

Beyond BIOS:

Implementing the Unified Extensible Firmware Interface with Intel's Framework

Vincent Zimmer
Michael Rothman
Robert Hale

Intel
PRESS

Copyright © 2006 Intel Corporation. All rights reserved.

ISBN 0-9743649-0-8

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Publisher, Intel Press, Intel Corporation, 2111 NE 25th Avenue, JF3-330, Hillsboro, OR 97124-5961. E-mail: intelpress@intel.com.

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.

Intel, the Intel logo, Celeron, Intel Centrino, Intel NetBurst, Intel Xeon, Itanium, Pentium, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

† Other names and brands may be claimed as the property of others.

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

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 storage¹ 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 conven-

tions. 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 re-targeted from platform to platform and even from one instruction set to another. EFI has been successfully re-targeted 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 struc-

tures 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.