maximum assigned PCI bus number.

Compatibility16 Functions

The EfiCompatibility module uses the Compatibility16 functions to communicate with the Compatbility16BIOS. The Compatibility16 functions are implemented in the Compatibility16BIOS module in addition to the legacy BIOS runtime services. EfiCompatibility uses these functions to generate the traditional runtime environment necessary to run the legacy software. Additionally, IBV may use these functions to include any proprietary information using its own methodology. The Compatibility16CallSegment and Compatibility16CallOffset fields in the Compatibility16 Table provide the entry point of the Compatibility16 functions. EfiCompatibility uses this entry point to invoke the Compatibility16 functions as a FAR CALL via Thunk using the FarCall86() function of Legacy BIOS Protocol. The CSM design allows addition of new Compatibility16 functions depending on future requirements, without deleting any of the existing function. A particular CSM implementation may support the following Compatibility16 functions:

```
typedef          enum     {
  Compatibility16InitializeYourself         0x0000,
  Compatibility16UpdateBbs                  0x0001,
  Compatibility16PrepareToBoot              0x0002,
  Compatibility16Boot                       0x0003,
  Compatibility16RetrieveLastBootDevice     0x0004,
  Compatibility16DispatchOprom              0x0005,
  Compatibility16GetTableAddress            0x0006,
  Compatibility16SetKeyboardLeds            0x0007,
  Compatibility16InstallPciHandler          0x0008
} EFI_COMPATIBILITY16_FUNCTIONS;
```

Compatibility16InitializeYourself()

This is the first function invoked by EfiCompatibility. The EfiCompatibility module supplies the necessary information as input parameters to this function. The Compatibility16InitializeYourself() function performs the initialization required for 16-bit legacy environment.

The different actions that this function may perform include, but are not limited to, installing Post Memory Manager (PMM), initializing the data area for legacy devices (such as Floppy, ATA, ATAPI), initializing BBS data area, installing the interrupt handlers, and installing memory manager for the memory region 0xE0000–0xFFFFF.

Compatibility16UpdateBbs()

This function allows the Compatibility16BIOS module to update the BBS data structures, if needed. EfiCompatibility supplies the necessary information as input parameters to this function. The IBV uses this function mainly to install USB mass storage devices by adding the corresponding entries in the BBS table, thereby emulating the USB mass storage devices in a BBS compliant fashion.

The Compatibility16UpdateBbs() function can also be used to emulate a BBS-noncompliant Option ROM as a BBS-compliant Option ROM. For example, if a non-BBS Option ROM traps INT19 vector during the execution of Option ROM initialization routine, this function may emulate this INT19 trapping by adding an extra entry in the BBS table and putting the trapped INT19 address as a BEV and restoring the INT19 vector to the original INT19 handler of Compatibility16BIOS.

Compatibility16PrepareToBoot()

This function allows the Compatibility16BIOS module to prepare the system environment prior to booting to a legacy operating system. The EfiCompatibility module supplies the necessary information as input parameters to this function. The different actions that this function may perform include, but are not limited to, updating INT15 E820 data table, initializing legacy CMOS locations for base memory size and extended memory size, preparing ACPI RSD PTR header, initializing SMBIOS installation structure, initializing data area for floppy, serial port, parallel port, keyboard, and PS/2 mouse, initializing internal data tables according to priority of different boot devices in the BBS table, uninstalling PMM and memory manager for the 0xE0000–0xFFFF memory region.

Compatibility16Boot()

This is the last function invoked by EfiCompatibility while booting to a legacy operating system. The memory region 0xE0000–0xFFFFF is write-protected. The EfiCompatibility module supplies the necessary information as input parameters to this function. The Compatibility16Boot() function should ensure that the system is in 16-bit real-mode and issue INT19 for booting.

Compatibility16RetrieveLastBootDevice()

This function returns the last boot device priority number. The EfiCompatibility module supplies the necessary information as input parameters to this function. The Compatibility16RetrieveLastBootDevice() function allows EfiCompatibility to determine the last boot device in the multiple boot device environment.

Compatibility16DispatchOprom()

This function allows the Compatibility16BIOS module to invoke the specified Option ROM initialization routine. The EfiCompatibility module supplies the necessary information as input parameters to this function. This function permits IBV to maintain its own Option ROM handling methodology.

The Compatibility16DispatchOprom() function can also be used to emulate a BBS-noncompliant Option ROM as a BBS-compliant Option ROM. For example, if a non-BBS Option ROM installs mass storage devices by trapping INT13 vector, this function may emulate this INT13 trapping by adding the extra entry (entries) in the BBS table, putting a simulated BCV, storing the trapped INT13 information internally, and restoring the INT13 vector to the original INT13 handler of Compatibility16BIOS. The simulated BCV, when called, should install the mass storage device using the trapped INT13 vector.

Compatibility16GetTableAddress()

The EfiCompatibility module uses this function to generate different data tables in legacy BIOS region 0xE0000–0xFFFFF. The EfiCompatibility module first invokes the Compatibility16GetTableAddress() function to allocate a memory of the requested size in the specified region between 0xE0000 and 0xFFFFF. The EfiCompatibility module then copies the corresponding data to the allocated memory in the legacy BIOS region. The Compatibility16BIOS module should implement a memory manager for the region 0xE0000–0xFFFFF for this function.

Compatibility16SetKeyboardLeds()

The EfiCompatibility module uses this function to synchronize the keyboard LED status. This function may update the legacy BIOS data area with the supplied LED status.

Compatibility16InstallPciHandler()

The EfiCompatibility module uses this function to install an IRQ handler for the mass storage devices that do not have any Option ROM associated with them. Examples of this type of devices include PATA and SATA devices.

Compatibility16 Data Structures

The EfiCompatibility module uses the Compatibility16 data structures as input/output parameters to different Compatibility16 functions and maintains/updates these data structures across different functional interfaces. These data structures are CSM-implementation–dependent and may vary from one CSM specification to the other (see the *CSM Specification* from Intel Corporation for more information). Some of the Compatibilty16 data structures used in a particular CSM implementation are:

- *EFI_TO_COMPATIBILITY16_INIT_TABLE*: The EfiCompatibility module uses this data structure as an input/output parameter to the Compatibility16InitializeYourself() function. This data structure contains information necessary for the Compatibility16BIOS to create its execution environment. The information in this data structure includes, but is not limited to, the low memory map and high memory map to be used by PMM, the location and size of Thunk and ReverseThunk module that need to be excluded from PMM allocation, and extended memory size used to initialize IBV-specific CMOS locations.
- *EFI_TO_COMPATIBILITY16_BOOT_TABLE:* The EfiCompatibility module uses this data structure as an input/output parameter to the Compatibility16UdateBbs() and Compatibility16PrepareToBoot() functions. This data structure contains information needed by Compatibility16 functions for a legacy boot environment. The information in this data structure includes, but is not limited to, the details of legacy devices such as a serial port, parallel port, floppy, keyboard, PS/2 mouse, and ATA/ATAPI, a BBS table containing the details of devices found, and unconventional device details such as PARTIES.
- *EFI_DISPATCH_OPROM_TABLE*: The EfiCompatibility module uses this data structure as an input/output parameter to the Compatibility16DispatchOprom() function. This data structure contains the necessary information needed by the Compatibility16 function to invoke the legacy Option ROM initialization routine and handle the information returned by the legacy Option ROM initialization routine.
- *EFI_LEGACY_INSTALL_PCI_HANDLER*: The EfiCompatibility module uses this data structure as an input parameter to the Compatibility16InstallPciHandler() function. This data structure contains the necessary information needed by the Compatibil-

ity16 function to install the interrupt handler for the PCI mass storage devices that do not have any associated Option ROM.

Compatibility16SMM

Compatibility16SMM is an optional module and implements the SMM functionalities that are not implemented in EFI. However, platform requirements, traditional features, or proprietary features may require this module. The Compatibility16SMM module is chipset-specific. This module allows IBV to maintain its traditional methodologies used in a legacy environment via SMI. Some examples include INT15 Function D042 support for processor microcode, USB legacy support for keyboard and mouse, and the support for USB mass storage devices.

Thunk and ReverseThunk

The Thunk module switches from a 32-bit protected mode environment into a 16-bit real mode environment. The ReverseThunk module switches from a 16-bit real mode environment to a 32-bit protected environment. Both Thunk and ReverseThunk ensure that the 8259 PIC is programmed properly for the concerned environment. Figure 17.3 shows how Thunk and ReverseThunk operate in 16-bit and 32-bit environments.

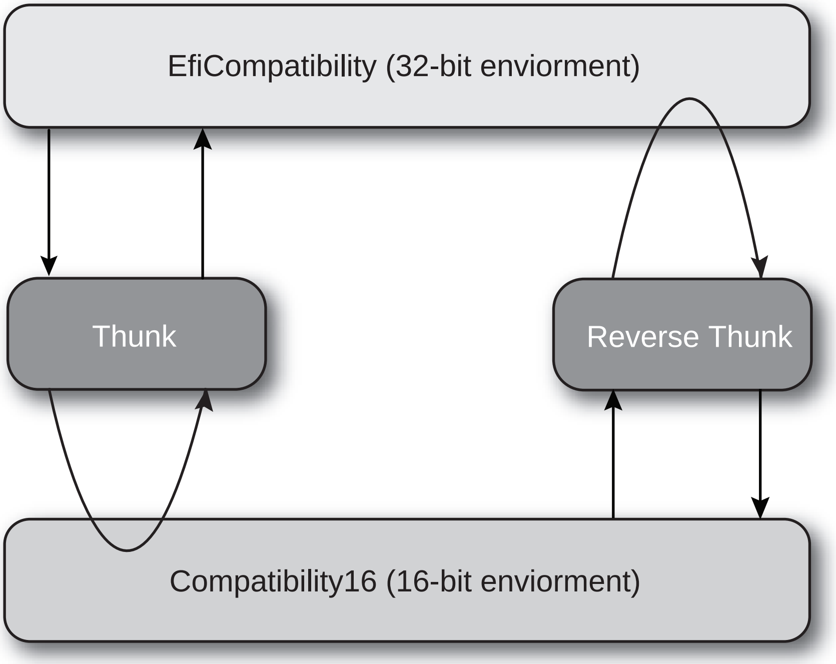

EfiCompatibility (32-bit enviorment)
Thunk
Reverse Thunk
Compatibility16 (16-bit enviorment)

Figure 17.3 Thunk and ReverseThunk in the Framework

The EFI environment (32-bit mode) operates in polled mode instead of interrupt-driven mode and supports only the system timer interrupt. Therefore changing the environment from a 32-bit mode to a 16-bit mode involves supporting additional hardware and software interrupts required by the legacy environment (16-bit mode). The EfiCompatibility module uses Thunk during the transition from EFI to Compatibility16BIOS or to Option ROM. The Thunk module performs the following actions:

- Handling any Interrupt Controller reprogramming.
- Loading of proper GDT and IDT tables.
- Changing to 16-bit mode.
- Initializing IA32 registers with supplied input data.
- Performing the required action FAR CALL or Software Interrupt.
- Setting the output data with the values from IA32 registers.
- Restoring the 32-bit interrupt environment.

- Returning to EFI.

The 16-bit FAR routine returns to Thunk by executing (or simulating the execution of) a RETF instruction. The 16-bit software interrupt handler returns to Thunk by executing (or simulating the execution of) an IRET instruction.

The ReverseThunk module is similar to Thunk except it is invoked by the 16-bit code during the transition from a 16-bit mode to a 32-bit mode. The Compatibility16BIOS module can use ReverseThunk if it needs to execute a 32-bit mode code.

Functional Visualization of CSM

After all necessary DXE drivers are executed and the DXE Foundation produces the EFI Boot Services and EFI Runtime Services, the DXE Dispatcher transfers the control to the BDS Architectural Protocol. The BDS Architectural Protocol establishes the console devices and then attempts booting to an operating system.

The BDS Architectural Protocol executes a variety of applications in the pre-boot environment. These applications may include setup, system configuration display, system diagnostics, OEM proprietary value-add applications, and the operating system boot loader.

Figure 17.4 shows an architectural view of the CSM under BDS. The BDS invokes the CSM if a traditional Option ROM is required, or a traditional boot option is found in the boot sequence, or the target operating system is legacy. The traditional Option ROM support is also required in an EFI environment when *no EFI driver exists,* that is, the device has no EFI driver but only a traditional Option ROM. In this case, the normal EFI binding of the device and driver fails. The BDS then binds the device with the traditional Option ROM via the EfiCompatibility module through the Legacy BIOS Protocol. The BDS then uses the Legacy BIOS Driver to dispatch and initialize the traditional Option ROM associated with the device.

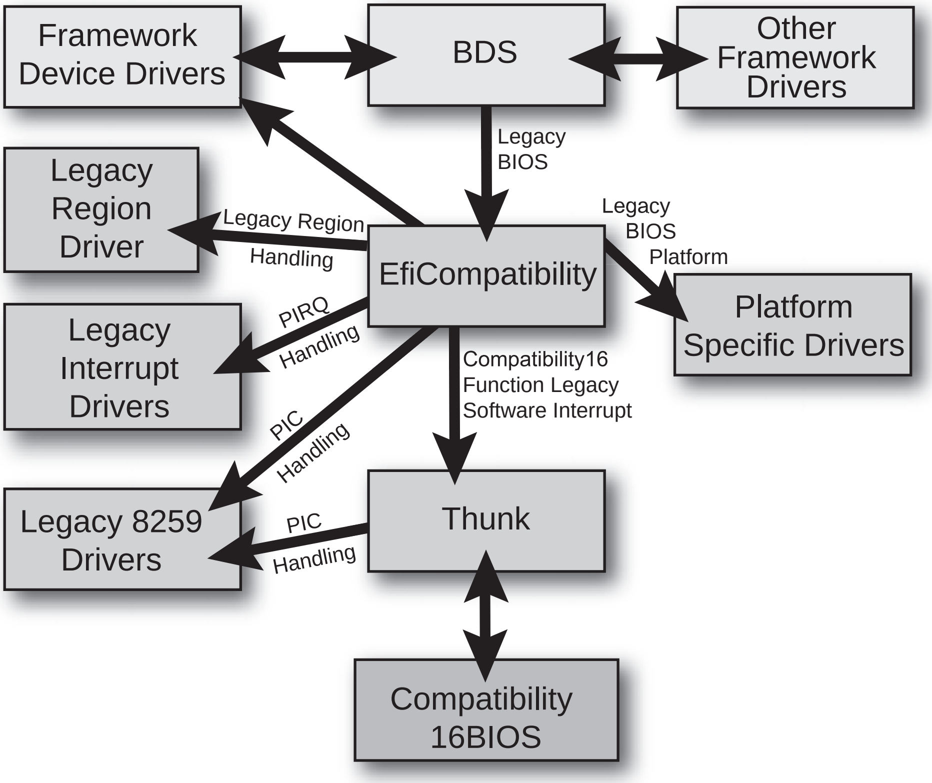


Figure 17.4 Architectural View of CSM under BDS

During a legacy boot path in an EFI environment, the traditional Option ROM support is also required for the devices with an associated Option ROM even when both EFI driver and the traditional Option ROM are available for the same device. The BDS needs to invoke the CSM in this case because the traditional Option ROM support is required to boot to a legacy operating system.

Figure 17.5 shows a functional view of the CSM under BDS. The BDS determines boot devices and performs various actions depending on the target operating system using Legacy BIOS Protocol. A device controlled by both an EFI and legacy Option ROM may produce different results. The EFI drivers generate a list of boot devices. The ShadowAllLegacyOproms() function disconnects all EFI devices. The GetBbsInfo() function gives the list of legacy boot devices that are identified. The CSM updates the internal BBS table information during the Option ROM initializations. The BDS provides the complete list of boot devices so that Setup can present the comprehensive list to the user for boot device selection. The user selects the boot device from the list of available boot devices. The legacy drive numbers are assigned to the available devices by the Compatibilit16BIOS module.

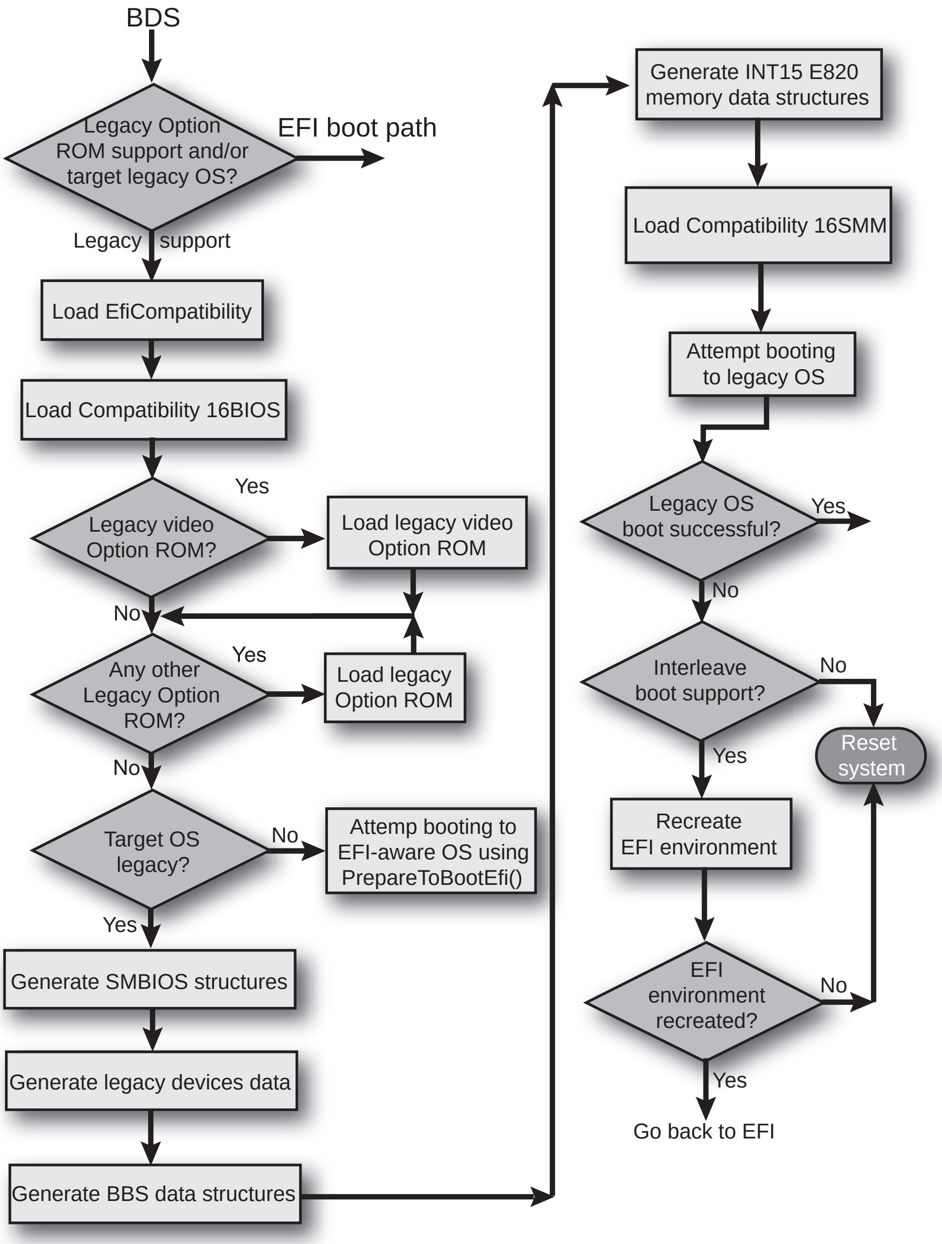

BDS
Legacy Option ROM support and/or target legacy OS?
EFI boot path
Legacy support
Load EfiCompatibility
Load Compatibility 16BIOS
Legacy video Option ROM?
Yes
Load legacy video Option ROM
No
Any other Legacy Option ROM?
Yes
Load legacy Option ROM
No
Target OS legacy?
No
Attemp booting to EFI-aware OS using PrepareToBootEfi()
Yes
Generate SMBIOS structures
Generate legacy devices data
Generate BBS data structures
Generate INT15 E820 memory data structures
Load Compatibility 16SMM
Attempt booting to legacy OS
Legacy OS boot successful?
Yes
No
Interleave boot support?
No
Reset system
Yes
Recreate EFI environment
EFI environment recreated?
No
Yes
Go back to EFI

Figure 17.5 Functional View of CSM under BDS

The EFI Device Path distinguishes between booting to an EFI-aware operating system (irrespective of whether the device is EFI Option ROM or legacy Option ROM controlled) or to a traditional operating system. The LegacyBoot() function is used in booting to a traditional operating system. A particular implementation may choose to check the removable media boot devices for a media presence prior to setting boot devices. This check may speed up the boot process and may prevent a possible system reset. Whether a failed traditional boot situation causes a system reset or returns to EFI is implementation-dependent. An intelligent implementation supporting interleave boot between legacy and EFI needs the control to be returned back to EFI in case of a legacy boot failure. If booting to an EFI-aware operating system and any legacy Option ROM has been initialized, then the PrepareToBootEfi() function is used.

Installing Legacy BIOS Environment

While installing the legacy environment, the BDS invokes the Legacy BIOS Driver to perform various actions required to initialize the environment for EfiCompatibility and Compatibility16BIOS. These actions may include, but are not limited to, the following:

- Finding the Legacy Region Protocol, the Legacy Interrupt Protocol, the Legacy BIOS Platform Protocol, and the Legacy 8259 Protocol.
- Allocating the memory for interrupt vectors and BDA from traditional memory region in 0x00000–0x00500.
- Allocating the memory for EBDA and EfiCompatibility usage. A particular implementation may choose to allocate 0x80000–0x9FBFF for EfiCompatibility and 0x9FC00–0x9FFFF for EBDA.
- Allocating memory for Thunk and ReverseThunk.
- Initializing Thunk including updating of any relocatable addresss.
- Allocate low memory (below 1MB) and high memory (above 1MB) that can be used by PMM in Compatibility16BIOS.
- Generating the traditional memory map.
- Searching for Compatibility16BIOS module in the firmware volume.
- Determining the size of the Compatibility16BIOS module and calculating the starting address of the Compatibility16BIOS module.
- Making the memory region 0xE0000–0xFFFFF, the final destination of the Compatibility16BIOS module read-write, and then copying the Compatibility16BIOS to the final destination.

- Searching and validating the Compatibility16 Table, saving the Compatibility16 functions entry point and updating internal data structures.
- Generating preliminary E820 table, initializing BDA and EBDA, and initializing legacy CMOS locations.
- Filling the necessary fields in the Compatibility16 Table.
- Invoking the Compatibility16InitializeYourself() function via Thunk and the Compatibility16 Functions entry point.
- Getting the Plug and Play installation check structure from Compatibility16 Table and initializing the internal data structures.
- Installing Legacy BIOS Protocol.

Booting to a Legacy Operating System

While booting to a legacy operating system, the BDS perform various actions using the LegacyBoot() function. These actions include, but are not limited to, the following:

- Ensuring that the legacy BIOS region is read-write enabled.
- Building SMBIOS data structures.
- Ensuring that all PCI interrupt lines are programmed properly for legacy operation using 8259 PIC.
- Building the information for serial ports, parallel ports, and floppy.
- Identifying and building the information for IDE ATA/ATAPI devices.
- Initializing timer data areas in BDA.
- Rebuilding E820 memory table and copying it to legacy BIOS region using the Compatibility16GetTableAddress() function.
- Building BBS tables and assigning boot priority of the devices.
- Copying MP table, OEM-specific 16-bit data, and OEM-specific 32-bit data in the legacy BIOS region using the Compatibility16GetTableAddress() function.
- Adjusting the vector base of legacy interrupt in 8259 PIC.
- Invoking the PrepareToBoot() function of Compatibility16BIOS.

- Making the legacy BIOS region read-only (write-protected).
- Invoking the Compatibility16Boot() function to issue INT19 to boot to the legacy operating system.

Appendix A

Data Types

Table A.1 contains the set of base types that are used in all EFI applications and EFI drivers. Use these base types to build more complex unions and structures. The file `EFIBIND.H` in the *EFI 1.10 Sample Implementation* contains the code required to map compiler-specific data types to the EFI data types. If you are using a new compiler, update only this one file; all other EFI related sources should compile unmodified. Table A.2 contains the modifiers you can use in conjunction with the EFI data types.

Table A.1 Common EFI Data Types

Mnemonic	Description
BOOLEAN	Logical Boolean. 1-byte value containing a 0 for FALSE or a 1 for TRUE. Other values are undefined.
INTN	Signed value of native width. (4 bytes on IA-32, 8 bytes on Itanium®-based operations)
UINTN	Unsigned value of native width. (4 bytes on IA-32, 8 bytes on Itanium®-based operations)
INT8	1-byte signed value.
UINT8	1-byte unsigned value.
INT16	2-byte signed value.
UINT16	2-byte unsigned value.
INT32	4-byte signed value.
UINT32	4-byte unsigned value.
INT64	8-byte signed value.
UINT64	8-byte unsigned value.

Mnemonic	Description
CHAR8	1-byte Character.
CHAR16	2-byte Character. Unless otherwise specified all strings are stored in the UTF-16 encoding format as defined by Unicode 2.1 and ISO/IEC 10646 standards.
VOID	Undeclared type.
EFI_GUID	128-bit buffer containing a unique identifier value. Unless otherwise specified, aligned on a 64-bit boundary.
EFI_STATUS	Status code. Type INTN.
EFI_HANDLE	A collection of related interfaces. Type VOID *.
EFI_EVENT	Handle to an event structure. Type VOID *.
EFI_LBA	Logical block address. Type UINT64.
EFI_TPL	Task priority level. Type UINTN.
EFI_MAC_ADDRESS	32-byte buffer containing a network Media Access Control address.
EFI_IPv4_ADDRESS	4-byte buffer. An IPv4 Internet protocol address.
EFI_IPv6_ADDRESS	16-byte buffer. An IPv6 Iinternet protocol address.
EFI_IP_ADDRESS	16-byte buffer aligned on a 4-byte boundary. An IPv4 or IPv6 Internet protocol address.
<Enumerated Type>	Element of an enumeration. Type INTN.

Table A.2 Modifiers for Common EFI Data Types

Mnemonic	Description
IN	Datum is passed to the function.
OUT	Datum is returned from the function.
OPTIONAL	Datum is passed to the function is optional, and a NULL may be passed if the value is not supplied.
STATIC	The function has local scope. This replaces the standard C static key word, so it can be overloaded for debugging.
VOLATILE	Declare a variable to be volatile and thus exempt from optimization to remove redundant or unneeded accesses. Any variable that represents a hardware device should be declared as VOLATILE.
CONST	Declare a variable to be of type const. This is a hint to the compiler to enable optimization and stronger type checking at compile time.
EFIAPI	Defines the calling convention for EFI interfaces. All EFI intrinsic services and any member function of a protocol must use this modifier in the function definition.

Appendix B

Status Codes

Most EFI interfaces return an EFI_STATUS code. Table B.1 lists the status code ranges. Tables B.2, B.3, and B.4 list these codes for success, errors, and warnings, respectively. Error codes also have their highest bit set, so all error codes have negative values. The range of status codes that have the highest bit set and the next to highest bit clear are reserved for use by EFI. The range of status codes that have both the highest bit set and the next to highest bit set are reserved for use by OEMs. Success and warning codes have their highest bit clear, so all success and warning codes have positive values. The range of status codes that have both the highest bit clear and the next to highest bit clear are reserved for use by EFI. The range of status code that have the highest bit clear and the next to highest bit set are reserved for use by OEMs.

Table B.1 EFI_STATUS Code Ranges

IA-32 Range	Intel® Itanium® Architecture Range	Description
0x00000000- 0x3fffffff	0x0000000000000000- 0x3fffffffffffffff	Success and warning codes reserved for use by EFI. See Tables B.2 and B.4 for valid values in this range.
0x40000000- 0x7fffffff	0x4000000000000000- 0x7fffffffffffffff	Success and warning codes reserved for use by OEMs.
0x80000000- 0xbfffffff	0x8000000000000000- 0xbfffffffffffffff	Error codes reserved for use by EFI. See Table B.3 for valid values for this range.
0xc0000000- 0xffffffff	0xc000000000000000- 0xffffffffffffffff	Error codes reserved for use by OEMs.

Table B.2 EFI_STATUS Success Codes (High Bit Clear)

Mnemonic	Value	Description
EFI_SUCCESS	0	The operation completed successfully.

Table B.3 EFI_STATUS Error Codes (High Bit Set)

Mnemonic	Value	Description
EFI_LOAD_ERROR	1	The image failed to load.
EFI_INVALID_PARAMETER	2	A parameter was incorrect.
EFI_UNSUPPORTED	3	The operation is not supported.
EFI_BAD_BUFFER_SIZE	4	The buffer was not the proper size for the request
EFI_BUFFER_TOO_SMALL	5	The buffer is not large enough to hold the requested data. The required buffer size is returned in the appropriate parameter when this error occurs.
EFI_NOT_READY	6	There is no data pending upon return.
EFI_DEVICE_ERROR	7	The physical device reported an error while attempting the operation.
EFI_WRITE_PROTECTED	8	The device cannot be written to.

Mnemonic	Value	Description
EFI_OUT_OF_RESOURCES	9	A resource has run out.
EFI_VOLUME_CORRUPTED	10	An inconsistency was detected on the file system causing the operation to fail.
EFI_VOLUME_FULL	11	The file system has no more space.
EFI_NO_MEDIA	12	The device does not contain any medium to perform the operation.
EFI_MEDIA_CHANGED	13	The medium in the device has changed since the last access.
EFI_NOT_FOUND	14	The item was not found.
EFI_ACCESS_DENIED	15	Access was denied.
EFI_NO_RESPONSE	16	The server was not found or did not respond to the request.
EFI_NO_MAPPING	17	A mapping to a device does not exist.
EFI_TIMEOUT	18	The timeout time expired.
EFI_NOT_STARTED	19	The protocol has not been started.
EFI_ALREADY_STARTED	20	The protocol has already been started.
EFI_ABORTED	21	The operation was aborted.
EFI_ICMP_ERROR	22	An ICMP error occurred during the network operation.
EFI_TFTP_ERROR	23	A TFTP error occurred during the network operation.
EFI_PROTOCOL_ERROR	24	A protocol error occurred during the network operation.
EFI_INCOMPATIBLE_VERSION	25	The function encountered an internal version that was incompatible with a version requested by the caller.
EFI_SECURITY_VIOLATION	26	The function was not performed due to a security violation.
EFI_CRC_ERROR	27	A CRC error was detected.

Table B.4 EFI_STATUS Warning Codes (High Bit Clear)

Mnemonic	Value	Description
EFI_WARN_UNKNOWN_GLYPH	1	The Unicode string contained one or more characters that the device could not render and were skipped.
EFI_WARN_DELETE_FAILURE	2	The handle was closed, but the file was not deleted.
EFI_WARN_WRITE_FAILURE	3	The handle was closed, but the data to the file was not flushed properly.
EFI_WARN_BUFFER_TOO_SMALL	4	The resulting buffer was too small, and the data was truncated to the buffer size.

Appendix C

Quick Reference

This appendix contains a summary of all the services and GUIDs available to EFI drivers. This includes EFI Boot Services, EFI Runtime Services, EFI Driver Library Services, GUID variables, protocol variables, and the various protocol services. Some of the GUIDs and protocols listed here are not part of the *EFI 1.10 Specification*. Instead, they are extensions that are included in the *EFI Sample Implementation*.

Tables C.1, C.2, and C.3 list the EFI Boot Services and EFI Runtime Services. These three tables list the most commonly used services, the rarely used services, and the services that should not be used from EFI drivers. Services labeled with type BS are EFI Boot Services, and services labeled with type RT are EFI Runtime Services. A detailed explanation of the services that are rarely used or should not be used by EFI drivers is included in Chapter 2. Chapters 5 and 6 of the *EFI 1.10 Specification* contain detailed descriptions of all the EFI Boot Services and EFI Runtime Services.

Table C.4 lists the EFI Driver Library Services. These services simplify the implementation of EFI drivers. Most of these services are layered in to of the EFI Boot Services and EFI Runtime Services. A detailed description of these services can be found in the *EFI Driver Library Specification*.

Table C.5 lists the constants that are commonly used in EFI.

Table C.6 lists the GUIDs that are available to EFI drivers in the *EFI Sample Implementation*. The Directory Name heading is the name of the GUID directory that can be used with the `EFI_GUID_DEFINITION()` macro. The GUID Variable Names heading contains the names of the global variables that are available to the EFI driver that uses the `EFI_GUID_DEFINITION()` macro for that directory. For example, assume the following statement is added to an EFI driver:

```
#include EFI_GUID_DEFINITION(PcAnsi)
```

Then, the following variables of type `EFI_GUID` will be available to that EFI driver:

```
gEfiPcAnsiGuid  gEfiVT100Guid  gEfiVT100PlusGuid
gEfiVTUTF8Guid
```

Table C.7 lists the protocols that are available to EFI drivers in the *EFI Sample Implementation*. The Directory Name heading is the name of the protocol directory that can be used with the `EFI_PROTOCOL_DEFINITION()` macro. The Protocol GUID Variable Name heading contains the names of the global variables that are available to a driver that uses the `EFI_PROTOCOL_DEFINITION()` macro for that directory. For example, assume the following statement is added to an EFI driver:

```
#include EFI_PROTOCOL_DEFINITION(PciIo)
```

Then, the following variable of type `EFI_GUID`, along with the definition of the `EFI_PCI_IO_PROTOCOL,` would be available to that EFI driver:

```
gEfiPciIoProtocolGuid
```

Table C.8 contains the list of the protocols that are available to EFI drivers along with the list of services that each protocol provides. This table does not show any of the data elements that a protocol interface may contain. Refer to the *EFI 1.10 Specification* for additional details on each protocol. Each protocol is named by its C data structure. These protocols are available to EFI drivers that use the `EFI_PROTOCOL_DEFINITION()` macro for a specific protocol. For example, if an EFI driver intends to consume or produce the EFI USB I/O Protocol, it would need to include the following statement in its implementation:

```
#include EFI_PROTOCOL_DEFINITION(UsbIo)
```

Table C.9 contains the list of debug error levels that can be used with the `DEBUG()` macros.

Table C.1 EFI Services Commonly Used by EFI Drivers

Type	Service	Type	Service
BS	gBS->AllocatePool()	BS	gBS->InstallMultipleProtocolInterfaces()
BS	gBS->FreePool()	BS	gBS->UninstallMultipleProtocolInterfaces()
BS	gBS->AllocatePages()	BS	gBS->LocateHandleBuffer()
BS	gBS->FreePages()	BS	gBS->LocateProtocol()
BS	gBS->SetMem()	BS	gBS->OpenProtocol()
BS	gBS->CopyMem()	BS	gBS->CloseProtocol()
		BS	gBS->OpenProtocolInformation()
BS	gBS->CreateEvent()		
BS	gBS->CloseEvent()	BS	gBS->RaiseTPL()
BS	gBS->SignalEvent()	BS	gBS->RestoreTPL()
BS	gBS->SetTimer()		
BS	gBS->CheckEvent()	BS	gBS->Stall()

Table C.2 EFI Services Rarely Used by EFI Drivers

Type	Service	Type	Service
BS	gBS->ReinstallProtocolInterface()	BS	gBS->LoadImage()
BS	gBS->LocateDevicePath()	BS	gBS->StartImage()
		BS	gBS->UnloadImage()
BS	gBS->ConnectController()	BS	gBS->Exit()
BS	gBS->DisconnectController()		
		BS	gBS->InstallConfigurationTable()
RT	gRT->GetVariable()		
RT	gRT->SetVariable()	RT	gRT->GetTime()
BS	gBS->GetNextMonotonicCount()	BS	gBS->CalculateCrc32()
RT	gRT->GetNextHighMonotonicCount()		
		RT	gRT->ConvertPointer()

Table C.3 EFI Services that Should Not Be Used by EFI Drivers

Type	Service	Type	Service
BS	gBS->GetMemoryMap()	RT	gRT->SetVirtualAddressMap()
BS	gBS->ExitBootServices()	RT	gRT->GetNextVariableName()
BS	gBS->InstallProtocolInterface()	RT	gRT->SetTime()
BS	gBS->UninstallProtocolInterface()	RT	gRT->GetWakeupTime()
BS	gBS->HandleProtocol()	RT	gRT->SetWakeupTime()
BS	gBS->LocateHandle()		
BS	gBS->RegisterProtocolNotify()	RT	gRT->ResetSystem()
BS	gBS->ProtocolsPerHandle()	BS	gBS->SetWatchDogTimer()
BS	gBS->WaitForEvent()		

Table C.4 EFI Driver Library Functions

Initialization Functions	**Link List Functions**	**Memory Functions**
EfiInitializeDriverLib()	InitializeListHead()	EfiCopyMem()
EfiLibInstallDriverBinding()	IsListEmpty()	EfiSetMem()
EfiLibInstallAllDriverProtocols()	RemoveEntryList()	EfiZeroMem()
	InsertTailList()	EfiCompareMem()
Device Path Functions	InsertHeadList()	EfiLibAllocatePool()
EfiDevicePathInstance()	SwapListEntries()	EfiLibAllocateZeroPool()
EfiAppendDevicePath()		
EfiAppendDevicePathNode()	**Math Functions**	**String Functions**
EfiAppendDevicePathInstance()	LShiftU64()	EfiStrCpy()
EfiFileDevicePath()	RShiftU64()	EfiStrLen()
EfiDevicePathSize()	MultU64x32()	EfiStrSize()
EfiDuplicateDevicePath()	DivU64x32()	EfiStrCmp()
		EfiStrCat()
Miscellaneous Functions	**Spin Lock Functions**	EfiLibLookupUnicodeString()
EfiCompareGuid()	EfiInitializeLock()	EfiLibAddUnicodeString()
CR()	EfiAcquireLock()	EfiLibFreeUnicodeStringTable()
EfiLibCreateProtocolNotifyEvent()	EfiAcquireLockOrFail()	
EfiLibGetSystemConfigurationTable()	EfiReleaseLock()	**Debug Functions**
		DEBUG()

Table C.5 EFI Constants

Mnemonic	Description
TRUE	One
FALSE	Zero
NULL	VOID pointer to zero.

Table C.6 EFI GUID Variables

Directory Name	GUID Variable Names
Bmp	gEfiDefaultBmpLogoGuid
ConsoleInDevice	gEfiConsoleInDeviceGuid
ConsoleOutDevice	gEfiConsoleOutDeviceGuid
DebugImageInfoTable	gEfiDebugImageInfoTableGuid
GlobalVariable	gEfiGlobalVariableGuid
Gpt	gEfiPartTypeUnusedGuid
	gEfiPartTypeSystemPartGuid
	gEfiPartTypeLegacyMbrGuid
PcAnsi	gEfiPcAnsiGuid
	gEfiVT100Guid
	gEfiVT100PlusGuid
	gEfiVTUTF8Guid
PciOptionRomTable	gEfiPciOptionRomTableGuid
PrimaryConsoleInDevice	gEfiPrimaryConsoleInDeviceGuid
PrimaryConsoleOutDevice	gEfiPrimaryConsoleOutDeviceGuid
PrimaryStandardErrorDevice	gEfiPrimaryStandardErrorDeviceGuid
SalSystemTable	gEfiSalSystemTableGuid
SmBios	gEfiSmbiosTableGuid
StandardErrorDevice	gEfiStandardErrorDeviceGuid

Table C.7 EFI Protocol Variables

Directory Name	Protocol GUID Variable Names
BIS	`gEfiBisProtocolGuid`
BlockIo	`gEfiBlockIoProtocolGuid`
BusSpecificDriverOverride	`gEfiBusSpecificDriverOverrideProtocolGuid`
ComponentName	`gEfiComponentNameProtocolGuid`
DebugPort	`gEfiDebugPortProtocolGuid`
DebugSupport	`gEfiDebugSupportProtocolGuid`
Decompress	`gEfiDecompressProtocolGuid`
DeviceIo	`gEfiDeviceIoProtocolGuid`
DevicePath	`gEfiDevicePathProtocolGuid`
DiskIo	`gEfiDiskIoProtocolGuid`
DriverBinding	`gEfiDriverBindingProtocolGuid`
DriverConfiguration	`gEfiDriverConfigurationProtocolGuid`
DriverDiagnostics	`gEfiDriverDiagnosticsProtocolGuid`
Ebc	`GEfiEbcProtocolGuid`
	`gEfiEbcDebugHelperProtocolGuid`
EfiNetworkInterfaceIdentifier	`gEfiNetworkInterfaceIdentifierProtocolGuid`
IsaAcpi	`gEfiIsaAcpiProtocolGuid`
IsaIo	`gEfiIsaIoProtocolGuid`
LegacyBoot	`gEfiLegacyBootProtocolGuid`
LoadedImage	`gEfiLoadedImageProtocolGuid`
LoadFile	`gEfiLoadFileProtocolGuid`
PciIo	`GEfiPciIoProtocolGuid`
PciRootBridgeIo	`gEfiPciRootBridgeIoProtocolGuid`
PlatformDriverOverride	`gEfiPlatformDriverOverrideProtocolGuid`
PxeBaseCode	`gEfiPxeBaseCodeProtocolGuid`
PxeBaseCodeCallBack	`gEfiPxeBaseCodeCallbackProtocolGuid`
ScsiPassThru	`gEfiScsiPassThruProtocolGuid`
SerialIo	`gEfiSerialIoProtocolGuid`
SimpleFileSystem	`gEfiSimpleFileSystemProtocolGuid`
	`gEfiFileInfoGuid`
	`gEfiFileInfoIdGuid`

Directory Name	Protocol GUID Variable Names
	gEfiFileSystemVolumeLabelInfoIdGuid
SimpleNetwork	gEfiSimpleNetworkProtocolGuid
SimplePointer	gEfiSimplePointerProtocolGuid
SimpleTextIn	gEfiSimpleTextInProtocolGuid
SimpleTextOut	gEfiSimpleTextOutProtocolGuid
UgaDraw	gEfiUgaDrawProtocolGuid
UgaIo	gEfiUgaIoProtocolGuid
UgaSplash	gEfiUgaSplashProtocolGuid
UnicodeCollation	gEfiUnicodeCollationProtocolGuid
UsbAtapi	gEfiUsbAtapiProtocolGuid
UsbHostController	gEfiUsbHcProtocolGuid
UsbIo	gEfiUsbIoProtocolGuid
VgaMiniPort	gEfiVgaMiniPortProtocolGuid
WinNtIo	gEfiWinNtIoProtocolGuid
	gEfiWinNtVirtualDisksGuid
	gEfiWinNtPhysicalDisksGuid
	gEfiWinNtFileSystemGuid
	gEfiWinNtSerialPortGuid
	gEfiWinNtUgaGuid
	gEfiWinNtConsoleGuid
WinNtThunk	gefiWinNtThunkProtocolGuid

Table C.8 EFI Protocol Service Summary

EFI_BIS_PROTOCOL	**EFI_DECOMPRESS_PROTOCOL**
Initialize()	GetInfo()
Shutdown()	Decompress()
Free()	
GetBootObjectAuthorizationCertificate()	**EFI_DEVICE_IO_PROTOCOL**
GetBootObjectAuthorizationCheckFlag()	Mem.Read()
GetBootObjectAuthorizationUpdateToken()	Mem.Write()
GetSignatureInfo()	Io.Read()
UpdateBootObjectAuthorization()	Io.Write()
VerifyBootObject()	Pci.Read()
VerifyObjectWithCredential()	Pci.Write()
	PciDevicePath()
EFI_BLOCK_IO_PROTOCOL	Map()
Reset()	Unmap()
ReadBlocks()	AllocateBuffer()
WriteBlocks()	Flush()
FlushBlocks()	FreeBuffer()
EFI_BUS_SPECIFIC_DRIVER_OVERRIDE_PROTOCOL	**EFI_DEVICE_PATH_PROTOCOL**
GetDriver()	
	EFI_DISK_IO_PROTOCOL
EFI_COMPONENT_NAME_PROTOCOL	ReadDisk()
GetDriverName()	WriteDisk()
GetControllerName()	
	EFI_DRIVER_BINDING_PROTOCOL
EFI_DEBUGPORT_PROTOCOL	Supported()
Reset()	Start()
Write()	Stop()
Read()	
Poll()	**EFI_DRIVER_CONFIGURATION_PROTOCOL**
	SetOptions()

EFI_DEBUG_SUPPORT_PROTOCOL	OptionsValid()
GetMaximumProcessorIndex()	ForceDefaults()
RegisterPeriodicCallback()	
RegisterExceptionCallback()	**EFI_DRIVER_DIAGNOSTICS_PROTOCOL**
InvalidateInstructionCache()	RunDiagnostics()
EFI_EBC_PROTOCOL	**EFI_PCI_IO_PROTOCOL**
CreateThunk()	PollMem()
UnloadImage()	PollIo()
RegisterICacheFlush()	Mem.Read()
	Mem.Write()
EFI_NETWORK_INTERFACE_IDENTIFIER_PROTOCOL	Io.Read()
	Io.Write()
EFI_ISA_ACPI_PROTOCOL	Pci.Read()
DeviceEnumerate()	Pci.Write()
SetPower()	CopyMem()
GetCurResource()	Map()
GetPosResource()	Unmap()
SetResource()	AllocateBuffer()
EnableDevice()	FreeBuffer()
InitDevice()	Flush()
InterfaceInit()	GetLocation()
	Attributes()
EFI_ISA_IO_PROTOCOL	GetBarAttributes()
Mem.Read()	SetBarAttributes()
Mem.Write()	
Io.Read()	**EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL**
Io.Write()	PollMem()
CopyMem()	PollIo()
Map()	Mem.Read()
Unmap()	Mem.Write()
AllocateBuffer()	Io.Read()
FreeBuffer()	Io.Write()
Flush()	Pci.Read()

	Pci.Write()
EFI_LEGACY_BOOT_PROTOCOL	CopyMem()
BootIt()	Map()
	Unmap()
EFI_LOADED_IMAGE_PROTOCOL	AllocateBuffer()
Unload()	FreeBuffer()
	Flush()
EFI_LOAD_FILE_PROTOCOL	GetAttributes()
LoadFile()	SetAttributes()
	Configuration()
EFI_PLATFORM_DRIVER_OVERRIDE_PROTOCOL	**EFI_SIMPLE_FILE_SYSTEM_PROTOCOL**
GetDriver()	OpenVolume()
GetDriverPath()	
DriverLoaded()	EFI_FILE Handle
	Open()
EFI_PXE_BASE_CODE_PROTOCOL	Close()
Start()	Delete()
Stop()	Read()
Dhcp()	Write()
Discover()	SetPosition()
Mtftp()	GetPosition()
UdpWrite()	GetInfo()
UdpRead()	SetInfo()
SetIpFilter()	Flush()
Arp()	
SetParameters()	**EFI_SIMPLE_NETWORK_PROTOCOL**
SetStationIp()	Start()
SetPackets()	Stop()
	Initialize()
EFI_PXE_BASE_CODE_CALLBACK_PROTOCOL	Reset()
Callback()	Shutdown()
	ReceiveFilters()

EFI_SCSI_PASS_THRU Protocol	StationAddress()
PassThru()	Statistics()
GetNextDevice()	MCastIPtoMAC()
BuildDevicePath()	NvData()
GetTargetLun()	GetStatus()
ResetChannel()	Transmit()
ResetTarget()	Receive()

SERIAL_IO_PROTOCOL	**EFI_SIMPLE_POINTER_PROTOCOL**
Reset()	Reset()
SetAttributes()	GetState()
SetControl()	
GetControl()	**EFI_SIMPLE_TEXT_IN_PROTOCOL**
Write()	Reset()
Read()	ReadKeyStroke()

EFI_SIMPLE_TEXT_OUT_PROTOCOL	**EFI_USB_HC_PROTOCOL**
Reset()	Reset()
OutputString()	GetState()
TestString()	SetState()
QueryMode()	ControlTransfer()
SetMode()	BulkTransfer()
SetAttribute()	AsyncInterruptTransfer()
ClearScreen()	SyncInterruptTransfer()
SetCursorPosition()	IsochronousTransfer()
EnableCursor()	AsyncIsochronousTransfer()
	GetRootHubPortNumber()
	GetRootHubPortStatus()
	SetRootHubPortFeature()
	ClearRootHubPortFeature()
	EFI_USB_IO Protocol
	UsbControlTransfer()

	UsbBulkTransfer()
	UsbAsyncInterruptTransfer()
	UsbSyncInterruptTransfer()
	UsbIsochronousTransfer()
	UsbAsyncIsochronousTransfer()
	UsbGetDeviceDescriptor()
UNICODE_COLLATION_PROTOCOL	UsbGetConfigDescriptor()
StriColl()	UsbGetInterfaceDescriptor()
MetaiMatch()	UsbGetEndpointDescriptor()
StrLwr()	UsbGetStringDescriptor()
StrUpr()	UsbGetSupportedLanguages()
FatToStr()	UsbPortReset()
StrToFat()	
	EFI_VGA_MINIPORT_PROTOCOL
EFI_USB_ATAPI_PROTOCOL	SetMode()
UsbAtapiPacketCmd()	
UsbAtapiReset()	**EFI_WIN_NT_IO_PROTOCOL**
	EFI_WIN_NT_THUNK_PROTOCOL
	Too many to list here. See public EFI distributions for more data on this protocol.

Table C.9 Error Levels

Mnemonic	Value	Description
EFI_D_INIT	0x00000001	Initialization messages
EFI_D_WARN	0x00000002	Warning messages
EFI_D_LOAD	0x00000004	Load events
EFI_D_FS	0x00000008	EFI File System messages
EFI_D_POOL	0x00000010	EFI pool allocation and free messages
EFI_D_PAGE	0x00000020	EFI page allocation a free messages
EFI_D_INFO	0x00000040	Informational messages
EFI_D_VARIABLE	0x00000100	EFI variable service messages
EFI_D_BM	0x00000400	EFI Boot Manager messages

Mnemonic	Value	Description
EFI_D_BLKIO	0x00001000	EFI Block I/O Protocol messages
EFI_D_NET	0x00004000	EFI Simple Network Protocol, PXE Base Code, BIS messages
EFI_D_UNDI	0x00010000	UNDI driver messages
EFI_D_LOADFILE	0x00020000	Load File Protocol messages
EFI_D_EVENT	0x00080000	EFI Event Services messages
EFI_D_ERROR	0x80000000	Critical error messages

Glossary

agent An EFI component that can consume a protocol in the handle database.

Agent handle A term used by some of the EFI Driver Model–related services in the *EFI 1.10 Specification* to represent an image handle, a driver handle, or a driver image handle.

Bootable Image Services A term used collectively to describe the Block I/O Protocol, Disk I/O Protocol, Simple File System Protocol, and Load File Protocol.

bus controller handle Managed by a bus driver or a hybrid driver that produces child handles. The term *bus* does not necessarily match the hardware topology, but in this book is used from the software perspective, and the production of the software construct—which is called a *child handle*—is the only distinction between a controller handle and a bus controller handle.

bus driver Nearly identical to a device driver except that it creates child handles. The main objectives of the bus driver are to initialize the bus controller, to determine how many children to create, to allocate resources and create a child handle for one or more child controllers, to install an I/O protocol on the child handle that abstracts the I/O operations that the controller supports (such as the PCI I/O Protocol or the USB I/O Protocol), if the child handle represents a physical device, to then install a Device Path Protocol, and to load drivers from option ROMs if present. (To date, the PCI bus driver is the only bus driver that loads from option ROMs.)

child handle A type of controller handle that is created by a bus driver or a hybrid driver. The distinction between a *child handle* and a *controller handle* depends on the perspective of the driver that is using the handle. A handle would be a child handle from a bus driver's perspective, and that same handle may be a controller handle from a device driver's perspective.

device handle Used interchangeably with *controller handle*.

driver handle Supports the Driver Binding Protocol. May optionally support the Driver Configuration Protocol, the Driver Diagnostics Protocol, and the Component Name Protocol. *DIG64* requires these optional protocols for Itanium®-based platforms. Other platform specifications may or may not require these protocols. See Chapter 9 of the *EFI 1.10 Specification*.

driver image handle The intersection of image handles and driver handles. Supports both the Loaded Image Protocol and the Driver Binding Protocol. May optionally support the Driver Configuration Protocol, the Driver Diagnostics Protocol, and the Component Name Protocol. *DIG64* requires these optional protocols for Itanium-based platforms. Other platform specifications may or may not require these protocols. See Chapter 9 of the *EFI 1.10 Specification*.

controller handle A handle that represents a physical device, and that therefore must support the Device Path Protocol. If the handle represents a virtual device, then it must not support the Device Path Protocol. In addition, a device handle must support one or more additional I/O protocols that are used to abstract access to that device.

hybrid driver A hybrid driver has features of both a device driver and a bus driver. The main distinction between a device driver and a bus driver is that a bus driver creates child handles and a device driver does not create any child handles. In addition, a bus driver is allowed only to install produced protocols on the newly created child handles. A hybrid driver creates new child handles, installs produced protocols on the child handles, and installs produced protocols onto the bus controller handle.

image handle Supports the Loaded Image Protocol. See Chapter 7 of the *EFI 1.10 Specification*.

Network Services Refers to Network Interface Identifier Protocol, Simple Network Protocol, and PXE Base Code Protocol.

PCI Services Refers to PCI Root Bridge I/O Protocol, PCI I/O Protocol, and Device I/O Protocol.

physical controller handle A controller handle that represents a physical device that must support the Device Path Protocol. See Chapter 8 of the *EFI 1.10 Specification.*

SCSI Services Refers to the SCSI Pass Thru Protocol

service handle A handle that does not support the Loaded Image Protocol, the Driver Binding Protocol, or the Device Path Protocol. Instead, it supports the only instance of a specific protocol in the entire handle database. This protocol provides services that may be used by other EFI applications or EFI drivers. The list of service protocols that are defined in the *EFI 1.10 Specification* include the Platform Driver Override Protocol, Unicode Collation Protocol, Boot Integrity Services Protocol, Debug Support Protocol, Decompress Protocol, and EFI Byte Code Protocol.

Signal event An event whose notification function is scheduled for execution whenever the event goes from the waiting state to the signaled state.

Tag GUID is a protocol that contains only a GUID.

USB Services refers to the USB Host Controller Protocol and USB I/O Protocol.

virtual controller handle A controller handle that represents a virtual device and does not support the Device Path Protocol.

Wait event is an event whose notification function is executed whenever the event is checked or waited upon.

Index

Continuing Education is Essential

It's a challenge we all face – keeping pace with constant change in information technology. Whether our formal training was recent or long ago, we must all find time to keep ourselves educated and up to date in spite of the daily time pressures of our profession.

Intel produces technical books to help the industry learn about the latest technologies. The focus of these publications spans the basic motivation and origin for a technology through its practical application.

Right books, right time, from the experts

These technical books are planned to synchronize with roadmaps for technology and platforms, in order to give the industry a head-start. They provide new insights, in an engineer-to-engineer voice, from named experts. Sharing proven insights and design methods is intended to make it more practical for you to embrace the latest technology with greater design freedom and reduced risks.

I encourage you to take full advantage of Intel Press books as a way to dive deeper into the latest technologies, as you plan and develop your next generation products. They are an essential tool for every practicing engineer or programmer. I hope you will make them a part of your continuing education tool box.

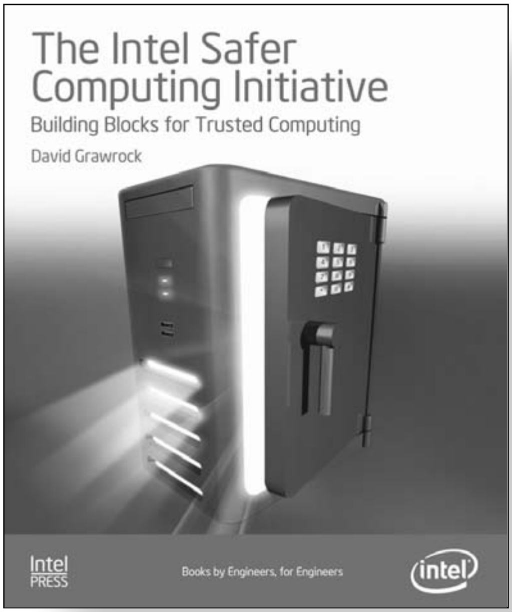
Sincerely,

Justin Rattner
Senior Fellow and Chief Technology Officer
Intel Corporation

Turn the page to learn about titles
from Intel Press for system developers

Enhance security and protection against software-based attacks

The Intel Safer Computing Initiative
Building Blocks for Trusted Computing
By David Grawrock
ISBN 0-9764832-6-2


The Intel Safer Computing Initiative
Building Blocks for Trusted Computing
David Grawrock
Intel PRESS
Books by Engineers, for Engineers
intel

With the ever-increasing connectivity of home and business computers, it is essential that developers understand how the Intel Safer Computing Initiative can provide critical security building blocks to better protect the PC computing environment. Security capabilities need to be carefully evaluated before delivery into the marketplace. Intel is committed to delivering security capabilities in a responsible manner for end users and the ecosystem.

A highly versatile set of hardware-based security enhancements, code-named LaGrande Technology (LT), will be supported on Intel processors and chipsets to help enhance PC platforms. This book covers the fundamentals of LT and key Trusted Computing concepts such as security architecture, cryptography, trusted computer base, and trusted channels.

Highlights include:

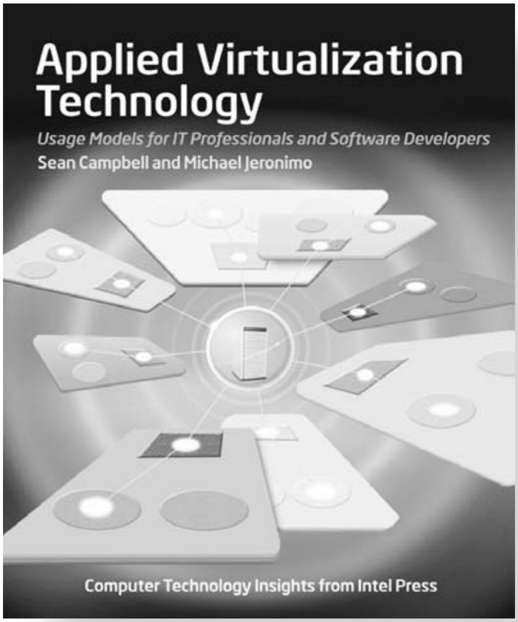
- History of trusted computing and definitions of key concepts
- Comprehensive overview of protections that are provided by LaGrande Technology
- Case study showing how access to memory is the focal point of an attack
- Protection methods for execution, memory, storage, input, and graphics
- How the Trusted Platform Module (TPM) supports attestation

In this concise book, the lead security architect for Intel's next-generation security initiative provides critical information you need to evaluate Trusted Computing for use on today's PC systems and to prepare your designs to respond to future threats.

Applied Virtualization Technology

Usage Models for IT Professionals and Software Developers

By Sean Campbell and Michael Jeronimo
ISBN: 0-9764832-3-8


Applied Virtualization Technology
Usage Models for IT Professionals and Software Developers
Sean Campbell and Michael Jeronimo
Computer Technology Insights from Intel Press

Server and desktop virtualization is one of the more significant technologies to impact computing in the last few years, promising the benefits of infrastructure consolidation, lower costs, increased security, ease of management, and greater employee productivity.

Using virtualization technology, one computer system can operate as multiple "virtual" systems. The convergence of affordable, powerful platforms and robust scalable virtualization solutions is spurring many technologists to examine the broad range of uses for virtualization. In addition, a set of processor and I/O enhancements to Intel server and client platforms, known as Intel® Virtualization Technology (Intel® VT), can further improve the performance and robustness of current software virtualization solutions.

This book takes a user-centered view and describes virtualization usage models for IT professionals, software developers, and software quality assurance staff. The book helps you plan the introduction of virtualization solutions into your environment and thereby reap the benefits of this emerging technology.

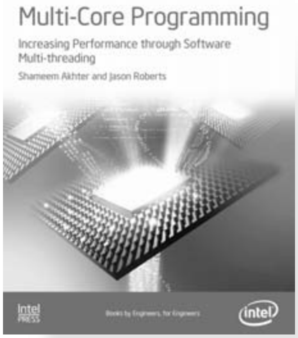
Highlights include:

- The challenges of current virtualization solutions
- In-depth examination of three software-based virtualization products
- Usage models that enable greater IT agility and cost savings
- Usage models for enhancing software development and QA environments
- Maximizing utilization and increasing flexibility of computing resources
- Reaping the security benefits of computer virtualization
- Distribution and deployment strategies for virtualization solutions

Multi-Core Programming
Increasing Performance through Software Multi-threading

By Shameem Akhter and Jason Roberts
ISBN 0-9764832-4-6

Software developers can no longer rely on increasing clock speeds alone to speed up single-threaded applications; instead, to gain a competitive advantage, developers must learn how to properly design their applications to run in a threaded environment. This book helps software developers write high-performance multi-threaded code for Intel's multicore architecture while avoiding the common parallel programming issues associated with multi-threaded programs. This book is a practical, hands-on volume with immediately usable code examples that enable readers to quickly master the necessary programming techniques.



Discover programming techniques for Intel multi-core architecture and Hyper-Threading Technology

The Software Optimization Cookbook, Second Edition
High-Performance Recipes for IA-32 Platforms

By Richard Gerber, Aart J.C. Bik, Kevin B. Smith, and Xinmin Tian
ISBN 0-9764832-1-1

Four Intel experts explain the techniques and tools that you can use to improve the performance of applications for IA-32 processors. Simple explanations and code examples help you to develop software that benefits from Intel® Extended Memory 64 Technology (Intel® EM64T), multi-core processing, Hyper-Threading Technology, OpenMP†, and multimedia extensions. This book guides you through the growing collection of software tools, compiler switches, and coding optimizations, showing you efficient ways to get the best performance from software applications.

> *❝ A must-read text for anyone who intends to write performance-critical applications for the Intel processor family. ❞*
>
> *—Robert van Engelen, Professor, Florida State University*

About Intel Press

Intel Press is the authoritative source of timely, technical books to help software and hardware developers speed up their development process. We collaborate only with leading industry experts to deliver reliable, first-to-market information about the latest technologies, processes, and strategies.

Our products are planned with the help of many people in the developer community and we encourage you to consider becoming a customer advisor. If you would like to help us and gain additional advance insight to the latest technologies, we encourage you to consider the Intel Press Customer Advisor Program. You can register here:

www.intel.com/intelpress/register.htm

For information about bulk orders or corporate sales, please send e-mail to
bulkbooksales@intel.com

Other Developer Resources from Intel

At these Web sites you can also find valuable technical information and resources for developers:

developer.intel.com	general information for developers
www.intel.com/software	content, tools, training, and the Intel® Early Access Program for software developers
www.intel.com/software/products	programming tools to help you develop high-performance applications
www.intel.com/netcomms	solutions and resources for networking and communications
www.intel.com/technology/itj	Intel Technology Journal
www.intel.com/idf	worldwide technical conference, the Intel Developer Forum

Intel
PRESS

If serial number is missing, please send an
e-mail to Intel Press at i**ntelpress@intel.com**

IMPORTANT

You can access the companion Web site for this book on the Internet at:

www.intel.com/intelpress/efi

Use the serial number located in the upper-right hand corner of this page to register your book and access additional material, including the *Digital Edition* of this book.