
Publisher

Richard Bowles

Managing Editor

Stuart Douglas

Content Architect

Vincent Zimmer

Michael Rothman

Program Manager

Stuart Douglas

Technical Editor

David Clark

Technical Illustrators

MPS Limited, a Macmillan Company

Technical and Strategic Reviewers

Jeff Bobzin

Cecil Lockett

Michael Turner

Michael Brinkman

Randy Murphy

Todd Selbo

Intel Technology Journal

Copyright © 2011 Intel Corporation. All rights reserved.
ISBN 978-1-934053-43-0, ISSN 1535-864X

Intel Technology Journal
Volume 15, Issue 1

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, Intel Atom, Intel AVX, Intel Battery Life Analyzer, Intel Compiler, Intel Core i3, Intel Core i5, Intel Core i7, Intel DPST, Intel Energy Checker, Intel Mobile Platform SDK, Intel Intelligent Power Node Manager, Intel QuickPath Interconnect, Intel Rapid Memory Power Management (Intel RMPM), Intel VTune Amplifier, and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

For more complete information about performance and benchmark results, visit www.intel.com/benchmarks

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

This book is printed on acid-free paper. 

Publisher: Richard Bowles
Managing Editor: Stuart Douglas

Library of Congress Cataloging in Publication Data:

Printed in United States of America

10 9 8 7 6 5 4 3 2 1

First printing: October, 2011

INTEL® TECHNOLOGY JOURNAL

UEFI TODAY: BOOTSTRAPPING THE CONTINUUM

Articles

Foreword	5
Beyond BIOS: Exploring the Many Dimensions of the Unified Extensible Firmware Interface	8
Silicon Enabling in a Modular Architecture	22
UEFI and the OEM and IHV Community	40
Booting in an Instant	68
UEFI Networking and Pre-OS Security	80
Debugging Firmware Based on the Unified Extensible Firmware Interface	102
Broad Use of UEFI in Hewlett Packard Systems	118

FOREWORD

by **Doug Fisher**

Intel Vice President

General Manager, Systems Software Division

From its roots in 1997 to support Intel® Itanium® based servers and the first published Extensible Firmware Interface (EFI) specification around 2000, Unified Extensible Firmware Interface (UEFI) has now eclipsed legacy BIOS across all computing platforms from high performance servers to mobile devices to deeply embedded devices. The success of UEFI would not have been possible without industry support of the open source implementation and industry-guided platform firmware specification. UEFI as an open standard specification ensured that it received broad support and investment from across the industry to evolve as required to support technology changes in our industry, for example boot time requirements and an ever increasing focus on security. This issue of the Intel Technology Journal is completely focused on UEFI.

The first article is an introduction to UEFI written by Mark Doran, Vincent J. Zimmer, and Michael A. Rothman; it provides both the details on the history of UEFI and how the specifications are managed within the UEFI Forum. The intent here is to not only better understand how UEFI is maintained as an open specification, but to also provide clarity on the usages and capabilities UEFI has made possible within our industry.

Silicon enabling provides vendor-specific value-added features on the platform with key chipset and CPU enabling code for the latest hardware. UEFI Platform Initialization (PI) specifications support this capability. In “Silicon Enabling,” authors Isaac Oram, Tim Lewis, and Vincent J. Zimmer focus on building block elements in PI that are the cornerstone of silicon enabling that provides OEMs with consistent and sufficient interfaces to build real systems without precluding opportunities for differentiation.

Regardless of how the computer industry has evolved over the years, the need for interoperability between the platform elements is unchanged. In “UEFI and the OEM and IHV Community,” authors Nathan Skalsky, Terry Kirch, Al Rickey, and Michael A. Rothman discuss how the UEFI standard introduces the composite pieces for such interoperability, and in so doing, illustrate how the end user benefits by both the OEM and IHV communities’ use of UEFI standards.

Target platforms supported by UEFI span deeply embedded devices to massive server clusters and everything in-between. One common desire

across all of these domains is the desire to reduce the time it takes for the platform to initialize (boot). Authors Michael A. Rothman, and John Mese, in “Booting in an Instant” cover the elements associated with boot performance from various points of view in the technology value chain. They also discuss how the evolution of standards (such as UEFI) have resulted in boot performance enhancements, and give examples of how these technology elements were incorporated into products to provide meaningful results to the end user.

The protection from security threats at all platform states, pre- and post-OS boot are increasingly important for our industry. In “UEFI Networking and Pre-OS security” authors Magnus Nystrom, Martin Nicholes, and Vincent J. Zimmer focus on how UEFI includes security in pre-OS state and networking for the platform boot process from local and remote media, assets to be protected, threats against those assets, and the various technologies that allow for their protection. They also go one step further to discuss forward-looking security capabilities and approaches enabled by UEFI.

For software developers, debugging can be a challenge and robust debugging tools are essential at every phase of development. UEFI is no different, and can be very difficult especially for embedded devices where access is limited. Stefano Righi, Brian Richardson, Jiewen Yao, and Elvin Li in “Debugging Issues with UEFI” provide an overview of common debug solutions including hardware-based debugging, system checkpoints, and source-level debugging. Firmware-specific concepts such as status codes, DEBUG/ASSERT macros and the UEFI debug protocol are introduced. They also discuss source-level debugging support using AMI and Intel solutions, comparing them to hardware-based alternatives.

To further emphasize the broad platform range (printers to high performing servers) associated with UEFI, the final article in this edition of the journal discusses one specific company’s use of UEFI across broad product lines. Dong Wei, Kimon Berlin, and Eugene Cohen all from Hewlett Packard discuss the specific strategy for value add for leveraging UEFI across the HP product portfolio.

This set of articles will show both a) the value of UEFI to our industry and how UEFI is positioned to continue to evolve and extend as required to meet the rapidly evolving requirements of our industry and customers, and b) the

impact of open standards like UEFI and how they are uniquely capable to provide the needed capability without precluding OEM differentiation. UEFI has covered tremendous ground since that first EFI specification around 2000, and I'm looking forward to watching how it evolves for the next 10 years to cover computing capabilities and usages that we've not yet imagined.

Doug Fisher

BEYOND BIOS: EXPLORING THE MANY DIMENSIONS OF THE UNIFIED EXTENSIBLE FIRMWARE INTERFACE

Contributors

Mark Doran

Intel Corporation

Vincent J. Zimmer

Intel Corporation

Michael A. Rothman

Intel Corporation

This article describes the basic capabilities of the specifications produced by the UEFI Forum as well as the history of how these standards evolved.

Introduction

The purpose of this article is to describe the basic capabilities of the UEFI Forum Specifications including the UEFI Specification version 2.3.1, the Platform Initialization Specification version 1.2, along with the structure and use of UDK2010, an open source implementation of the UEFI Forum Specifications. The history and evolution of these technologies provides some context for those descriptions.

The Chicken and the Egg

The very first effort that is considered a direct ancestor of UEFI technology had a very specific tactical goal. In the course of 1997 people at Intel were working on how to boot computers based on the prospective Itanium® Processor family. The original plan was to use the conventional BIOS code base for this job: while more or less everything else about the machines would be new—processors, chipsets, board designs, operating systems, and so on—it was felt that keeping stability in one element of the machine recipe a known quantity would be of some advantage. Without getting to specifics, this plan ultimately proved infeasible for technical and business reasons. This left the problem of how to boot an OS on these platforms, with something less than a year of time for resolution.

This challenge spawned the effort inside Intel that became known as the Intel Boot Initiative (IBI), specifically targeting development of a boot paradigm for Itanium Processor based machines. The IBI effort considered a set of alternatives, “make” versus “buy,” and that included among others adoption of the IEEE Open Firmware standard, use of the ARC platform standard, and of course building a solution from scratch. The Open Firmware standard offered a good technical solution but fell short in terms of business infrastructure for deployment in the time available while the ARC platform standard ended up being too prescriptive on platform design. Similarly other “buy” alternatives offered no clear path to deployment in the time available. Thus the decision was taken to pursue in-house development of a new mechanism.

“A high-level C language interface between platform firmware and the OS loader seemed like a natural.”

A high-level C language interface between platform firmware and the OS loader seemed like a natural for Itanium Processor machines given the complexity of low level programming and the desirability of having the OS know as little about the platform hardware specifics as possible in advance

of being able to load OS drivers. Having made that leap, it was a short hop from there to imagine a CPU-architecture-neutral API for firmware and OS communication for the boot process.

An Abstract Interface to Promote Innovation for OS and Firmware

The value of having such an interface and having it be broadly applicable to computers in general was driven home by experience on IA-32 platforms: there, the OS historically had hard-coded assumptions about the presence and operation of platform hardware devices and intimate familiarity with internals of many parts of system BIOS. All these factors on IA-32 discouraged change and made innovation tricky and relatively expensive. Obviously a successor technology that alleviates the need for the platform and OS to share intimate details for their respective implementation would provide an opportunity to decouple the rate of innovation for both the platform and the operating system.

This set of realizations led to the first big scope increase for the fledgling firmware interface. By taking on board the CPU-ISA-neutral approach, the scope of the program could increase to cover more of the Intel Processor family. Even in the late 1990s it was apparent that the conventional BIOS used on IA-32 computers was starting to become something of a drag on innovation. The designers of the new firmware interface thus turned their attention to including the ability to boot IA-32 processors as well as the original mission. It was around this time in late 1999 that the IBI program produced initial specifications for the new interface. The change in scope and the deliberate intention to foster a pro-innovation environment in the pre-OS space informed choice of the name for the new specification: the Extensible Firmware Interface (EFI)—neutral to any particular type of computer, deliberately describing only the interface (and explicitly not implementation of either producer [BIOS] or consumer [OS]), and calling out the idea that the interface would be a baseline for future additions.

EFI was adopted for the very first generation of Itanium Processor based computers and has been the boot interface there ever since. The initial version of the EFI specification 1.02 was published by Intel in December 2000 covering the operational scope needed to transfer control from platform firmware to the operating system. From the outset Intel kept the barriers to adoption for EFI as low as practical with little if any licensing restrictions and royalty-free sample implementation code. The principle of low barrier to adoption remains central to the management of the technology right up to present day.

To that initial publication the EFI Specification 1.10 was added in late 2002. This updated specification added a firmware driver model that addressed the problem of using add-in card devices in the boot process and providing code to operate those without requiring changes to the operating system boot loaders per device. In essence this provided a path to replace the fragile system of 16-bit option ROMs first advanced for ISA bus devices and later adopted for PCI boot devices as well.

“Even in the late 1990s it was apparent that the conventional BIOS used on IA-32 computers was starting to become something of a drag on innovation.”

“EFI was adopted for the very first generation of Itanium Processor based computers and has been the boot interface there ever since.”

“a group of industry stakeholders comprising BIOS vendors, OS vendors, system manufacturers, and silicon production companies agreed to form the Unified EFI Forum.”

“In addition to implementations of the various specifications, the Forum has also promoted the creation of test suites both for the UEFI Specification and for the PI Specification.”

Industry Backing: Advent of the Unified EFI Forum

Adoption on the IA-32 family would follow gaining momentum slowly but by early 2005 business conditions and technical constraints made it clear that the conventional BIOS technology would eventually run out of steam. In recognition of the fact that becoming a critical piece of the infrastructure for delivering IA-32 platforms to market is a multilateral industry intercept, a group of industry stakeholders comprising BIOS vendors, OS vendors, system manufacturers, and silicon production companies agreed to form the Unified EFI Forum in mid-2005 and the long-term home and governance model for this technology.

Intel contributed the EFI Specification as a starting point for the new Forum's work and the founding Promoter members worked in a truly unified fashion to produce a specification with broad industry support and endorsement. This initial publication from the Forum, the UEFI Specification 2.0, was published in January 2006.

In parallel with work on the interface between firmware and operating system, the Forum agreed to take on work to standardize interfaces for the internal construction mechanisms within an implementation of the UEFI Specification. This work led to the publication of the Platform Initialization (PI) Specification 1.0 in October 2006. This five-volume set aims to make it possible for silicon component support firmware to work unmodified with firmware on platforms developed by a variety of system building companies, simplifying and shortening deployment work for new product generations. The latest version of the PI Specification is version 1.1 published in February 2008.

The Forum continued to build consensus around updates to the UEFI Specification, publishing version 2.1 in January 2007. Among other things this introduced infrastructure that results in more graphical, better localized user interfaces for the pre-OS space.

Version 2.2 came along in September 2008 introducing IPv6 support for networking and also improved platform security primitives. Version 2.2's reign as the latest/greatest was relatively short-lived, however, largely as a result of work in the implementation world behind the specification.

Open Source Firmware Implementation

Intel had initially made available open source sample implementations of the original EFI Specification. That work continued as the EFI Specification evolved into the UEFI Specification and also delivered an implementation of the PI Specifications. This implementation found a permanent home as the EFI Developers Kit open source project still housed at www.TianoCore.org. This is known as the EDK for historical reasons although today of course the implementation conforms to the UEFI Forum's Specifications in its EDK II (second generation) form.

In addition to implementations of the various specifications, the Forum has also promoted the creation of test suites both for the UEFI Specification and for the PI Specification. These tests are designed to help developers build high

quality implementations of the specifications and are yet another example of the philosophy of making UEFI an easy technology to adopt in this case by making useful tools freely available for developers.

Completing the Specification Picture

As commonly happens with open source projects, interested parties come along and find new and interesting ways to use the code. In the case of the EDK, several companies found that the code was useful on ARM based platforms. Following successful ports to ARM platforms, it was proposed to add an ARM binding for the UEFI Specification. This was completed by the Forum in May of 2009 leading to the publication of the 2.3 version of the specification.

The 2.3 version of the Specification represents an interesting milestone for the community around Intel Architecture firmware. That specification represents the first point in time where all the interfaces for the boot process are written down in a formal document with industry-wide agreement on the content.

Looking Forward

The most recent version of the UEFI Specification is now 2.3.1, which as the name suggests, is an incremental release based on the 2.3 version. The new areas refine support for scalable platform security solutions and help to support faster and more sophisticated look and feel for the boot process.

With the state of the specifications now caught up to the present day platform design needs, attention is turning to driving technology forward to improve and expand capabilities in the pre-OS space. One of the first such efforts, radical reduction in boot time, may seem counterintuitive in that frame—the innovation is in fact to do less not more in the pre-OS space. However, this clearly represents a step forward in terms of appeal to the market as a whole and it is equally something that depends in large part on the abstraction of firmware and OS implementation from each other that is integral to the UEFI design—each part of the implementation of boot, firmware platform component initialization, and operating system startup can be optimized to work best with each other, yielding significant improvements overall.

UEFI technology is already in widespread use, in everything from smart phones to printers, notebooks, servers, and even supercomputers. There are new devices and platform technologies in prospect that will benefit from easier enabling through UEFI shortening time to market. There are new types of platforms like system-on-chip starting to adopt UEFI technologies for infrastructure in new product categories. In short, UEFI technology is helping to power the leading edge of compute platform innovations backed by broad industry collaboration for deployment and support.

“With the state of the specifications now caught up to the present day platform design needs, attention is turning to driving technology forward to improve and expand capabilities in the pre-OS space.”

“UEFI technology is already in widespread use, in everything from smart phones to printers, notebooks, servers, and even supercomputers.”

“UEFI is a pure interface specification that does not dictate how the platform firmware is built; the firmware is built; the “how” is relegated to PI.”

Where Does This Fit in the Ecosystem?

When we discuss UEFI, we need to emphasize that UEFI is a pure interface specification that does not dictate how the platform firmware is built; the “how” is relegated to PI. The consumers of UEFI include but are not limited to operating system loaders, installers, adapter ROMs from boot devices, pre-OS diagnostics, utilities, and OS runtimes (for the small set of UEFI runtime services). In general, though, UEFI is about *booting*, or passing control to a successive layer of control, namely an operating system loader, as shown in Figure 1. UEFI offers many interesting capabilities and can exist as a limited runtime for some application set, in lieu of loading a full, shrink-wrapped multi-address space operating system like Microsoft Windows*, Apple OS X*, HP-UX*, or Linux, but that is not the primary design goal.

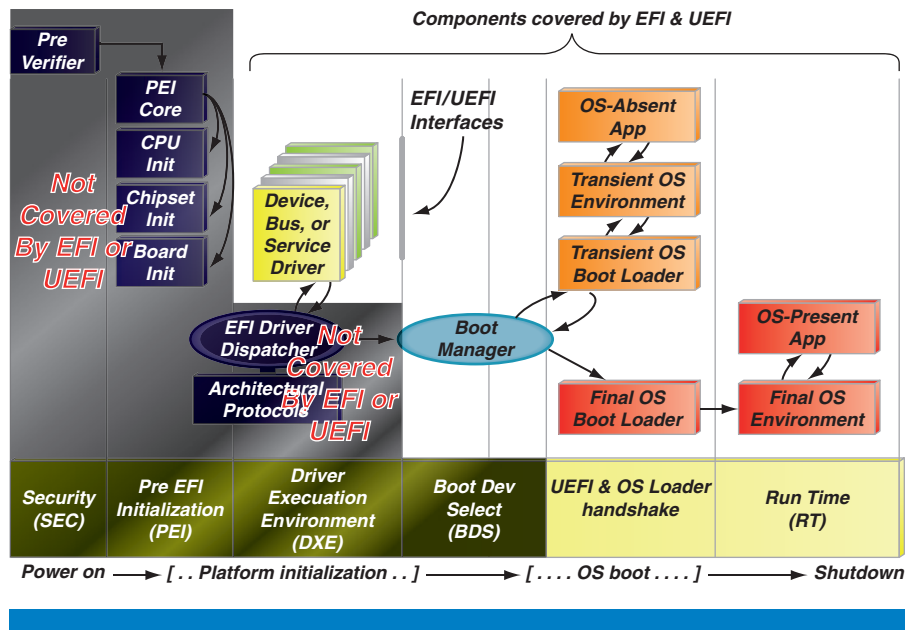


Figure 1: Where EFI and UEFI Fit into the Platform Boot Flow
(Source: Intel Corporation, 2010)

PI, on the other hand, should be largely opaque to the pre-OS boot devices, operating systems, and their loaders since it covers many software aspects of platform construction that are irrelevant to those consumers. PI instead describes the phases of control from the platform reset and into the success phase of operation, including an environment compatible with UEFI, as shown in Figure 2. In fact, the PI DXE component is the preferred UEFI core implementation.

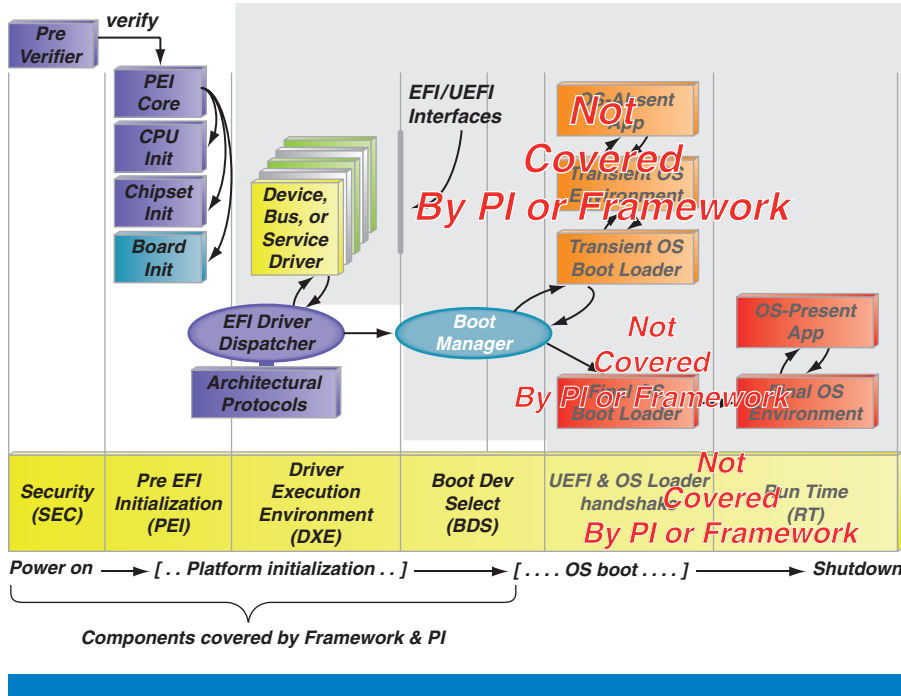


Figure 2: Where PI and Framework Fit into the Platform Boot Flow
 (Source: Intel Corporation, 2010)

Within the evolution of Framework to PI, some things were omitted from inclusion in the PI specifications. Specifically, the CSM specification abstracted booting on a PC/AT system. This requires an x86 processor, PC/AT hardware complex (for example, 8254, 8259, RTC). The CSM also inherited other conventional BIOS boot limitations, such as the 2.2-TB disk limit of Master Boot Record (MBR) partition tables. For a world of PI and UEFI, you get all of the x86 capabilities (IA-32 and x64, respectively), ARM*, Itanium®, and future CPU bindings. Also, via the polled driver model design, UEFI APIs, and the PI DXE architectural protocols, the platform and component hardware details are abstracted from all consumer software. Other minor omissions also include data hub support. The latter has been replaced by purpose-built infrastructure to fill the role of data hub in Framework-based implementations, such as SMBIOS table creation and agents to log report status code actions.

What has happened in PI beyond Framework, though, includes the addition of a multiprocessor protocol, Itanium E-SAL and MCA support, the above-listed report-status code listener and SMBIOS protocol, an ACPI editing protocol, and an SIO protocol. With Framework collateral that moved to PI, a significant update was made to the System Management Mode (SMM) protocol and infrastructure to abstract out various CPU and chipset implementations from the more generic components. On the DXE front,

“Some additions occurred in the PEI foundation for the latest evolution in buses, such as PCI Express.”*

“Given that there’s never enough ROM space, and also in light of the customer requirements for boot time such as the need to be “instantly on,” this overhead must be balanced by the business value of PI module enabling.”

“There is a large body of Framework-based source-code implementations, such as those derived or dependent upon EDK I (EFI Developer Kit version 1), which can be found on www.tianocore.org.”

small cleanup was added in consideration of UEFI 2.3 incompatibility. Some additions occurred in the PEI foundation for the latest evolution in buses, such as PCI Express*. In all of these cases, the revisions of the SMM, PEI, and DXE service tables were adjusted to ease migration of any SMM drivers, DXE drivers, and PEI module (PEIM) sources to PI. In the case of the firmware file system and volumes, the headers were expanded to comprehend larger file and alternate file system encodings, respectively. Unlike the case for SMM drivers, PEIMs, and DXE drivers, these present a new binary encoding that isn’t compatible with a pure Framework implementation.

The notable aspect of the PI is the participation of the various members of the UEFI Forum, which will be described below. These participants represent the consumers and producers of PI technology. The ultimate consumer of a PI component is the vendor shipping a system board, including multinational companies such as Apple, Dell, HP, IBM, Lenovo, and many others. The producers of PI components include generic infrastructure producers such as the independent BIOS vendors (IBVs) like AMI, Insyde, Phoenix, and others. And finally, the vendors producing chipsets, CPUs, and other hardware devices like AMD, ARM, and Intel would produce drivers for their respective hardware. The IBVs and the OEMs would use the silicon drivers, for example. If it were not for this business-to-business transaction, the discoverable binary interfaces and separate executable modules (such as PEIMs and DXE drivers) would not be of interest. This is especially true since publishing GUID-based APIs, marshalling interfaces, discovering and dispatching code, and so on take some overhead in system board ROM storage and boot time. Given that there’s never enough ROM space, and also in light of the customer requirements for boot time such as the need to be “instantly on,” this overhead must be balanced by the business value of PI module enabling. If only one vendor had access to all of the source and intellectual property to construct a platform, a statically bound implementation would be more efficient, for example. But in the twenty-first century with the various hardware and software participants in the computing industry, software technology such as PI is key to getting business done in light of the ever-shrinking resource and time-to-market constraints facing all of the UEFI forum members.

There is a large body of Framework-based source-code implementations, such as those derived or dependent upon EDK I (EFI Developer Kit version 1), which can be found on www.tianocore.org. These software artifacts can be recompiled into a UEFI 2.3, PI 1.2-compliant core, such as UDK2010 (the UEFI Developer Kit revision 2010), via the EDK Compatibility Package (ECP). For new development, though, the recommendation is to build native PI 1.2, UEFI 2.3 modules in the UDK2010 since these are the specifications against which long-term silicon enabling and operating system support will occur, respectively.

Terminology

The following list provides a quick overview of some of the terms that have existed in the industry associated with the BIOS standardization efforts.

- *UEFI Forum*. The industry body which produces UEFI, Platform Initialization (PI), and other specifications.
- *UEFI Specification*. The firmware-OS interface specification.
- *EDK*. The EFI Development Kit, an open sourced project that provides a basic implementation of UEFI, Framework, and other industry standards. It is not however, a complete BIOS solution. An example of this can be found at www.tianocore.org.
- *UDK*. The UEFI Development Kit is the second generation of the EDK (EDK II), which has added a variety of codebase-related capabilities and enhancements. The inaugural UDK is UDK2010, with the number designating the instance of the release.
- *Framework*. A deprecated term for a set of specifications that define interfaces and how various platform components work together. What this term referred to is now effectively replaced by the PI specifications.
- *Tiano*. An obsolete codename for an Intel codebase that implemented the Framework specifications.

Managing the Specifications in UEFI

Regarding the UEFI Forum, there are various aspects to how it manages both the UEFI and PI specifications. Specifically, the UEFI forum is responsible for creating the UEFI and PI specifications.

When the UEFI Forum first formed, a variety of factors and steps were part of the creation process of the first specification:

- The UEFI forum stakeholders agree on EFI direction
- Industry commitment drives need for broader governance on specification
- Intel and Microsoft contribute seed material for updated specification
- EFI 1.10 components provide starting drafts
- Intel agrees to contribute EFI test suite

As this had established the framework of the specification material that was produced and that the industry used, the forum itself was formed.

The UEFI Forum was established as a Washington nonprofit corporation. It develops, promotes, and manages evolution of Unified EFI Specification and continues to drive low barrier for adoption.

The UEFI Forum has a form of tiered membership: Promoters, Contributors, and Adopters. More information on the membership tiers can be found at www.uefi.org. The Promoter members for the UEFI forum are AMD, AMI, Apple, Dell, HP, IBM, Insyde, Intel, Lenovo, Microsoft, and Phoenix.

“the UEFI forum is responsible for creating the UEFI and PI specifications.”

“The UEFI Forum has several work groups.”

The UEFI Forum has several work groups. Figure 3 illustrates the basic makeup of the forum and the corresponding roles.

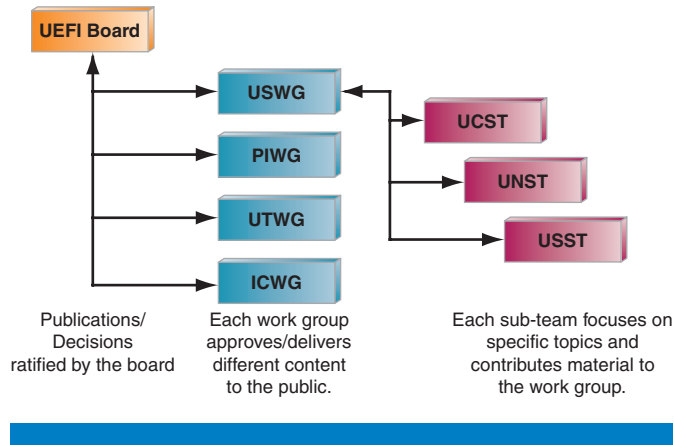


Figure 3: Forum Group Hierarchy
(Source: Intel Corporation, 2011)

Sub-teams are created in the main owning workgroup when a topic of sufficient depth requires a lot of discussion with interested parties or experts in a particular domain. These teams are collaborations amongst many companies who are responsible for addressing the topic in question and bringing back to the workgroup either a response or material for purposes of inclusion in the main working specification. Some examples of sub-teams that have been created are as follows as of this writing:

- UCST – UEFI Configuration Sub-team. Chaired by Michael Rothman (Intel), this sub-team is responsible for all configuration related material and the team has been responsible for the creation of the UEFI configuration infrastructure commonly known as HII, which is in the UEFI Specification.
- UNST – UEFI Networking Sub-team. Chaired by Vincent Zimmer (Intel), this sub-team is responsible for all network related material and the team has been responsible for the update/inclusion of the network related material in the UEFI specification, most notably the IPv6 network infrastructure.
- USST – UEFI Security Sub-team. Chaired by Tim Lewis (Phoenix), this sub-team is responsible for all security related material and the team has been responsible for the added security infrastructure in the UEFI specification.

PIWG and USWG

The Platform Initialization Working Group (PIWG) is the portion of the UEFI forum that defines the various specifications in the PI corpus. The UEFI Specification Working Group (USWG) is the group that evolves the main UEFI specification. Figure 4 illustrates the layers of the platform and shows the scope for the USWG and PIWG, respectively.

“The Platform Initialization Working Group (PIWG) is the portion of the UEFI forum that defines the various specifications in the PI corpus.”

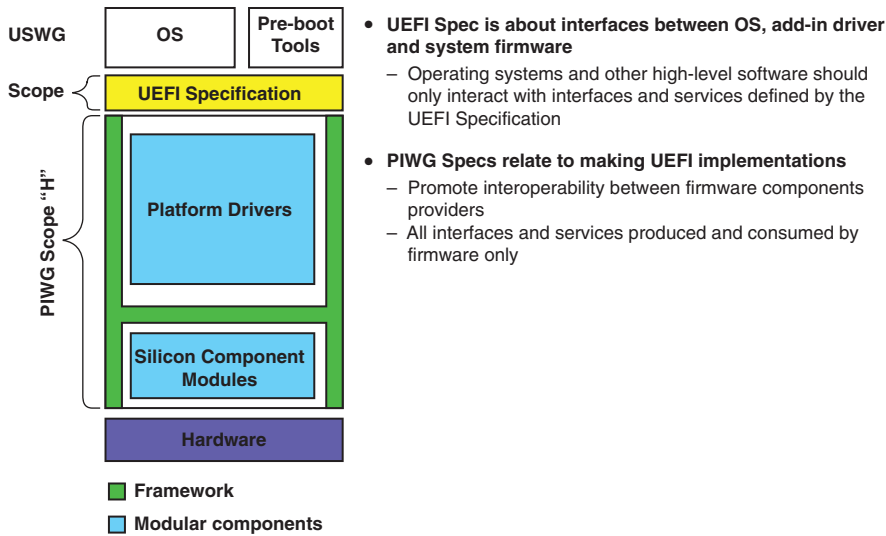


Figure 4: PI/UEFI Layering
 (Source: Intel Corporation, 2011)

Figure 5 shows how the PI elements evolve into UEFI. The left half of the diagram with SEC, PEI, and DXE are described by the PI specifications. BDS, UEFI+OS Loader handshake, and RT are the province of the UEFI specification.

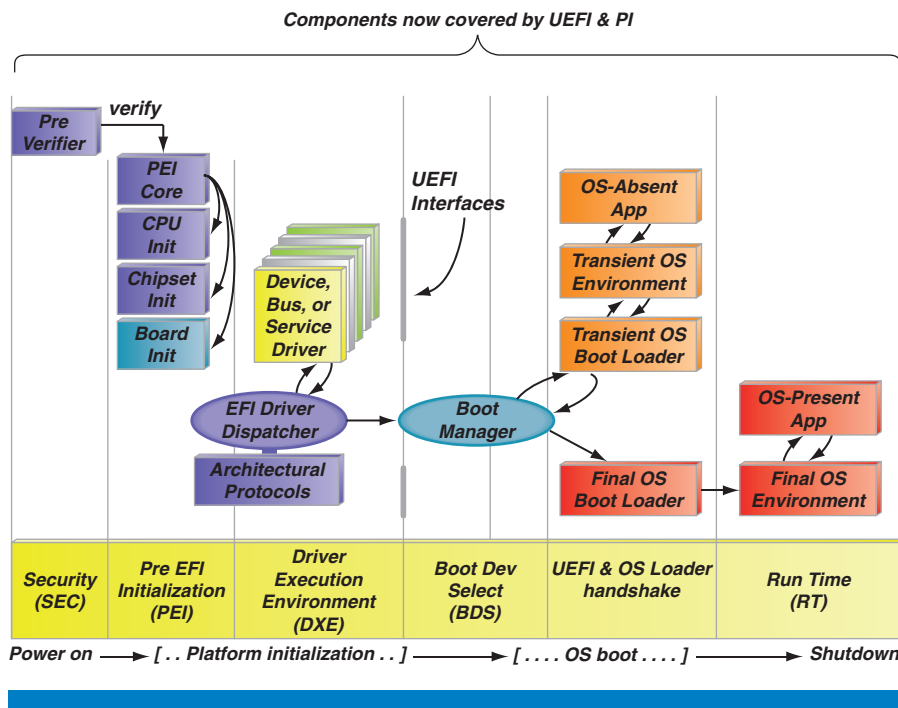


Figure 5: Where PI and Framework Fit into the Platform Boot Flow
 (Source: Intel Corporation, 2011)

In addition, as time has elapsed, the specifications have evolved. Figure 6 is a timeline for the specifications and the implementations associated with them.

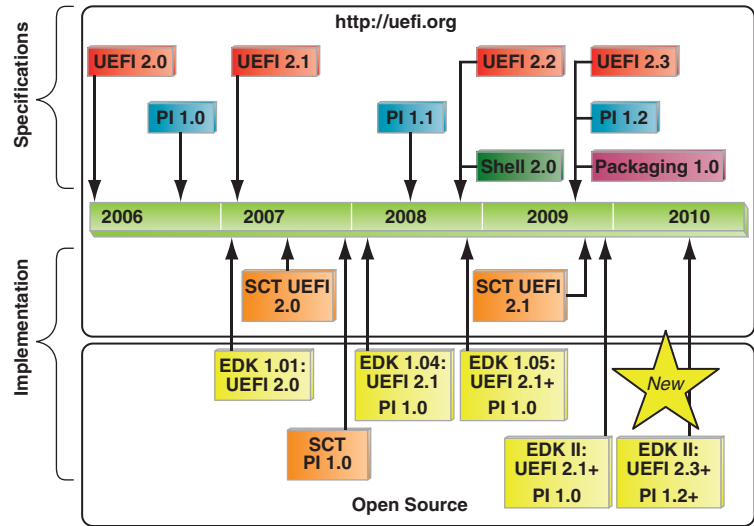


Figure 6: Specification and Codebase Timeline
(Source: Intel Corporation, 2011)

Platform Trust/Security

Recall that PI allowed for business-to-business engagements between component providers and system builders. UEFI, on the other hand, has a broader set of participants. These include the operating system vendors that built the OS installers and UEFI-based runtimes; BIOS vendors who provide UEFI implementations; platform manufacturers, such as multinational corporations who ship UEFI-compliant boards; independent software vendors who create UEFI applications and diagnostics; independent hardware vendors who create drivers for their adapter cards; and platform owners, whether a home PC user or corporate IT, who must administer the UEFI-based system.

“PI differs from UEFI in the sense that the PI components are delivered under the authority of the platform manufacturer and are not typically extensible by third parties.”

PI differs from UEFI in the sense that the PI components are delivered under the authority of the platform manufacturer and are not typically extensible by third parties. UEFI, on the other hand, has a mutable file system partition, boot variables, a driver load list, support of discoverable option ROMs in host-bus adapters (HBAs), and so on. As such, PI and UEFI present different issues with respect to security. Chapter 10 treats this topic in more detail, but in general, the security dimension of the respective domains include the following: PI must ensure that the PI elements are only updateable by the platform manufacturer, recovery, and that PI is a secure implementation of UEFI features, including security; UEFI provides infrastructure to authenticate the user, validate the source and integrity of UEFI executables, network authentication and transport security, audit (including hardware-based measured boot), and administrative controls across UEFI policy objects, including write-protected UEFI variables.

A fusion of these security elements in a PI implementation is shown in Figure 7.

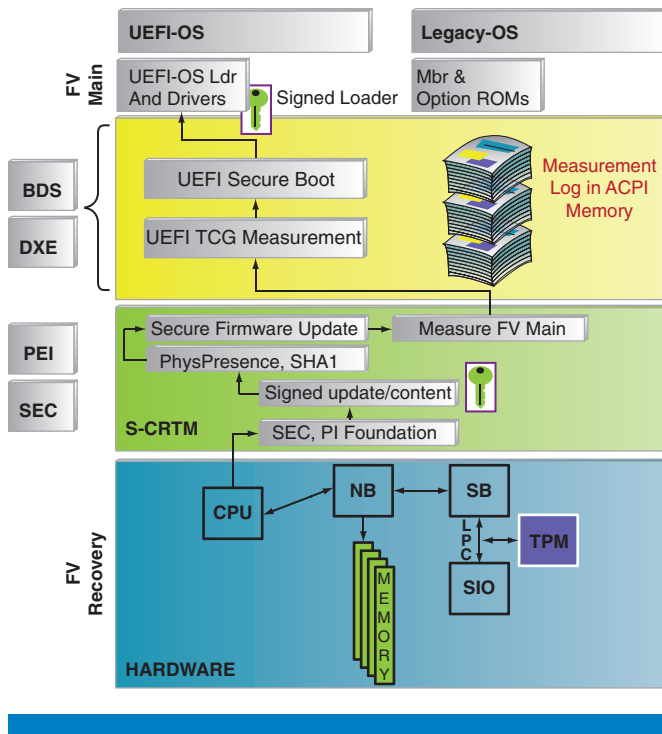


Figure 7: Trusted UEFI/PI stack
(Source: Intel Corporation, 2011)

Embedded Systems: The New Challenge

As UEFI took off and became pervasive, a new challenge has been taking shape in the form of the PC platform evolution to take on the embedded devices, more specifically the consumer electronic devices, which have a completely different set of requirements driven by user experience factors like instant power-on for various embedded operating systems. Many of these operating systems required customized firmware with OS-specific firmware interfaces and did not fit well into the PC firmware ecosystem model.

The challenge now is to make the embedded platform firmware have similar capabilities to the traditional model such as being OS-agnostic, being scalable across different platform hardware, and being able to lessen the development time to port and to leverage the UEFI standards.

How the Boot Process Differs between a Normal Boot and an Optimized/Embedded Boot

Figure 8 illustrates that, from the point of view of UEFI architecture, there are no design differences between the normal boot and an optimized boot. Optimizing a platform's performance does *not* mean that one has to violate any of the design specifications. It should also be noted that to comply with UEFI, one does not

“The challenge now is to make the embedded platform firmware have similar capabilities to the traditional model such as being OS-agnostic, being scalable across different platform hardware, and being able to lessen the development time to port and to leverage the UEFI standards.”

need to encompass all of the standard PC architecture, but instead the design can limit itself to the components that are necessary for the initialization of the platform itself. Chapter 2 in the *UEFI 2.3 Specification* does enumerate the various components and conditions that comprise UEFI compliance.

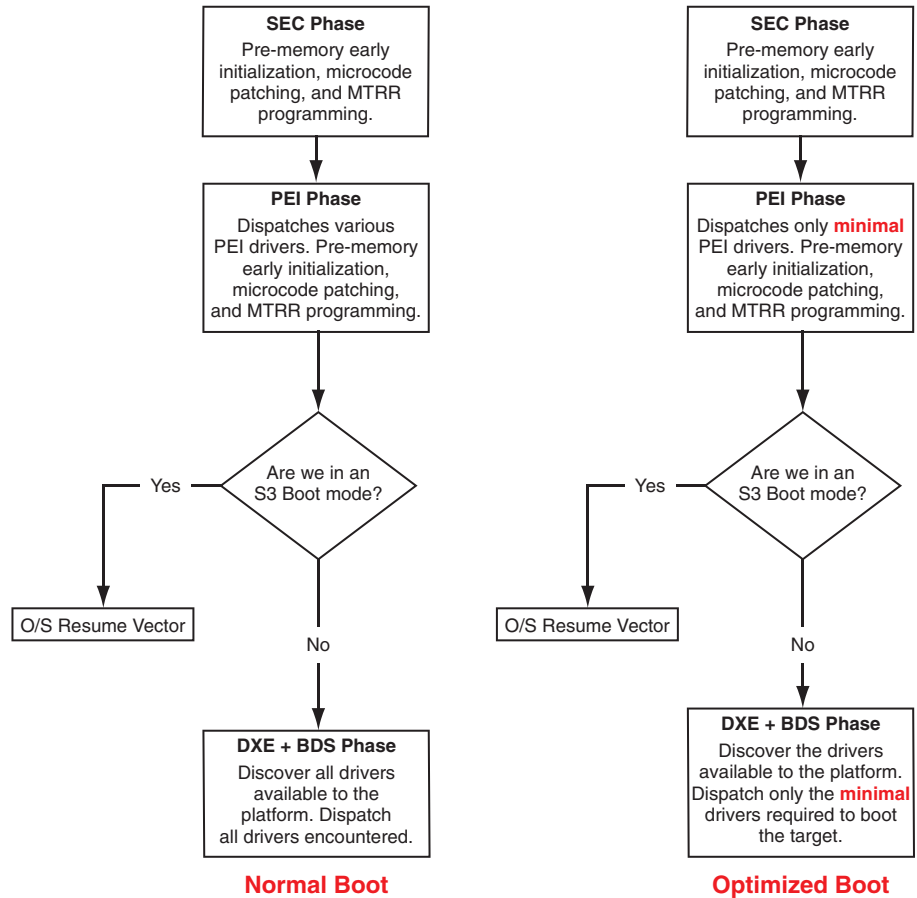


Figure 8: Architectural Boot Flow Comparison
(Source: Intel Corporation, 2011)

Summary

We have provided some background about the history that led to the creation of the BIOS standards that are developed today. In addition, we have hopefully provided some insight on how the UEFI forum operates and opened the door for people to understand how UEFI applies within their platform. Finally, we have given some pointers to the open source aspect of UEFI such that people can follow the evolution of the codebase technology to help realize implementations of this technology. As you read the other articles in this journal, you should see a very clear indication of some of the usage and capabilities exhibited by various members of the industry.

So fasten your seatbelt and dive into a journey through industry standard firmware.

Authors' Biographies

Mark Doran is a Senior Principal Engineer with Intel Corporation. He is Intel's lead architect for UEFI work. His prior work includes OS kernel development and IEEE POSIX standards content development. His first venture into standards for the firmware space was the Intel Multiprocessor Specification, the first recipe for building Intel Architecture symmetric multiprocessor computers that run shrink-wrap operating system binaries. Mark is originally from the UK and received a BSc in Computer Science with Electronic Engineering from University College, University of London.

Vincent J. Zimmer is a Principal Engineer in the Software and Services Group at Intel Corporation and has over 18 years experience in embedded software development and design, including BIOS, firmware, and RAID development. Vincent received an Intel Achievement Award and holds over 200 patents. He has a Bachelor of Science in Electrical Engineering degree from Cornell University, Ithaca, New York, and a Master of Science in Computer Science degree from the University of Washington, Seattle.

Can be contacted at <http://www.twitter.com/VincentZimmer> and vincent.zimmer@gmail.com

Michael A. Rothman is a Senior Staff Engineer in the Software and Services Group at Intel and has more than 20 years of operating system and embedded software development experience. Michael holds over 200 patents and was awarded an Intel Achievement Award for some of his systems work. He started his career with kernel and file system development in OS/2 and DOS and eventually migrating to embedded operating systems work and firmware development.

Can be contacted at <http://www.twitter.com/MichaelARothman> and michael.a.rothman@gmail.com

SILICON ENABLING IN A MODULAR ARCHITECTURE

Contributors

Isaac Oram

Intel Corporation

Tim Lewis

Phoenix Technologies

Vincent Zimmer

Intel Corporation

The Unified Extensible Interface UEFI Platform Initialization (PI) specifications allow for modular silicon enabling using defined building blocks. This includes system initialization, boot, and the unique requirements of the pre-boot space.

The adoption of UEFI specifications in the BIOS industry has reached critical mass in recent years. UEFI is now a component of firmware, operating systems, add-in devices, and other industry standards. Most products in the Intel® Architectures Personal Computer ecosystem are based on designs derived from the original EFI specifications and their current counterparts: the UEFI specification, and extensions such as the Intel® Platform Innovation Framework for EFI (Framework) and UEFI Platform Initialization (PI) specifications. These specifications have become the cornerstone for Intel silicon enabling in the Intel Architectures Personal Computer ecosystem.

This article will explore the use of the UEFI Platform Initialization (PI) specifications as a framework for silicon enabling. We will examine the building block elements provided for by the specification as well as the platform boot process, unique modes, and common uses. Additionally, we will examine drivers for different processors, memory and graphics controllers, and support chips.

Introduction

Personal computing has undergone a quiet revolution in the last five years, which was primarily driven by silicon enabling. As the industry evolved from personal computers (PCs) into many distinct product lines built on several architectures, an extensible and component-oriented industry standard firmware architecture was introduced and generally accepted into the BIOS environment. This revolution started for a number of reasons, but ultimately was driven by the need to initialize and utilize increasingly complex silicon and products in a cost-effective manner.

Definition: Silicon enabling can be defined as the activities required for OEM to deliver products to market utilizing a particular set of silicon products. This goal can be met through delivery of silicon specifications, reference code, sample code, binary modules, default settings, and the like.

Industry standards typically arise from a distinct need. Even with an identified need an industry standard relies on more than simply being documented in order to be successful. The “generally accepted extensible industry standard

“Definition: Silicon enabling can be defined as the activities required for OEM to deliver products to market utilizing a particular set of silicon products. This goal can be met through delivery of silicon specifications, reference code, sample code, binary modules, default settings, and the like.”

architecture” in this case is the UEFI Forum’s Platform Initialization (PI) architecture and is effectively realized as a standard through five vectors:

- It is documented in the UEFI Platform Initialization Specification. This provides a binary interoperable component architecture that is predominately applicable for silicon initialization prior to loading an operating system (OS). It is commonly thought of as a BIOS construction architecture specification.
- It is publicly instantiated in two open source software development environments, the EFI Development Kit (EDK) and EDK II; a common location for these kits is the website www.tianocore.org. These provide widely available and readily reusable implementations that module developers can utilize as a reference for building products that conform to the various UEFI and PI specifications.
- It is instantiated in BIOS products as a de facto standard implementation through a concept known as the Intel Green H, which is fundamentally a specific version of interface header files (corresponding to industry standards), library implementations, and some core modules. This provides a means to deliver source code into multiple BIOS codebases with minimal integration required.
- It is testable utilizing the UEFI published Self-Certification Tests (SCT). The SCT exercise the interfaces defined in the various UEFI specifications.
- It is used by leading hardware, software, and systems companies and thus is effectively required in the marketplace.

A BIOS software standard has the special constraints of having to support many different activities, most germane to this discussion are hardware initialization and system debug. These special constraints necessitate both source code and binary module compatibility and interoperability. These are required in order to effectively realize a component-oriented architecture used by a multitude of companies across the industry that support the Intel Architectures Personal Computer ecosystem. In the case of PI, binary compatibility is supported through the specifications and tests. Source compatibility is supported through the EDK and Intel Green H. Widespread requirement is realized through the support of BIOS, hardware, and software vendors throughout the industry.

In order to explore the modern silicon enabling environment, it is necessary to discuss the PI definition itself, the extensive silicon initialization performed within this infrastructure, the opportunities that are available, examples of the relevance, and the future opportunities not yet realized.

Platform Initialization Architecture

What is a BIOS? The term BIOS stands for basic input/output system. BIOS is a class of firmware that runs on the in-band processors or CPUs of a system in order to initialize the platform hardware complex and pass control to an

“A BIOS software standard has the special constraints of having to support many different activities, most germane to this discussion are hardware initialization and system debug.”

“In the case of PI, binary compatibility is supported through the specifications and tests.”

“The original PC/XT had an 8-KB BIOS that initialized the system and passed control to DOS.”*

“In UEFI, there is a separate set of standards referred to as the Platform Initialization (PI) standards (UEFI PI Specification).”

operating system. For purposes of the security architecture mentioned in the earlier chapter, it is the “firmware” layer.

The original PC/XT had an 8-KB BIOS that initialized the system and passed control to DOS*. This was 1982. Since that time, the BIOS domain has evolved significantly, including the transition of the industry to the Unified Extensible Firmware Interface (UEFI).

We will refer to the original BIOS as the *conventional BIOS* and the UEFI-based boot code as *UEFI*. Conventional BIOS and UEFI (UEFI Specification) based-systems must carry out several roles. First, each has the concept of platform initialization. This phase is the code that commences execution immediately after a platform restart (S3, S5, and so on). In a conventional BIOS, this is a vendor-specific flow and construction, but is sometimes referred to as the *stackless assembly or boot-block code*. In UEFI, there is a separate set of standards referred to as the Platform Initialization (PI) standards (UEFI PI Specification). In a PI-based platform initialization, the SEC and PEI phases commence this early execution.

The temporal evolution of a UEFI PI-based boot is shown in Figure 1.

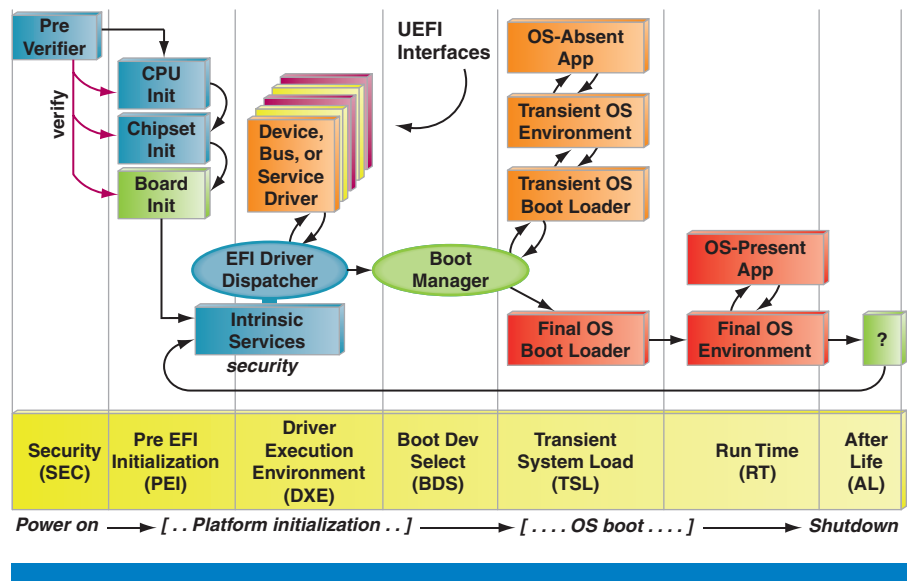


Figure 1: UEFI PI boot flow
(Source: Intel Corporation, 2011)

Afterward, each platform needs to discover I/O buses, dispatch option ROMs from host-bus adapter cards, and so on. In conventional BIOS, this I/O enumeration happens in the power-on-self test (POST) phase. There is no real standard for POST on a conventional BIOS. For UEFI PI-based firmware, though, this phase of execution occurs in the Driver Execution Environment (DXE).

DXE also serves as the UEFI core for purposes of supporting UEFI-based operating systems.

After BIOS POST and DXE, though, the “standard” part of the interface to the platform appears. For a PC/AT BIOS, the standard includes the de facto interrupt-callable interface (for example, Int13h for disk, Int10h for video) that executes in 16-bit real mode on the x86 architecture. For UEFI, this option ROM and loader interoperability includes the UEFI boot services and protocols (for example, EFI_BLOCK_IO_PROTOCOL as analog to BIOS int13h) and is described by the UEFI specification. It is during this phase of execution where third party content can appear from disk or adapters that did not necessarily ship with the platform manufacturer’s (PM) system board.

This taxonomy above is critical because the transition from an execution regime provided by the PM into a space where third party codes can run has implications on the construction of a trusted platform.

Figure 2 shows the generic BIOS initialization flow. This flow includes the initialization of the platform CPU, memory, and I/O devices during POST. The POST flow again is analogous to the DXE flow in Figure 1.

“the transition from an execution regime provided by the PM into a space where third party codes can run has implications on the construction of a trusted platform.”

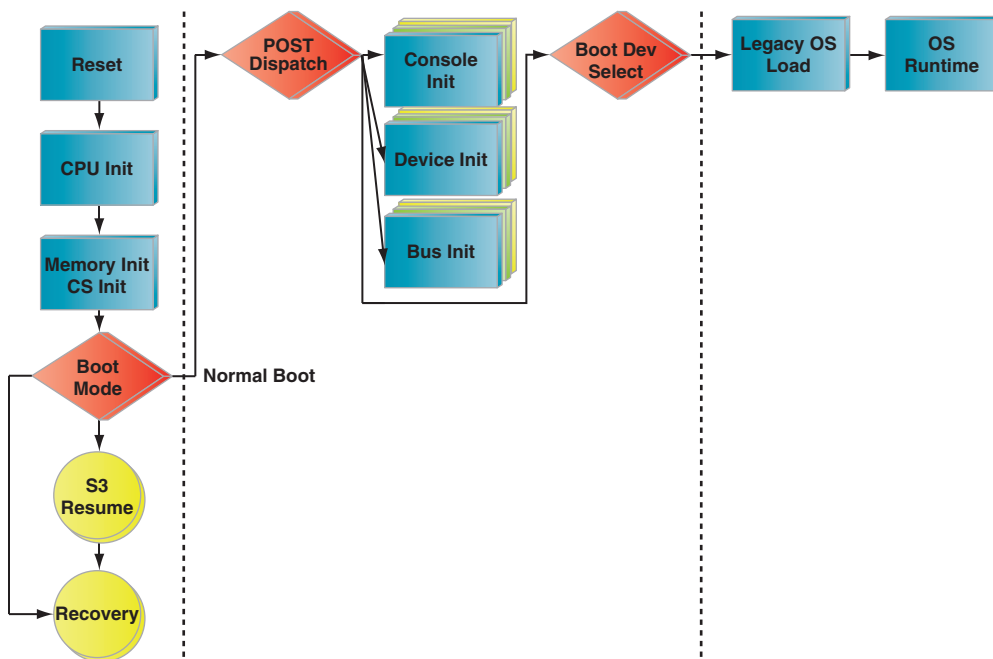


Figure 2: High-level BIOS flow
(Source: Intel Corporation, 2011)

UEFI Firmware

Above the hardware layer of the security architecture is the firmware, as shown in Figure 3.

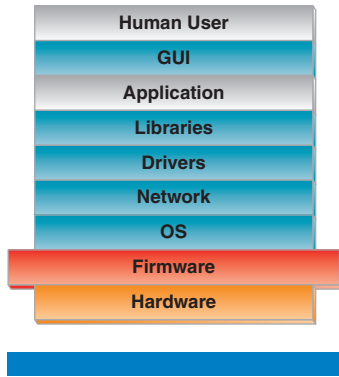


Figure 3: Firmware layer of the security architecture
(Source: Intel Corporation, 2011)

Since there are several code-base implementations of platform firmware today and possibly in the future, this discussion will be held in the context of the Platform Initialization (PI) standards and some representative codebases thereof.

As a backgrounder, the Platform Initialization Working Group of the UEFI Forum delivers the Platform Initialization Architecture Specifications, based on Intel PEI and DXE specifications. The Platform Initialization Architecture Specifications are independent of the UEFI 2.0 Specification. The PI Architecture platforms can still boot today’s operating systems. AMD, AMI, Apple, Dell, HP, IBM, Insyde, Intel, Lenovo, Microsoft, and Phoenix own the specifications. The goal of the Platform Initialization Working Group is to allow silicon vendors who create “reference code” today to package this reference code as modules that snap into PI Architecture firmware implementations. Figure 4 illustrates the PI scope.

What About Firmware Practices?
UEFI PI Overview

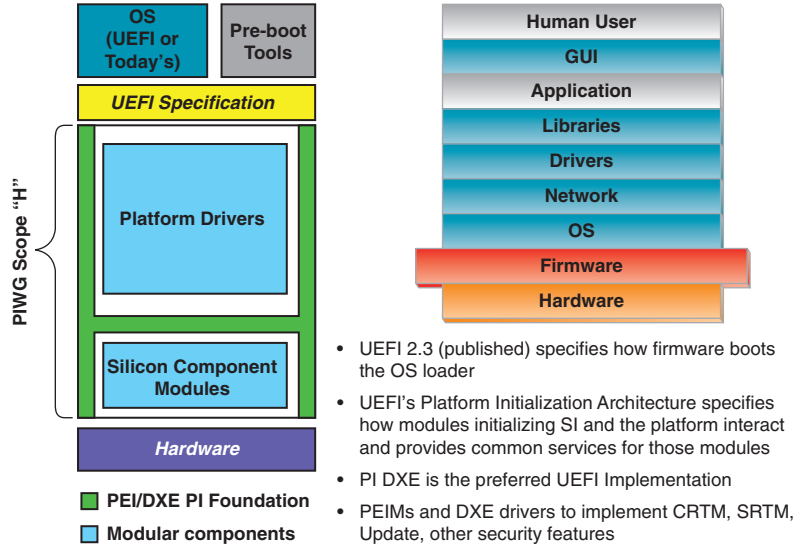


Figure 4 UEFI PI
(Source: Intel Corporation, 2011)

UEFI and PI specifications only describe normative material around interface definition and mechanism, but no informative content, such as why and how. The latter is intended to be the purview of design guides.

More information on the UEFI PI Specifications can be found at the UEFI Web site.[1]

In contrast to the monolithic nature of a BIOS, the UEFI PI allows hardware agility via software extensibility by exposing a driver model, such as dependency-expression–based PEI Modules (PEIMs) and DXE drivers. Extensibility points can serve as a point of attack for malware. This type of

“In contrast to the monolithic nature of a BIOS, the UEFI PI allows hardware agility via software extensibility by exposing a driver model.”

malware is essentially undetectable by OS-hosted protection technologies, such as antivirus software, since the malware could execute well before the OS. Consequently, the security integrity foundation of the pre-OS boot environment provided by UEFI and other pre-OS extensibility must be solid, while still providing enough flexibility to support hardware agility.

The PI phase is intended to only be extensible by the platform manufacturer (PM), not third party (in contrast to UEFI and its option ROM/loader/driver model). The installation and behavior of code in this early PI flow is said to act under the authority of the platform manufacturer; this will be referred to as “PM_AUTH” below.

Preservation of this PI as PM-extensible-only-intent in both construction and survivability in the field is a goal of the system design.

Figure 5 describes the UEFI PI boot flow, including annotation of the PM-extensible-only PI code and the third party extensible UEFI.

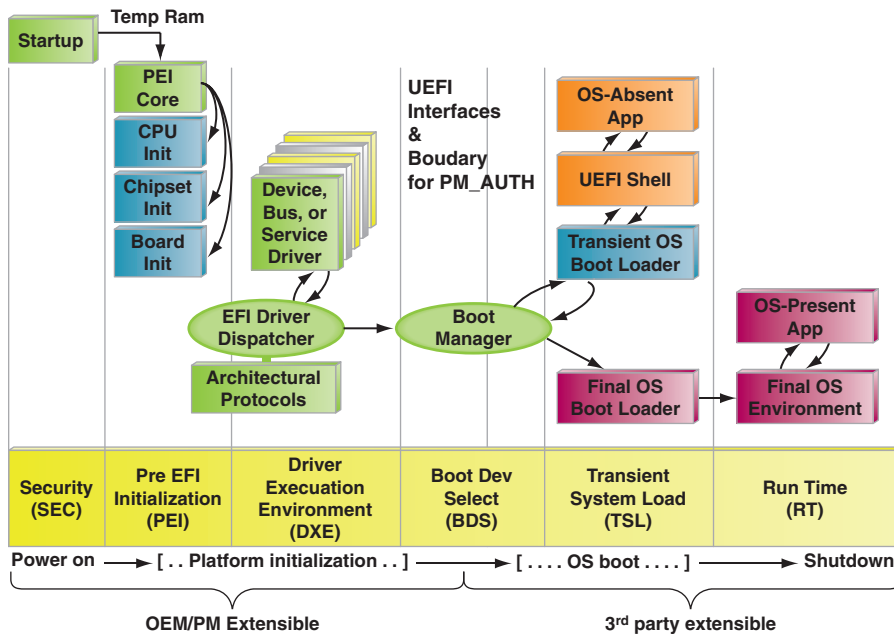


Figure 5: UEFI PI boot flow
(Source: Intel Corporation, 2011)

Silicon Initialization

The UEFI Platform Initialization or Intel® Platform Innovation Framework for EFI (Framework) specifications provide a software environment that allows a silicon vendor to deliver nearly all required core silicon initialization. For the purposes of this discussion, we are not distinguishing between the two sets of specifications, as they largely cover similar infrastructure for component development and interoperability.

“Software remains the most efficient method of initializing and supporting this increasingly complex hardware.”

“Core silicon initialization relies on a phased approach.”

Silicon initialization remains required due to the costs and complexities associated with initialization in hardware and the relative inflexibility of hardware as opposed to software. In the last 15 years, we have seen processors grow from <10 to >800 million transistors.[2] Accompanying this growth was a range of increasingly complex features and capabilities. Software remains the most efficient method of initializing and supporting this increasingly complex hardware.

For the purposes of this discussion, we are focused on the critical components (core silicon) that are a part of every Intel architecture system: processors, memory controllers, graphics controllers, storage controllers, system bus controllers, IO controllers, and the like. There are a host of additional devices that comprise a modern Intel architecture system, but their needs are primarily met by the UEFI Specification. Additionally, these devices tend to be initialized later in the boot process with more reliance on other industry standards, such as USB, PCIE, and ACPI. In some cases, BIOS plays a minimal role in their initialization.

Where Does Silicon Initialization Occur?

Core silicon initialization relies on a phased approach. At initial system reset, a very limited set of hardware resources is available; devices are inaccessible, memory is not available, and so forth. This leads BIOS to change the basic operating environment for software as hardware is initialized and resources expand. In PI:

- The system commences at reset (the first instruction fetched by the host bootstrap processor) in the SEC phase, where it makes memory available by initializing the processor cache and then transitions to PEI phase.
- The system starts PEI with a small amount of stack and heap available, and initializes enough hardware to have permanent memory available and then we transition to DXE phase.
- The system starts DXE with permanent memory, initializes core silicon, and then transitions to BDS phase.
- The system starts BDS with core silicon initialized and proceeds to initialize hardware required to boot an OS (input, output, and storage devices). At a high level, BDS corresponds to “executing the UEFI driver model” for the purposes of booting an OS.

There are special sub-phases in the flow:

- PEI pre-mem
- PEI post-mem
- SMM
- CSM

SEC	PEI (Pre-memory)	PEI (Post-Memory)	DXE	SMM	BDS	CSM
Native Processor Mode	Debug Infrastructure	Final Stack/Heap	MP Support	Code Infrastructure	PCI	Legacy OS interfaces
Initial Stack/Heap	Memory	Full Config (S3)	SMM	Protection mechanisms	USB	
Microcode Update	Processor/Chipset interfaces	Flash Update (Capsule)	Power/Terminal Management	Flash Write	SATA	
	Long Initialization devices	Cache Configuration	PCI		Graphics	
					Storage	

Table 1: Mapping of core SI to phases

As Table 1 shows, silicon initialization is spread throughout the boot process. The reasons are generally based on cost and complexity. As the boot progresses, the cost and complexity of initializing a set of silicon functionality goes down as more infrastructure becomes available. This is not to say that SMM initialization is not complex, but it is significantly less complex after permanent memory and DXE services are available. The cost stems from the need to execute directly from uncompressed FLASH memory prior to memory availability. The end result is that various core silicon initialization activities included in the table can be accomplished reasonably cheaply by PI modules and typically comprise greater than 90 percent of the silicon initialization required.

How Is Silicon Initialization Implemented?

In the Intel case, silicon initialization takes the form of a set of packages that correspond roughly to silicon products. These silicon reference packages may support a single product, multiple products, a subset of the features associated with a single product, or even a capability that spans multiple products. Examples include the platform power management (PPM) reference package, which has often supported multiple generations of silicon, the platform controller hub (PCH) reference package, which typically supports a single generation of silicon, and the integrated clock controller (ICC) reference package, which is only applicable when the PCH is used in a specific configuration. The complexity in the object model arises from the complexity of the overall platform as well as the variability in the use of the reference package. An example of the latter is that PPM reference package is often enabled later in the product development cycle, so packaging it separately from the “always necessary” processor code gives customers flexibility in developing their product.

Each package effectively contains dynamic linked libraries (DLLs) for the different BIOS environments, such as pre-memory (PEIM), UEFI boot

“As the boot progresses, the cost and complexity of initializing a set of silicon functionality goes down as more infrastructure becomes available.”

“The complexity in the object model arises from the complexity of the overall platform as well as the variability in the use of the reference package.”

services and runtime services (DXE), system management mode (SMM), and so on. These DLLs are usually delivered in source form, in a silicon reference package typically containing:

1. Source code
2. Custom interfaces as well as their documentation
3. Build files for use in an open source EDK
4. Sample code
5. Static libraries
6. Design and integration documentation

In order to reduce the number of supported configurations, the silicon reference packages are developed and tested against a specific version of industry standard interfaces and useful libraries, the Intel Green H. By defining the Intel Green H as a specific set of files, it is reasonable to deliver source code that has the attributes of being easily integrated and reusable without modification. By agreeing on such basics as the header files containing the services tables, common protocol structure definitions, and the like, a host of source code portability problems are avoided.

“By agreeing on such basics as the header files containing the services tables, common protocol structure definitions, and the like, a host of source code portability problems are avoided.”

While PI provides a rich set of basic services, and in some cases abstracts more complex services, it is not sufficient to cover all possible silicon features with industry standard interfaces. In order to address this, reference packages provide custom interfaces. These interfaces typically take the form of variables, HOB, PPI, and protocols, as well as dependency expressions, callbacks, and other UEFI and PI services. With this toolbox available, silicon vendors can provide rich services to abstract silicon initialization. In the current silicon reference packages, it is common for there to be a “policy” protocol allowing the consumer to pass in board and design specific parameters. In other cases, interfaces, such as the I/O trap protocol, are provided. These services allow even the most complex silicon feature implementation to be shared between the producer and the consumer.

In total, with a silicon reference package that delivers the code and the Intel Green H giving it a “socket” to plug into, the consumer can build the modules using a widely available EDK and then focus their efforts on integrating into their BIOS build environment and enabling the various features of the silicon reference package in their BIOS. This allows the silicon vendor to deliver silicon initialization that is reused in widely divergent BIOS codebases that have unique requirements and attributes.

“This allows the silicon vendor to deliver silicon initialization that is reused in widely divergent BIOS codebases that have unique requirements and attributes.”

Silicon Enabling: Changing the Role of BIOS

The advent of the UEFI PI standards represents a watershed moment for the industry.

Implement Once

PI has allowed core silicon initialization to be developed by the silicon provider and used everywhere that silicon is used. Previously, core silicon initialization was done by many different companies based on the limited documentation available for said silicon (SI). The downside of this set of behaviors is that the various implementations of said SI initialization by the SI consumers may deviate from the intent of the SI producers. Also, the validation effort applied internally by the SI producer towards the SI producer's implementation of the initialization code does not directly map to the results of the SI consumer; the latter will have its own SI initialization implementation and possibly different system board design.

Deployment

With many of the SI consumers aligning system board designs closely with the SI producers because of sensitive analog signaling and layout considerations, the point of variability more becomes the SI initialization code. As noted above, the SI producer creates SI initialization code to validate the SI internally. If the SI producer and SI consumer both support the UEFI PI specification for their system board firmware, the SI producer can release its SI initialization modules at the same time the physical SI components are released. This allows for a deployment model wherein there is no time delay between “hardware and firmware” with respect to deployment.

Integration

The advent of systems supporting the UEFI and UEFI PI specifications is changing the relationship and responsibilities of the players in the PC firmware space.

In the “old” days (before UEFI and PI), the silicon vendor produced reference source code that showed the important initialization and configuration steps for initializing their specific chip or chips. The BIOS vendor took that source code, modified it for their build system, and hooked up each of the reference code fragments to their code base. Rolling out bug fixes to all customers was time-consuming because each one required an integration step that was unique to the target code base. Quality assurance for the BIOS focused on functional test or homegrown API testing, because there were few well-defined APIs. The only standard means of extending the BIOS was the option ROM, which was essentially unchanged for 20 years.

As time went by, many silicon vendors began to create a “core” source code package for their chips, as well as a “plug-in” layer for each of the BIOS code bases that they supported. This eased some of the problems for the silicon vendor, because now the single change they made to the “core” would work in all of the target code bases. But it created a problem for the BIOS vendors. Each silicon vendor was creating a unique “plug-in” style, with its own quirks, often favoring the BIOS code base that the silicon vendor used internally.

“allows for a deployment model wherein there is no time delay between “hardware and firmware” with respect to deployment.”

“Now each silicon vendor could package the support for their chips into drivers.”

With UEFI and then, later, PI, there was a jump forward. Now each silicon vendor could package the support for their chips into drivers. The specifications defined how they were launched, how they published interfaces and how they discovered interfaces. Because of the standardization, there is no longer a separate “plug-in” layer for different BIOS vendor code bases. This has led to a decoupling of the code producer (silicon vendor) from the code consumer (BIOS vendor or OEM).

There have been a few consequences of this new model:

1. Write Once, Run Anywhere Drivers. If the BIOS complies with these specifications, a driver can be inserted in any BIOS and it should work.
2. Silicon-Vendor Produced Drivers. With UEFI, over time, the responsibility for all source code related to silicon support is shifting to the silicon vendor. Previously, despite the large amount of reference code given out in the “old” days, the BIOS vendor was still responsible for filling in the gaps.
3. Increased Testability. UEFI provides well-defined APIs for all services required for booting the system. The robustness of the implementation for these APIs is critical for insuring interoperability. Taking advantage of the well-defined APIs, the UEFI Testing Working Group has produced the Self-Certification Test (SCT) for each generation of the specifications.
4. Monolithic To Modular BIOS. More and more, the BIOS functions as a platform running a collection of drivers and applications, rather than a single body of code with a few strange appendages.
5. Drivers Offer Built-In Customization. In the “old” model, customization for a specific platform or product was handled by the OEM or BIOS vendor much later in the development cycle and often involved an insertion of hooks and flags directly into vendor-provided code. Now, the silicon vendors are inserting customizability into their drivers, using PI’s Platform Configuration Database (PCD), policy protocols, and EFI variables.

“Once the inner workings of the BIOS world were exposed via a public specification, many new ideas came forward.”

Once the inner workings of the BIOS world were exposed via a public specification, many new ideas came forward. The UEFI BIOS is gaining new capabilities because UEFI lowers the barrier to implementing new ideas that work on every PC.

Interoperability in Practice

The UEFI PI specification does not give you everything you need to build a PC platform. Many of the small silicon components—embedded controllers, super I/O controllers, flash devices—are represented in the specification but are not fully developed. However, the PI specification provides a solid foundation on which production-quality firmware can be built.

A good example of this can be found in the PI specification’s support for the System Management Bus (SMBus). Originally developed for supporting battery-management subsystems, this two-wire multi-master bus interface has

evolved into a standard motherboard side-band bus for sensor and platform management.[3] Since its introduction in 1995, it has become an integral part of other industry standards, including PCI, IPMI[4], DASH[5] and ASF[6].

The specification describes two APIs that abstract the SMBus host controller's low-level features: `EFI_PEI_SMBUS2_PPI` (for the PEI phase) and the `EFI_SMBUS_HC_PROTOCOL` (for the DXE phase). These APIs allow commands to be sent and received on the SMBus address without specific knowledge of the hardware interface.

```
typedef struct _EFI_SMBUS_HC_PROTOCOL {
    EFI_SMBUS_HC_EXECUTE_OPERATION      Execute;
    EFI_SMBUS_HC_PROTOCOL_ARP_DEVICE    ArpDevice;
    EFI_SMBUS_HC_PROTOCOL_GET_ARP_MAP   GetArpMap;
    EFI_SMBUS_HC_PROTOCOL_NOTIFY        Notify;
} EFI_SMBUS_HC_PROTOCOL;
```

Code 1. SMBus Host Controller Protocol.

(Source: UEFI Forum, Inc.)

This protocol interface structure has function pointers to different abstracted functions. *Execute* will send a command to a targeted SMBus device over the SMBus. *ArpDevice* and *GetArpMap* will handle the SMBus address-resolution protocol to assign unique addresses to SMBus devices and report the results. *Notify* allows other drivers to register for a callback when SMBus devices send event notifications.

For example, the following code fragment sends the Get UDID (directed) command and then prints the error message or else the device's Unique Device Identifier (UDID).

```
EFI_STATUS s;
EFI_SMBUS_DEVICE_ADDRESS DeviceAddr;
SMBUS_GET_UDID_DIRECTED_RESP GetUdidResp;
s = SmbusHc->Execute (
    SmbusHc,
    SMBUS_ADDR_DEV_DEFAULT_WRITE,
    DeviceAddr | 1,
    EfiSmbusReadBlock,
    TRUE,
    &DataSize,
    &GetUdidResp);
if (EFI_ERROR (s)) {
    printf ("Failed (%r). Skipped\n", s);
} else {
    DumpUdid (&GetUdidResp.Udid);
}
```

Code 2. SMBus Host Controller Protocol Execute() Example

(Source: Phoenix Technologies Ltd.)

“The caller is not required to know anything about how the SMBus Host Controller is implemented in hardware or, indeed, whether or not hardware is present at all.”

The caller is not required to know anything about how the SMBus Host Controller is implemented in hardware or, indeed, whether or not hardware is present at all.

Where’s the SMBus Bus?

So the PI specification provides access to the host controller. But where’s the bus? Where’s the SMBus device driver? In the UEFI driver model, support for an industry standard bus is usually broken into three drivers:

1. *Host Controller Device Driver.* This driver abstracts a specific type of host controller and produces a Host Controller Protocol to allow the attributes and features to be discovered and manipulated.
2. *Bus Driver.* This driver implements the requirements of the industry standard bus. It uses the Host Controller Protocol to enumerate the bus, finding all child devices, creating handles for them and installing an instance of the I/O Protocol on each.
3. *Device Driver.* This driver implements the requirements of a specific device on the bus. It uses the I/O Protocol to discover device features, send commands, and produce new protocols used by other drivers and applications.

This model is seen in the UEFI Specification with the PCI bus. The chipset drivers produce the PCI Root Bridge I/O Protocol (*Host Controller*). The PCI bus driver produces the PCI I/O Protocol (*I/O*). The graphics device drivers (for example) use the PCI I/O Protocol and produces the Graphics Output Protocol.

“The USB bus driver uses the USB Host Controller Protocol and produces the USB I/O Protocol (I/O).”

The USB bus follows a similar model. The PCI-XHCI drivers produce the USB Host Controller2 Protocol (*Host Controller*). The USB bus driver uses the USB Host Controller Protocol and produces the USB I/O Protocol (*I/O*). The USB keyboard driver (for example) uses the USB I/O Protocol and produces the Simple Text Input Protocol.

However, when we look back at what the PI specification describes, we can see that it provides just the basics: the Host Controller protocol. But where is the bus driver? Where is the I/O protocol? The subsequent sections talk about the methods that layer on top of the host controller itself so a consumer can talk to the necessary device.

The SMBus Driver Model

Phoenix’s SecureCore Tiano* builds on the strong foundation provided by the PI specification to create a full UEFI-style driver model. This allows SMBus device drivers to be started, stopped, and enumerated. It also allows DASH, ASF, and IPMI to be layered on top in a well-understood fashion.

For example, Figure 6 shows the ASF architecture layered on top of a driver producing the PI specification’s SMBus Host Controller protocol.

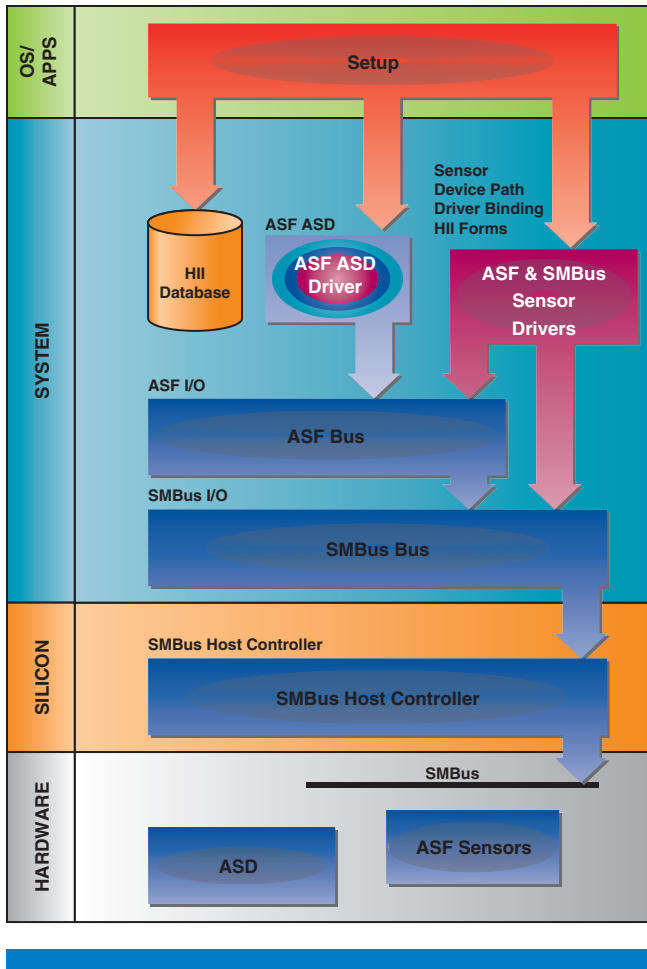


Figure 6: ASF/SMBus UEFI driver model
 (Source: Phoenix Technologies Ltd., 2011)

The SMBus Host Controller driver still occupies the critical role, abstracting communication across the SMBus. But a new driver, the SMBus Bus driver, enumerates all the devices on the bus, creates child handles for each of the discovered devices and installs an instance of the SMBus I/O protocol on each. This follows the UEFI driver model shown earlier for PCI and USB.

```
typedef struct _SCT_SMBUS_IO_PROTOCOL {
    UINT32 Size;

    SMBUS_ADDR Addr;

    SCT_SMBUS_IO_IDENTIFY Identify;
    SCT_SMBUS_IO_EXECUTE Execute;
} SCT_SMBUS_IO_PROTOCOL, *PSCT_SMBUS_IO_PROTOCOL;
```

Code 3. Secure Core Tiano (SCT) I/O Protocol
 (Source: Phoenix Technologies Ltd., 2011)

“The device drivers can, in turn, produce additional protocols and even setup pages.”

The *Execute* member function is a pass-through function to the SMBus Host Controller protocol for the host controller to which this device is attached. The new member function *Identify* is used to return the Unique Device Identifier (UDID), which provides the vendor identifier and device identifier (similar to PCI configuration space fields of the same name) and some capabilities flags.

This UDID is used by the *Supported()* and *Start()* member functions of the Driver Binding protocol produced by the ASF and SMBus device drivers.

The device drivers can, in turn, produce additional protocols and even setup pages. For example, sensor drivers can provide real-time temperature, voltage, or fan-speed values to the user.

ASF-Beyond SMBus

The Alert Standard Format is one of the remote platform management standards created by the Desktop Management Task Force. On the local PC platform, it uses the SMBus to communication between different ASF-compliant devices, such as NICs (or Alert Signaling Devices/ASDs), serial EEPROMs, and sensors. Each of these is an SMBus device, and more.

In Figure 6, the ASF bus driver examines each instance of the SMBus I/O protocol created previously to see if it is also an ASF-compliant device. It does this by examining the capabilities flags in the SMBus device’s UDID. From the *Supported()* function of the ASF Bus driver:

```
SMBUS_UDID Udid;
s = SmbusIo->Identify (SmbusIo, &Udid);
if (EFI_ERROR (s)) {
    DPRINTF_ERROR (“Could Not Return Device UDID.\n”);
    goto Done;
}

if ((Udid.Interface & SMBUS_UDID_INTERFACE_ASF) == 0) {
    s = EFI_UNSUPPORTED;
}

...install an instance of the ASF I/O protocol...
s = EFI_SUCCESS;
Done:
return s;
```

Code 4. ASF device identification
(Source: Phoenix Technologies, Ltd., 2011)

In this case, the protocol is installed on the same device handle, since it is referring to the same device. The ASF I/O protocol converts ASF-style commands into SMBus commands and sends them to the ASF device.

Going one step further, the Alert Signaling Devices (or ASD) drivers or ASF sensor drivers in Figure 6 examine each device handle to see whether it supports ASF I/O and its UDID matches a specific device and vendor identifier, or responds to a specific command.

Building on the UEFI PI Specification

The same idea described for ASF is used for other industry specifications, such as DASH and IPMI. Each adds a layer on top of the SMBus I/O, such as DASH's MCTP and PLDM or IPMI's system interface.

The UEFI PI specification describes the base level APIs necessary to create a fully functional SMBus driver stack. Using this, device drivers can be made to support individual SMBus devices and additional standards, such as ASF, DASH, and IPMI. Phoenix's SecureCore Tiano* has employed this similar model for other small silicon devices, including embedded controllers, Super I/O controllers, and flash devices.

Making the SMBus devices into UEFI driver-model devices also allows the developer to take advantage of the wealth of debugging and development tools available, including the UEFI Shell.[7]

“The UEFI PI specification describes the base level APIs necessary to create a fully functional SMBus driver stack.”

Future Directions

Going forward, the PI architecture allows for coordinated release of silicon and silicon initialization. Given the UEFI PI-defined interfaces and binary image format, the full advantages of implementations of the architecture can be realized. Specifically, the module implementations are decoupled from the code base implementation instance; the only criteria is a level of UEFI PI specification compatibility. The modules can be built as self-describing binaries that allow for SI consumers to do source or binary debug. And ultimately, the modules can be released only as binaries depending upon the support agreement or other business criteria. The UEFI PI also allows for automated integration and testing, wherein a new module can only demand testing related to its functionality, not a full testing regression regime.

“The modules can be built as self-describing binaries that allow for SI consumers to do source or binary debug.”

Finally, PI allows for more of an OS-like model. Just as there are well-defined buses and sockets for SI, PI provides the same pin-outs and sockets for “firmware.” Removing the time and effort to produce independent SI initialization modules will move the bar to providing more vendor-specific value added features on the platform that are end-customer visible, such as accelerated boot time or a rich graphical user interface. The advantages of UEFI PI become even more powerful for both the small and highly-integrated system-on-a-chip (SOC). The intent of UEFI PI specifications is to cover enough interfaces to “build real systems” but not so much as to make every implementation look the same or remove the opportunity to innovate and differentiate.

UEFI PI will continue to offer a robust execution environment for system board manufacturers to innovate and provide assurance around the implementation of UEFI and UEFI PI features.

“the UEFI PI provides a menu of items from the SI producer, BIOS vendor, and others who can provide content.”

And finally, the UEFI PI provides a menu of items from the SI producer, BIOS vendor, and others who can provide content. And it is from this menu that the system board vendors can choose elements that they need to build basic platform capabilities, freeing up their development resources to build differentiated added value while covering basic tasks from these menu items.

References

- [1] <http://www.uefi.org/specs/>
- [2] <http://www.intel.com/technology/timeline.pdf>
- [3] System Management Bus (SMBus) Specification, Version 2.0 (August 3, 2000) Copyright © 1994, 1995, 1998, 2000 Duracell, Inc., Energizer Power Systems, Inc., Fujitsu, Ltd., Intel Corporation, Linear Technology Inc., Maxim Integrated Products, Mitsubishi Electric Semiconductor Company, PowerSmart, Inc., Toshiba Battery Co. Ltd., Unitrode Corporation, USAR Systems, Inc.
- [4] Intelligent Platform Management Interface Specification, Version 2.0 (February 12, 2004, June 12 2009 Markup), Copyright © 2009 Intel Corporation, Hewlett-Packard Company, NEC Corporation, Dell Inc.
- [5] Management Component Transport Protocol (MCTP) SMBus/I2C Transport Binding Specification, Version 1.0.0 (July 28, 2009), Copyright © 2009 Distributed Management Task Force, Inc.
- [6] Alert Standard Format Specification, Version 2.0 (April 23, 2003), Copyright © 2000-2002 Distributed Management Task Force, Inc.
- [7] *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*, M Rothman, T Lewis, V Zimmer, R Hale (Intel Press, 2010)

Authors' Biographies

Isaac Oram is a platform firmware architect in the PC Client Group at Intel Corporation with 14 years of experience in research and development for notebook, desktop, server, and tablet product firmware.

Tim Lewis is the Chief BIOS Architect for Phoenix Technologies Ltd., responsible for the architecture of Phoenix's SecureCore Tiano* firmware for x86 and ARM processors. With over 24 years of firmware experience, Tim has worked on everything from compiler design to low-power-ASICs to firmware and Windows* application design. He represents Phoenix on the UEFI board of directors and to the ACPI 5.0 promoters. Tim has a B.A. in History from San Jose State and an M. Div. from Western Seminary.

Vincent J. Zimmer is a Principal Engineer in the Software and Services Group at Intel Corporation and has over 18 years experience in embedded software

development and design, including BIOS, firmware, and RAID development. Vincent received an Intel Achievement Award and holds over 200 patents. He has a Bachelor of Science in Electrical Engineering degree from Cornell University, Ithaca, New York, and a Master of Science in Computer Science degree from the University of Washington, Seattle.
<http://www.twitter.com/VincentZimmer> and vincent.zimmer@gmail.com

UEFI AND THE OEM AND IHV COMMUNITY

Contributors

Nathan Skalsky

IBM

Terry Kirch

Emulex

Al Rickey

Emulex

Michael A. Rothman

Intel

“Database Services. A series of UEFI protocols that are intended to be an in-memory repository of specialized databases.”

“Configuration Routing Services. The interface that manages the movement of configuration data from drivers to target configuration applications.”

The computer industry has evolved a great deal over the years, experiencing various design changes over time. One thing that has remained the same has been the need for interoperability between the platform elements. This article describes how the UEFI standard introduces the composite pieces for such interoperability, and in so doing, the article illustrates how the end user benefits from the advantages gained by from such standards both the OEM and IHV communities.

Introduction to Pre-OS Platform Configuration

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

- *Database Services.* A series of UEFI protocols that are intended to be an in-memory repository of specialized databases. These database services are focused on differing types of information:
 - *Database Repository* – This is the interface that drivers interact with to manipulate configuration related contents. It is most often used to register data and update keyboard layout related information.
 - *String Repository* – This is the interface that drivers interact with to manipulate string-based data. It is most often used to extract strings associated with a given token value.
 - *Font Repository* – The interface to which drivers may contribute font-related information for the system to use. Otherwise, it is primarily used by the underlying firmware to extract the built-in fonts to render text to the local monitor. Note that since not all platforms have inherent support for rendering fonts locally (think headless platforms), general purpose UI designs should not presume this capability.
 - *Image Repository* – The interface to which drivers may contribute image-related information for the system to use. This is for purposes of referencing graphical items as a component of a user interface. Note that since not all platforms have inherent support for rendering images locally (think headless platforms), general purpose UI designs should not presume this capability.
- *Configuration Routing Services.* The interface that manages the movement of configuration data from drivers to target configuration applications. It then serves as the single point to receive configuration information from configuration applications, routing the results to the appropriate drivers.

- *Browser Services.* The interface that is provided by the platform's BIOS to interact with the built-in browser. This service's look-and-feel is implementation-specific, which allows for platform differentiation.
- *Configuration Access Services.* The interface that is exposed by a driver's configuration handler and is called by the configuration routing services. This service abstracts a driver's configuration settings and also provides a means by which the platform can call the driver to initiate driver-specific operations.

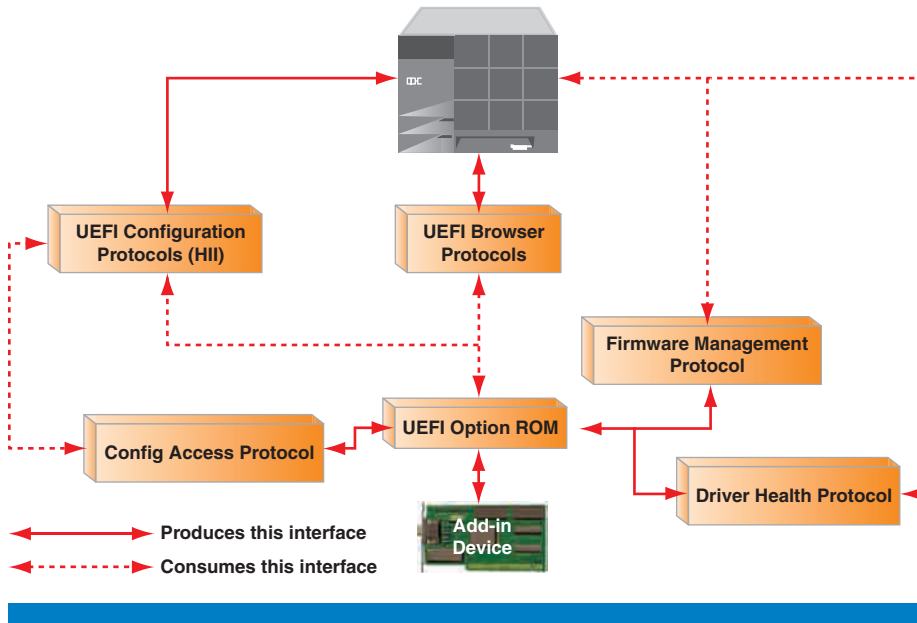


Figure 1: Software Interface Relationships with Devices

(Source: Intel Corporation, 2011)

Driver Health Protocol

The UEFI driver model has also introduced the Driver Health Protocol. The Driver Health Protocol exposes additional capabilities that a boot manager might use in concert with a device. These capabilities include `EFI_DRIVER_HEALTH_PROTOCOL.GetHealthStatus()` and `EFI_DRIVER_HEALTH_PROTOCOL.Repair()` services. The former will allow the boot manager to ascertain the state of the device, and the latter API will allow for the invocation of some recovery operation. An example of the usage may include a large solid-state disk cache or redundant array of inexpensive disks (RAID). If the system were powered down during operating system runtime in an inconsistent state, say not having the RAID5 parity disk fully updated, the Driver Health Protocol would allow for exposing the need to synchronize the cache or RAID during the pre-OS without “disappearing” for a long period during this operation and making the user believe the machine had failed. More information on the Driver Health Protocol can be found in Chapter 10 of the UEFI 2.3 Specification.

“The Driver Health Protocol exposes additional capabilities that a boot manager might use in concert with a device.”

“the Firmware Management Protocol, which provides an abstraction for devices to manage the firmware images associated with them.”

“By standardizing the means by which platform error records are produced/consumed, a much more robust mechanism of information interchange can be achieved between the hardware and software elements of the platform.”

Firmware Management Protocol

One of the interfaces that also can be found in the latest UEFI specifications is the Firmware Management Protocol, which provides an abstraction for devices to manage the firmware images associated with them. For instance, a particular device (such as a RAID controller) would often have an option ROM associated with it that resides on a nonvolatile storage component on the physical controller itself. There normally would not be a mechanism for the platform to interact with the content of that nonvolatile storage aside from a PCI configuration space interaction. Having the device expose abstractions that allow the manipulation (such as GetImage, SetImage, CheckImage, and GetImageInfo) of the image contents allows the device to have its own option ROM content inventoried or even updated. In absence of such an interface, data directly associated with the image in the option ROM is largely opaque to the platform owners.

Error Reporting

The error interface consists of a set of OS runtime APIs implemented by system firmware accessible through runtime UEFI interface mechanisms (that is, GetVariable/SetVariable). In the UEFI specification a UEFI variable named HwErrRec#### is defined.

By using this abstraction for a common hardware error record format, the following capabilities are provided:

- Error reporting to the OS through standardized error log formats.
- The ability to store OS- and OEM-specific records to the platform nonvolatile storage in a standardized way and to manage these records based on an implementation-specific usage model.

By standardizing the means by which platform error records are produced/consumed, a much more robust mechanism of information interchange can be achieved between the hardware and software elements of the platform.

Why UEFI Matters to IBM and the x86 Server/HPC Marketplace

This section provides targeted examples on how the advent of UEFI as a standard has helped address prior limitations and improve the offerings that IBM has been able to provide in the pre-OS environment.

Limitations of BIOS

BIOS originated with the first IBM PC in 1981 and has grown over time to accommodate the ever-increasing demands of enterprise and high-performance computing. Significant architectural limitations restrict the further growth of BIOS. One major architectural restraint is that BIOS runs in 16-bit processor mode, causing the following functionality limitations:

- Generally, only 1 MB of memory is addressable at any time.

- The space for PCI option ROMs and how much code they can run are limited, restricting both the number of adapters that can be installed (approximately four) and how much functionality they can contain.
- Space for advanced BIOS functionality is limited.
- The firmware image is monolithic and nonmodular.

Although much of the x86 processor, memory, and I/O technology has greatly evolved over the past 30 years, the system BIOS has remained essentially unchanged.

The original BIOS was not intended to accommodate server technologies such as scalable multi-way systems, advanced power and energy management and capping, remote console, and systems management.

Advantages of UEFI for High-Performance and Enterprise Systems

The primary advantages of UEFI for high-performance and enterprise systems are:

1. The ability to boot 2.2+ TB Storage Partitions
2. The ability to configure/deploy HII-complaint adapters
3. The ability to grow beyond 128k PCI OPROM space limitations and other BIOS memory map restrictions
4. 64-bit UEFI pre-boot space for add-on pre-OS technologies

Recent History and Future of IBM x86 UEFI Development

As platforms within IBM adopted the UEFI standard, the immediate goal was to achieve at least parity with our current pre-OS capabilities. This was achieved, and as we developed ongoing solutions we were able to outstrip our prior capabilities and add further innovation as the platforms evolved, as shown in Figure 2.

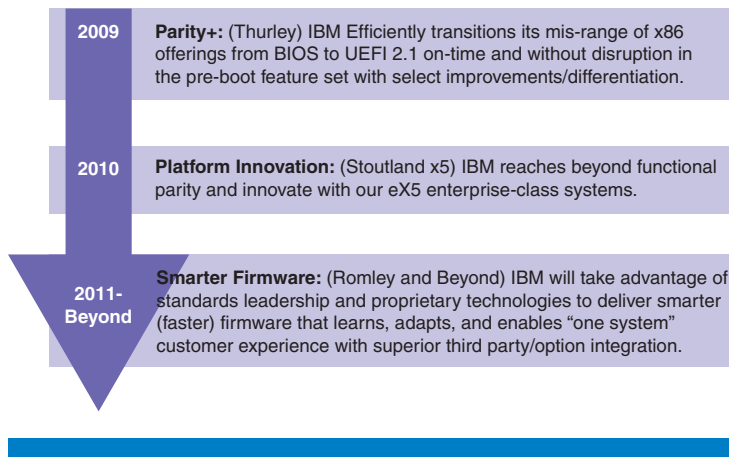


Figure 2: Evolution of UEFI within some IBM platforms

(Source: IBM Corporation, 2011)

“Although much of the x86 processor, memory, and I/O technology has greatly evolved over the past 30 years, the system BIOS has remained essentially unchanged.”

IBM UEFI-Compliant Firmware Boot Manager

IBM UEFI firmware supports UEFI booting and BIOS booting without requiring that boot modes or settings be changed. Unlike in other UEFI/BIOS compatible systems, IBM UEFI firmware supports both boot methods within one managed boot order.

The UEFI boot process is significantly different from the BIOS boot process: instead of booting devices, the system boots specific targets from physical or logical device paths, so boot media can contain virtually unlimited boot targets (operating-system instances).

By default, the firmware favors UEFI boot targets if both are available; for example, if the CD drive is first in the boot list and a dual bootable media (booted BIOS or UEFI) is present, the firmware detects that the CD is UEFI-bootable and hands off to the EFI boot loader. You can override this preference by inserting the *Legacy Only* flag above a boot target in the boot list that you want to boot BIOS only.

Notes:

- If you insert the *Legacy Only* flag into the boot order list, the server invokes the compatibility module for booting regardless of whether there appears to be anything there to boot (that is, UEFI is not aware of media behind non-UEFI compliant adapters). When the Legacy Only option is invoked, the server is committed to a BIOS boot, and the server eventually resets if nothing is booted during the BIOS boot process.
- Some UEFI-aware operating systems that also have BIOS support are installed according to how the dual-boot installation media was booted. Therefore, if you want to install Microsoft Windows* Server 2008 x64 as a UEFI-aware operating system, you must boot the UEFI boot loader on the installation media. This happens by default unless you use the *Legacy Only* flag to force the media to be booted as a BIOS boot target.

The Boot Process

The UEFI specification extends the boot process to allow several new boot management capabilities. Key features of UEFI include the ability to create user-defined boot options, associate additional parameters to pass to the boot loader, assign unique names (for example, “Emma’s Maintenance OS”), and manage multiple operating-system installations on one boot device.

Table 1 compares the boot processes and highlights some of the differences between BIOS booting and UEFI booting.

“The UEFI specification extends the boot process to allow several new boot management capabilities.”

BIOS Boot Process	UEFI Boot Process
Hands off control to the master boot record (MBR)	Hands off control to a boot loader on the UEFI partition
One MBR per boot device	Possible multiple boot loaders on one device
Options are limited to categories such as CD or hard disk drive.	User-defined boot options can be created in addition to generic boot device categories.
Configuration information cannot be passed to the MBR and operating system.	The user or operating-system agent can add parameters to the boot options.
The boot order information is stored in CMOS memory.	Boot options and their order are stored in NVRAM.

Table 1: Comparison of BIOS and UEFI Boot Processes

(Source: IBM Corporation, 2011)

Notes:

- IBM UEFI firmware supports both boot processes through the System x boot manager with the limitation that BIOS booting is terminal; that is, when the compatibility support module (CSM) is invoked for a boot, the system cannot return to the UEFI boot manager.
- In UEFI bootable media, the main UEFI boot loader is usually at `\efi\boot\bootx64.efi`. It must be at that location if the firmware is to boot it automatically without requiring that you manually add a boot option for where the applicable boot loader binary is located and named. Any valid UEFI boot loader at `efi\boot\bootx64.efi` is invoked when the boot option for the device is enumerated.

Generic Boot Options

The generic boot options shown in Table 2 are, for the most part, smart options. If a CD or DVD contains a UEFI boot record in the master catalog, the CD/DVD boot is an EFI boot. If the CD or DVD does not contain a UEFI boot record but it does contain an x86 boot record, the CD/DVD boot is a BIOS boot.

Dual-boot media is media that can simultaneously meet UEFI and BIOS boot requirements.

“Dual-boot media is media that can simultaneously meet UEFI and BIOS boot requirements.”

Generic Boot Option	Dual-Boot Media Support
CD or DVD	Yes*
Diskette	Yes*
USB storage	Yes*
Hard disk 0,1, 2, 3, 4	No; either master boot record (MBR) or GUID partition table (GPT)
Network	N/A; will try EFI network boot first*
Embedded hypervisor	N/A; hard-wired to boot the hypervisor option

Table 2: Meeting both UEFI and Legacy BIOS boot requirements

* If there is no Legacy Only flag in the boot order list, the boot manager gives preference to booting dual-boot media as UEFI. If you insert the Legacy Only flag above a boot target in the boot order list, that media will be forced into a BIOS boot.

(Source: IBM Corporation, 2011)

Note: All UEFI bootable media must be FAT formatted.

Legacy Only is not a boot option but a flag that indicates to the firmware that any generic boot options below it in the boot order list are to be BIOS booted, even if they could be UEFI booted. There are two designed reasons for the Legacy Only option:

- To force a CD/DVD BIOS installation for a dual-boot CD or DVD
- To boot to a hard disk that is not visible to the EFI environment (a non-IBM disk controller without EFI firmware)

CD or DVD

A table of contents describes partitions and indicates whether those partitions are bootable and, if they are, which architecture (BIOS x86 or UEFI).

Notes:

- The Microsoft Windows Server 2008 installation DVD contains two bootable partitions: one for BIOS x86 and one for UEFI.
- Some versions of SLES 11 installation media contain two bootable partitions; however, both are designated for BIOS x86 and therefore will be booted through the BIOS compatibility mechanism.

Diskette

Diskettes use a FAT file system, and they are not able to store a large amount of data. However, a diskette is tested to determine whether it is bootable, and if it is, the server goes into BIOS mode to boot the diskette. Note that a formatted diskette always appears to be bootable, even if it has no DOS or other bootable image; therefore, the presence of a diskette in the drive causes the server to attempt to boot from it. Remove any diskette from the drive before you boot the server unless you intend to boot from it.

USB Storage

USB storage is similar to a hard disk in that it has a master boot record (MBR). However, because it is removable media, the specification allows for USB storage to include an \efi\boot\bootx64.efi file. If a USB key contains the file, it is EFI bootable. You can place a fullshell.efi file on a USB key and rename it to \efi\boot\bootx64.efi. Then, if you boot from the USB key, it EFI boots to the shell. If there is no \efi\boot\bootx64.efi file on the USB key, the UEFI boot manager examines the MBR, and if it is designated as bootable, the server goes into BIOS mode and boots from the USB key. (See “Starting the UEFI Shell” for information about obtaining and running the shell environment.)

Hard disk

A BIOS-bootable operating system can be installed only to a master boot record (MBR) partition, and an EFI-bootable operating system can be installed only to a GUID partition table (GPT) partition. MBR partitions and GPT partitions may not coexist on the same volume. Therefore, if a selected partition is MBR, it is BIOS-booted; if it is GPT, it is EFI-booted.

Network

The server first attempts an EFI network boot. If that fails, the server passes control to the compatibility source module (CSM) to attempt a BIOS network boot. You can configure the iSCSI and PXE settings through the Setup utility (System Settings → Network panel). You can specify Legacy, UEFI, Both, or None.

Embedded Hypervisor

An embedded hypervisor key is a USB key with a specific vendor ID/product ID (VID/PID). It is excluded from processing in the USB storage entry, and USB keys are excluded when the Embedded Hypervisor entry is processed. An embedded hypervisor key can be installed internally or externally.

Operating System Deployment

The UEFI boot manager processes devices in the boot order list, one at a time, as it searches for a potential bootable device. If the boot manager detects a UEFI-bootable device, it attempts to boot that device. If that attempt fails, the boot manager returns to the boot order list. For most devices, the boot manager goes to the next device and tries again. For PXE and iSCSI, the boot manager checks for Legacy at this time.

If the firmware detects that the boot target is only BIOS bootable (that is, it has only a bootable MBR), it will take the current boot order list and perform the same step as Legacy Only, and transition to legacy mode.

“A BIOS-bootable operating system can be installed only to a master boot record (MBR) partition, and an EFI-bootable operating system can be installed only to a GUID partition table (GPT) partition.”

“If the boot manager successfully inspects hard disk drive 0 and determines that it is has a bootable BIOS MBR, the boot manager sends the boot order list to the compatibility support module (CSM), unloads UEFI from memory, and starts the CSM for a BIOS boot.”

Example:

The boot order is as follows:

1. DVD
2. Hard disk drive 0 (formatted with a bootable BIOS MBR)
3. USB

The UEFI boot manager inspects for a DVD and determines that no DVD is present.

Next, the boot manager attempts to inspect hard disk drive 0. If the boot manager successfully inspects hard disk drive 0 and determines that it is has a bootable BIOS MBR, the boot manager sends the boot order list to the compatibility support module (CSM), unloads UEFI from memory, and starts the CSM for a BIOS boot. However, if the boot manager is unable to inspect hard disk drive 0 (for example, because it is managed by an atypical legacy storage adapter that does not have a UEFI device driver and cannot have UEFI support emulated through thunking), the boot manager does not recognize a BIOS operating system. In this case, you must manually add the *Legacy Only* flag above *HardDrive0* to instruct the boot manager to assume that a BIOS operating system is associated with hard disk drive 0.

For information about adapter support issues and use of the *Legacy Only* flag, see “BIOS Support” and “Optimizing Boot-time Performance.”

Notes:

- Before you attempt to install a UEFI-aware operating system on a hard disk that is already formatted with an MBR, you must delete all partitions from the disk or reformat the disk with GPT.
- The UEFI 2.3.1 specification allows the UEFI boot manager to look in `\efi\boot\bootx64.efi` for a valid UEFI boot loader on a hard disk drive when no specific boot loader is specified in the boot option, because a generic boot option (such as *HardDrive0*) or a partial media path is being used. Earlier UEFI specifications provide this capability only for removable media such as USB keys.

Boot Management Limitations

This section describes the boot management limitations that are associated with the UEFI firmware and how to work around those limitations.

Non-UEFI-Compliant Boot Devices

Some non-UEFI-compliant adapters, such as storage controllers, provide UEFI emulation interface wrappers (known as *legacy adapter thunking*) through the IBM UEFI firmware. A controller adapter that is not UEFI compliant (that is, it does not have a UEFI device driver) and is not enabled for legacy adapter

thinking requires special consideration when you configure the server to boot from targets that are managed by that controller. You must insert the Legacy Only flag above the generic boot option that represents the target adapter. For more information, see “Using the Legacy Only Flag.” Inserting the *Legacy Only* flag instructs the UEFI firmware to invoke the compatibility support module (CSM) and attempt a legacy boot, even though it cannot detect a boot target. The CSM detects the boot target because the legacy option ROM will have run in this case.

Generic Boot Options

Generic boot options provide BIOS-like ease of configuration; however, they do not allow you to specify which specific device in a category to boot first. To overcome this limitation, you can reorder the option ROM execution order (for legacy boots from adapters) or add specific device path boot options (for UEFI boots).

Multi-Path Controllers

Although you can use the Setup utility to effectively create UEFI and BIOS boot options, there are cases in which it is preferable to use the boot option that is created by a UEFI-aware operating system (for example, multipath controllers). Manually adding a boot option by traversing the file system and adding the boot loader results in a specific device path, whereas the operating system can create a media path that allows a multipath controller to lose a path and still boot.

Understanding Boot Performance on x86 Systems

Optimizing Boot-time Performance

The simplest way for customers to achieve quicker boot times for a configuration is to install and boot UEFI-aware operating systems whenever possible. For deployments in which a UEFI-aware operating system is not available, this section introduces concepts and techniques for optimally configuring adapter support for legacy boots.

The major variable determinants of server boot-time performance are how much memory capacity is available and what adapters are installed. Other determinants of boot-time performance are inherent to the design and core technologies of the server design (such as CRTM/TPM, platform self-test, and power management) and are not configurable. Always use the latest available firmware, because any optimizations to these core features will be in the latest firmware releases.

Memory

The more memory that is installed, the more there is to initialize ECC and test. You can install less memory, but that usually does not result in significant boot-time improvement. The best approach for optimizing boot time and memory use is to balance DIMMs and memory capacity among installed processors.

“The simplest way for customers to achieve quicker boot times for a configuration is to install and boot UEFI-aware operating systems whenever possible.”

“The best approach for optimizing boot time and memory use is to balance DIMMs and memory capacity among installed processors.”

Balancing memory optimizes memory initialization on servers with integrated memory controllers (such as Intel Xeon® 5500 based servers). For details, see the *Optimizing the Performance of IBM System x and BladeCenter Servers Using Intel Xeon 5500 Series Processors* whitepaper.

Adapters

Some classes of server adapters, such as network or RAID controllers, can take considerable time to initialize in the pre-operating-system UEFI or BIOS environment. Because IBM UEFI firmware simultaneously supports both UEFI and BIOS boot mechanisms, there can be unwanted repetition of adapter initialization when BIOS operating systems are booted. This repetition can occur with an adapter that includes native UEFI device drivers in addition to BIOS code on its adapter ROM. Figure 3 illustrates how there are some factors that show where time is spent during the initialization process in a platform.

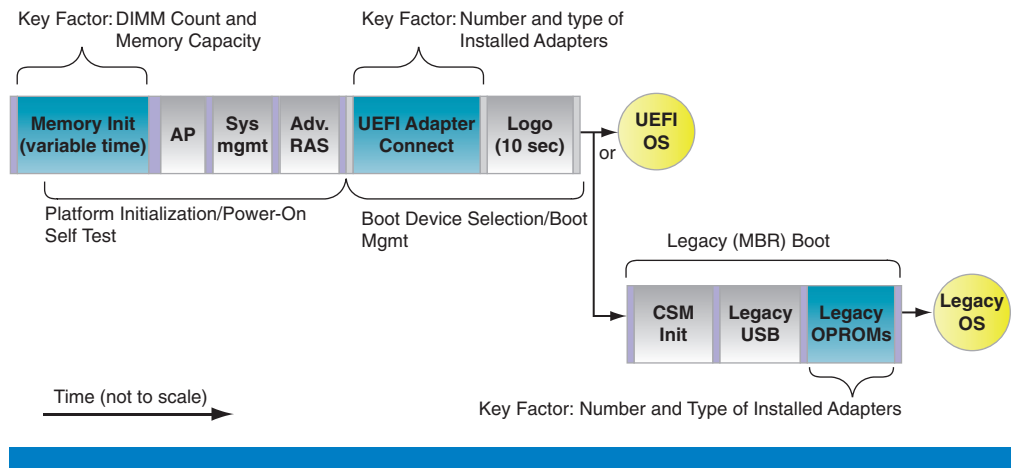


Figure 3: Relative determinants of boot-time performance
(Source: IBM Corporation, 2011)

UEFI-Compliant and Legacy BIOS Adapter Support

There are three places where adapters can be initialized:

- *UEFI*: Driver execution environment (DXE) dispatcher and boot device selection (BDS) connect controllers.
- *UEFI legacy compatibility*: 16-bit thunk device drivers initialize legacy option ROMs and provide a UEFI compatibility wrapper.
- *Legacy compatibility module boot process*: Legacy option ROMs are initialized as part of the legacy boot process (only when booting legacy, non-UEFI-aware operating systems).

Optimal Scenario

The best way to ensure the fastest boot time is to use UEFI-compliant adapters and a UEFI-aware operating system. Figure 4 illustrates the firmware interactions of an adapter with both a UEFI-compliant device driver and legacy

ROM. The legacy code of the adapter is never invoked; therefore, the time penalty of that initialization is not incurred. Because the server is booting a UEFI-aware operating system, there is no need for BIOS compatibility for this boot.

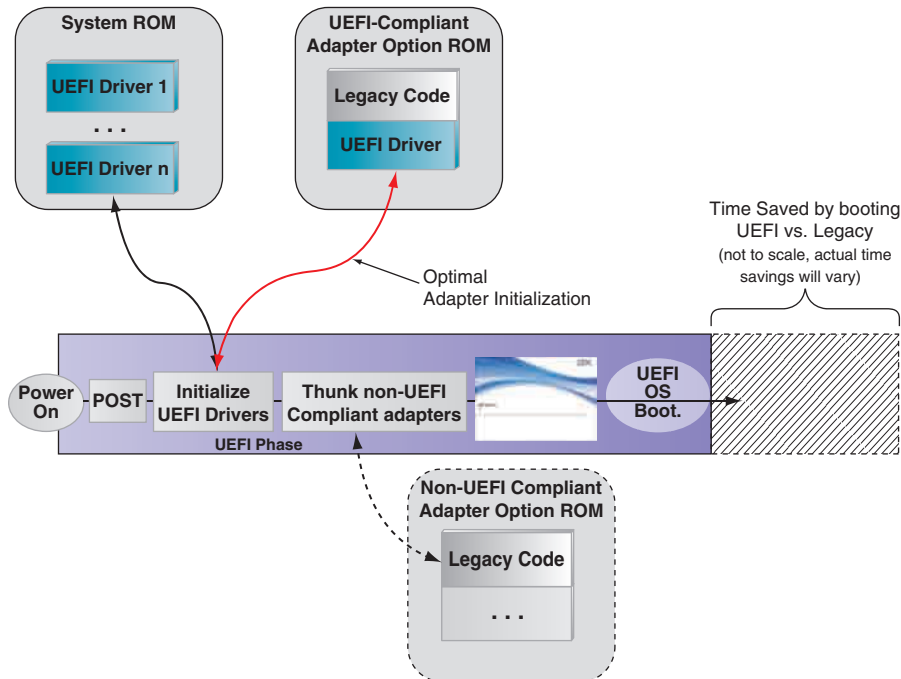


Figure 4: Optimal scenario—default dual BIOS/UEFI-compatibility adapter behavior during a UEFI boot

(Source: IBM Corporation, 2011)

Non-Optimized BIOS Boot Scenario

Figure 5 illustrates the worst-case scenario, in which an adapter with both a UEFI-compliant device driver and a legacy option ROM has its UEFI device driver connected during UEFI preboot and its legacy option ROM run during a BIOS boot. This redundancy is due to the UEFI preboot requirement to detect whether there is bootable media behind the adapter. The preferred method of doing this is to connect the UEFI device driver and check for bootable media. If the media contains a UEFI-aware operating system, there is no redundancy; however, if the media contains a legacy master boot record (MBR) operating system, the redundancy occurs because UEFI unloads (including the adapter UEFI device driver) and loads the compatibility support module (CSM), which calls the adapter legacy option ROM.

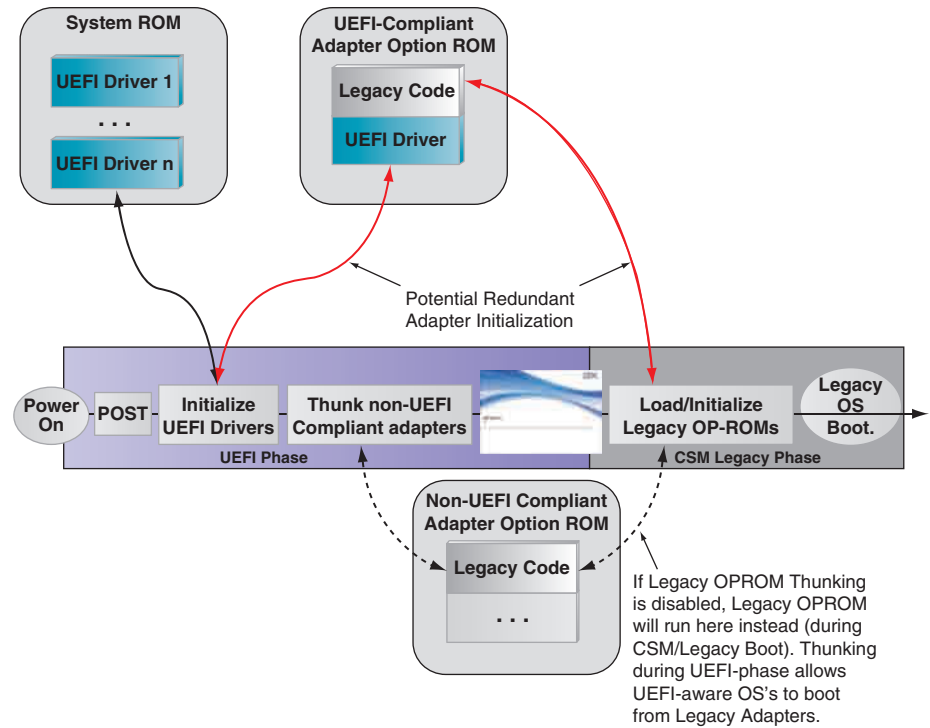


Figure 5: Worst-case scenario—default full-compatibility adapter behavior during a BIOS boot
 (Source: IBM Corporation, 2011)

Optimized Legacy BIOS Boot Scenario

Figure 6 illustrates a scenario in which the server is going to boot a BIOS operating system and, therefore, boot-time performance can be optimized by disabling UEFI support for the adapter. The result depends on the PCI class of the adapter:

- *Storage adapters:* The IBM UEFI firmware invokes thinking support to provide a UEFI compatibility wrapper, enabling the UEFI boot manager to detect the media that the adapter is managing, if UEFI or legacy boot targets are available. This is critical, because if the UEFI boot manager cannot detect the media, it cannot recognize that an operating system is available to be booted.
- *Other adapters:* The UEFI boot device selection (BDS) logic cannot recognize that an operating system is available to be booted, so you must insert the *Legacy Only* flag above the category that the adapter is in (for example, *Network*). This flag instructs the firmware to assume that a BIOS operating system is available to be booted and that the compatibility support module (CSM) will detect it after the legacy adapter ROM is called.

“if the UEFI boot manager cannot detect the media, it cannot recognize that an operating system is available to be booted.”

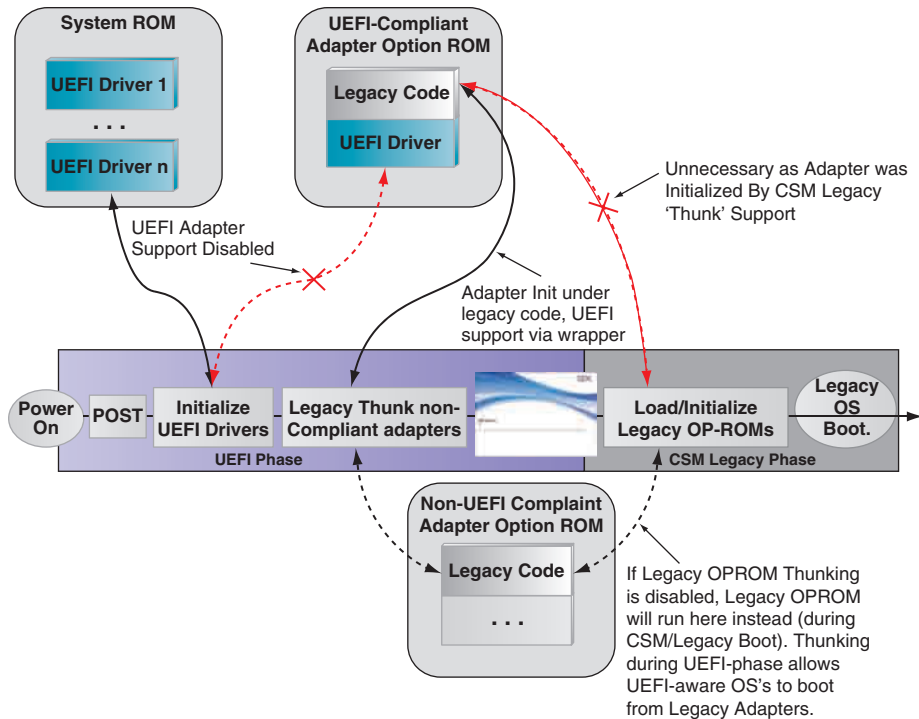


Figure 6: Optimized BIOS boot scenario—adapter behavior with UEFI support disabled on UEFI/BIOS adapter ROM during a BIOS boot (Source: IBM Corporation, 2011)

Using UDK2010 to Profile, Understand, and Improve IHV Adapter and Driver Boot Times

There are six significant steps to using UDK2010 to profile, understand, and improve IHV adapter and driver boot times:

1. Use System x UEFI/UDK2010-based firmware with performance monitoring enabling. Contact the platform OEM for instructions or test build to enable this functionality. (Contact your IBM/OEM IHV building block owner for development platform availability.)
2. Use the UDK2010 DP Tool (Driver Performance) that is compiled for the target system. Note: this driver can be platform-unique, and in the case of IBM x86 servers and blades the use of the IBM-version of the DP tool is highly recommended. (Contact your IBM/OEM IHV building block owner.)
3. Run the system under the desired profiling conditions and configuration.
4. Enter the UEFI Shell (internal or using “load from file”) and invoke the DP tool (from USB or other FAT formatted media).
5. Analyze the output form DP tool to understand overall driver execution time.

6. If execution time is longer than desired, additional performance start/end checkpoints can be added to understand what functions/calls are consuming the most time and return to step 3 (repeat until the performance issues are understood and accounted for).

UEFI HII Configuration Support and Legacy OPRoMs

IBM System x Servers and Blades released since the first half of 2009 support both the UEFI 2.1 and 2.0/1.10 means of driver configuration (Human Interaction interface and Driver Configuration Protocol respectively).

Future IBM systems will include a UEFI 2.3-compliant HII browser (aka F1 Setup and Boot Management). As such, IHV support should be tested for its ability to present and function properly on UEFI 2.3-compliant HII browsers.

IBM typically prefers that x86 boot adapters support both UEFI 2.3 and include a legacy OPRoM image. The legacy OPRoM should take its configuration from the UEFI driver and accordingly the OPRoM is not expected to present the user with a Ctrl sequence to enter a adapter configuration UI.

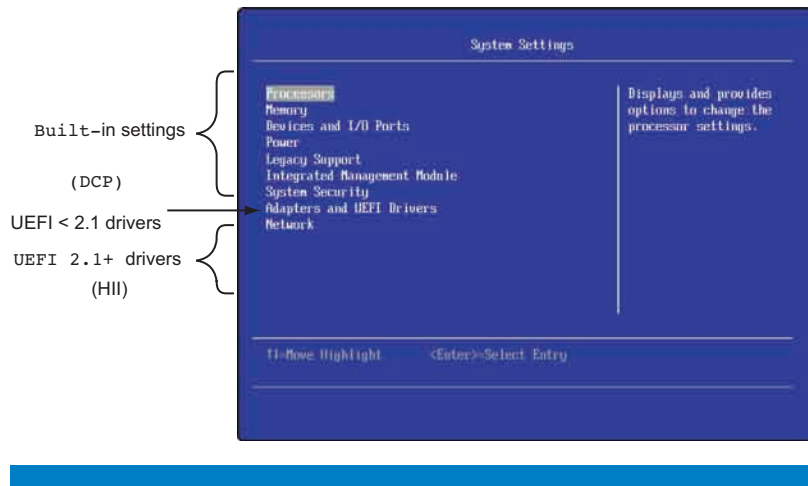


Figure 7: Organization of a Platform Configuration Screen.
(Source: IBM Corporation, 2011)

Developing UEFI2.3 HII Support for New IBM x86-based Servers and Blades

When migrating existing or building new add-on driver support for adapters or pre-boot technologies, there are two major requirements to be aware of:

1. Drivers must follow the HII code definition in the *Unified Extensible Firmware Interface Specification* whose version is no later than v2.3.
2. Driver developers must use *VFR Programming Language* whose version is no later than v1.3, which supports UEFI2.3 HII IFR opcodes.

HII 2.3 Protocols

For HII driver support developed using the Intel framework HII and currently use EFI HII protocol *gEfiHiiProtocolGuid*, they should instead use UEFI HII database protocol *gEfiHiiDatabaseProtocolGuid*. Additionally, UEFI HII configuration access protocol should be used instead of *gEfiHiiConfigAccessProtocolGuid*.

Driver Implementation of Configuration Access Protocol

HII-compliant drivers must implement both *EFI_HII_CONFIG_ACCESS_PROTOCOL.ExtractConfig()* and *EFI_HII_CONFIG_ACCESS_PROTOCOL.RouteConfig()*. The configuration setting values output in the *Result* parameter of *ExtractConfig()* should be those extracted from permanent storage, and the extract action should be done by *ExtractConfig()*. The configuration setting values input in the *Configuration* parameter of *RouteConfig()* should be stored into permanent storage and the store action should be done by *RouteConfig()*.

VFR Writing Syntax

Driver developers should define VFR files in the UEFI 2.3 VFR syntax. VFR files defined in framework VFR syntax should be migrated into UEFI 2.3 VFR syntax.

UDK2010 VFR Compiler

When using the VFR compiler, developers should *not* indicate *-c* or *--compatible-framework* to avoid the VFR compiler treating the VFR file to be built as a framework VFR file. The VFR compile option is indicated in DSC file.

VFR compiler options example:

```
$(R8_DIR)\Platform\$(PROJECT_FAMILY)\$(PROJECT_NAME)\Setup\
Dxe\DxePlatform.inf {
  <BuildOptions>
    *_*_*_VFR_FLAGS = -c # This line should be removed when using
UefiSetup/Dxe/DxePlatform.inf
    *_*_*_BUILD_FLAGS = -c
} #flashmap
```

Default Setting Stores

Driver developers should at define at least one default setting store.

Default setting store example:

```
defaultstore MyStandard, prompt =
STRING_TOKEN(STR_STANDARD_DEFAULT), attribute=
EFI_HII_DEFAULT_CLASS_STANDARD
IBM UEFI-compliant firmware will set standard defaults to drivers if they
cannot find out the current configurations.
```

Defining Settings on the Fly

IHV Driver developers should statically list as many of the configuration settings in the VFR file as possible. If some settings have to be created

dynamically, try to do this action in a driver binding protocol service, such as Start() and Stop(), or when IFR data is populated into HII database. Avoid adding new questions to the formset from within *EFI_HII_CONFIG_ACCESS_PROTOCOL.Callback()* whenever possible.

All settings should be manipulated via the IFR opcodes (that is, suppress-if, gray-if, and so on) and not through *EFI_HII_CONFIG_ACCESS_PROTOCOL.Callback()*, which should only be used for non-setting questions.

Question ID Value

In the *VFR Programming Language* specification, assigning a value to questionid is optional; if it is not defined the compiler assigns a unique ID automatically. IBM strongly prefers that IHV drivers assign a *fixed ID number* for all questions *whether the question represents a configuration setting or not*.

If a question is removed in future updates, the questionID number reserved for it should not be used again.

For questions that are dynamically created, questionid should be fixed and unique in the scope of existing static questionIDs defined in VFR and other dynamic questions created in runtime.

Question Prompt Style Guidance

IHV drivers should use prompt strings as short as possible, capitalize the first letter of each word, and ensure that a question's "prompt" is unique within a FormSet.

Prompts should remain consistent from build to build to maintain consistency for consumers.

Question Duplication

Question duplication is a case where two or more formsets use the same varstore location. IBM prefers that IHV drivers avoid duplicate questions as this can complicate customer experience and inhibit the ability of automated deployment entities to accurately model the driver configuration.

Note/Tip: if developers need to use the value of a question in another formset to build an expression, question reference can be used instead of creating a duplicate question.

Defining Settings for OEM/MFG Use

Questions representing settings that are not intended for customer use can use the "suppressif TRUE;" VFR syntax to contribute the question to the HII database while keeping production versions of the system HII browser from rendering the question. Note that this is not a secure means of protecting settings, but rather an alternative way of representing settings deemed useful to manufacturing processes but not useful for end users.

Emulex Driver Development: Legacy, EFI 1.10, and UEFI 2.x

Emulex has collaborated with Intel for nearly ten years on driver development for their HBAs (Host Bus Adapter) in a Fibre Channel storage area network (SAN). Beginning with the Itanium® servers, Emulex developed the legacy (X86) driver. With Emulex's expertise adding to the driver development process, their legacy X86 boot driver quickly evolved to EFI, with a monolithic driver architecture that combined Block I/O, SCSI I/O and SCSI Pass Thru protocols. As EFI evolved, Emulex took a lead role in replacing the X86 driver with the EFI (1.10) driver. Most recently, Emulex developed the HII driver, based on UEFI 2.x, for its HBAs (see Figure 8).

This section provides a historical view of the driver evolution, with the legacy development covered first, followed by the EFI 1.10 development, and then finishing with the UEFI 2.x development. For these three phases, the challenges and advantages are noted.

Legacy Driver: Understanding Expansion ROMs

Legacy Expansion ROMs are the mechanism by which IHVs include initialization, software interrupt services, and configuration utilities with their add-in cards. These services allow system BIOS, software, and operating systems to access devices that are not natively supported by the BIOS.

During Power-On Self-Test (POST), expansion ROMs are loaded into the upper memory area by the system BIOS and initialized by calling a well-known initialization entry point. It is the responsibility of the expansion ROM to initialize the add-in card, discover any devices managed by the add-in card, install the appropriate interrupt services, and prompt the user for keystrokes to enter the IHV's configuration utility.

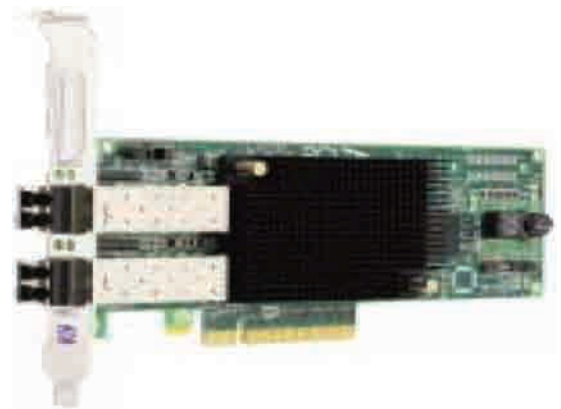
The Constraints and Restrictions of Option ROMs in Legacy BIOS Systems

Expansion ROMs are single-threaded real-mode drivers that provide software interrupt services for the devices they control. The basic model for expansion ROMs and BIOS ROM scan date back to IBM XT's.

In the ensuing decades, a number of enhancements were introduced in hardware and in the BIOS to extend the capabilities to meet evolving needs and to mitigate architectural limitations. Despite this, system resources, particularly in fully configured enterprise systems, remain at a premium. The size of the upper memory region places a limit on the number and size of expansion ROMs that can be loaded and initialized by the system BIOS.

Other aspects of the PC/AT architecture impose limits and challenges for IHVs, BIOS developers, and system integrators as well, described below. In other cases, such as allocation of Extended BIOS Data Area (EBDA), there are multiple algorithms that have been employed over the years that can conflict with BIOS or other expansion ROMs.

“As EFI evolved, Emulex took a lead role in replacing the X86 driver with the EFI (1.10) driver.”



IBM 42D0494
8Gb/s Dual Channel HBA

Figure 8: An example of a storage adapter with HII support
(Source: Emulex Corporation, 2011)

“Legacy Expansion ROMs are the mechanism by which IHVs include initialization, software interrupt services, and configuration utilities with their add-in cards.”

“system resources, particularly in fully configured enterprise systems, remain at a premium.”

“Lack of published standards - BIOS documentation and standards are incomplete and often exist as tribal knowledge.”

An expansion ROM configuration utility is a stand-alone embedded application relying exclusively on BIOS services, with the following development challenges:

- Size restrictions at runtime and in flash
- Stack and variable space
- Screen management must be developed from primitive BIOS video services. These are used to construct menus
- Obsolete build environment and tools
- Assembly language
- No provision for multi-language
- Configuration changes require system reboot
- Small window of time to invoke configuration utility during POST

Following are some of the complications that have to be dealt with by expansion ROM developers:

- *Extended BIOS Data Area (EBDA)* - The algorithm for allocation of EBDA is covered in the PCI 3.0 specification, but was not standardized prior to that. Several algorithms for allocating EBDA memory were used by different OEMs, causing various compatibility issues with BIOS features that utilize EBDA, such as CD-ROM boot.
- *Memory allocation* - Options for memory allocation vary such as POST Memory Manager (PMM), use of EBDA for scratch memory during initialization, or looking for unused segments in the lower 640 K of conventional memory.
- *No hard drive boot failover* - By convention only the first enumerated hard drive in a legacy system will be used as a boot device.
- *Access of memory and I/O above 1 MB* - In order to access memory above 1 MB, expansion ROMs have to switch in and out of Big Real Mode.
- *Lack of published standards* - BIOS documentation and standards are incomplete and often exist as tribal knowledge. This can lead to interoperability issues.
- *Reliance on obsolete development tools* - Expansion ROM developers often rely on unsupported and out-of-date tools capable of building 16-bit executables, such as MASM, and TASM. In addition, developers often have to write their own utilities to convert the compiled and linked executable into the final ROM image. Setting up the initial build environment and the basic framework for an option ROM is not a trivial task.

Legacy Expansion ROM Configuration Utility User Interface Development Challenges

Expansion ROM configuration utilities are small embedded applications that can be invoked by the user during POST by entering a key sequence when prompted by the expansion ROM. They are typically developed in C or assembly language and compiled into the expansion ROM image.

Implementations vary from simple text-driven prompts for user input to fairly elaborate menu driven utilities. They are, however, all designed to run in a pre-boot environment and are limited to using the available BIOS services.

In practice, because of the variety of BIOS implementations, such utilities are often designed for the lowest common denominator. In this way, they are not prevented from running on systems that lack optional advanced BIOS services, such as PMM.

Furthermore, the basic constructs of a user interface, such as menus, must be developed from scratch using a relative sparse set of primitive services provided by the video BIOS.

Figure 9 shows a screenshot of the X86 BIOS entry point during POST.

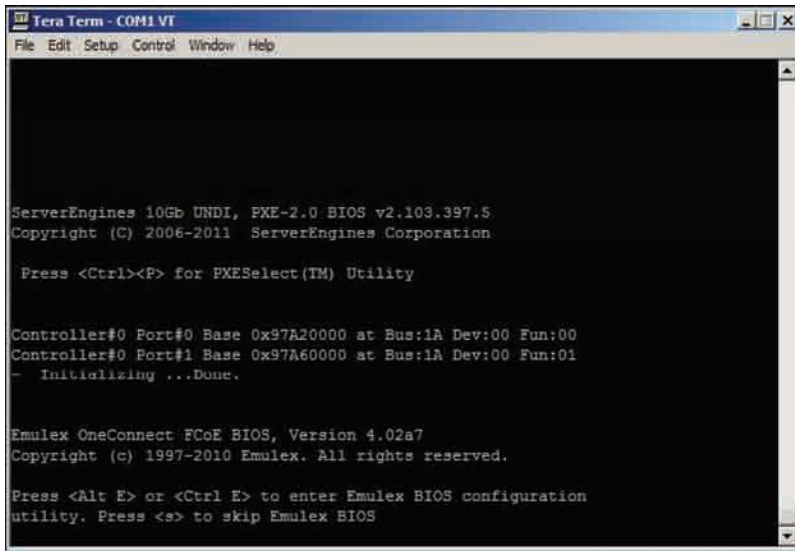


Figure 9: Legacy Post Screen Banner
(Source: Emulex Corporation, 2011)

Early Emulex EFI Drivers

When Emulex first switched from a legacy driver to one that was EFI compatible, this was early in the history of technology adoption with the EFI 1.1 version of the specification. This section describes some of the history and key learnings achieved from that effort.

EFI 1.10 Driver Development History

The EFI 1.10 specification was published by Intel Corporation in 2002. Intel also developed an EFI 1.10 sample implementation that was available for download from their Web site.

The EFI sample implementation included source code for a variety of EFI drivers, as well as makefile examples that could be easily leveraged by a developer to create a new EFI driver that would build along with the sample implementation.

The Intel EFI 1.10 was adopted as the basis of the UEFI 2.0 specification. The reader should be aware that all references to the 1.10 specification apply equally to UEFI 2.0.

The Intel Sample Implementation is no longer maintained and has been replaced by the TianoCore open source project, which is based on the Intel Sample Implementation.

EFI 1.10 Driver Development

Emulex's early EFI focus was on porting the existing Fibre Channel expansion ROM to an EFI 1.10 driver, in order to provide basic boot from SAN support (see Figure 10) on Itanium platforms and only support switched fabric.

“The Intel Sample Implementation is no longer maintained and has been replaced by the TianoCore open source project, which is based on the Intel Sample Implementation.”

Subsequent releases focused primarily on expanding Fibre Channel support (that is, arbitrated loop) and enabling the user to access adapter firmware features, such as forcing specific link speeds, as well as options for depth of device discovery.

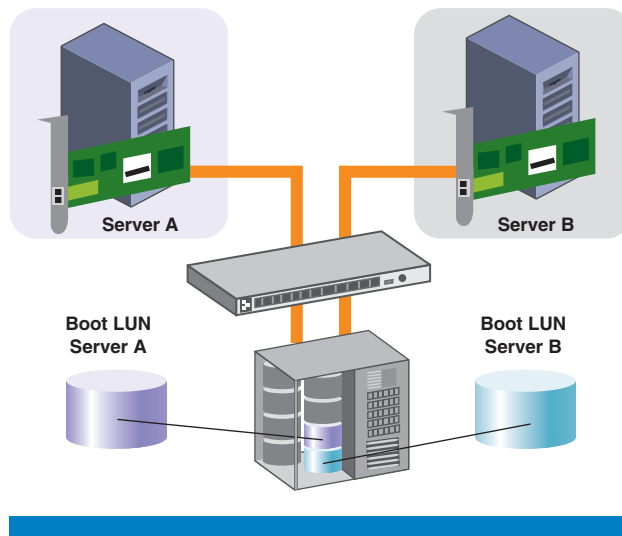


Figure 10: This “boot from SAN” diagram shows two servers using Fibre Channel HBAs to access their boot images, which are located on an external array. (Source: Emulex Corporation, 2011)

SCSI Driver Model

The Emulex Fibre Channel driver is based on the SCSI Driver Model as originally defined in the Intel SCSI Driver Model document, which is incorporated in the UEFI 2.3 specification as Section 14. The SCSI Driver Model defines a driver stack with abstraction layers for block I/O, SCSI bus, and a SCSI passthrough protocol, which provides the abstraction layer for the host interface of an IHV’s storage adapter.

The SCSI Driver Model document suggests producing separate drivers for the three abstraction layers that comprise the SCSI driver stack. Following this, Emulex’s first implementation did consist of three drivers and lacked provisions for configuration and diagnostics.

Customer feedback and support issues led Emulex to produce a single monolithic driver capable of producing all of the handles and protocols called for in the SCSI driver model documentation.

EFI Architecture Advantages over Legacy

Moving from legacy development to EFI resulted in the following notable advantages:

- Sample implementations available in the UEFI specification and EFI Developer’s Kit (EDK) make it easy to put together the basic framework of a driver

“Customer feedback and support issues led Emulex to produce a single monolithic driver capable of producing all of the handles and protocols called for in the SCSI driver model documentation.”

- EFI drivers more closely resemble a modern OS device driver
- A rich set of system services
- An exclusively C language development environment (no assembly language)
- System firmware services are more standardized, and better documented
- Modern build tools
- Full access to system memory
- Capable of multilanguage support
- Configuration changes don't require reboot
- EFI byte code allows a single driver image to support multiple processor architectures
- Standard means of compressing the boot driver image
- Decompression code is not carried in the driver

Emulex development engineers discovered that though working with the UEFI standard, OEMs implement the standard in slightly different ways. With the 1.10 driver, there were very few differences across OEMs and no noted conflicts with other storage drivers in the system, whereas expansion ROM conflicts are a fairly common occurrence.

The 1.10 Configuration Utility User Interface Development Challenges

The EFI 1.10 specification provides a mechanism for driver configuration through the DriverConfigurationProtocol. The protocol defines three functions, which can be entered through the EFI shell command “drvcfg”.

EFI 1.10 configuration utilities are fairly similar to Expansion ROM utilities. They are embedded configuration programs that rely on the available system services and are responsible for implementing the screen management functions required to render the application menus and screens.

In some cases, EFI configuration utilities have slight advantages over their expansion ROM counterparts, such as better access to system memory. In other cases they have slight disadvantages, with the most notable being the relative sparseness of the Simple Text Output protocol over the services provided by the BIOS interrupt 10h interface.

The model for the 1.10 driver configuration requires a significant amount of developer effort and code space in order to produce a reasonably useful user interface for configuring the driver and the IHV's add-in card. The following section provides details on topics the developer should be aware of.

Simple Text Output

The SimpleText Output protocol comprises the standard set of text-based console I/O services available to EFI drivers. The protocol provides very basic functions for setting cursor position, setting output attributes, setting screen modes, and outputting text strings.

“Screen management is the sole responsibility of the developer who needs to create whatever screen management routines and constructs, such as menus and windows, are required.”

Menu Navigation / User Experience

Screen management is the sole responsibility of the developer who needs to create whatever screen management routines and constructs, such as menus and windows, that are required. The sample implementation does include some library functions for console I/O that leverage the Simple Text Output protocol for things like formatted output.

Among the challenges encountered by Emulex were some behavioral changes noted in library functions when changing EDKs. Additionally, there were difficulties displaying data types that vary in size from machine to machine.

Input of User Data

User input is read through the Simple Input protocol, which provides a single function for reading keyboard input, similar to interrupt 16 function 0h. All user input data has to be parsed and validated by the developer.

Shell Requirement

EFI supports loading of a command interpreter through the EFI system boot manager known as the EFI Shell. However, the EFI specification does not require OEMs to include an EFI shell in the system’s ROM. The shell optionally supports an internal command for launching EFI driver configuration utilities by calling into the Driver Binding protocol.

To launch a configuration utility, the user must load the EFI shell, then run the shell command “drivers” to obtain the handle for the driver that they would like to configure. Once the user has the driver image handle, the user can call the driver’s configuration protocol by passing the handle to the shells “drvcfg” command (see Figure 11).

```

Tera Term - COM1 VT
File Edit Setup Control Window Help
A3 0000000A ? - - - - VGA Class Driver          VgaClass
A4 0000000A D - - 1 - Generic Disk I/O Driver      DiskIo
A7 0000000A ? - - - - FAT File System Driver      Fat
A8 00000010 D - - 1 - IDE Controller Init Driver    IdeController
A9 00000010 D - - 1 - SATA Controller Init Driver    SataController
AA 0000000A B - - 2 1 PCI IDE/ATAPI Bus Driver      IdeBus
AB 0000000A ? - - - - AHCI Bus Driver          Ahci
AE 0000000A B - - 1 6 ISA Bus Driver          IsaBus
AF 0000000A B - - 1 1 ISA Serial Driver          IsaSerial
B0 0000000A ? - - - - ISA Floppy Driver          LegacyFloppy
B1 0000000A ? - - - - Partition Driver (MBR/GPT/EL Torito) Partition
B2 0000000A B - - 1 34 PCI Bus Driver          PciBus
B8 00000010 D - - 1 - Usb Ehci Driver          Ehci
B9 0000000A ? - - - - Usb Keyboard Driver          UsbKb
BA 00000011 ? - - - - Usb Mass Storage Driver      UsbMassStorage
BB 0000000A ? - - - - Usb Mouse Driver            UsbMouse
BC 00000020 D - - 4 - Usb Uhci Driver          Uhci
BD 0000000A D - - 5 - Usb Bus Driver            UsbBus
BE 0000000A D - - 1 - SCSI Bus Driver            ScsiBus
BF 0000000A ? - - - - Scsi Disk Driver            ScsiDisk
C2 0000000A D - - 1 - PS/2 Mouse Driver          Ps2Mouse
E0 00040111 D X X 1 - Emulex SCSI Pass Thru Driver  MemoryMapped(0xB,0)

Shell> drvcfg -s e0

```

Figure 11: Example of EFI shell “drvcfg” command
(Source: Emulex Corporation, 2011)

This mechanism for launching configuration utilities has caused difficulties for Emulex and other IHVs. OEMs are not required to include an EFI shell in the system's ROM and some implementations of the shell have had commands removed to free up system ROM space.

Emulex HII UEFI Driver

This section describes the latest driver development: HII UEFI 2.x.

Emulex HII Driver Development History

With the release of the UEFI 2.1 specification, which added the EFI HII Configuration Access Protocol, Emulex was one of the first IHVs to port its EFI 1.10 (drvcfg) driver to be compatible with any system BIOS that had implemented the HII. In the development effort, Emulex worked closely with IBM, as they were one of the first IHVs to have a UEFI 2.1-compliant system BIOS.

Use of the EDK Sample Driver as a Template

At the start of this task, the discovery of the DriverSample code in the UEFI EDK, now called the UEFI Developer's Kit 2010 (UDK2010), was a great help in jump-starting the HII driver development effort. This code has many examples showing how a device driver's configuration utility could present forms, navigate forms, and input data in collaboration with a system BIOS HII browser.

User of VFR for Menus

One of the most helpful examples in the DriverSample code was demonstrating the use of Visual Forms Representation (VFR) files. In addition to facilitating a quick, easily maintainable implementation of menus and basic navigation, the VFR file also allows the developer to implement localization. A given string token used in the VFR file can be translated into different languages and tagged with a language code. With these tagged strings in the database, the system UEFI browser can prompt users to select their preferred language. Thus, configuration utility menus are displayed in the chosen language.

Useful Documents

The most useful documents for this driver development are:

- Unified Extensible Firmware Interface Specification 2.1: Chapters 27, 28, and 29
- Intel VFR Programming Language
- VfrCompiler_Utility_Man_Page

The Challenges of Keeping Up with the UEFI Specification Versions (<2.1)

The UEFI specification has evolved over time. With each revision of the specification, new protocols are added, while some older protocols get deprecated. Emulex adds the appropriate new protocols to their driver as they are published with each revision. However, there is a challenge in maintaining backward compatibility with the previous EFI versions that continue to run on older platforms.

“In addition to facilitating a quick, easily maintainable implementation of menus and basic navigation, the VFR file also allows the developer to implement localization.”

“there is a challenge in maintaining backward compatibility with the previous EFI versions that continue to run on older platforms.”

Emulex uses the EDK to compile code. Emulex has always made use of the EDK infrastructure for building the UEFI Fibre Channel driver. The EDK has also evolved over time to support the changes in the UEFI specification. There have been occasions where library functions contained in the then new EDK version were modified from the previous EDK. This caused some issues in the code, requiring modifications to ensure that the calls to the EDK library function were producing the expected results.

Emulex HII Driver Implementation Challenges

The following describes the HII driver implementation challenges.

Challenges of “Hybrid” x86/UEFI Systems

The first EFI systems were primarily based on the Itanium processor. With the release of the x64 processor series, Emulex began seeing more systems that had the ability to boot both EFI-aware and legacy operating systems. The Emulex HBA carries both EFI and legacy option ROM drivers. Both of these carry their own configuration utility.

On these systems, if an EFI boot device is not found, the system will “thunk” into legacy mode. At that point, the legacy option ROMs are loaded to facilitate booting a non-EFI operating system, making it possible for a user to enter the Emulex Fibre Channel legacy utility on this system to configure the HBA.

Traditionally, the configuration data areas of these two drivers were independent of each other. With the appearance of these systems, Emulex felt it was important to ensure that if either configuration utility was used to enter new HBA configuration settings and/or to configure boot LUNs, both utilities would reflect those settings.

Use of Callbacks to Display Dynamically Discovered Data

Emulex has a single driver that manages all the Fibre Channel adapters in a system. The number of adapters in a system and the number of targets connected to those adapters are not static entities. The driver must discover all the Emulex adapters, display them, and allow the user to configure them. The utility must also have the ability to discover and display all the targets connected to an adapter. This dynamic discovery necessitates the use of HII callbacks to update the form that the browser uses to display the devices that were found.

Differences in System BIOS Browsers

OEMs implement their browser interface in different ways. Even though standards exist, each OEM will have some differences in how their browser interface looks, navigates, and interacts with installed devices. Emulex performs extensive testing to ensure that the UEFI Fibre Channel boot driver configuration utility is fully compatible with all platforms that the Emulex HBA is installed on.

Emulex Documentation (Where to Enter the Configuration Utility)

During development, it was noticed that the Emulex FC driver utility entry point could appear under different menus in a given platform’s setup menu

“dynamic discovery necessitates the use of HII callbacks to update the form that the browser uses to display the devices that were found.”

screens. This is because the platform's HII system BIOS makes that decision. This presents a challenge in generating the HII driver utility documentation as to specifying where the entry point to the driver utility can be found. Figure 12 shows the Emulex driver entry point presented by an IBM system BIOS browser.

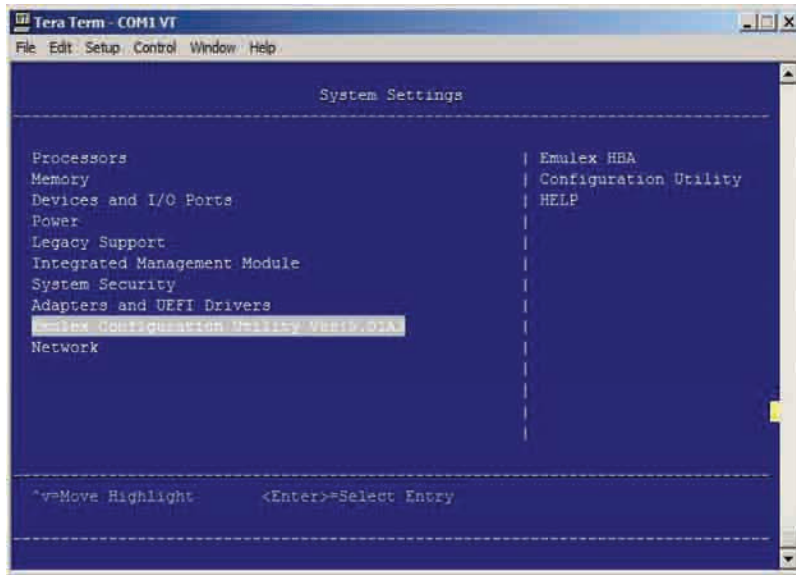


Figure 12: UEFI entry point screen
(Source: Emulex Corporation, 2011)

Flash ROM Space Growth

One development concern was the ROM space size increase due to the need to carry the 1.10 (drvcfg) code. Emulex decided to continue to carry the 1.10 (drvcfg) code because this allowed maintenance of a single codebase that could support both pre- and post-UEFI 2.1 systems. However, Emulex decided that only one configuration utility would be exposed on a given platform and this would be dependent on the UEFI support level of the system the driver was running on.

To determine what version of UEFI a system supports, you access the Revision entry in the `EFI_TABLE_HEADER` structure contained in the EFI System Table.

Implementation and Advantages of Having HII

The following describes the implementation and advantages of using HII.

Integration with System BIOS Browser Menu and Navigation

With HII, Emulex adapter configuration menus integrate into the BIOS browser, and adapter setup is significantly changed. HII makes the integration much easier, but at first, it was a challenge to determine how to transition configuration utility menus and navigation. They had previously existed in an environment where they were completely coded and controlled by the Emulex utility (using the Simple Text Protocol).

“To determine what version of UEFI a system supports, you access the Revision entry in the `EFI_TABLE_HEADER` structure contained in the EFI System Table.”

Now with HII, the system BIOS HII browser controls all these processes. Since it was not possible to keep the same look and feel of the 1.10 configuration utility, developers designed the new configuration menu forms to flow in a logical manner, allowing the user to configure the same set of parameters as they had in the past.

As mentioned above, developers needed the ability to update forms to display dynamically discovered devices. These device entries would require submenus. Emulex also needed the ability to navigate into those submenus. To accomplish this, the “Create OpCode” library functions contained in the UEFI EDK are very useful.

User Input and Saving the Configuration Data

Acquiring user data input is made considerably easier by leveraging the HII system BIOS browser. Instead of using the Simple Input Protocol to return a single input character, one can define a buffer storage structure in the VFR file. The browser populates the members of this structure according to the data type that is defined for the corresponding question.

“Any function that requires a configuration parameter value can call the library function GetBrowserData.”

Any function that requires a configuration parameter value can call the library function GetBrowserData. This call returns the current buffer storage structure values. The configuration utility stores these values in the HBA’s flash memory, allowing each individual adapter to be programmed back to the desired state after a restart.

HII Protocols for External Applications to Push Configuration Data to the Driver

OEMs typically use a proprietary web-based configuration utility for managing their blade servers. The Emulex Fibre Channel boot driver contains code that accepts various commands from these utilities. Common commands sent to the Emulex driver may be to specify new adapter World Wide Names (WWNs) and to discover specific boot LUNs. The Configuration Routing and Configuration Access Services introduced in the UEFI 2.1 specification allow for a more standardized way of provisioning a platform.

UEFI Adoption

Working over the past ten years with Intel on driver development, Emulex has initiated and developed many driver improvements using UEFI and HII. UEFI is the most current and useful system firmware development standard today, also providing useful development tools for OEM and IHV driver engineers. As UEFI continues to enjoy widespread adoption by OEMs, Emulex expects they will see the significant added value using HII to manage their server platforms locally and remotely.

Authors’ Biographies

Nathan Skalsky is an advisory firmware engineer / software development manager at IBM’s Systems and Technology group in Raleigh, North Carolina. Nathan received a Outstanding Technical Achievement Award for his

leadership role in the development of IBM's first, second, and third generation UEFI-compliant x86 Servers and Blades. Nathan's development interests include enterprise reliability (RAS), security/trusted platform technologies, and high performance computing (HPC). He has coauthored several research papers on embedded systems, human computer interaction, and trusted computing, and now manages a platform security development team at IBM.

Terry Kirch is a senior principal engineer in the HBA boot driver group at Emulex Corporation and has more than 30 years software and embedded software development experience. Terry's first exposure to BIOS code development came when he joined Phoenix Technologies in 1995. Over the years Terry has provided code for development projects that have ranged from avionics test boxes to video games.

Al Rickey is a Principal Software Engineer in the HBA boot driver group at Emulex Corporation and has been involved in storage driver development for 20 years. Al holds three patents related to storage media formatting. He started his career writing educational software (college-level interactive math tutor software suites for textbook publishing companies), migrated to storage drivers, ATA diagnostics, and data recovery in the early 1990s.

Michael A. Rothman is a Senior Staff Engineer in the Software and Services Group at Intel and has more than 20 years of operating system and embedded software development experience. Michael holds over 200 patents and was awarded an Intel Achievement Award for some of his systems work. He started his career with kernel and file system development in OS/2 and DOS and eventually migrated to embedded operating systems work and firmware development.

<http://www.twitter.com/MichaelARothman> and michael.a.rothman@gmail.com

BOOTING IN AN INSTANT

Contributors

Michael A. Rothman
Intel Corporation

John Mese
Lenovo

“Power On Self Test (POST) is a longstanding term that typically refers to the actions a BIOS undergoes during its initialization of a platform.”

The PC platform today scales from the deeply embedded solution space to giant server cluster solutions and everything in between. One common desire across all of these domains is the desire to reduce the time it takes for the platform to initialize (boot). With this article, many elements associated with boot performance will be discussed from various points of view in the technology food chain. In addition, methods for measuring boot performance will be discussed, how the evolution of standards (such as UEFI) resulted in boot performance enhancements, and examples of how these technology elements were incorporated into products to provide meaningful results to the end-user.

The Elements of the Boot Process

This section deconstructs the elements of the boot process and the items that might affect the behavior or characteristics of a platform’s initialization.

What Is POST?

Power On Self Test (POST) is a longstanding term that typically refers to the actions a BIOS undergoes during its initialization of a platform. Even though many of these actions can vary in their details, there are a variety of activities that a platform undergoes that drive a series of different behaviors based on certain technical and nontechnical requirements. For instance, whether or not a platform allows certain devices to be initialized during the platform boot process, whether or not the boot can be interrupted by the user, supporting older boot standards, as well as a myriad of other desired behaviors can have a factor in the boot performance characteristics of a system.

Figure 1 illustrates the evolution of the platform initialization from the first moment that power is applied until the point where the BIOS hands-off to the target O/S.

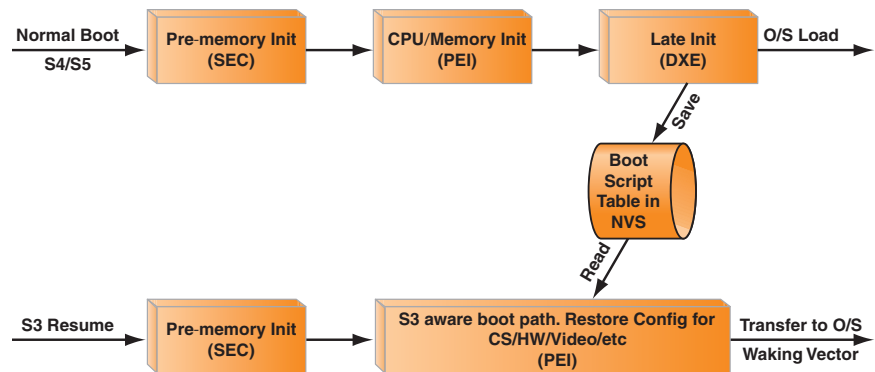


Figure 1: UEFI boot phases and ACPI behavior (Source: Intel Corporation, 2011)

For a more detailed description of each of these phases, refer to “Reducing Platform Boot Time”[1].

Factors Affecting Boot Performance

Boot performance is not necessarily isolated to the hardware characteristics of a platform. Often there are various elements that affect how long it takes to initialize a platform, and this section discusses what those elements may be.

Marketing Requirements

Marketing requirements are not the first thing that comes to mind when an engineer sits down to optimize BIOS performance. However, the reality is that marketing requirements form the practical limits for how the technical solution can be adjusted.

- *What are the design goals?* – How does the user need to use the platform? Is it a “closed box” system? Is it a traditional desktop? Is it a server? How the platform is thought of will ultimately affect what the user expects. Making conscious design choices to either enable or limit some of these expectations is where the platform policy can greatly affect the resulting performance characteristics.
- *Platform policy* – One of the first considerations when looking at a BIOS and the corresponding requirements is whether or not an engineer can limit the number of variables associated with what the user can do “to” the system. For instance, it might be reasonable to presume that in a platform with no add-in slots, a user will not be able to boot from a RAID controller since the user cannot physically plug one in. This is where a designer enters the zone of platform policy. Even though a platform may not expose a slot, the platform might expose a USB connection. A conscious decision needs to be made for how and when these components are used. A good general performance optimization statement would be: “If you can put off doing something in BIOS that the OS can do, then put it off!” Since a user can connect anything from a record player to a RAID chassis via USB, the user might think that they would be able to boot from a USB-connected device if physically possible. Though this is physically possible, it is within the purview of the platform design to enable or disable such a behavior.
- *What type of boot media is supported?* – Even though most people are often considering the total size of the media as the primary factor and don’t necessarily give much thought beyond that, this is often a mistake. Though it may not be obvious, the choice of boot media can be a significant element in the boot time when you consider that some drives require 1–5 seconds (or much more) to spin up. The characteristics of the boot media are very important since, regardless of whatever else you might do to optimize the boot process, the platform still has to read from the boot media and there are some inherent tasks associated with doing that. Spin-up delays are one of those tasks that are unavoidable in today’s rotating magnetic media. Even though today there might be a cost and size differential between solid-state devices (SSDs) and the typical rotating

“the reality is that marketing requirements form the practical limits for how the technical solution can be adjusted.”

“A good general performance optimization statement would be: “If you can put off doing something in BIOS that the OS can do, then put it off!”

“Spin-up delays are one of those tasks that are unavoidable in today’s rotating magnetic media.”

media, the SSDs usually save not only time associated with spin-up, they do not have the seek times required for a request to moving read heads to the appropriate location on the platters associated with magnetic storage rotating media.

New Behaviors Attributable to UEFI

With the advent of UEFI, certain behavior is possible that may not have been available in what is often termed legacy BIOS. This section attempts to illustrate what some of those differences are and how they relate to platform initialization complexity.

Avoid Unnecessary Driver Execution

It is useful to understand the details of how we avoided executing some of the extra drivers in our platform. It is also useful to reference the appropriate sections in the UEFI specification to better understand some of the underlying parts that cannot, for conciseness, be covered in this article.

“The BDS phase of operations is where various decisions are made regarding what gets launched and what platform policy is enacted.”

The BDS phase of operations is where various decisions are made regarding what gets launched and what platform policy is enacted. That being said, this is the code that will frequently get the most attention in the optimization efforts.

At its simplest, the BDS phase is the means by which the BIOS completes any required hardware initialization so that it can launch the boot target. At its most complex, you can add a series of platform-specific, extensive, value-added hardware initialization that is not required for launching the boot target.

- *What is a boot target?* – The boot target is defined by something known as an EFI device path (see UEFI specification). This device path is a binary description of where the required boot target is physically located. This gives the BIOS sufficient information to understand what components of the platform need to be initialized to launch the boot target. Below is an example of just such a boot target:

```
Acpi(PNP0A03,0)/Pci(1F|1)/Ata(Primary,Master)/HD(Part3,Sig00110011)/  
“\EFI\Boot”/“OSLoader.efi”
```

- *Loading a boot target* – The logic associated with the BDS optimization focuses solely on what the minimal behavior is associated with initializing the platform and launching the OS loader. When customizing the platform BDS, you can avoid calling routines that attempt to connect all drivers to all devices recursively and instead only connect the devices directly associated with the boot target. Figure 2 is an example of that logic:

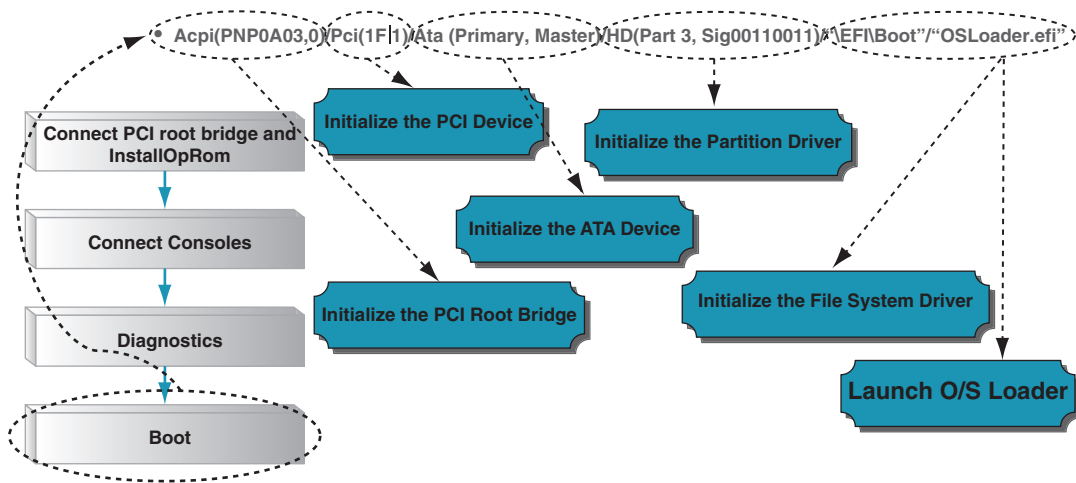


Figure 2: Deconstructing the launching of a boot target
(Source: Intel Corporation, 2010)

- *Steps taken in an optimized versus a normal boot* – Figure 3 indicates that between the normal boot and an optimized boot, there are *no* design differences from a UEFI architecture point of view. Optimizing a platform’s boot performance does *not* mean that one has to violate any of the design specifications.

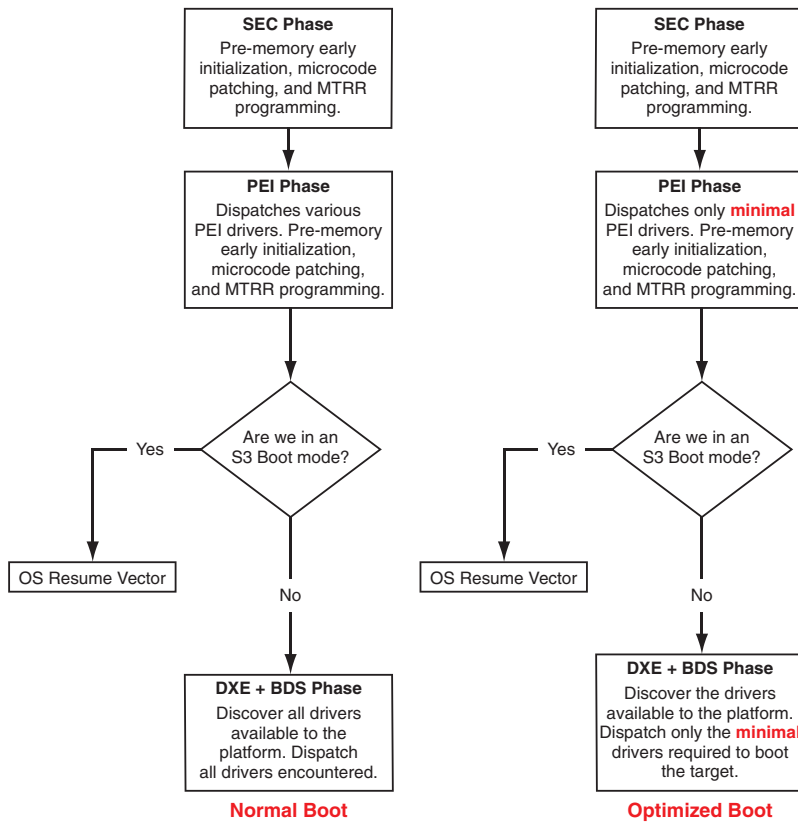


Figure 3: Architectural boot flow comparison
(Source: Intel Corporation, 2010)

“Normally the overall goal is to boot the target OS as quickly as possible and the only expected user interaction is with the OS.”

Allow Third Party Drivers to Delay the Boot? Not Anymore!

Normally the overall goal is to boot the target OS as quickly as possible and the only expected user interaction is with the OS. That said, the main reason for people today to interact with the BIOS is to launch the BIOS setup. Admittedly, there are some settings within this environment that are unique and cannot be properly configured outside of the BIOS. However, in many platforms the ability to interact with the user has caused a significant level of added slow-down due to third party drivers (such as option ROMs) being initialized and during their initialization soliciting hot-keys to be hit, delaying the boot by multiple seconds. This can potentially be multiplied by however many instances of such behavior are found in the platform.

With the advent of UEFI 2.1, and more specifically the Human Interface Infrastructure (HII) content in the UEFI specification, it became possible for configuration data in the BIOS to be exposed to the OS. Many of the BIOS settings can now have methods exposed and configured in what are not traditional (pre-OS) ways. This also allows for third party components to silently post their configuration data to the platform and it leaves up to the platform the decision on how that gets exposed to the user. This avoids numerous extraneous delays that might otherwise be encountered due to third party components trying to capture the attention of the user.

If it is deemed unnecessary to interact with the BIOS, there is very little reason (except as noted in prior sections) for the BIOS to probe for a hot-key.

Some Discovered Performance Guidelines

This section describes some guidelines to enhance the system boot performance. These guidelines are not specific for a given BIOS codebase, but could be treated as generic guidelines in firmware performance tuning.

Guideline #1: Make Good Use of System Cache

CPU cache is the fastest memory bank in the system. Correctly using CPU cache can significantly improve the boot performance especially if the original code does not set the cache optimally.

Since reading data or code from flash is very slow, it is very important to cache the range of flash when executing code from flash in the PEI phase.

Guideline #2: Proper Flash Layout

In a BIOS that complies with the Platform Initialization (PI) specification, there is a flash component concept known as a firmware volume (FV). This is typically an accumulation of BIOS drivers. It is a reasonable expectation that these FVs are organized into several logical collections that may or may not be associated with their phase of operations or functions. The access from flash is significantly slower than access from memory. Minimizing the flash access will significantly improve the boot performance. For some quick boot paths, flash access may take a very large percentage of the boot time.

“Correctly using CPU cache can significantly improve the boot performance.”

“Minimizing the flash access will significantly improve the boot performance.”

Some drivers read data or binaries from an FV, such as CSM binary, legacy option ROM, or BIOS ID. As a common design, the BIOS is intended to scan all FVs. But for a specific platform, files can be well organized so that the code can avoid scanning all FVs.

The less space a BIOS occupies, the shorter the time it takes for routines within the BIOS to read content into faster areas of the platform (such as memory). This can be done by minimizing the drivers that are required by the platform, at least for a specific boot path. For example, if you want the splash screen to appear as early as possible, you can create a FV that only contains drivers required for a graphic console. After the graphic console has connected, then it will dispatch other DXE drivers and boot.

Guideline #3: Skip Unnecessary Drivers!

If the system is designed to boot the OS as fast as possible, you can enable a special boot path that only initializes the devices required for the boot.

There are two levels of hardware initialization. The first level is initializing the hardware to make it meet the industry standard infrastructure that the OS phase would expect. This also includes initializing any hardware necessary to discover and launch the OS. For example, set the SATA controller mode to AHCI, native, or legacy so that the BIOS can launch the OS.

The second level of hardware initialization is to make certain devices (such as USB) that might not be necessary for the pre-OS phase be accessible in the pre-OS phase. Accessing files from USB mass storage devices requires a lot of driver connections in the BDS phase and can take quite a bit of time.

To boot the OS quickly, some of the second level initialization can be skipped. Those devices still can work correctly in the OS. For example, if we want to boot the OS from a hard disk, we can skip USB, network, and CD-ROM initialization, which can save a lot of time. When there is no initialization for these type of devices in the pre-OS phase, the OS has native drivers that can initialize these devices when the OS is running. In fact, the OS will typically reinitialize these devices regardless of whether the BIOS has initialized them anyway.

Guideline #4: Avoid Unnecessary Hardware Resets

Some drivers reset the device and wait for the device ready signal. For example, a PS2 keyboard often needs more than one second to do a full reset. Even for an SSD device, it takes about 800 ms to do a reset of that device.

Guideline #5: Use Saved Data

If the hardware has not changed, we can use saved data instead of accessing slow I/O to get the data. For example, reading Serial Presence Detect (SPD) data from System Management Bus (SMBus) is much slower than reading it from nonvolatile RAM. Since it takes a long time to enumerate the PCI bus, we can use saved data from the last successful boot directly to reduce the overall boot time.

“If the system is designed to boot the OS as fast as possible, you can enable a special boot path that only initializes the devices required for the boot.”

“Since it takes a long time to enumerate the PCI bus, we can use saved data from the last successful boot directly to reduce the overall boot time.”

Guideline #6: Parallelize When Possible

Firmware does not support multitasking like an OS, but we can do some parallel work according to hardware behaviors. For example, the hard disk needs seconds of time to spin up. So we can send the spin-up signal to the hard disk as early as possible, then do other initialization work, which can save the boot time.

In addition, where devices might provide DMA capabilities, there are methods that can be used to initiate a DMA transaction (that is, pass a command to a controller to fill a buffer) while the main CPU is actively doing something else.

Optimization Case Study by Lenovo

Lenovo has been at the vanguard of reducing platform boot times in the PC industry, and has illustrated this with some of the platforms they have shipped. Lenovo partnered closely with Microsoft during the development of Windows* 7 to improve on/off times on Lenovo systems. This culminated in the Lenovo Enhanced Experience, a development process at Lenovo wherein BIOS, drivers, and software are analyzed for performance and tuned to provide an optimized Windows experience with superfast boot, shutdown, and standby/resume times.

As a result of these efforts Lenovo built deep engineering contacts within key partners such as Microsoft, Intel, Phoenix, Symantec, and other IHV/ISVs that has allowed for a significant accumulation of subject matter expertise and allowed Lenovo to actively pursue performance-focused initiatives.

As an example of this, in January 2011 at the Consumer Electronics Show in Las Vegas, Lenovo demonstrated six Windows 7 PCs (two IdeaPads*, one IdeaCentre*, one ThinkCentre*, and two ThinkPads*) that *boot in under 10 seconds* to promote the release of the Lenovo Enhanced Experience 2.0. This started internally at Lenovo as a challenge to break through existing barriers to create an ultra-fast boot, to build a proof-of-concept system so much better than what was out there that it would help drive industry level improvements much like concept cars do in the automotive industry.

So what went into the 10-second PC? All phases of boot were analyzed (see Figure 4) and progressive opportunities for improvement were identified.

“Lenovo demonstrated six Windows 7 PCs (two IdeaPads, one IdeaCentre*, one ThinkCentre*, and two ThinkPads*) that boot in under 10 seconds.”*

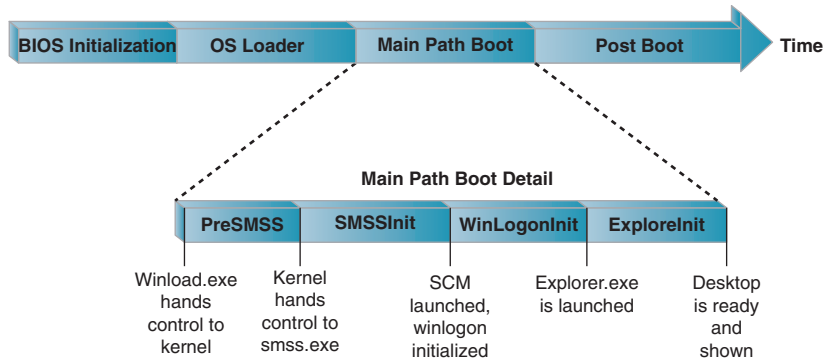


Figure 4: Analysis of the boot process

(Source: Lenovo, 2011)

First and foremost, BIOS time had to be addressed. The new generation of Lenovo systems (Intel Huron River-based) are UEFI-capable but by default expect a legacy configuration.

- Legacy support means having to invoke a Compatibility Support Module (CSM), an option ROM that adds significant time to platform initialization. By utilizing a native-UEFI-only mode, the time delay caused by having to load the CSM can be avoided.
- Furthermore, UEFI allows for multithreaded platform initialization, allowing for parallelism, whereas a legacy-compatible boot requires more serialized loading.
- Additionally, disabling devices not required for booting, like alternate boot devices such as network and optical drives, further reduces initialization time. An extremely high percentage of the time, a user just boots to the internal mass-storage drive anyway. Lenovo optimizes for the most common case.
- Finally, one last optimization, and by-product of legacy days, is displaying a startup logo—removing that yields a small but measurable time savings.

These improvements yielded BIOS initialization improvements that took the default BIOS time of 6.5 seconds down to 2.25 seconds!

The Main Path Boot phases (OS) were attacked in three ways:

1. Driver optimizations, especially video and boot start drivers
2. Minimizing CPU activity
3. Minimizing I/O activity

By reducing the shotgun blast of resource requests at boot, individual components can start quicker. As you can see in the CPU sampling and disk utilization graphs in Figure 5, reducing down to only boot-critical components yields a very clean and therefore very fast boot experience.

“improvements yielded BIOS initialization improvements that took the default BIOS time of 6.5 seconds down to 2.25 seconds!”

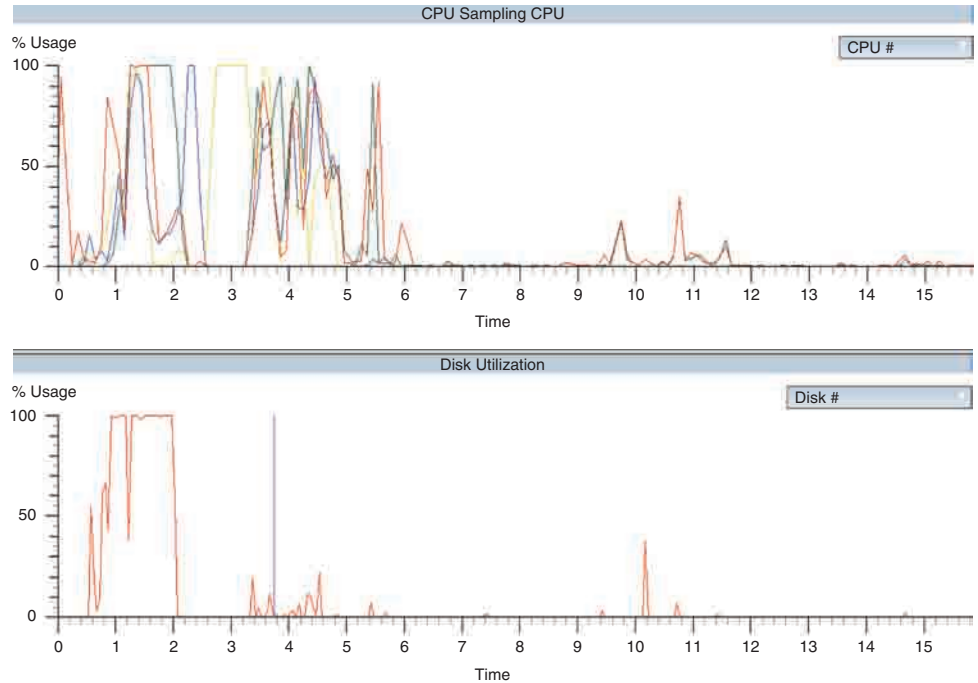


Figure 5: Optimized system: Intel® Core™ i5 2.60 GHz, 4 GB memory, Intel 160 GB SSD. This is a production version of a Lenovo ThinkPad® T420s Rapid Boot Extreme edition. OS Boot Time measured by Microsoft Velocity Test Suite 4.3.1 = 6 s. (Source: Lenovo International, 2011)

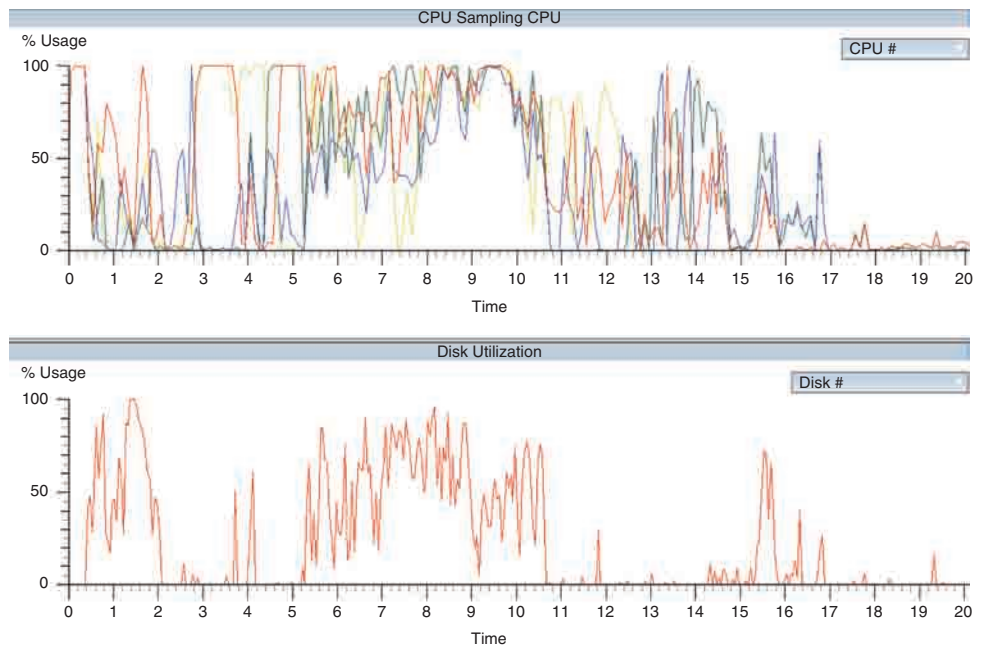


Figure 6: Unoptimized system, same hardware as in Figure 5. This, however, is a standard Lenovo ThinkPad T420s. OS Boot Time = 17 s. (Source: Lenovo International, 2011)

Observe the significant differences in CPU and I/O activity between the optimized system shown in Figure 5 and the unoptimized system shown in Figure 6. The CPU is very busy for the entire boot period in Graph 6, maxing out at several points. The SSD is fast so I/O activity, which is traditionally the bottleneck for HDD systems, is not as significant a factor as it can be in this instance. However, the difference is still very compelling.

As you can see from the graphs, many IHVs and ISVs (and even the OS) include a lot of advanced function that loads at boot by default. In practice very little of the value-added software is required to boot the machine. Driver and software vendors need to adopt better system resource “conscientiousness” and produce software that does not “crowd” on/off transitions like boot, shutdown, and standby/resume.

Conclusion

Ultimately, the level of performance optimization that is achievable is largely subject to the requirements associated with the platform. Given sufficient probing, there are almost always methods to achieve boot speed gains by leveraging various aspects of UEFI-compliant codebases.

Figure 7 is an illustration from a separate case study that indicates the percentage of overall boot time savings for a typical BIOS codebase when applying some of the previously enumerated techniques. It highlights some of the items to focus on and areas within each BIOS codebase that deserve further investigation.

“Given sufficient probing, there are almost always methods to achieve boot speed gains by leveraging various aspects of UEFI-compliant codebases.”

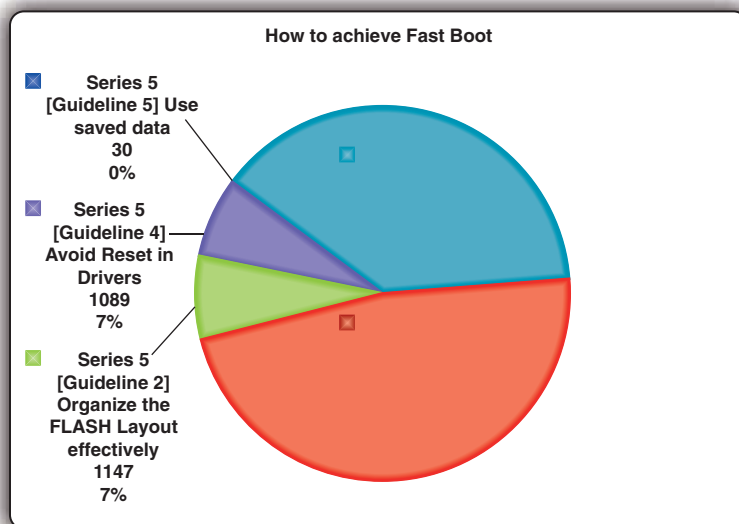


Figure 7: Effectiveness of some of the performance guidelines techniques on an example BIOS codebase
(Source: Intel Corporation, 2010)

Reference

- [1] Michael A. Rothman, Genliu Xing, Yan Wang, Jiong Gong, “Reducing Platform Boot Time”. Available at <ftp://download.intel.com/design/intarch/PAPERS/324891.pdf>

Authors' Biographies

Michael A. Rothman is a Senior Staff Engineer in the Software and Services Group at Intel and has more than 20 years of operating system and embedded software development experience. Michael holds over 200 patents and was awarded an Intel Achievement Award for some of his systems work. He started his career with kernel and file system development in OS/2 and DOS and eventually migrated to embedded operating systems work and firmware development. Can be contacted at <http://www.twitter.com/MichaelARothman> and michael.a.rothman@gmail.com

John Mese is a Senior Software Architect in the Software and Peripherals Group at Lenovo specializing in Performance and Responsiveness on the Think-branded products. His optimization work has included improvements to BIOS, firmware, device drivers, applications, and operating systems. He is also a Master Inventor at Lenovo and earned a Master of Science degree in Computer Engineering from the University of Florida.

UEFI NETWORKING AND PRE-OS SECURITY

Contributors

Magnus Nyström

Partner Architect at Microsoft Corporation

Martin Nicholes

Security Architect at Insyde Software

Vincent J. Zimmer

Principal Engineer at Intel Corporation

Readers will get an understanding for the scope and objective of the Unified Extensible Firmware Interface (UEFI) specification and the UEFI technology's role as a foundation for pre-OS security and networking. This includes the platform boot process from local and remote media, assets to be protected, threats against those assets, and the various technologies that allow for their protection. In addition to a review of these technologies, forward-looking capabilities and approaches related to UEFI are discussed.

UEFI Architecture

This section provides a basic overview of the UEFI firmware architecture. For a more detailed view of UEFI architecture, see the UEFI Web site [1] and *Beyond BIOS*[2]. UEFI provides a standard interface to shield the operating system from hardware changes. It has the ability to host chipset and peripheral boot drivers. It provides services both during the boot process and during runtime, available through architected tables.

Architecture Diagram

Figure 1 shows how an operating system loader relies upon the services provided by UEFI to launch the OS kernel. Other industry standard interfaces, such as ACPI, are available on a UEFI platform.

UEFI provides an orderly method for loading drivers and handling interdependence between them. Once the necessary drivers are loaded, a series of preconfigured boot paths or boot devices are attempted by a boot manager or dispatcher. During a boot attempt, a file is loaded using UEFI boot-time services and is executed. This file load operation is a good point to secure, as will be discussed in the section on UEFI 2.3.1 Secure Boot.

Failures of these boot attempts are handled by returning to the UEFI boot manager. However, once a boot attempt executes an OS boot loader that exits the UEFI boot-time environment, the UEFI boot-time environment cannot be re-entered, as shown in Figure 2.

“UEFI provides an orderly method for loading drivers and handling interdependence between them.”

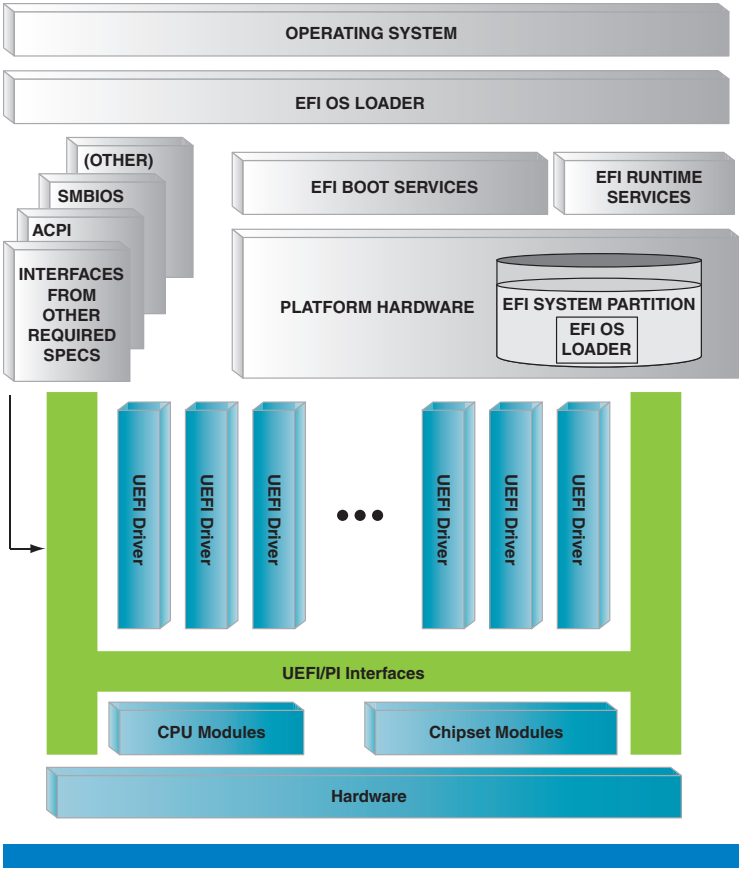


Figure 1: Firmware layering, including image loading
(Source: Intel Corporation, 2011)

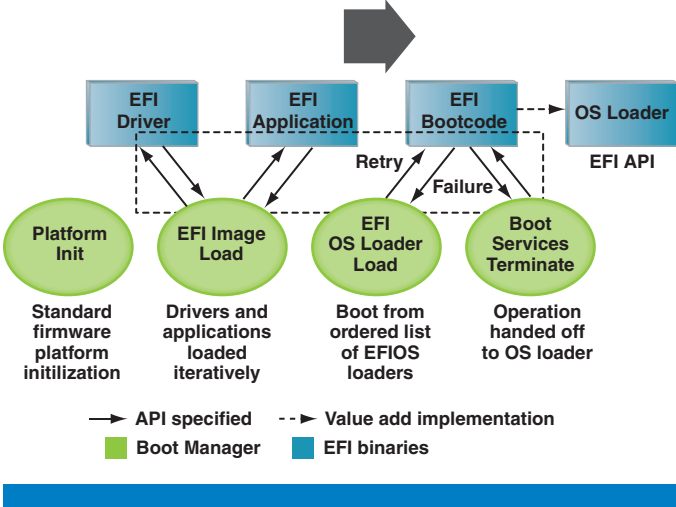


Figure 2: Firmware boot flow
(Source: Intel Corporation, 2011)

Services

Table 1 shows the boot-time services categories available to a UEFI application. The most critical services surround image handling, as this is way that new code is loaded and executed.

“Control of platform state is another critical area to review with respect to platform security.”

Name (Boot-time)	Description
Events	Create, Close, Signal, WaitFor, Check, SetTimer
Task Priority Level	Raise, Lower
Memory	Allocate, Free, GetMemoryMap
Protocol	Install, UnInstall, ReInstall, Notify, Locate, Open, Close, Information, Connect, Disconnect
Image	Load, Start, Unload, EntryPoint, Exit, ExitBootServices
Misc	SetWatchDogTimer, Stall, CopyMem, SetMem, GetNextMonotonicCount, InstallConfigurationTable, CalculateCrc32

Table 1: UEFI boot services
(Source: Intel Corporation, 2011)

Table 2 shows the runtime service categories available in the UEFI environment. These calls can change the state of the platform by modifying volatile data, nonvolatile data and real-time clock settings. Control of platform state is another critical area to review with respect to platform security.

Name (runtime)	Description
Variables	Get, GetNext, Set, Query
Time	Get, Set, GetWakeup, SetWakeup
Virtual Memory	SetVirtualAddressMap, ConvertPointer
Misc	GetNextHighMonotonicCount, ResetSystem, UpdateCapsule, QueryCapsuleCapabilities

Table 2: UEFI runtime services
(Source: Intel Corporation, 2011)

Earlier versions of UEFI (UEFI 2.0) provided little in the way of security services. For example, there was no way to save platform data such that only the creator of the data could modify or delete it. In addition, earlier versions of UEFI did not provide basic authentication tools, such as hashing or decryption capabilities. This led to several development directions in an attempt to add security to the UEFI environment: one involved measuring and recording platform state (trusted computing), and the other moved to enhance UEFI security capabilities (secure boot). The next section starts with conceptual models of a computer system and leads into a detailed review of newer UEFI capabilities that enhance UEFI security.

Underlying UEFI Implementation

Although the prior discussions have been focused on aspects of UEFI, UEFI can be built upon another firmware standard called the Platform Initialization (PI) Standard [3]. The PI Standard provides a standard flow and architecture for early machine initialization. The security issues related to PI are discussed here, along with some related issues with respect to UEFI.

“The PI Standard provides a standard flow and architecture for early machine initialization.”

UEFI by design allows files to be discovered and run from arbitrary locations in the system. Following is an overview of the power of this facility and touches on the security risks associated with loading files during the boot process.

Platform Initialization (PI) Standard

The PI specification allows a UEFI-based system to ride upon a standard firmware architecture. The boot process of such a system proceeds through a series of phases. Each phase has its own unique security advantages as well as its own risks.

SEC, PEI, and DXE

The security (SEC) phase of the PI platform boot must handle different types of platform reset events. SEC is also the root of trust for the system, providing a control point for further launch of firmware on the system. The main advantage of the SEC phase is to provide an anchor point from which to build an authenticated boot process. The SEC phase must find and transfer control to the Pre-EFI Initialization (PEI) phase of the boot process, once temporary memory is available. Of course, depending on location of the PEI code, as well as platform policy, the PEI code must be authenticated before execution.

The main purpose of the PEI phase is to provide an environment for PEI module execution. So early in the PEI phase, the PEI dispatcher is started. PEI modules typically perform low-level platform initialization of embedded devices and chipset, such as serial port initialization. Another requirement of the PEI phase is to discover platform information, create a database of this information in hand-off blocks (HOBs), and pass the database onto the DXE phase of platform boot. Again, PEI modules can be made subject to authentication before running the module. Typically PEI modules are part of the core firmware of the platform, and could be considered static and trusted for a particular platform model. However, there is no requirement in the PI specification regarding location of PEI modules, and so there could be platforms where some or all of the PEI modules would need to be authenticated to maintain platform integrity.

“The main purpose of the PEI phase is to provide an environment for PEI module execution.”

DXE as Preferred UEFI Core Embodiment

The final stage of the PI specification starts when the DXE Initial Program Load (IPL) is located and executed by the PEI phase. The DXE stage consumes information about the platform through the HOBs and creates a more complete environment for drivers. The DXE dispatcher is responsible for finding and launching DXE drivers, and in this phase drivers may come from many sources, as will be described later.

“The DXE dispatcher is responsible for finding and launching DXE drivers.”

Platform Manufacturer (PM) Authority

PI code and UEFI core only come from a system board manufacturer and are not arbitrarily extensible by third parties. A platform manufacturer is responsible for the delivery, authenticity, and protection of firmware that implements the PI and core UEFI services. In order to maintain platform integrity, this firmware must be changed under platform manufacturer control.

Achieving this control is platform-dependent, but true security methodology, including hashing and cryptographic authentication, is required. Some tools for firmware update have been provided in the UEFI specification and are touched upon later. Controlled firmware update is required for secure boot, as well as for TCG measured boot.

CRTM for TCG Boots

The Trusted Computing Group (TCG)[4] standard requires a Core Root of Trust for Measurement (CRTM). One method to implement a CRTM is to use a Static CRTM (S-CRTM), which is the core platform firmware provided in the flash part that comes with the system, as described earlier in this section. The S-CRTM is responsible for measuring any code that executes after the S-CRTM. Unlike secure boot, the measured boot only provides a record of all the firmware modules that have been run and does not provide any judgment about the integrity of the firmware modules.

Secure, Rollback-Protected Updates

In order to meet the need for a controlled method of updating platform firmware, several tools have been developed. UEFI 2.3 provides a protocol called Firmware Management Protocol. This protocol provides a standard way to control firmware on a platform. Any device in the system, from a PCIe card to the main system firmware, can expose this protocol so that a single tool can track and update firmware revisions. Although this protocol does not explicitly define firmware image authentication support, the support can be added into a system in various ways.

Another mechanism supporting secure firmware updates is the use of an EFI Update Capsule. If possible, firmware can perform the firmware update at runtime, or perhaps during the next reset of the platform. Again, security of a capsule update is the responsibility of the consumer of the capsule. Therefore, the structure of the capsule should contain authentication information necessary to maintain platform firmware integrity.

Firmware update solutions based on either of these techniques must provide the ability to detect firmware rollback conditions. This is critical, due to the possibility of an attacker using an older firmware image with a known vulnerability. There may be situations where firmware must be rolled back due to some failure, but in those cases, platform operator intervention is required. This prevents an automated firmware rollback attack.

UEFI 2.3.1 Secure Boot

An introduction to UEFI's Secure Boot support is provided here as well as coverage of the various components associated with securely booting a platform.

“the structure of the capsule should contain authentication information necessary to maintain platform firmware integrity.”

Background

A core aspect of UEFI Secure Boot is its ability to leverage digital signatures to determine whether an EFI driver or application is trustworthy. After an initial brief introduction to the concept of digital signatures, this section describes how UEFI Secure Boot makes use of them—and other cryptographic constructs such as cryptographic hash functions—to establish a trustworthy system boot. We also discuss deployment aspects as well as how Secure Boot assists in the secure load of an operating system.

Cryptography Primer

Naturally, an article such as this one cannot provide anything but a high-level introduction to cryptography. The interested reader is referred to, for example, [5] for a more complete treatment of the topic.

Cryptographic Hash Functions

As is well known, a *hash function* h is a mathematical function from a large domain X of values into a smaller range Y . Normally, it is desirable that the projection of X into Y is such that the values of Y are evenly distributed; this reduces the risk of “hash collisions”, that is, that two values x_1 and x_2 that belong to X will have the same image y : $h(x_1) = h(x_2)$.

A *cryptographic hash function* f is a hash function that meets some additional properties:

- It shall be computationally expensive to find x_1, x_2 such that $f(x_1) = f(x_2)$ (this property is often referred to as *collision resistance*)
- It shall be computationally expensive to find an x such that $f(x) = y$ for a given y (this property is often referred to as *pre-image resistance*)

In addition, cryptographic hash functions are usually designed to allow the quick computation of $y = f(x)$ given x . Some examples of well-known cryptographic hash functions are MD5 [6], SHA-1 [7], and the SHA-2 family [7].

Cryptographic hash functions are useful in applications that need to protect data integrity since their properties make it difficult for an attacker to modify a datum (such as the image of an executable) without detection of a party with access to the original hash value. Likewise, replacement of an image for another one will be infeasible if the party that checks the image has stored the original image’s hash value (pre-image resistance).

Public-Key Cryptography and Digital Signatures

Public-key cryptography was discovered in the late 1970s by Whitfield Diffie, Ralph Merkle, and Martin Hellman [8]. The main notion until then was that in order to carry out confidential communication between two parties the two parties had to agree—out of band—on a shared secret. This secret was then used to encrypt the communication. Diffie, Merkle, and Hellman’s discovery of a technique to establish shared secrets without the need for pre-agreements and without having to know in advance with whom to communicate securely was

“cryptographic hash functions are usually designed to allow the quick computation of $y = f(x)$ given x .”

“The public key can be presented to anyone whereas the private part of the key is known only to its owner.”

revolutionary and was based on the concept of a new form of cryptographic keys consisting of a *public key* part and a *private key* part. The public key can be presented to anyone whereas the private part of the key is known only to its owner.

Ronald Rivest, Adi Shamir, and Leon Adelman built upon Hellman, Merkle, and Diffie’s ideas when they invented the RSA cryptosystem a few years later. With RSA, an entity *A* holding a private key may not only use that key to decrypt data sent to him by someone knowing the public key of *A* but *A* may also use his private key to *digitally sign* some data, as shown in Figure 3. Anyone with knowledge of the public key of *A* and access to the signed data may then verify mathematically that the data could only have been signed with knowledge of *A*’s private key—that is, only *A* could have generated the message—and that it has not been modified.

This property is extremely useful and has numerous applications. One of them is of interest in the context of UEFI Secure Boot: an issuer of a firmware driver may *digitally sign* this driver to vouch for the driver’s authenticity as well as provide assurance to a verifier that the driver has not been tampered with, as shown in Figure 4.

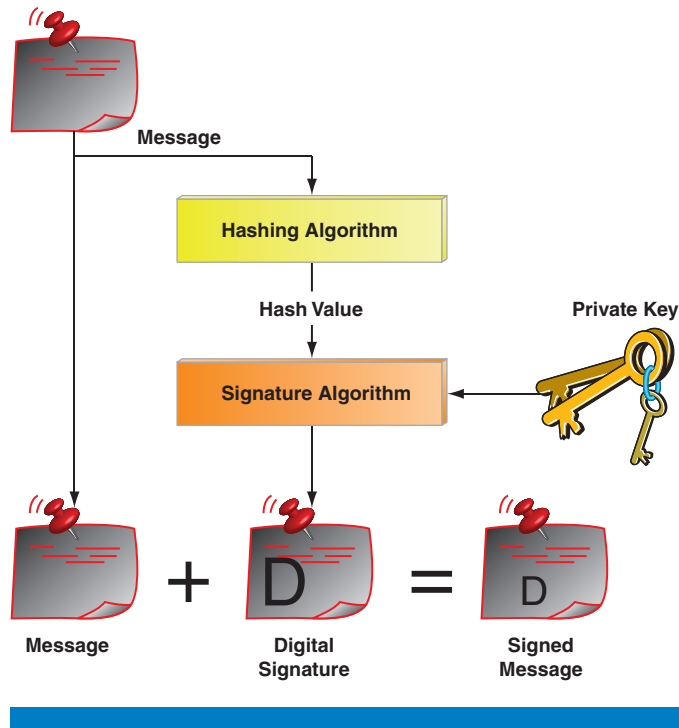


Figure 3: Digital signature creation process
(Source: UEFI Specification)

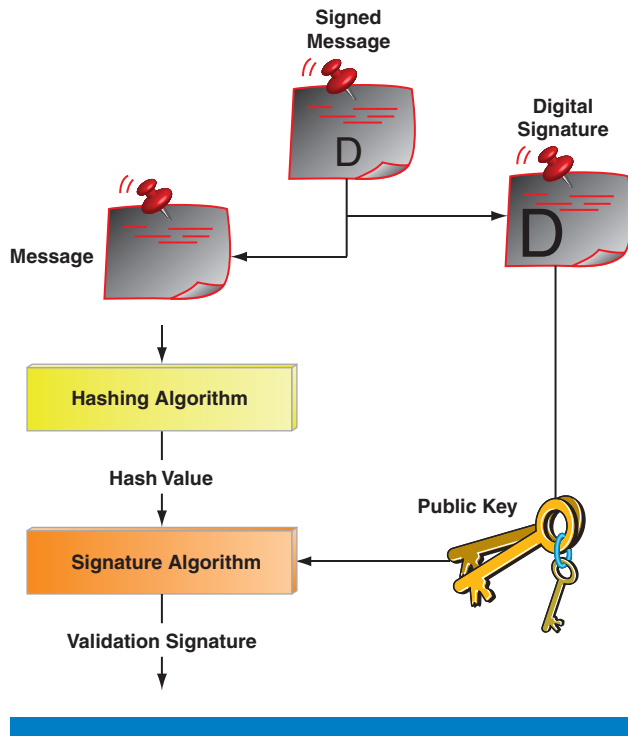


Figure 4: Digital signature verification process
(Source: UEFI Specification)

Public-Key Infrastructures

One issue conveniently overlooked in the previous discussion is *how* a relying party (a party that needs to verify a digital signature) may determine that *A* actually is the holder of the public-private key pair that is claimed to be held by *A*. To achieve this, the concept of *Public-Key Infrastructures* (PKI) has been introduced. At a high level, a third party produces digital signatures binding a public key to the holder of the private key corresponding to the public key. Such signatures are commonly referred to as *digital certificates* and the issuer of the certificates are referred to as a *certificate authority* or *CA*. The problem is now transferred to one of distributing and trusting CAs. CAs may also be specialized; for example, a given CA may only issue certificates for individuals (stating that individual *Alice* holds public key *A*) whereas another CA only issues certificates for code-signing purposes (certificates stating that signer *S* can use his private key only to digitally sign computer code). Often these specialized CAs have certificates issued by a higher-layer CA. This is convenient as it reduces the distribution problem to the problem of distributing top-level CAs (commonly referred to as *trust anchors* or *root CAs*). The top-level CAs self-sign their certificates, meaning that they sign their own certificates. Technically this is not required; all that is required is for a relying party to know that a given root CA is in possession of a given public key, but for convenience's sake (consistent use of certificates) self-signed certificates are often used.

“At a high level, a third party produces digital signatures binding a public key to the holder of the private key corresponding to the public key.”

“Once a certificate has been revoked, it can no longer be used to verify signatures made with the corresponding private key.”

“Typically, the PK is held by a device manufacturer, but it may also be held by an enterprise that wishes to have full control of the UEFI Secure Boot environment of PCs in their organization.”

PKIs with CAs also allow for *revocations*. A revocation is essentially a statement made by a CA that it is no longer the case that the unique binding between an entity *B* and the private key whose public key counterpart was present in *B*'s certificate holds. This could be due to *B* losing his private key or because it is suspected to have been compromised or some other reason. Once a certificate has been revoked, it can no longer be used to verify signatures made with the corresponding private key. For this reason, it is sometimes important to augment a digital signature with a *time stamp* allowing a relying party to determine if a signature was created before a given point in time (the time stamp itself is usually created by a third party called a timestamping authority, who also needs to be trusted by the relying party). In particular, it allows a relying party to determine if the signature was made before the certificate used to verify the signature was revoked.

For a technical description of PKIs, see [9].

Functionality

With this introduction to the cryptographic building blocks that hash functions and digital signatures constitute, we are now ready to look at their application in UEFI Secure Boot.

Root of Trust

As mentioned, in order to rely on a digital signature, the relying party needs to know that the purported signer indeed was in possession of the private key that was used to create the digital signature. As CAs are commonly used as “trusted third parties” to convey this knowledge, the distribution of CAs (and in particular root CAs) is what enables a relying party to establish trust in a signature.

In UEFI Secure Boot, there is a particular key that constitutes the basis for the trust environment. This key is the Platform Key (PK) and the holder of this key is able to modify any of the other trust anchor lists that exist on a platform. Typically, the PK is held by a device manufacturer, but it may also be held by an enterprise that wishes to have full control of the UEFI Secure Boot environment of PCs in their organization.

Trust Anchors

In addition to the PK, UEFI Secure Boot maintains two additional trust anchor databases:

- The Key Exchange Key (KEK) database
- The Allowed signature database

The former database contains those trust anchors that are allowed to modify the Allowed signature database. The Allowed signature database in turn contains those trust anchors that are used when verifying the signature on UEFI firmware images.

There is also a third database, the Forbidden database. This database identifies signers that have been revoked and can therefore no longer be trusted.

Signed Firmware

In UEFI Secure Boot, firmware that isn't explicitly *White-listed* (see below) must be digitally signed and time-stamped in such a way that the UEFI Boot Manager can verify the image's signature. UEFI uses the PE/COFF [10] format and the Microsoft Authenticode Specification [11] for signatures over PE/COFF images. The PE/COFF format contains a date/time field that may be used for timestamp purposes.

White-Listing and Black-Listing

UEFI Secure Boot extends beyond mere digital certificates for determining whether a signed UEFI firmware component is trustworthy or not, however. Besides digital certificates, UEFI Secure Boot also allows an authorized entity to identify a particular image hash as trustworthy (or not). If the hash of a given image is present in the Allowed database but not in the Forbidden database, then the fact that only an authorized entity (either the PK or a trust anchor in the KEK database) can modify the contents of the databases suffices to determine the validity of an image.

This scheme carries an advantage over most existing PKIs in that individual images can be allowed (white-listed) or revoked (black-listed); there is no requirement to revoke signers—an operation with potentially much more far-reaching implications since *all* images signed by that signer would be inoperable on a Secure Boot system. The usefulness of this is apparent when one considers a scenario where a specific driver from a vendor has been found to contain a vulnerability. Clearly, the vendor need (and most often should) not be considered malicious because of this and hence the natural action is to just black-list the driver (by adding its hash to the Forbidden database). A useful application of white-listing is when a new driver is detected but isn't signed. In this situation, an authorized entity may explicitly add the hash of the driver image to the Allowed database, allowing future system boots with it.

Authenticated Variables

From the above logic regarding malicious code, it follows that a party that wishes to update any of the databases (KEK, Allowed, Forbidden), must be able to do so only with proper authorization. The authorization model chosen for UEFI Secure Boot follows a consistent design in that it leverages digital signatures itself and hence UEFI variables that requires authentication of the caller in order to be updated are referred to as *Authenticated Variables*. In short, in order to update an authenticated variable, the caller creates a signature over the new variable value. The trusted firmware then verifies this signature before updating the value of the variable.

What would happen if an early version of a Forbidden database update was captured by an attacker and later on replayed after the attacker's key had been revoked? In a worst-case scenario, the attacker would be able to continue to have his firmware booted because the platform's black-list would never include

“UEFI Secure Boot extends beyond mere digital certificates for determining whether a signed UEFI firmware component is trustworthy or not.”

“The authorization model chosen for UEFI Secure Boot follows a consistent design in that it leverages digital signatures.”

“Associated with the rollback prevention of the authenticated variables is the rollback prevention of drivers themselves.”

“When Secure Boot is enabled, the Platform Initialization module is expected to contain a public RSA key held in read-only memory.”

the attacker’s certificate. To prevent this, any authenticated variable update includes a timestamp. Trusted UEFI firmware must ensure that the timestamp (which is part of the signature) is later in time than the timestamp currently associated with the authenticated variable before updating the variable’s value. There is one exception to this, though: UEFI 2.3.1 introduces functionality that allows an entity to append a value to an existing authenticated variable such as the Forbidden database. In this case, the timestamp is unimportant as the update does not affect values already present in the database. However, even in this case the firmware must update the time associated with the variable if the timestamp is later than the current time associated with the variable, since it will serve as a “last known update” time and may be required if a later “write” (rather than “append”) update request is made.

Rollback Prevention

Associated with the rollback prevention of the authenticated variables is the rollback prevention of drivers themselves. Imagine a scenario where a driver is known to have a vulnerability and hence a new version is distributed. Unless the UEFI implementation has maintained some means to determine the version (in an abstract sense) of the driver, it would not be able to tell if the new driver version actually is an older version of the driver currently installed rather than a newer one. This situation would allow an attacker to redistribute the vulnerable version of the driver and potentially gain control over a large number of platforms.

To remediate this situation, compliant UEFI 2.3.1 platforms must implement rollback prevention of firmware components. They may do so by using the TimeDateStamp field from the PE/COFF image header of the firmware in question or by otherwise associating a version with the driver. Firmware updates are verified against the time (or version) of the currently used driver and only if the new firmware is found to be “later” than the currently used does the update occur. NIST has recently released [12], which describes further guidelines for the protection of firmware and firmware updates.

Integration

A UEFI Secure Boot–capable platform may consist of many firmware components, including:

- UEFI Boot Code
- UEFI Boot Manager
- UEFI Drivers
- UEFI Applications
- UEFI OS Loader

When Secure Boot is enabled, the Platform Initialization module is expected to contain a public RSA key held in read-only memory. This key is used to verify the UEFI Boot Manager. Once the image has been verified, the UEFI

Boot Manager is launched and continues the platform boot by loading and initializing the EFI images as specified by boot order variables. Before the loading of each EFI image, the Boot Manager uses the Allowed and the Forbidden signature database to determine if the image should be allowed to be loaded. Only images whose signature matches an entry in the Allowed database (or whose signer is present in the Allowed database) and is not present in the Forbidden database (and whose signer is not present in the Forbidden database) are allowed to be loaded and initialized.

Information about images that are rejected due to failed signature verification or due to being associated with an entry in the Forbidden database will be stored in a UEFI table for later consumption (and possible remediation) by the platform operating system.

Once the UEFI Boot Time and UEFI Runtime services are available, updates are possible to the KEK, Allowed, and Forbidden databases (the PK may also be modified, but this would be a rare operation). In order to protect against unauthorized updates to these databases (that normally are held in NVRAM) by a faulty EFI component, implementations should preferably only allow such updates to occur via a trusted platform component, thereby guaranteeing that the signatures on such updates are valid before committing to the updates.

Deployment

There are various phases of a platform lifetime, some of which are described in [22]. For UEFI Secure Boot, this provisioning and field maintenance entails additional considerations described below.

Manufacturing Time

During manufacturing of a platform that supports Secure Boot, it is expected that the platform vendor will provision the initial Secure Boot configuration. In a nutshell, this configuration will consist of:

- Creating the initial Allowed database
- Creating the initial Forbidden database
- Creating the initial KEK database
- Setting the Platform Key (PK).

Once the final step above has occurred, the platform will no longer be in Setup mode and any further changes to the databases will require the update to be properly authorized (digitally signed).

Assuming that the Allowed database contains signers for the OS loaders that later on will be booted on the platform, the initial Secure Boot configuration is now complete. If the default mode for the platform boot is Secure Boot, the platform manufacturer may now also set the Secure Boot variable to True; this provides information to the OS loader about the EFI platform's enforced security policy.

“Information about images that are rejected due to failed signature verification or due to being associated with an entry in the Forbidden database will be stored in a UEFI table.”

“any update to the Authenticated database variables needs to be signed.”

“Since the provenance of the OS and its loader is often different than that of the system board UEFI firmware, the secure boot of the loader is how a vendor’s OS is cryptographically bound to a platform likely produced by another vendor.”

“The EFI Boot Manager concludes by verifying the signature of the OS Loader and then relinquishing control to the OS Loader.”

Field Management

Once a Secure Boot device has been deployed in the field, there may, as previously indicated, occasionally be a need for modifications to the Secure Boot-related databases. For example, a firmware image may have been found to be vulnerable to a security threat and hence should not be allowed to be in the boot path any longer. Likewise, the platform owner may want to add a new KEK signer to allow for more flexible updates to the Allowed or Forbidden database.

Such management occurs through the EFI Runtime Services and in particular the Variable Services. As described, any update to the Authenticated database variables needs to be signed. The signature must be in the form of a PKCS #7 [20] signature and must contain a timestamp to allow the EFI Runtime Services to verify that the update is not a replay of a previous update.

Recovery Situations

There may be situations when a new driver has been found to be incompatible with the platform or contain a flaw that makes it unsuited for use on the device. Since Secure Boot does not allow for programmatic rollback of firmware (because this would open an attack vector in that earlier versions of the firmware with known vulnerabilities could be reintroduced), this could present an issue that in the worst case would render the device unusable.

Fortunately, however, platforms and platform administrators may avoid this situation in at least two ways:

- By requiring a physically present user to accept the “older” firmware component to be reintroduced.
- By re-signing the earlier firmware image, thereby creating a new image with a “fresher” time-stamp than the recently installed driver.

Such functionality provides robustness without compromising system security.

OS Load Aspects

One usage of UEFI Secure Boot includes the invocation of the operating system loader. Recall that in UEFI, the loader is a UEFI executable with a subsystem type of boot service application. Since the provenance of the OS and its loader is often different than that of the system board UEFI firmware, the secure boot of the loader is how a vendor’s OS is cryptographically bound to a platform likely produced by another vendor. The various modalities of this bootstrap are described below.

Fixed Local Storage

In the most common case, the OS is present on some fixed local storage, such as a local hard disk. The EFI Boot Manager concludes by verifying the signature of the OS Loader and then relinquishing control to the OS Loader. Once in control, the OS Loader performs the OS load and, at some point in this process, calls the `ExitBootServices()` Boot Service function. After this call, only UEFI Runtime Services will be available.

Since EFI Variable Services are Runtime services, the OS may still perform updates to Secure Boot-related variables; however, the responsibility for verifying the validity of such updates still rests with the trusted firmware.

Removable Media

UEFI also allows OS load from removable media. The EFI Boot Manager reads the Boot order variable in order to determine whether to prioritize boot from removable media or from fixed local storage. Secure Boot still determines the validity of the OS Loader.

Network-based Access

A third alternative is to perform a network-based OS load, such as using the PXE [13] pre-boot environment protocol to download an OS loader.

The BIOS Integrity Services (BIS) protocol was deployed prior to the advent of the UEFI 2.3.1 Secure Boot. This was done because legacy BIOS loaders could not accommodate an embedded signature. As such, a detached signature and the Boot Object Authorization (BOA) were used for BIOS. During EFI1.02 definition, this usage was simply mapped to EFI. The BOA had limitations, like no chaining or multiple authorities, for either BIOS or EFI.

UEFI BIS also did not have a way to securely provision the BOA. Going forward, UEFI Secure Boot replaces BIS usage since Secure Boot mitigates these limitations via 1) certificate-based key storage with chaining, 2) authenticated variables to authorize the updates, and 3) a multiple-entry list of allowed signers.

In addition to authenticating the integrity of the code objects, the transport can either be integrity-protected and/or have confidentiality controls applied by means of the UEFI IPsec protocol [14].

Relationship with Measured Boot

There is often confusion surrounding the terms Secure Boot and Measured Boot. The previous section clearly described various components associated with Secure Boot, and this section describes Measured Boot and how the terms relate to one another.

Background

Measured Boot is the term used for a boot in which each component loaded during the boot process is measured into a trusted environment such as a TPM [4]. The measurements taken during the boot process may be provided to a third party for attestation purposes (“Authenticated Boot”), that is, a statement about the device’s posture as indicated by the boot measurements. The combination of Secure Boot and Measured Boot is commonly referred to as *Trusted Boot* [15].

For a more thorough treatment of Measured Boot, the reader is referred to [16].

Complementary Functionality

While Secure Boot provides *local verification* of boot integrity, Measured Boot provides the basis for attested statements about the platform’s configuration. These statements may be provided to third parties for *remote* verification of the

“Measured Boot is the term used for a boot in which each component loaded during the boot process is measured into a trusted environment such as a TPM.”

“Measured Boot provides the basis for attested statements about the platform’s configuration.”

platforms configuration, such as, for example, in order to prove a platform's conformance to some defined security policy.

UEFI Secure Boot therefore works in tandem with, but independent of, Measured Boot. UEFI supports Measured Boot and the process for performing Measured Boot on a UEFI platform is described in [17] and [18]. [18] states that any UEFI variables that have an effect of platform configuration shall be included in the measurement process. Because of this, Measured Boot must include the Allowed, Forbidden, KEK, and PK variables (databases) in its measurements of a Secure Boot-configured platform.

Example Usage Scenarios

This section illustrates use of Secure Boot functionality and Signature Databases.

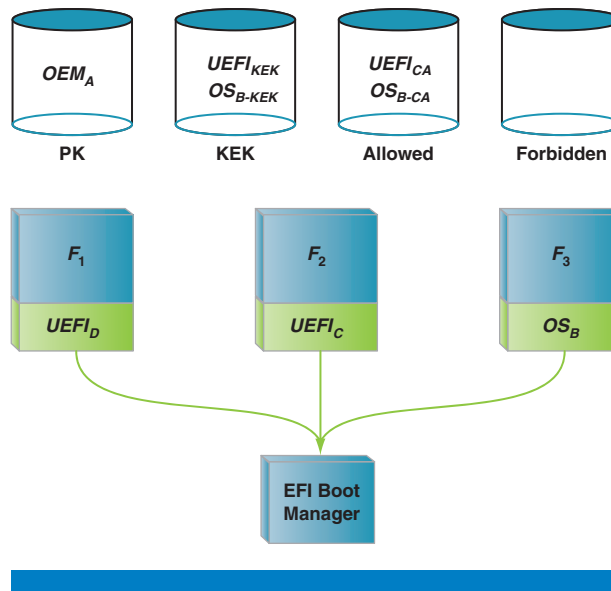


Figure 5: Boot succeeds
(Source: Microsoft Corporation, 2011)

“the boot succeeds because all of the image signatures are verified and no image (or image signer) is present in the Forbidden database.”

In Figure 5, the boot succeeds because all of the image signatures are verified and no image (or image signer) is present in the Forbidden database. (In Figure 5, image F_1 is signed by a certificate issued by the UEFI CA $UEFI_{CA}$ identifying the vendor as D , image F_2 is signed by a certificate issued by the UEFI CA $UEFI_{CA}$ identifying the vendor as C , and image F_3 is signed by a certificate issued by the OS vendor B CA OS_{B-CA} (identifying the vendor as B).

At this point, an update to the Forbidden database with the image hash of firmware image F_1 occurs through a UEFI $SetVariable()$ call. The update is authenticated with a $UEFI_{KEK}$, and since the timestamp on the signature was fresher than the timestamp associated with the existing Forbidden database value (which was empty) the update succeeds. We now have the situation illustrated in Figure 6.

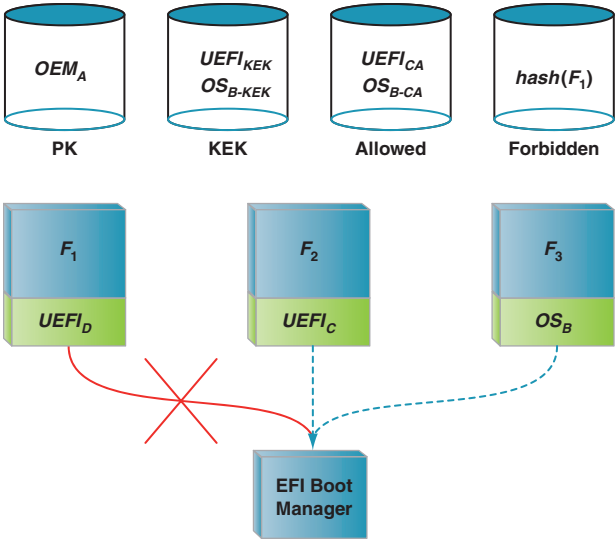


Figure 6: Failure due to presence of F_1 in the boot path (Source: Microsoft Corporation, 2011)

A Secure Boot along this boot path will fail because the hash of image F_1 is now in the Forbidden database. The dashed line indicates that the UEFI Boot Manager never attempts to load F_2 or F_3 .

Next, a firmware update occurs in which F_1 is replaced with F'_1 . The update is signed with a key chaining back to $UEFI_{CA}$ and the timestamp in the signed image as well as the signature itself is verified before the update is committed. This gives the situation illustrated in Figure 7.

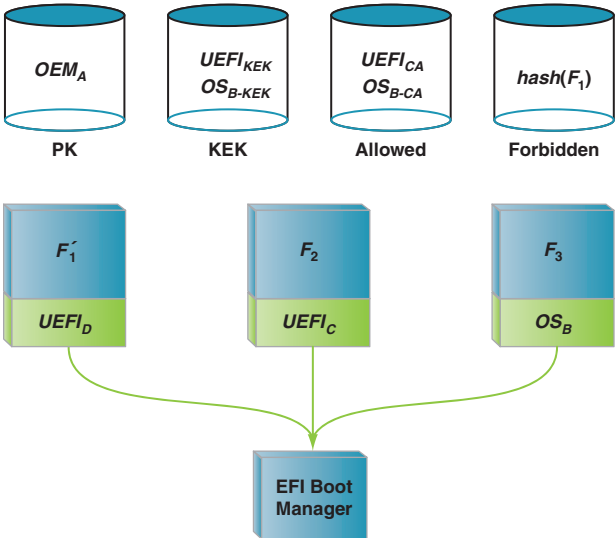


Figure 7: Boot succeeds thanks to new F'_1 image (Source: Microsoft Corporation, 2011)

A Secure Boot now succeeds since F'_1 is signed by a signer chaining back to an entry in the Allowed database and none of the encountered images have a hash value equal to $hash(F_1)$.

Finally, it is determined that no images issued by vendor C shall be allowed to be booted. While a rare event in practice, it is included here for illustrative purposes. In this situation, an update to the Forbidden database occurs through a UEFI `SetVariable()` call that *appends* the certificate of C to the Forbidden database. The `SetVariable()` call must naturally be signed by an entity whose certificate chains back to an entity in the KEK database, pass all authentication checks, and so on. The current platform state becomes that shown in Figure 8.

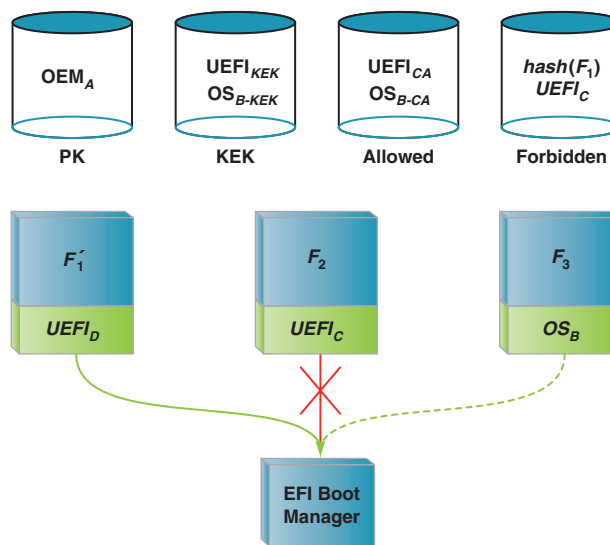


Figure 8: Boot fails due to image F_2 signature chaining to $UEFI_C$
 (Source: Microsoft Corporation, 2011)

In this situation, Secure Boot in this boot path again fails because of the recognition by the UEFI Boot Manager that F_2 was signed by an entity now in the Forbidden database.

Networking

One of the classic areas of concern with respect to security has been networking. The network should be considered a hostile environment because it is outside the perimeter of the platform protections. Some of the classic attacks include Man-in-the-Middle (MitM), wherein an agent can intercept traffic and corrupt traffic in flight. In addition, the end-point of the communication can impersonate the intended party; this is often referred to as the “*lying endpoint problem*” [19].

“The network should be considered a hostile environment because it is outside the perimeter of the platform protections.”

For purposes of network boot, there are two perspectives. The first is identifying the machine to the network infrastructure. This is for corporate IT to protect its network from a possibly rogue machine. Technology to address this includes authentication protocols, such as 802.1X-controlled ports, and challenge-response capabilities in the Extensible Authentication Protocol (EAP) over LAN (EAPOL). EAP handlers can include EAP-CHAP, EAP-TLS, or EAP-KRB for Challenge-Handshake Application Protocol, Transport Layer Security or Kerberos, respectively. In this case, the UEFI machine needs to prove its identity or proof of some capability, such as a pre-shared key (PSK) for CHAP or Kerberos, or an asymmetric key pair for TLS, to the network infrastructure authentication server.

Technology such as hardware VLAN can be used to segregate this machine from the rest of the main network or quarantine the UEFI machine in case of an authentication failure/remediation.

The other perspective is for the client machine to protect itself from the network. This can include using the above-listed authenticated protocols in order to have the network prove it is authorized. A common use case here is ISCSI logon using CHAP, with the network operator setting the PSK in both the SAN target and the UEFI ISCSI initiator.

But the more common protection mechanism is the UEFI Secure boot. Just as a mutable EFI System Partition is an attack vector for UEFI images from host malware, the network opens up a larger class of malware. As such, the ability to verify the UEFI executable loaded across the network represents the last defense against network-delivered malware. This problem was appreciated in the late 1990s with the BIOS Integrity Services (BIS), which EFI 1.02 inherited in 1999. As mentioned earlier, the problem with BIS deployment is that it had a single certificate, the Boot Object Authentication (BOA). BIS was also a commercial failure in that it never specified a credential deployment strategy, such as that described for UEFI 2.3.1 Secure Boot. It was a single root, it didn't define any BOA protection mechanism/update, it didn't leverage industry-standard tool-flows, and it didn't cover bootstrap from boot paths other than the network.

An additional set of capabilities for authentication, confidentiality, and integrity includes UEFI IPsec support. This provides for secure Internet Protocol communication. This protects any application traffic across an IP network and is mandatory for IPv6. Features include the Authenticated header (AH), the Encapsulating Security Payload (ESP), and the Internet Key Exchange (IKEv2). In addition, integrity mechanisms such as HMAC-SHA1, and encryption ciphers such as TripleDES-CBC and AES-CBC are incorporated. Finally, the stack can operate in both Transport or Tunnel modes for both IPv4 and IPv6 connections. And the authentication support includes pre-shared key and X.509 certificates.

Beyond the IPv6 and IPsec UEFI interfaces, the wire-protocol for network booting has commensurate evolution to the UEFI APIs. Specifically, in the

“A common use case here is ISCSI logon using CHAP, with the network operator setting the PSK in both the SAN target and the UEFI ISCSI initiator.”

“An additional set of capabilities for authentication, confidentiality, and integrity includes UEFI IPsec support.”

“Beyond the IPv6 and IPsec UEFI interfaces, the wire-protocol for network booting has commensurate evolution to the UEFI APIs.”

DHCPv6 extensions for IPv6 network booting, the boot file information is sent as a Uniform Resource Locator (URL); the network boot option details are described in both the UEFI 2.3.1 specification and in IETF RFC 5970. As such, the UEFI client machine and the boot server can negotiate various types of downloads, including TFTP, FTP, HTTP, NFS, or iSCSI. This allows the network capabilities to track the needs of the market and the machine's firmware capabilities. The default bootstrap on IPv6 is referred to as *netboot6*; this is a generalization of PXE 2.1 bootstrap on IPv4.

In order to exercise some of these capabilities, there are open source examples of these codes at [21].

Summary

This article has described how the various integrity preserving technologies in UEFI allow for the “extensibility” of UEFI to be a boon for companies, not a sharp edge that damages them. Technology that can be applied to that end includes support for measured boot and the trusted platform module and UEFI Secure Boot. UEFI was designed as a policy-driven boot loader environment, and it is with Secure Boot that a cryptographically strong set of controls are introduced, which ensures that the party with the appropriate administrative role, whether platform or OS, can manage this policy. And the policy-driven controls can be applied for boot scenarios that entail either locally-attached media or a network-attached server.

References

- [1] Unified Extensible Firmware Interface (UEFI), at <http://www.uefi.org/home>
- [2] V. Zimmer et al. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*, Second edition. Intel Press. November, 2010
- [3] The UEFI Platform Initialization (PI) Specification, Volumes 1–5, July 2010
- [4] Trusted Computing Group (TCG), at <http://www.trustedcomputinggroup.org/>
- [5] A. Menezes et al. *Handbook of Applied Cryptography*, Fifth edition, CRC Press, August 2001
- [6] R. Rivest, “The MD5 Message-Digest Algorithm,” RFC 1321, April 1992
- [7] “Secure Hash Standard”, NIST, FIPS Publication 180–3, October 2008
- [8] W. Diffie et al. “New Directions in Cryptography,” IEEE Transactions on Information Theory, 22(6):644–654, November 1976

- [9] R. Housley et al. *Planning for PKI: Best Practices Guide for Deploying Public Key Infrastructure*, Wiley, March 2001
- [10] “Microsoft PE and COFF Specification”, Microsoft, Revision 8.2, at <http://msdn.microsoft.com/en-us/windows/hardware/gg463119>
- [11] “Windows Authenticode Portable Executable Signature Format”, Microsoft, August 2008, at <http://msdn.microsoft.com/en-us/windows/hardware/gg463180>
- [12] D. Cooper, et al. “BIOS Protection Guidelines,” NIST SP 800–147, April 2011
- [13] Preboot Execution Environment (PXE) Specification, Version 2.1
- [14] S. Kent et al. “Security Architecture for the Internet Protocol,” IETF RFC 4301, December 2005
- [15] W. Arbaugh et al. “A Secure and Reliable Bootstrap Architecture,” in *Proceedings 1997 IEEE Symposium on Security and Privacy*, pp. 65–71, May 1997
- [16] TCG Architecture Overview, Version 1.4, <http://www.trustedcomputinggroup.org>, August 2007
- [17] TCG EFI Protocol Specification, Version 1.20, Revision 1.0, <http://www.trustedcomputinggroup.org>, June 2006
- [18] TCG EFI Platform Specification, Version 1.20, Revision 1.0, <http://www.trustedcomputinggroup.org>, June 2006
- [19] R. Sahita et al. “Mitigating the lying-endpoint problem in virtualized network access frameworks”, in *Proceedings of the Distributed Systems: operations and management 18th IFIP/IEEE international conference*, 2007
- [20] B. Kaliski “PKCS#7: Cryptographic Message Syntax Version 1.5,” IETF RFC 2315, March 1998
- [21] UEFI Developer Kit, at <http://www.tianocore.org>
- [22] M. Rothman et al., *Harnessing the UEFI Shell: Moving the Platform Beyond DOS*, Intel Press, January 2010

Authors' Biographies

Magnus Nyström is a Partner SDE in the Windows Security team where he works on architectural matters related to the next Windows release. Most recently, he was a Distinguished Engineer at RSA, the Security Division of EMC, where he worked on various aspects of RSA's technical strategy. Magnus has extensive experience from international standardization work, both from intergovernmental standards bodies as well as industry consortiums. In 2005,

he was selected to serve in the European Union's Network and Security Agency ENISA's permanent stakeholder's group together with 29 other experts from across the world. The tenure was 2.5 years and Magnus was reappointed for a second term in 2007. Magnus has about twenty patents or pending patents—all of them in the computer security space.

Martin O. Nicholes is a Firmware Architect at Insyde Software and has over 25 years of experience in software development, including firmware and operating system driver development. He worked 23 years for Hewlett-Packard in the areas of server firmware development, server firmware architecture, and server security. He holds 10 patents in the area of firmware architecture. He is CISSP certified. He received his B.S. in Engineering from San Jose State University and a M.S. and Ph.D. in Electrical and Computer Engineering from the University of California, Davis

Vincent J. Zimmer is a Principal Engineer in the Software and Services Group at Intel Corporation and has over 19 years experience in embedded software development and design, including BIOS, firmware, and RAID development. Vincent received an Intel Achievement Award and holds over 200 US patents. He has a Bachelor of Science in Electrical Engineering degree from Cornell University, Ithaca, New York, and a Master of Science in Computer Science degree from the University of Washington, Seattle. He can be contacted at <http://www.twitter.com/VincentZimmer> and vincent.zimmer@gmail.com

DEBUGGING FIRMWARE BASED ON THE UNIFIED EXTENSIBLE FIRMWARE INTERFACE

Contributors

Stefano Righi

American Megatrends, Inc.

Brian Richardson

American Megatrends, Inc.

Jiewen Yao

Intel

Elvin Li

Intel

Every software developer knows that debugging is more difficult than coding. Debugging is one of the most common tasks in product development and maintenance. In each phase of software engineering, engineers need debug tools. For UEFI firmware, this is an even larger problem. An environment with rich debug capability will speed up development for UEFI firmware, but specialized tools are often required.

This article provides an overview of common debug solutions including hardware based debugging, system checkpoints, and source-level debugging. Firmware specific concepts such as status codes, DEBUG/ASSERT macros, and the UEFI debug protocol are introduced. This article also demonstrates source-level debugging support using AMI and Intel solutions, comparing them to hardware-based alternatives in various scenarios.

Common Debug Scenarios in UEFI

This section describes the various aspects of debugging within a UEFI environment as well as the resources available to the developer during debug.

Hardware-Based Debugging

JTAG-style connectors such as the Intel XDP debug port allow complete software execution control and direct hardware inspection. JTAG is widely used for early silicon debugging on Intel platforms, especially when the system is not stable. We can use it to configure the system, fuse registers before boot, add workarounds, view the system state even when the system is stuck and the CPU is out of control at that time.

However, JTAG-style connectors are not commonly available on production hardware. This can limit debugging on systems in production, especially if physical intrusion is not possible. There are also scenarios when only a subset of JTAG capabilities is required to debug a firmware issue, so the cost of a hardware-based debugger is not justified.

System Checkpoints

A checkpoint is a hexadecimal value sent to I/O port 0x80. This checkpoint system was developed in legacy BIOS implementations and is still used in many UEFI implementations. The BIOS outputs checkpoints through PEI, DXE, and BDS to indicate the task the system is currently executing. Checkpoints are very useful in aiding software developers or technicians in debugging problems that occur during the pre-boot process, since they are available before the local display is initialized.

“JTAG-style connectors are not commonly available on production hardware.”

“The BIOS outputs checkpoints through PEI, DXE, and BDS to indicate the task the system is currently executing.”

For BIOS developers, field technicians, and quality assurance technicians, the POST checkpoints are like the diagnostic codes used in today's automotive computers. When the "check engine light" comes on, customers expect a mechanic to read the engine code and diagnose the problem. Checkpoints expose the same feature in the BIOS, providing debug information before the OS boots.

The use of checkpoint codes for firmware debugging is limited by a number of factors:

- It requires access inside the system enclosure, which is not possible on some server, embedded, and mobile applications.
- It requires a PCI slot, which is no longer standard on newer computer platforms. Today's systems use PCI Express (PCIe) for internal expansion slots and Universal Serial Bus (USB) for external devices. PCIe and USB are preferable to system vendors due to the reduced size of the expansion slot.
- If all of the system PCI slots are populated, then the hardware configuration has to be changed to use the PCI Checkpoint Card. This may change the problem that the technician or developer is trying to debug.
- Technicians must use additional documentation to translate the hexadecimal checkpoint into useful information, typically via tables in BIOS documentation.
- Checkpoint cards do not store any history of the checkpoints from any boot session, so checkpoint information must be manually recorded during testing.

Opening a consumer desktop system to insert a PCI-based POST Checkpoint Card is simple, but the same basic diagnostic on a tablet or embedded system is difficult. Computers in point of sale (POS), gaming, digital signage, and rugged computing applications are not designed to be easily opened, which makes system diagnostics difficult if an add-in card has to be installed.

Checkpoints are also limited by the I/O port implementation, restricting checkpoints to 8-bit or 16-bit values that are specific to the firmware vendor's implementation. This legacy interface could be replaced with a larger bit field value over an interface-agnostic implementation.

Extended Debug Information

System checkpoints give us information, but the information is limited, since the use of checkpoints is just a simple method to let us know where we are.

UEFI debug strings are typically disabled on production firmware, but are often enabled on evaluation platforms and test firmware builds. When enabled, UEFI debug strings provide verbose details throughout the boot process such as driver entry notices, GUID for published protocols, and special messages reserved for code compiled in debug mode.

This example output comes from a memory detection routine in PEI:

```
[AmiDbg]MemDetect.Entry(FFFF6582)
[AmiDbg]Memory Installed: Address=1C700000; Length=3000000
```

“Computers in point of sale (POS), gaming, digital signage, and rugged computing applications are not designed to be easily opened.”

“UEFI debug strings are typically disabled on production firmware.”

```

[AmiDbg]PEI_STACK: Address=1C700000; Length=100000
[AmiDbg]HOBLIST address before memory init = 0xfef00400
[AmiDbg]HOBLIST address after memory init = 0x1c800000
[AmiDbg]PEI core reallocated to memory
[AmiDbg]Total Cache as RAM: 7168 bytes.
[AmiDbg] CAR stack ever used: 3580 bytes.
[AmiDbg] CAR heap used: 3016 bytes.
[AmiDbg]Notify: PPI Guid: f894643d-c449-42d1-8ea8-85bdd8c65bde, Peim
notify entry point: fffbda9f
[AmiDbg]Notify: PPI Guid: 36164812-a023-44e5-bd85-05bf3c7700aa, Peim
notify entry point: fffe7810
[AmiDbg]Capsule.Entry(1F6F0A5E)
[AmiDbg]Capsule Read variable service installed

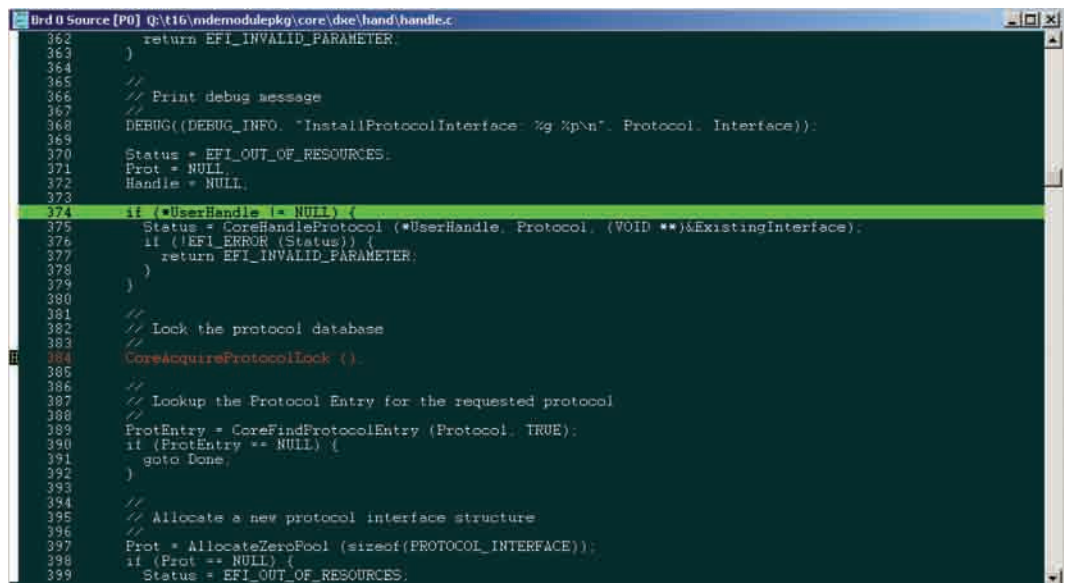
```

In many cases, UEFI debug strings provide enough advanced information to resolve firmware issues. The problem for developers is finding a simple way to view and capture the strings. UEFI debug strings were primarily designed for systems with RS-232 serial ports, which are not commonplace on modern computer systems.

Source-Level Debugging

JTAG is very powerful, and most JTAG software provides the capability for source-level debugging, if the image has a debug section, for instance PE/COFF PDB information and ELF Dwarf information. That helps firmware developers use tools commonly employed in software development.

Figure 1 is a source code window of the ITP/JTAG debugger. C code of the firmware can be displayed in source code window. The execution stops at line 374, and there is a breakpoint at line 384.



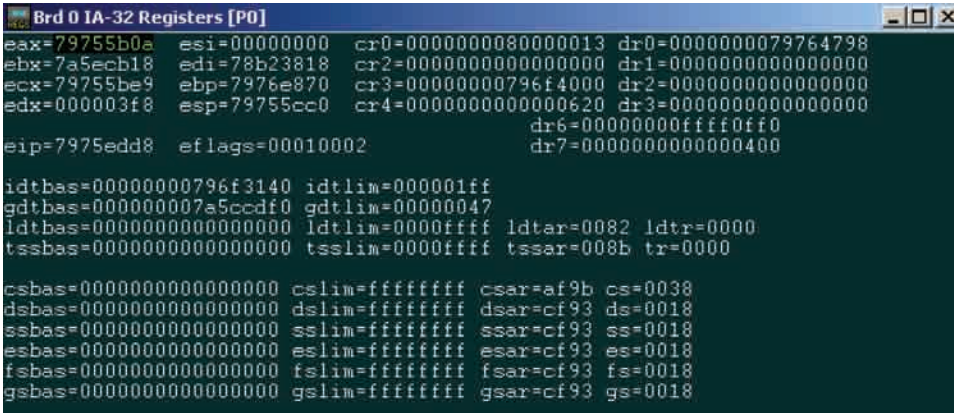
```

Drd 0 Source [P0] Q:\16\mdemodulepkg\core\dx\hand\handle.c
362     return EFI_INVALID_PARAMETER;
363 }
364
365 //
366 // Print debug message
367 //
368 DEBUG((DEBUG_INFO, "InstallProtocolInterface: %g %p\n", Protocol, Interface));
369
370 Status = EFI_OUT_OF_RESOURCES;
371 Prot = NULL;
372 Handle = NULL;
373
374 if (*UserHandle != NULL) {
375     Status = CoreHandleProtocol (*UserHandle, Protocol, (VOID **)&ExistingInterface);
376     if (!EFI_ERROR (Status)) {
377         return EFI_INVALID_PARAMETER;
378     }
379 }
380
381 //
382 // Lock the protocol database
383 //
384 CoreAcquireProtocolLock ();
385
386 //
387 // Lookup the Protocol Entry for the requested protocol
388 //
389 ProtEntry = CoreFindProtocolEntry (Protocol, TRUE);
390 if (ProtEntry == NULL) {
391     goto Done;
392 }
393
394 //
395 // Allocate a new protocol interface structure
396 //
397 Prot = AllocateZeroPool (sizeof (PROTOCOL_INTERFACE));
398 if (Prot == NULL) {
399     Status = EFI_OUT_OF_RESOURCES;

```

Figure 1: Example of a source level debug screen
(Source: Intel Corporation, 2011)

Figure 2 shows a processor registers window. It displays processor registers. Register values can be changed directly in this window.



```

eax=79755b0a  esi=00000000  cr0=0000000080000013  dr0=0000000079764798
ebx=7a5ecb18  edi=78b23818  cr2=0000000000000000  dr1=0000000000000000
ecx=79755be9  ebp=7976e870  cr3=00000000796f4000  dr2=0000000000000000
edx=000003f8  esp=79755cc0  cr4=0000000000000620  dr3=0000000000000000
eip=7975edd8  eflags=00010002  dr6=00000000ffff0fff
                                dr7=0000000000000400

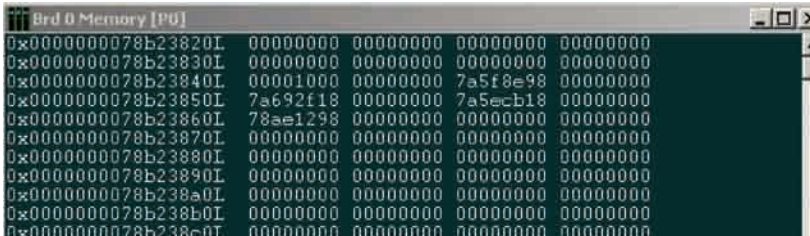
idtbas=00000000796f3140  idtlim=000001ff
gdtbas=000000007a5ccd0f  gdtlim=00000047
ldtbas=0000000000000000  ldtlim=0000ffff  ldtar=0082  ldtr=0000
tssbas=0000000000000000  tsslim=0000ffff  tssar=008b  tr=0000

csbas=0000000000000000  cslim=fffffff  csar=af9b  cs=0038
dsbas=0000000000000000  dslim=fffffff  dsar=cf93  ds=0018
ssbas=0000000000000000  sslim=fffffff  ssar=cf93  ss=0018
esbas=0000000000000000  eslim=fffffff  esar=cf93  es=0018
fsbas=0000000000000000  fslim=fffffff  fsar=cf93  fs=0018
gsbas=0000000000000000  gslim=fffffff  gsar=cf93  gs=0018

```

Figure 2: Dump screen of the registers
(Source: Intel Corporation, 2011)

Figure 3 shows a host memory window; we can change memory in this window.



```

0x0000000078b23820L  00000000  00000000  00000000  00000000
0x0000000078b23830L  00000000  00000000  00000000  00000000
0x0000000078b23840L  00001000  00000000  7a5f8e98  00000000
0x0000000078b23850L  7a692f18  00000000  7a5ecb18  00000000
0x0000000078b23860L  78ae1298  00000000  00000000  00000000
0x0000000078b23870L  00000000  00000000  00000000  00000000
0x0000000078b23880L  00000000  00000000  00000000  00000000
0x0000000078b23890L  00000000  00000000  00000000  00000000
0x0000000078b238a0L  00000000  00000000  00000000  00000000
0x0000000078b238b0L  00000000  00000000  00000000  00000000
0x0000000078b238c0L  00000000  00000000  00000000  00000000

```

Figure 3: Dump screen of a memory region
(Source: Intel Corporation, 2011)

UEFI Solutions for Pre-Boot Debugging

This section describes some of the data elements one would deal with when debugging software within UEFI.

UEFI Status Codes

Appendix D of the UEFI Specification defines a series of Status Codes (EFI_STATUS) used by UEFI interfaces to indicate successes, errors, and warnings. These codes expand on the concept of checkpoints defined by legacy BIOS, with standardized values and use of larger bit fields. They are summarized in Table 1.

Supported 32-bit Range	Supported 64-bit Architecture Ranges	Description
0x00000000-0x1fffffff	0x0000000000000000-0x1fffffffffffffff	Success and warning codes reserved for use by UEFI main specification.
0x20000000-0x3fffffff	0x2000000000000000-0x3fffffffffffffff	Success and warning codes reserved for use by the Platform Initialization Architecture Specification.
0x80000000-0x9fffffff	0x8000000000000000-0x9fffffffffffffff	Error codes reserved for use by UEFI main specification.
0xa0000000-0xbfffffff	0xa000000000000000-0xbfffffffffffffff	Error codes reserved for use by the Platform Initialization Architecture Specification.

Table 1: UEFI status codes
(Source: UEFI Specification 2.3.1)

Status codes are not restricted to a specific hardware interface, such as ISA or PCI as used by legacy checkpoints. The UEFI Platform Interface (PI) specification provides Status Code Services and code definitions for the PPI and protocols used in a Report Status Code Router.

DEBUG() and ASSERT() Macros

DEBUG and ASSERT are very useful for developers who want to debug without using source level tools. These macros produce human readable debug output that can directed to a standard serial port or USB debug port. These macros can be enabled or disabled during firmware build. Figure 4 is an example of DEBUG() output from a UEFI system.

“DEBUG and ASSERT are very useful for developers who want to debug without using source level tools.”

```

Install PPI: 30DC90C6-13FB-4A75-9E79-59E90078B9FA
Notify: PPI Guid: 30DC90C6-13FB-4A75-9E79-59E90078B9FA, Peim notify entry point: FFF402F0
Install PPI: 7400D748-FC8C-4EE6-9288-C4BEC092A410
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF54F20 EntryPoint=0x000FFF55180 CapsulePeim.efi
PROGRESS_CODE: U3020002 10
Install PPI: 3AC335E1-3832-40F4-A2FC-3054D2E13230
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF573A0 EntryPoint=0x000FFF57670 PchPolicyInitPeim.efi
PROGRESS_CODE: U3020002 10
Install PPI: 22074E71-60C0-4517-8757-7995EDFD8032
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF58A20 EntryPoint=0x000FFF59CF0 SaPeimPolicyInit.efi
PROGRESS_CODE: U3020002 10
Install PPI: 150CE416-EE63-46B6-8BA9-7322B8E04637
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF61420 EntryPoint=0x000FFF616F0 CpuPolicyInitPeim.efi
PROGRESS_CODE: U3020002 10
CpuPeimWrapper: OverClockSetup Variable was not found, using defaults
CpuPeimWrapper: Setup Variable was not found, using defaults
Install PPI: 7B8EE7B1-4E35-4556-BB56-6797E24445C2
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF6F320 EntryPoint=0x000FFF6F580 S3Resume2Peim.efi
PROGRESS_CODE: U3020002 10
Install PPI: 60582DBC-DB85-4514-8FCC-5A0F6227B147
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF7D20 EntryPoint=0x000FFF77F0 HeciInit.efi
PROGRESS_CODE: U3020002 10
Force an SS exit path.
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF7F1C0 EntryPoint=0x000FFF7F490 MdrAppPeim.efi
PROGRESS_CODE: U3020002 10
PROGRESS_CODE: U3020002 10
Loading PEIM at 0x000FFF82680 EntryPoint=0x000FFF82950 PeimHostInterface.efi
PROGRESS_CODE: U3020002 10
Register PPI Notify: AB042095-78CF-4872-8444-1B5C1006FBDA
Install PPI: F00BAEED-0837-410F-B586-2D271577FA20
CPUSU Post: C0 PEI PCI ENUM START.
    
```

Figure 4: Sample DEBUG() output from a UEFI system
(Source: Intel Corporation, 2011)

UEFI Debug Protocol

Chapter 17 of the UEFI 2.3.1 specification describes the UEFI debug architecture, a minimal set of protocols and data structures used to enable source-level debugging. UEFI debug support is presented as a pair of protocol interfaces:

- Debug Support – this protocol abstracts the processor’s debugging facilities, namely a mechanism to manage the processor’s context via caller-installable exception handlers.
- Debug Port – this protocol abstracts the device that is used for communication between the host and target. Typically this will be a serial stream style interface.

The best example of UEFI Debug Protocol implementation uses the USB Debug Port. USB-based debug solutions function thanks to the USB 2.0 Debug Port. The debug port is a function of an EHCI USB 2.0 host controller, but is implemented by most Intel peripheral controller hubs. The debug port uses a simplified USB protocol that does not require a full memory stack, unlike the standard USB protocol. This allows the debug port to be initialized in the SEC or early PEI phase, almost immediately after the firmware gets control.

USB-based debug solutions offer the most flexibility for the platform developer and field technician:

- Externally accessible – USB ports are designed for external expansion, so technicians don’t have to open the case to connect the device
- USB 2.0 enables early debugging – accessible via the USB EHCI debug port
- No additional hardware cost – use the same USB port for debugging devices or with standard USB 1.1 and USB 2.0 devices
- USB is ubiquitous – users expect USB to be enabled on today’s systems

Since debug firmware contains extra strings and information for synchronizing the host and target systems, the UEFI debug protocol is not enabled by default. UEFI firmware and drivers must be compiled in “debug mode” to enable support for the UEFI debug protocol. This allows debug capabilities to be quickly disabled between builds, allowing non-debug firmware to be properly optimized before release.

The UEFI specification only describes low-level interfaces, not the entire debug architecture. Different solutions take advantage of the UEFI debug protocols, either for simple debug output or full source-level debugging at the firmware level. This allows the developer to choose from a variety of tools.

Practical Examples of UEFI Debug Solutions

This section covers several examples of debugger interfaces and tools that a user can experience both from open source repositories as well as AMI.

“The debug port is a function of an EHCI USB 2.0 host controller, but is implemented by most Intel peripheral controller hubs.”

“The UEFI specification only describes low-level interfaces, not the entire debug architecture.”

Intel® UEFI Development Kit (Intel UDK) Debugger Tool

The Intel® UDK Debugger Tool provides the ability to debug UDK-based firmware running on an IA-32 family processor through a simple debug cable (serial or USB).

In conjunction with the Microsoft Windows* Debug Tool (WinDbg) and Linux GDB, the Intel UDK Debugger Tool provides the ability to debug UDK-based firmware on UEFI IA-32 and UEFI x64 platforms. The target side “debug agent” (SourceLevelDebugPkg) is part of the EDK II project used to create UDK2010.

Figure 5 shows the WinDbg window. It can show the C source code, and the execution is stopped at the highlighted line.

The screenshot shows the WinDbg interface with the following content:

```

v:\sourceleveldebugpkg\library\pecoffextraactionlibdebug\pecoffextraactionlib.c
AsaWriteDr7 (0x20000480);
AsaWriteCr4 (Cr4 | BIT3);
// Do an IN from IO_PORT_BREAKPOINT_ADDRESS to generate a
// returns a read value other than DEBUG_AGENT_WAIT
//
do {
  DebugAgentStatus = IoRead8 (IO_PORT_BREAKPOINT_ADDRESS);
} while (DebugAgentStatus == DEBUG_AGENT_IMAGE_WAIT);
} else if (LoadImageMethod == DEBUG_LOAD_IMAGE_METHOD_SOFT_
// Generate a software break point.
CpuBreakpoint ();
}
// Restore Debug Register State only when Host didn't change
// E.g. User halts the target and sets the HV breakpoint
// in the above exception handler.
NewDr7 = AsaReadDr7 ();
if (!IsDrxEnabled (0, NewDr7)) {
  AsaWriteDr0 (Dr0);
}
if (!IsDrxEnabled (1, NewDr7)) {
  AsaWriteDr1 (Dr1);
}
if (!IsDrxEnabled (2, NewDr7)) {
  AsaWriteDr2 (Dr2);
}
if (!IsDrxEnabled (3, NewDr7)) {
  AsaWriteDr3 (Dr3);
}
if (AsaReadCr4 () == (Cr4 | BIT3)) {
  AsaWriteCr4 (Cr4);
}

```

The right pane shows the kernel debugger connection log:

```

Microsoft (R) Windows Debugger Version 6.11.0001.404 X86
Copyright (c) Microsoft Corporation. All rights reserved.

Kernel Debugger connection established
Debugger data list address is NULL
Connected to eXDI Device 0 x86 compatible
Symbol search path is: SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
Executable search path is:
eXDI Device Kernel Version 0 HP Free x86 c
Machine Name:
Primary image base = 0x00000000 Loaded mod
System Uptime: not available
Break instruction exception - code 80000000
ffffeab6 cc int 3
0: kd> .sympath V:\BUILD\OVMFIA32\DEBUG_MY
Symbol search path is: V:\BUILD\OVMFIA32\N
Expanded Symbol search path is: v:\build\o
0: kd> .reload /f SECMAN=0x0'FFFE064
0: kd> g
SECMAN\PeCoffLoaderRelocateImageExtraActi
ffffeab7 0f21f8 mov eax,dr7
0: kd> .sympath V:\BUILD\OVMFIA32\DEBUG_MY
Symbol search path is: V:\BUILD\OVMFIA32\N
Expanded Symbol search path is: v:\build\o
0: kd> .reload /f SECMAN=0x0'FFFE064

```

Figure 5: Snapshot of the WinDbg interface with which the open source software debug package interacts
(Source: Intel Corporation, 2011)

Figure 6 shows the Linux GDB execution; it demonstrates the calling of `PeiServicesAllocatePages`.

```

0x07fccbf7 <HandOffToDxeCore+15>:  89 45 b8      mov   %eax, -0x48(%ebp)
(gdb) l
60      VOID
61      HandOffToDxeCore (
62          IN EFI_PHYSICAL_ADDRESS  DxeCoreEntryPoint,
63          IN EFI_PEI_HOB_POINTERS  HobList
64      )
65      {
66          EFI_STATUS                Status;
67          EFI_PHYSICAL_ADDRESS      BaseOfStack;
68          EFI_PHYSICAL_ADDRESS      TopOfStack;
69          UINTN                     PageTables;
(gdb) s
The target architecture is set automatically (currently i386)
The target architecture is assumed to be i386:intel
76      Status = PeiServicesAllocatePages (EfiBootServicesData, EFI_SIZE_TO_PAGES (STACK_SIZE), &BaseOfStack);
=> 0x07fccbfa <HandOffToDxeCore+18>:  8d 45 d8      lea  -0x28(%ebp),%eax
0x07fccbfd <HandOffToDxeCore+21>:  50           push  %eax
0x07fccbfe <HandOffToDxeCore+22>:  6a 20        push  $0x20
0x07fcc00 <HandOffToDxeCore+24>:  6a 04        push  $0x4
(gdb) s
The target architecture is set automatically (currently i386)
The target architecture is assumed to be i386:intel
65      {
=> 0x07fcc02 <HandOffToDxeCore+26>:  89 55 bc      mov   %edx, -0x44(%ebp)
(gdb) s
The target architecture is set automatically (currently i386)
The target architecture is assumed to be i386:intel
76      Status = PeiServicesAllocatePages (EfiBootServicesData, EFI_SIZE_TO_PAGES (STACK_SIZE), &BaseOfStack);
=> 0x07fcc05 <HandOffToDxeCore+29>:  e8 93 0b 00 00 call  0x7fcd79d <PeiServicesAllocatePages>
0x07fcc08a <HandOffToDxeCore+34>:  89 c3        mov   %eax,%ebx
(gdb) █

```

Figure 6: Snapshot of the gdb interface with which the open source software debug package interacts

(Source: Intel Corporation, 2011)

AMIDebug Rx*

AMIDebug Rx*, shown in Figure 7, is the first of its kind: a low-cost debug tool built around the debug port feature common to today's USB 2.0 EHCI controllers. AMI Debug Rx is designed as replacement for the PCI POST Checkpoint Card, and also serves as a host-to-target interface for AMIDebug for UEFI and the Intel UDK 2010 firmware debug solutions.

“AMI Debug Rx is designed as replacement for the PCI POST Checkpoint Card, and also serves as a host-to-target interface for AMIDebug for UEFI and the Intel UDK 2010 firmware debug solutions.”



Figure 7: Snapshot of the AMI DebugRx device

(Source: American Megatrends Inc., 2011)

Key features:

- USB-based replacement for the PCI port 80h “POST Checkpoint” card
- Checkpoints can be captured and stored to one of four “sessions” for later review
- Measures elapsed time between checkpoints to analyze boot performance timing
- Display descriptive text for each checkpoint, based on built-in string table or custom table
- USB Virtual COM (VCOM) port for data transfer and configuration
- UEFI debug messages redirected over USB VCOM or saved in device memory

Diagnosing small form factor platforms with AMIDebug Rx is nonintrusive, allowing technicians to see checkpoints without opening the case. AMIDebug Rx replaces the POST checkpoint card’s LED display with an easy-to-read LCD screen. This debug method produces more descriptive debugging messages than the checkpoint card, along with extended features such as boot speed timing and UEFI debug message redirection.

AMIDebug for UEFI

AMIDebug for UEFI is a powerful debug solution for Aptio, AMI’s product solution for UEFI firmware. AMIDebug for UEFI uses the UEFI debug protocols to provide an alternative to ITP/JTAG debugging for IA-32 and x64 firmware. Developers have access to source-level debugging and control the debug target hardware through the Visual eBIOS (VeB) development interface, shown in Figure 8.

AMIDebug for UEFI provides functionality similar to hardware-based development tools:

- Source-level symbolic debugging
- Access to hardware resources (CPU registers, PCI configuration space, memory, and I/O locations)
- Debug Aptio firmware, UEFI/DXE drivers, PEIMs, and pre-boot applications in the UEFI Shell

AMIDebug for UEFI and the UDK 2010 debugger have many common features, but AMIDebug is an example of a commercial software debugger with additional features.

“AMIDebug for UEFI uses the UEFI debug protocols to provide an alternative to ITP/JTAG debugging for IA-32 and x64 firmware.”

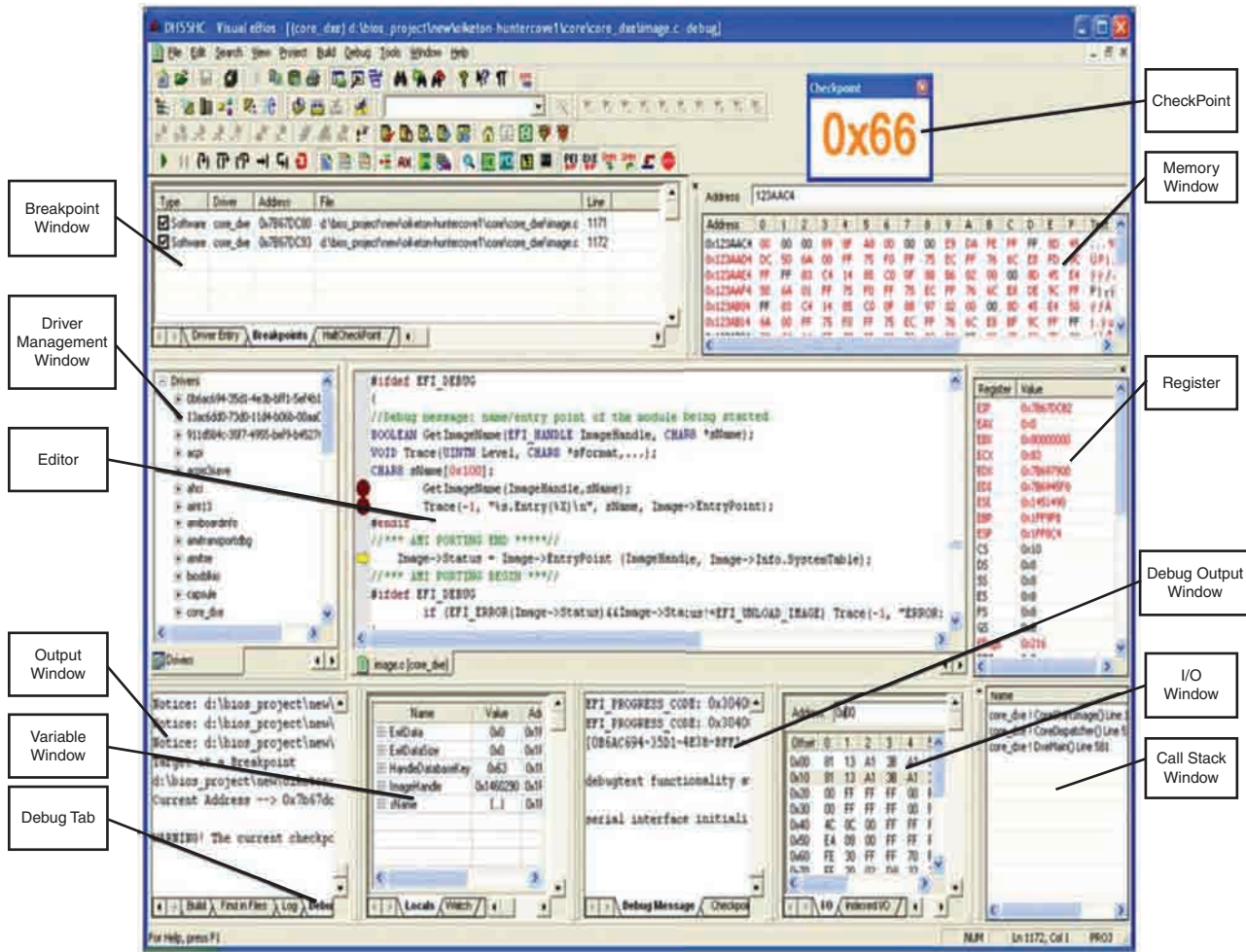


Figure 8: Snapshot of the AMI Debug interface.
 (Source: American Megatrends Inc., 2011)

The AMIDebug interface is integrated with Visual eBIOS (VeB), a development environment specifically built for developing BIOS and UEFI firmware. This allows debugging and development in the same interface, which speeds up issue resolution. AMIDebug extensions for VeB also incorporate extended debug information, such as checkpoints or debug messages, which normally appear in secondary redirection consoles. AMIDebug for UEFI also adds specialized firmware debug interfaces and can be extended to debug System Management Mode (SMM) routines on IA32 and x64 processors.

Firmware Debugging in UDK 2010

This section covers the various aspects of debugging code within the UDK codebase along with the pertinent architectural phases within which such debugging might take place.

General Architecture

Debugging UDK-based firmware requires two machines: a target and a host. The target contains the UDK firmware to be debugged and the host executes the debug interface software (Figure 9).

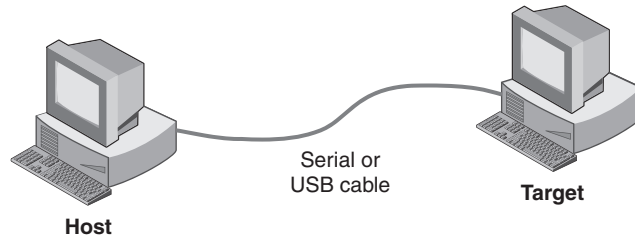


Figure 9: Cable connection between the target and host machines

(Source: Intel Corporation, 2011)

The architecture of WinDbg debugging support is depicted in Figure 10.

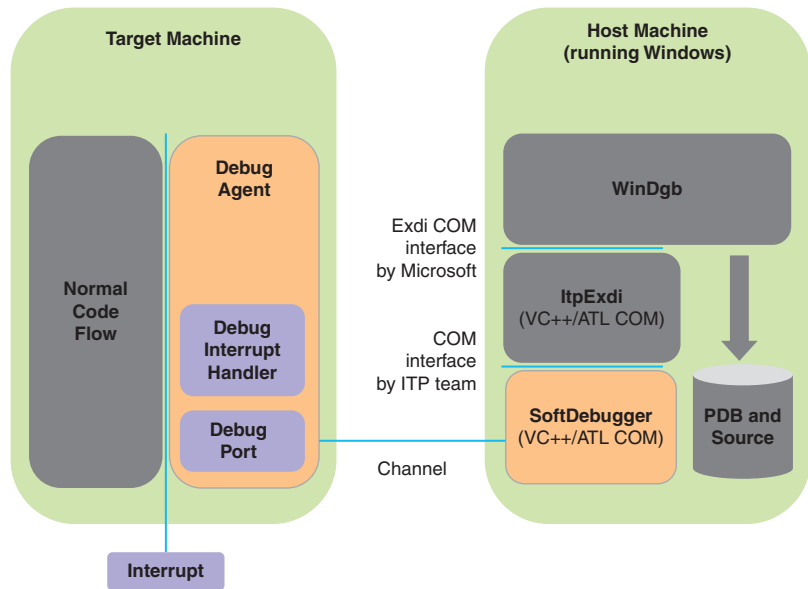


Figure 10: General architecture (Windows tool chain)

(Source: Intel Corporation, 2011)

The host's responsibility:

- Communicate with target for debug
- Implement SoftDebugger APIs
- Communicate with debug front end (WinDbg, Gdb, and so on)
- Read a configuration file to accept user configuration

The target's responsibility:

- Respond to host's request to set hardware breakpoint in debug registers in processor
- Handle interrupt (INT1, INT3, and so on) and give response to host, and wait for next step from host

Figure 11 shows the architecture of using GDB to debug UDK-based firmware compiled with GCC in Linux.

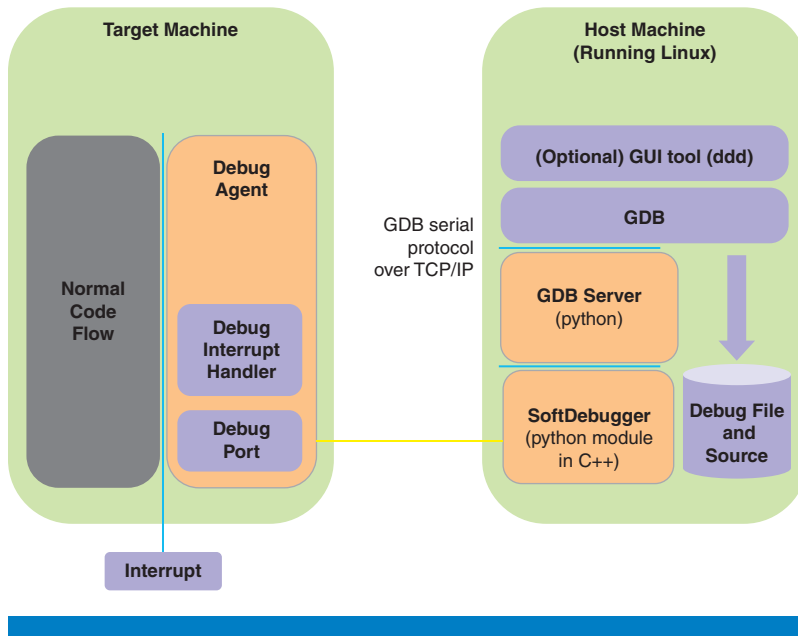


Figure 11: General architecture (Linux/GCC tool chain)

(Source: Intel Corporation, 2011)

WinDbg Debug

A UEFI firmware developer can use the Microsoft Windows Debug Tool (WinDbg) and the Intel UEFI Development Kit Debugger Tool (Intel UDK Debugger Tool) to debug UDK-based firmware on UEFI IA-32 and UEFI x64 platforms. The target side component (debug agent) is “SourceLevelDebugPkg” in UDK 2010.

The host is a Microsoft Windows XP (SP3) platform executing WinDbg and the Intel UDK Debugger Tool. The target and host interconnect via a serial null modem or USB 2.0 debug cable (as shown in Figure 9).

The Intel UDK Debugger Tool supports the following:

- Source-level debugging using Microsoft Windows Debug Tool (WinDbg) – host running Microsoft Windows XP (SP3)
- Debugging as early as late SEC (after temporary RAM setup) for the normal boot path
- Starting debugging SMM code by requesting target to stop at next SMI

“The basic debugging flow includes three major steps: compiling, programming, and launching.”

- Cable interconnect: serial null modem cable or USB host to host cable (USB 2.0 debug device cable)
- Setting unresolved breakpoints

The basic debugging flow includes three major steps: compiling, programming, and launching.

1. Compile the firmware that includes the target side debug agent, as depicted in Figure 12.

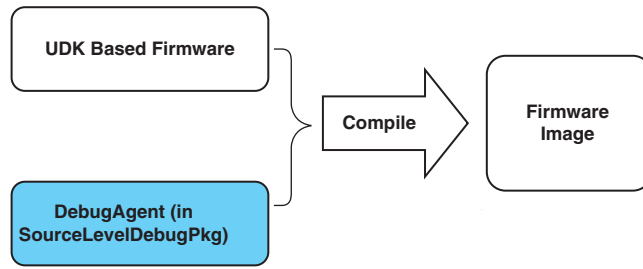


Figure 12: Compiling firmware image with debug agent (Source: Intel Corporation, 2011)

2. Program the firmware image into flash on the target system.
3. Launch a debugger on the host to debug the firmware on the target system.

Figure 13 shows how the components interact during a debug session in Windbg debugging.

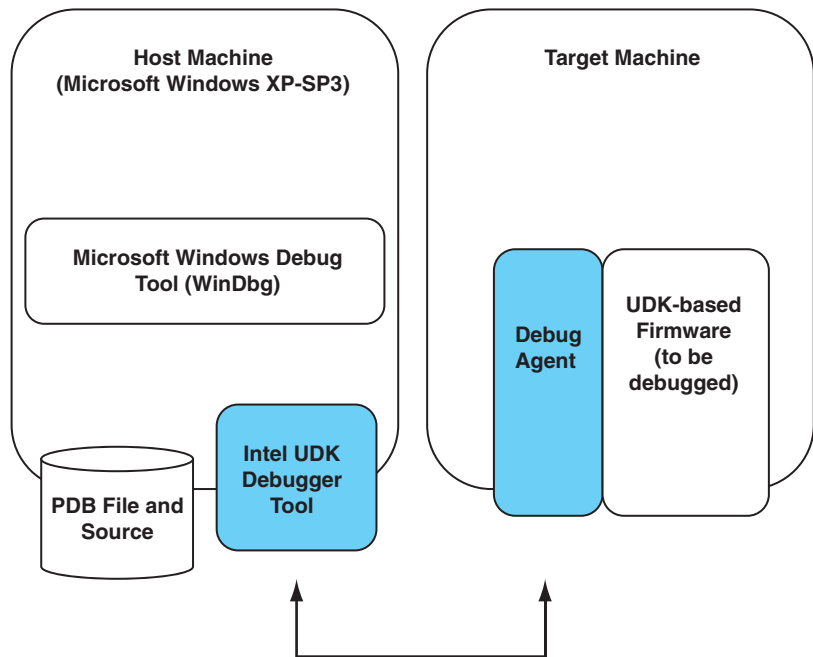


Figure 13: Debug session active components (Source: Intel Corporation, 2011)

Debugging SEC and PEI Code

Most of the code for SEC and PEI phase executes from read-only memory. The Intel UDK Debugging Tool automatically uses a hardware breakpoint if it detects the address is within the read-only memory flash range.

Debugging DXE Code

Some PI firmware implementations execute SEC/PEI in 32-bit mode and DXE/SMM in 64-bit mode. When the UEFI Debugger Tool detects a mode switch from either 32-bit mode to 64-bit mode (or from 64-bit mode to 32-bit mode), the Microsoft Windows Debug Tool (WinDbg) is automatically re-launched.

Debugging SMM Code

The Smmentrybreak command must be used to set a flag so the next entry into SMM will force the target to break into the debugger. The smmentrybreak switch must be set to inform the target whether it should stop the next time it enters SMM mode. Breakpoints can be set after the target enters SMM mode and debugging can continue. When the target stops at the SMM entry, the source for SMM handlers may be opened, and software breakpoints may be set.

“The Intel UDK Debugging Tool automatically uses a hardware breakpoint if it detects the address is within the read-only memory flash range.”

Comparing Hardware and Software Debug Capabilities

Debuggers are critical tools for the development of software. Intel offers hardware debug solutions based on the ITP/JTAG model. Other solutions exist in the marketplace for hardware-level debugging, such as the WindRiver* ICE 2 JTAG Debugger. Software debugger solutions also exist, such as the Intel UEFI Development Kit Debugger Tool or the AMIDebug for UEFI solution that integrates into the Aptio Visual eBIOS (VeB) development environment.

Table 2 illustrates differences between hardware- and software-based firmware debug solutions.

	Hardware Debugger	Software Debugger
Dependency	Requires platform with JTAG (Joint Test Action Group) or TAP (Test Access Port) interface	Requires support for a low-level debug port (serial, USB debug port, 1394)
Special Features	Hardware event. (Like init, reset, smmentry/smmexit) SMM/UEFI runtime service debug in OS environment. SEC phase debug. Debug mode switch (like 16-bit legacy code, 32/64 switch code) Debug software debugger agent. Some special features, like firmware update.	Software logic. Condition breakpoint (break if condition is satisfied). Unresolved breakpoint (Set breakpoint on an unloaded module). View data structures. EBC (EFI byte code) debugger. Debug UEFI driver and option ROM (OPROM). View UEFI debug strings, BIOS checkpoints and UEFI status codes.
Usage Scenarios	Debug silicon before production for hardware or firmware issues.	Debug production board (without JTAG) for firmware issues (BIOS/UEFI debugging, driver/OpROM compatibility)
Cost	More costly than software-based solutions	Some tools are open source and free.

Table 2: Differences between hardware and software debuggers
(Source: American Megatrends, Inc. 2011)

“For early silicon enabling, the hardware debugger is good for debugging hardware design issues.”

For early silicon enabling, the hardware debugger is good for debugging hardware design issues. The direct interface to the processor eliminates communication issues with other platform hardware. The ability to trap hardware events (SMI, INIT, mode switch) is useful in early hardware debugging.

After the initial board power-on, a software debugger can be used to debug complex firmware issues. Full support for hardware and software breakpoints gives developers a great deal of flexibility without using an ITP/JTAG solution. The reduced cost of a software solution enables a larger number of developers access to power debugging tools.

One area of concern is debugging UEFI drivers based on EFI Byte Code (EBC). EBC is executed in a processor-independent virtual machine that interprets a predefined instruction set, similar to Java. EBC code interpretation on the target is hard to debug using a hardware debugger solution. This is an issue that may be addressed in future software-based debug solutions.

Summary

The article has described debugging solutions during UEFI firmware development. We can choose any of debugging solutions based on the specific platform hardware design and the firmware problems we meet. Checkpoint solution can be used with PCI POST Checkpoint Card. A serial port can print out debug messages. A USB debug port can be used as well when a serial port and PCI POST Checkpoint card cannot be used. These solutions can quickly let developers know where firmware execution is. If problems are encountered with silicon problem before production, JTAG software with a JTAG-connector is the most powerful debugging method to do source-level debugging. If the platform has no JTAG-style connectors, AMIDebug and the Intel UDK Debugger Tool can be employed to do source-level debugging as well. These debugging solutions make it possible to develop and debug firmware as general software.

“If problems are encountered with silicon problem before production, JTAG software with a JTAG-connector is the most powerful debugging method to do source-level debugging.”

Authors' Biographies

Stefano Righi is Vice President of Software Utilities at American Megatrends, Inc. (AMI). He has over 25 years of experience in research and development, technologies, product and resource management, with special focus on BIOS and system architecture. During his tenure of more than 12 years at AMI he has been responsible for UEFI BIOS architecture, Firmware Development Environment, Graphical User Interfaces and Pre-Boot solutions including AMIDiag. Stefano is representing AMI in the UEFI Board of Directors.

Brian Richardson started with American Megatrends, Inc. (AMI) in 1996. After spending years in BIOS and test development, Brian moved to technical

marketing to support BIOS and UEFI customers. Brian has been a frequent speaker at Intel Developer Forum (IDF) and produced video content for AMI's YouTube Channel (AMITV).

Jiewen Yao is an EFI/Tiano BIOS engineer in the Software and Solutions Group at Intel and has 8 years' experience on firmware design, development, and testing. He received an MS degree from Shanghai Jiaotong University and a BS degree from Fudan University. Jiewen holds approximately 10 patents in the firmware area.

Elvin (Wei), Li is a BIOS Engineer in the Software and Solutions Group at Intel and has more than 5 years of UEFI BIOS development experience. He received his MS degree in 2006 and BS degree in 2003 from Beijing Jiaotong University, both in Electrical Engineering. His interests lie predominantly in the areas of embedded system design and system software. His e-mail is elvin.li at intel.com.

BROAD USE OF UEFI IN HEWLETT PACKARD SYSTEMS

Contributors

Dong Wei
Hewlett Packard

Kimon Berlin
Hewlett Packard

Eugene Cohen
Hewlett Packard

“EFI also defined a modular, flexible and extensible architecture, enabling the use of high-level programming language.”

“In 2005, the industry came together and decided to form the UEFI Forum to own the interface definitions that EFI and Framework covered.”

The reader of this article will understand the value that Hewlett Packard has gained from leveraging UEFI across their product portfolios. By providing a variety of case studies, HP shows how UEFI solutions have been used within HP’s products ranging from printers to supercomputers.

Introduction

Before UEFI, there was EFI (Extensible Firmware Interface). EFI was created initially in 1998 for systems based on the Intel® Itanium® processor to overcome BIOS dependencies that presented formidable challenges to the “big iron” system design. For example, the dependency on the legacy 8259 interrupt controller, the 64K I/O port space, the 192K option ROM execution space, the single PCI segment group, all impacted the scalability of the system. We also foresaw the coming of the 2.2-TB Master Boot Record (MBR) hard drive partition limitation and defined a new GUID Partition Table (GPT) format.

While addressing all these limitations, EFI also defined a modular, flexible and extensible architecture, enabling the use of high-level programming language. It was created with the processor architecture agnostics in mind, supporting Itanium, x86, and a processor-independent byte code, EFI Byte Code (EBC).

EFI was an Intel-owned specification defining the interfaces between the operating systems and the system firmware, as well as the device boot driver and the system firmware. Intel created the Framework defining the system firmware internal interfaces to further make the EFI implementation modular.

In 2005, the industry came together and decided to form the UEFI Forum to own the interface definitions that EFI and Framework covered. Intel contributed the EFI and Framework specifications to the UEFI Forum as the starting point. The change of name from EFI to UEFI (U stands for Unified) signified that the tasks of definition, promotion, and adoption were on the shoulders of the industry from then on.

The first specification, the UEFI 2.0 Specification, was published by the UEFI Forum and defined the binding for x64 processors with help from AMD and Intel. The Framework also evolved into the Platform Initialization (PI) Specification. ARM-binding for UEFI was published in 2009 as part of the UEFI 2.3 Specification. Most recently, the UEFI 2.3.1 Specification finalized the support for the Secure Boot capability and the IPv6 pre-boot support.

HP's History with EFI

In the mid-1990s, while HP partnered with Intel to develop the Itanium® architecture for the enterprise server systems, the system firmware architecture was initially Itanium-specific as a combination of processor abstraction layer (PAL) and system abstraction layer (SAL). The PAL/SAL concept inherited much from the firmware architecture used in the PA-RISC systems called processor- and I/O-dependent code (PDC-IODC).

Due to the desire to leverage the x86 ecosystems to support the Windows* operating system, the boot services in SAL were directly wrapped around the x86 BIOS services like INT 19h, INT 13h, and INT10h. However the BIOS limitations presented formidable challenges to the “big iron” system design. For example, the dependency of BIOS on legacy PC-AT hardware such as the 8259 interrupt controller and the 8254 timer created problems like single-point of failure, inflexible configurations, and resource duplication for multi-partition systems. The shortage of the I/O port space, option ROM execution space, and PCI bus numbers also made boot device configuration support severely constrained for enterprise servers that may have hundreds of I/O devices that are potentially bootable devices.

HP introduced the concept of an Enhanced Mode (EM, another name for the Itanium architecture at the time) option ROM, but it did not get finalized due to the Windows support requirement. Fortunately, when Microsoft considered the Windows support on Itanium in the late 1990s, they also saw the need to create a new boot model to replace BIOS.

There were some existing alternatives, but they all had some major technical, business, or legal issues. Intel put out a proposal called Intel Boot Initiative (IBI), which later became EFI. EFI 1.02 was a good start, but still did not address the option ROM limitation issue. It was not until EFI 1.10 when the boot model for Itanium was complete, in time for the first Itanium launch. Since then, EFI and later on UEFI, has been the only boot model supported on all Itanium systems. HP Integrity Systems have been leading in the EFI/UEFI adoptions and supporting value-add features such as boot-from-tape, virtual partitioning, device parameter configuration, system management, and RAS capabilities.

The two additional efforts during the evolution of EFI have made significant impact to HP's system designs. The first is the processor-architecture-agnostic design of the interfaces. This originated from the desire to have one binary image for the Option ROM on the add-in card to support booting on both x86 and Itanium systems. Therefore, EFI was created to have three processor bindings: x86, Itanium, and an EFI Byte Code (EBC). EFI drivers compiled into EBC can support boot on both x86 and Itanium systems as long as they are EFI-compliant. UEFI now also supports x64 and ARM bindings. Another effort is the push to modularize the implementation of UEFI by Intel via its Framework development, which led to the Platform Initialization (PI) specifications. These efforts have enabled HP to modularize

“However the BIOS limitations presented formidable challenges to the “big iron” system design.”

“Therefore, EFI was created to have three processor bindings: x86, Itanium, and an EFI Byte Code (EBC).”

the implementation to share code across multiple business units and across products based on different processors.

Starting from 2008, HP has shipped UEFI-based systems using Itanium, x64, and ARM processors from servers to clients, from printers to networking gears, as shown in Figure 1. UEFI is becoming the converged firmware infrastructure.

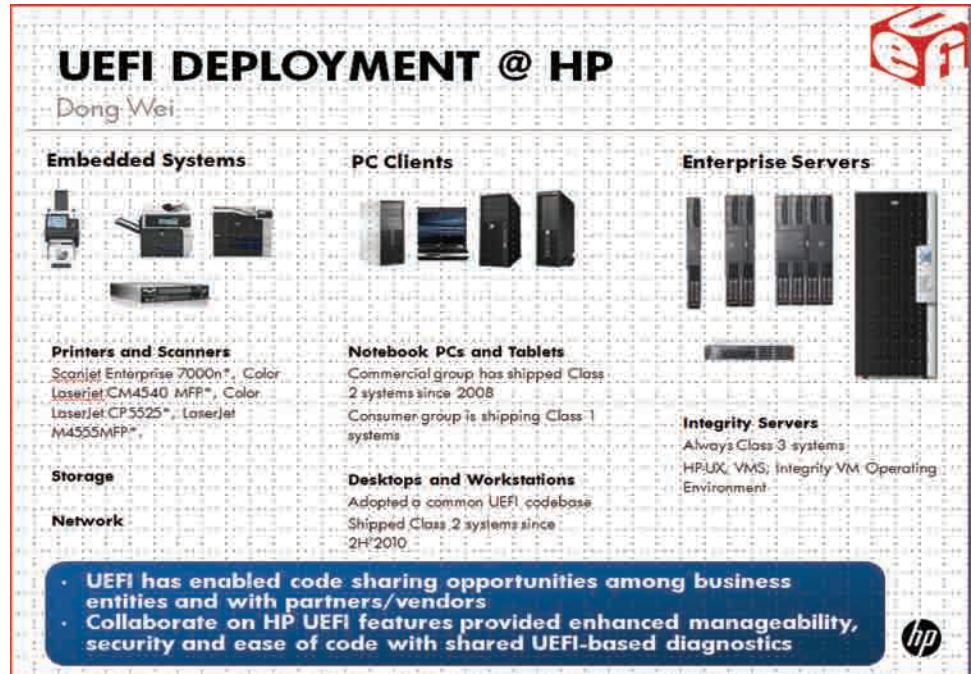


Figure 1: UEFI deployment at HP
(Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

“Accompanying the UEFI and PI specifications, the UEFI Forum is responsible for providing the self-certifying compliance tests (SCT).”

“Intel sponsors an open source component development at tianocore.org.”

Accompanying the UEFI and PI specifications, the UEFI Forum is responsible for providing the self-certifying compliance tests (SCT). On the UEFI front, the UEFI Forum is responsible for the development and delivery of the SCT. For PI SCT, there is an open source effort at <http://sourceforge.net/projects/pi-sct/>. The UEFI Forum’s task is to endorse the PI SCT when it is ready. Currently, the UEFI Forum has made available the UEFI 2.3 SCT and has endorsed the PI 1.0 SCT.

Intel sponsors an open source component development at tianocore.org. Two major projects are EDK and EDK II. The EDK is the open-source component of the Framework, Intel’s implementation of the EFI Specification, which was developed under the project code-named Tianocore. The EDK was released under the BSD License. The EDK II project is the response to the EFI community’s request for a better build and version tracking environment for UEFI and PI development. The main difference between the EDK II to the original EDK is the Enhanced Build Environment of the EDK II. EDK is no longer kept up. Newer features and components are provided via EDK II. In 2010, Intel

released UEFI Developer Kit (UDK) 2010 based on EDK II as a snapshot of the components implementing to the UEFI 2.3 and PI 1.2 specifications that were available at the beginning of January, 2010.

When HP Integrity* enterprise servers first launched in 2002 with Itanium 2 processors, there were two codebases. Both had Intel's Sample EFI Implementation layered on top of proprietary implementations underneath. One was optimized for the entry-level servers and the other was optimized for the mid-range and Superdome* servers. When Intel first created EDK, HP was not quite sure if such a framework was actually scalable and flexible enough to support the HP systems that are quite different from Intel development vehicles. HP Integrity Systems have powered some of the world's most demanding mission-critical environments for enterprise customers who require high availability, rich virtualization capabilities, and manageability.

At that time, we were working on systems using the HP zx2 chipset on the entry-level and the sx2000 chipset on the mid-range and Superdome servers. We collected and reexamined our requirements, which can be summarized as follows:

- Ability to support advanced features
- Path to support network boot over IPv6, and so on
- Ability to provide HP platform innovations
- Ability to deliver platform value-add module
- Ability to protect intellectual property
- Improve execution excellence
- Deliver with limited engineering resources
- Deliver faster time to market
- Separate the hardware basic execution away from HP innovations
- Reduce integration and validation time
- Use packaging supplied by silicon driver modules from silicon supplier
- Maximize proper code reuse
- Build-once, use by multiple platforms

To evaluate whether EDK met these requirements, we prototyped EDK on top of systems based on the HP zx1 chipset. We were able to focus on the attributes of EDK since zx1-based systems were of shipping quality then. The results were promising. However, EDK II was emerging; it was the response to the UEFI community's request for a better build and version tracking environment for UEFI and PI development. EDK II also has superior package support.

After careful preparation, HP was ready to intercept the product deployment with EDK. We decided to skip EDK and transition directly to EDK II on all new HP Integrity Systems based on the Intel Itanium 9300 processors. A single source tree is used to support Blades and Rack Servers using the Intel 7500 chipset (as shown in Figure 2), and Superdome 2 using the HP sx3000 enterprise systems chipset (as shown in Figures 3–5).

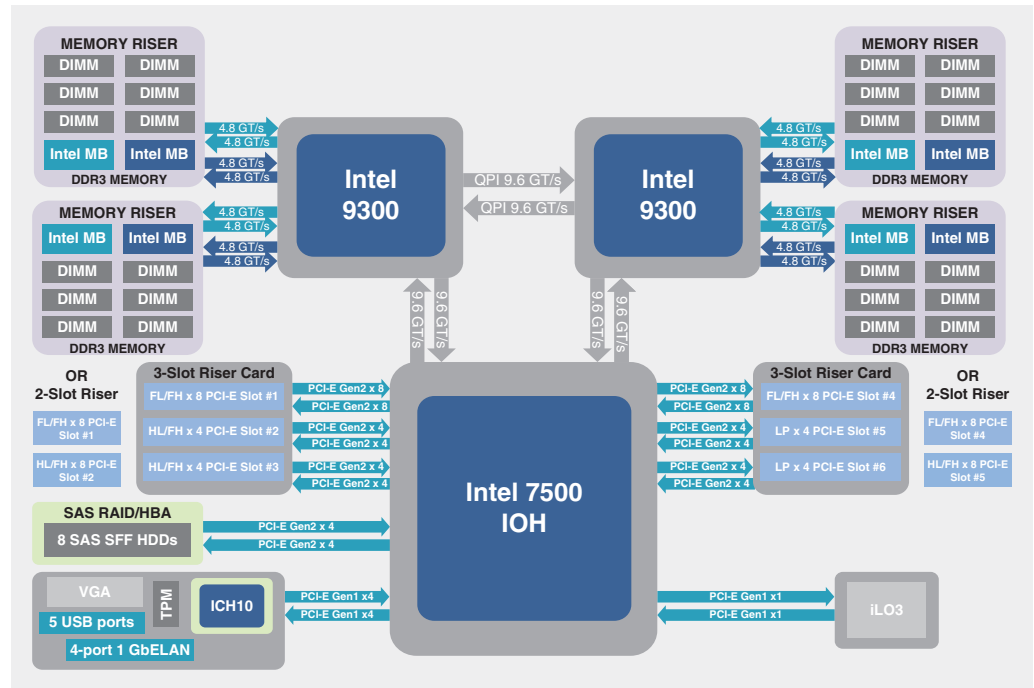


Figure 2: Blades and Rack Servers using the Intel 7500 chipset
(Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

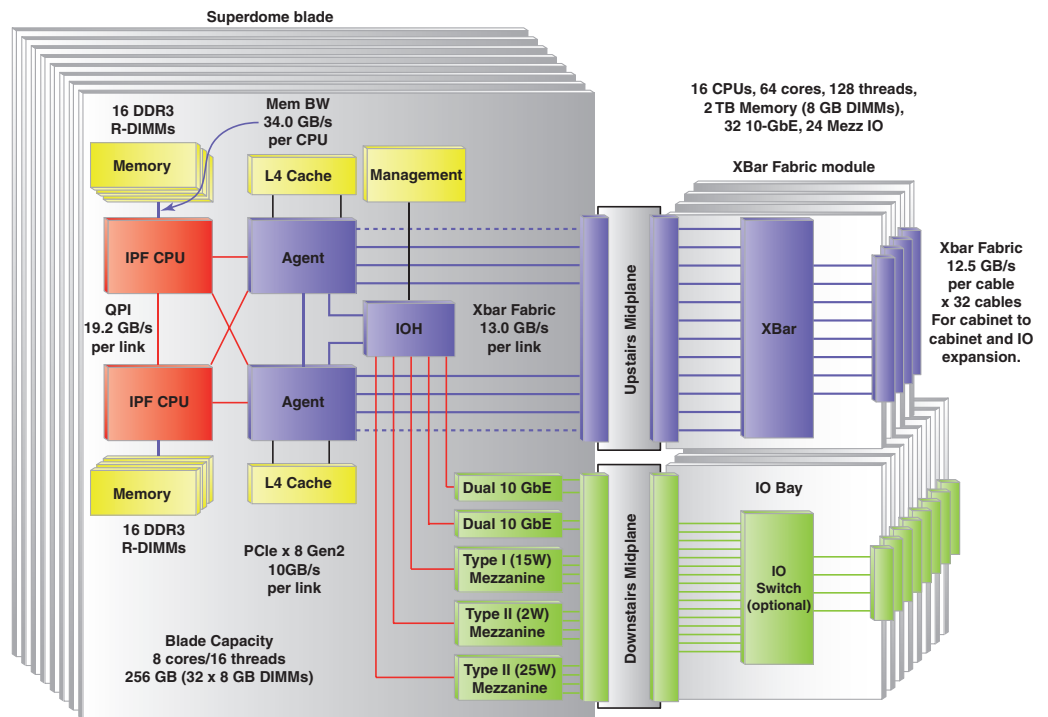


Figure 3: Superdome 2
(Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

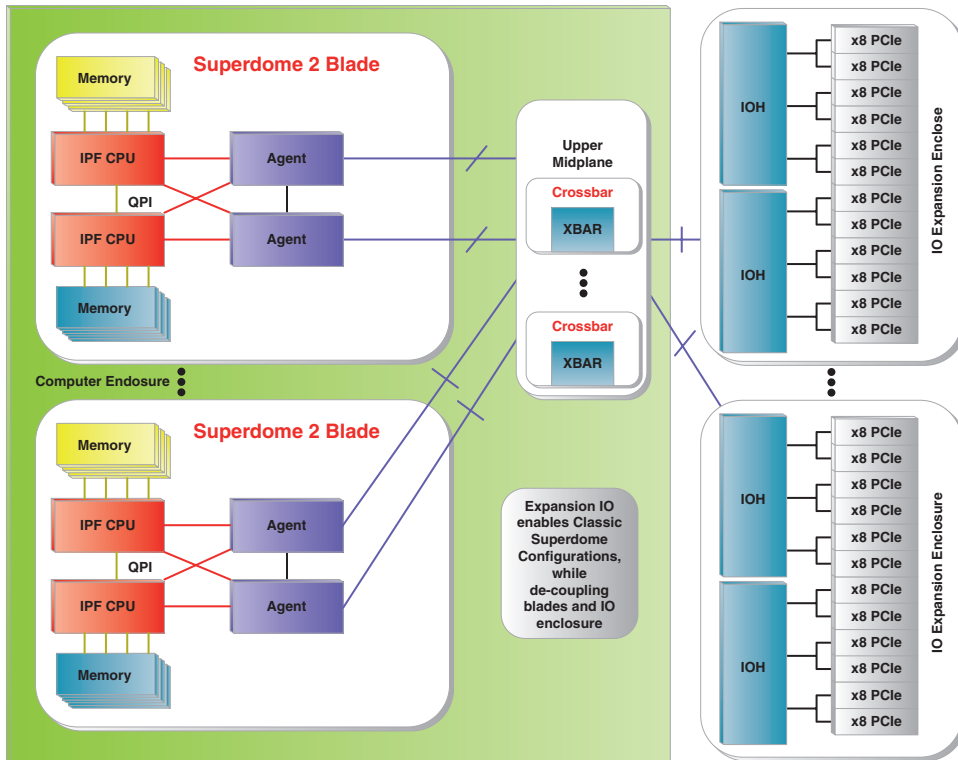


Figure 4: Superdome 2 IO Expansion
 (Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

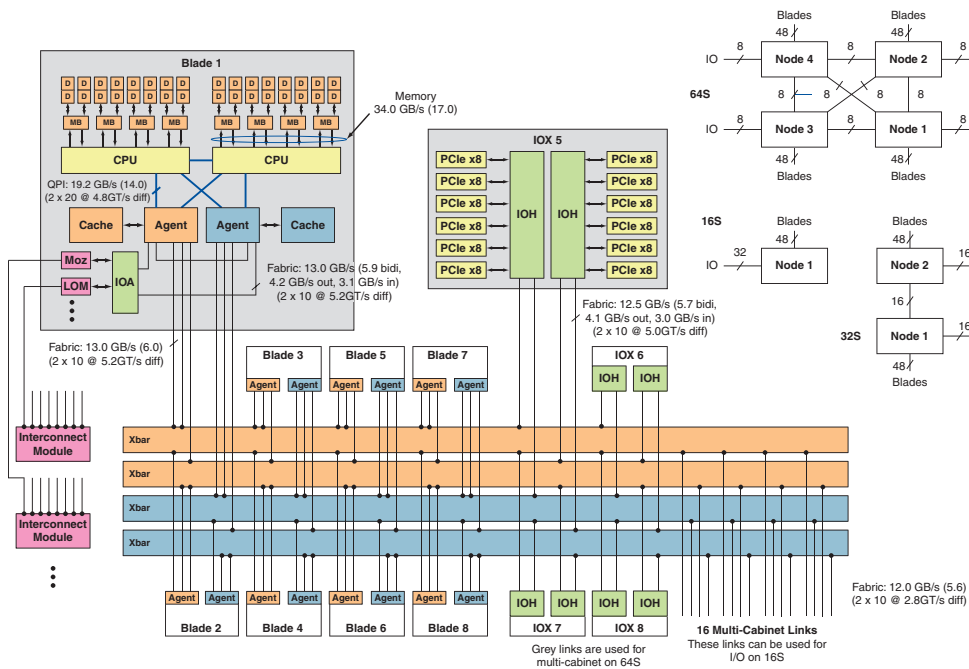


Figure 5: Superdome 2 Interconnect
 (Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

EDK II's superior package support enables better ability to reuse, global visibility for bug fixes, and allows for single module/solution ownership. Some silicon modules and applications were developed or leveraged from the EDK environment. Intel designed EDK Compatibility Package (ECP) made it possible to reuse these modules and applications as-is without having to port into the EDK II construct. This made the EDK II transition smooth.

HP, as an early adopter for EDK II, provided reviews and guidance that helped refine EDK II to the present form. We provided multiple rounds of feedback on simplifications. We recommended the use of industry-standard tools instead of proprietary tools. We provided fixes of build tool bugs and identified EDK II issues that arose when enabling compiler optimization with the Intel C compiler. We also discovered multiple EDK II bugs, for example, a subtle design issue with the UEFI network stack that led to severe performance degradation on large systems. HP contributions benefited the entire open-source community.

“The challenge faced by HP as an early adopter for EDK II is that the code development required large-scale source tree updates.”

The challenge faced by HP as an early adopter for EDK II is that the code development required large-scale source tree updates. Updates were needed on average every two to three months, but that was expected in the early adoption phase. The UDK2010 release based on the EDK II codebase addressed this challenge through codebase maturity, packaging technology, and catching up with the latest specifications.

HP has the following recommendations for the developers:

- Pay close attention to the specifications/errata
- Employ parallel versions for different specification versions
- Maintain the infrastructure support and compatibility: keep a “deprecated” version of lib/include/PCD and avoid changing build tools/lib/include/PCD
- Proactively communicate when a bug is fixed

HP also has the following recommendations for the OEMs/IBVs:

- Take advantage of parallel versions if available
- Get small-scale source updates needed
- Pull in the latest code at least every two months
- Use the EDK II package solution
- Create vendor-specific modules

UEFI on Client Systems

When EFI was created, HP did various experiments of porting it to the x86 environment. But actual product integration plans started after the formation of the UEFI Forum and the publication of the UEFI 2.0 Specification, where the x64 binding was first published. By 2008, commercial notebooks started

to ship Class 2 UEFI systems (supporting both legacy and UEFI operating systems), while consumer notebooks started to ship Class 1 UEFI systems (supporting legacy operating systems only). Class 3 UEFI systems only boot UEFI operating systems and do not contain a CSM [1].

In 2011, desktops and workstations started shipping Class 2 UEFI systems based on the Intel “Sandy Bridge” microarchitecture (see Figure 6). These systems adopted a common UEFI codebase descended from the EDK and ported existing features from their legacy HP BIOS to this new environment. The runtime portion of the HP BIOS was reimplemented as a CSM. The shell, network stack, and network tools are delivered as Web downloads, and they reuse fixes and improvements previously developed on Itanium servers and printers.

“The shell, network stack, and network tools are delivered as Web downloads, and they reuse fixes and improvements previously developed on Itanium servers and printers.”

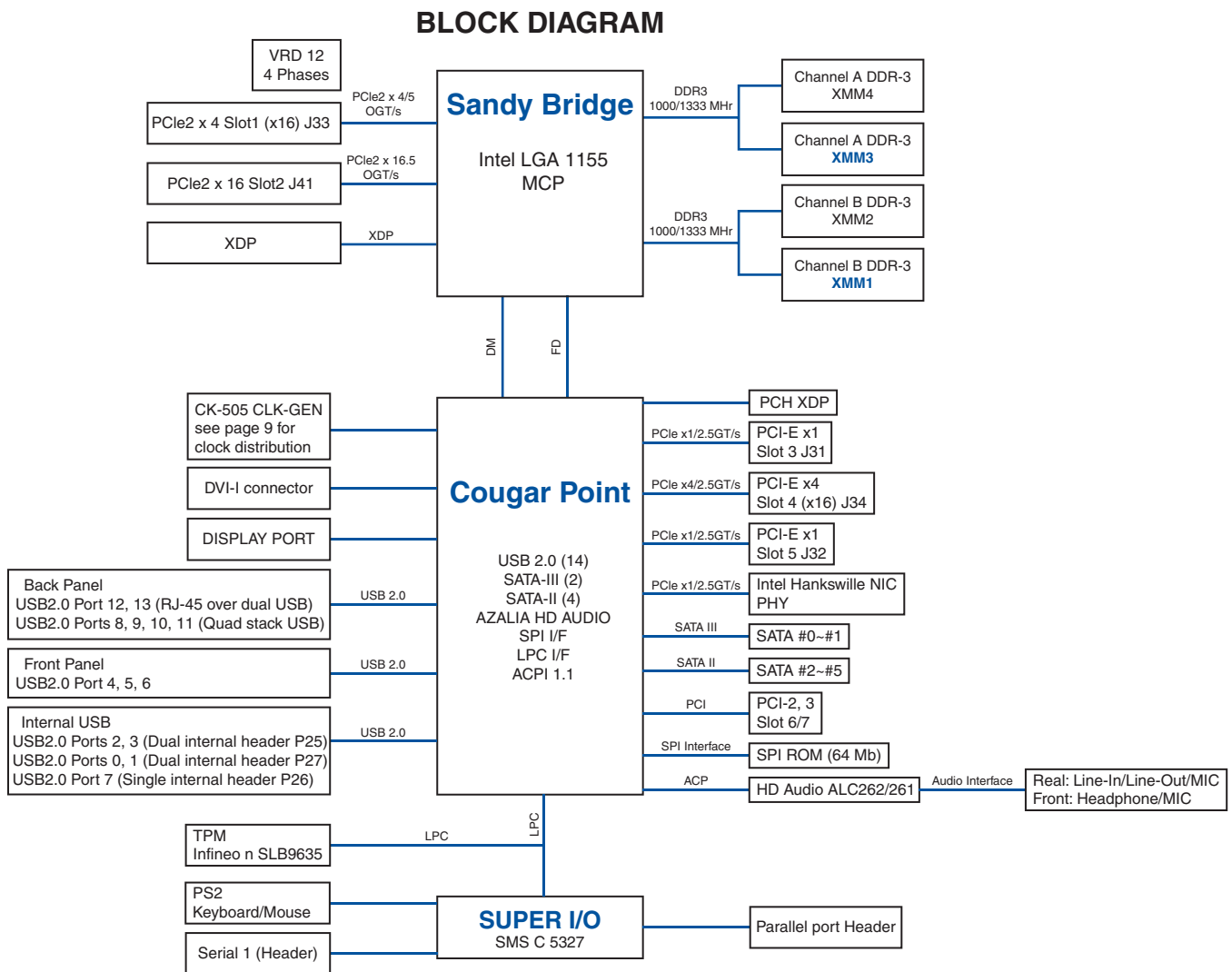


Figure 6: Block diagram for the HP Z210 Workstation
(Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

“HP’s UEFI diagnostics for client systems established a common diagnostics tool for these systems that converged to the UEFI technology.”

HP’s UEFI diagnostics for client systems established a common diagnostics tool for these systems that converged to the UEFI technology. It has enhanced diagnostics, provided DIMM fault isolation, and enabled concurrent memory, hard drive, and battery tests. It has improved the test coverage and optimized test cycles.

UEFI technology makes it easier to provide value-add to our systems. System boot time has been a challenge that needs collaborative work among ecosystem players such as OEMs, OSVs, and IHVs. The industry is collectively working to address this painful user experience for future products. However, the modularity of UEFI also makes it easier for HP to innovate. HP DayStarter is a simple value-add to the system allowing users to have access to productivity information while waiting for the system to boot. Although ultimately this feature may not be needed when the industry collectively reduce the boot time, it is a small investment that resulted in differentiated better user experience in the meantime while the overall boot time is still pretty long.

Figure 7 shows a typical boot sequence to Windows, and how the new HP DayStarter kicks in to improve the user experience.

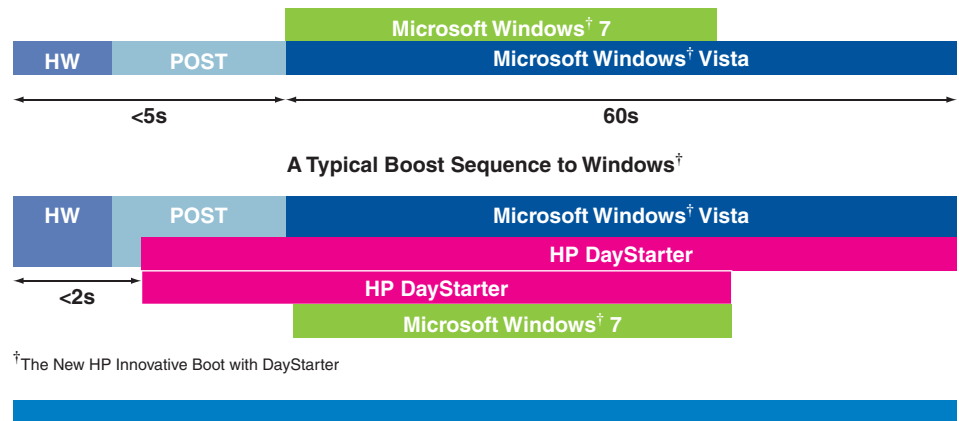


Figure 7: HP Innovative Boot with DayStarter
 (Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

Figure 8 is a screenshot of what is present to the users while Windows boots in the background.

The benefits to the customers are the instant-on user experience with user productivity information (such as calendar, to-do list and customizable information) available for display before and while Windows is booting. The main technology behind it is for the UEFI BIOS to locate the proper JPEG images and use the System Management Mode (SMM) to update the frame buffer content until Windows is ready for system login. At OS runtime, HP implements an Outlook plug-in to capture the calendar information.

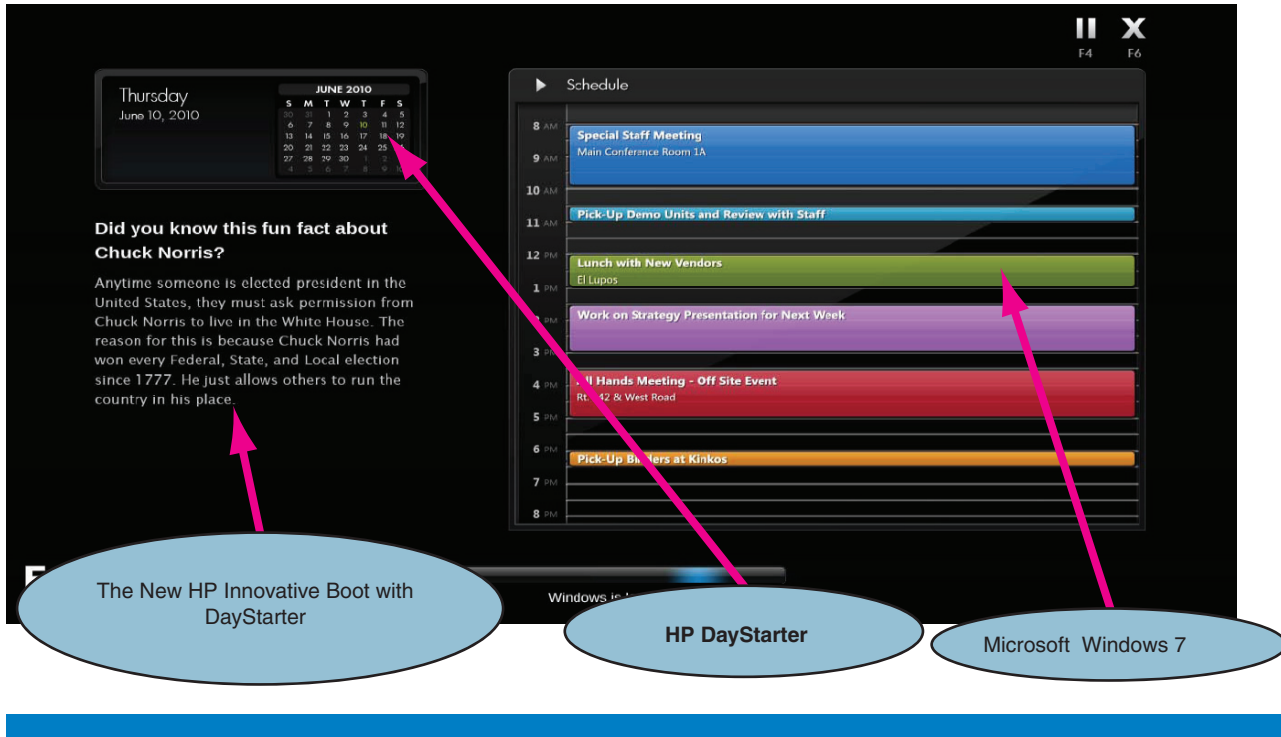


Figure 8: HP DayStarter Screenshot
 (Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

Details on ARM and UEFI/PI

ARM processors have been predominant in the smartphone market and are becoming increasingly mainstream in the overall embedded space. However, historically ARM systems did not have a preboot firmware standard. This led each design to have its own distinct firmware model that was tightly coupled to the operating system being booted. This traditional approach means the firmware developers would need to maintain completely different codebases even though the systems may use many of the same types of peripheral devices (network, SATA, USB controllers, and so on) and feature sets across the designs. Generations of ARM cores relied on boot packages such as UBoot, [2], Redboot, or proprietary software.

How to efficiently develop and ship these products and meet time-to-market demands becomes a challenge. Some form of converged firmware infrastructure is necessary to maximize proper code reuse, enable the products to achieve faster time to market with limited engineering resources, and concurrently add innovative features.

UEFI is a new opportunity for preboot firmware on ARM-based systems. The UEFI Forum defined the ARM-binding for UEFI. With UEFI, it is now possible to maximize the code reuse among different designs, including those using different processor architectures.

“Generations of ARM cores relied on boot packages such as UBoot, Redboot, or proprietary software.”

With the publication of the ARM-binding, Apple and HP contributed UEFI reference implementations, including one for the Beagle Board (beagleboard.org), to tianocore.org, enabling silicon vendors to supply UEFI drivers for their hardware. ARM Ltd. recently contributed reference code for the Versatile Express reference platform with Cortex A9 MP cores, in addition to:

- Build environment fixes for the ARM GCC toolchains
- Updates for the ARM RealView Emulation Baseboard code
- A new ArmPlatformPkg containing common components for ARM reference platforms
- TrustZone controller support
- MP Core support
- PL18x MMC Controller Support
- A customized Boot Device Select (BDS) library that supports booting Linux directly

“There are many advantages for ARM preboot firmware to standardize on UEFI.”

There are many advantages for ARM preboot firmware to standardize on UEFI. OEM/ODMs are always looking into reduced development cost. Code sharing among products is one way to achieve that. UEFI not only enables code sharing among ARM products or among x86 products, it also enables code sharing across processor architectures. Products may share many of the peripheral devices (network, SATA, USB controllers, and so on) and feature sets across the designs.

Figure 9 shows an ARM port where 99.42 percent of the lines of code need not change from an x86 port.

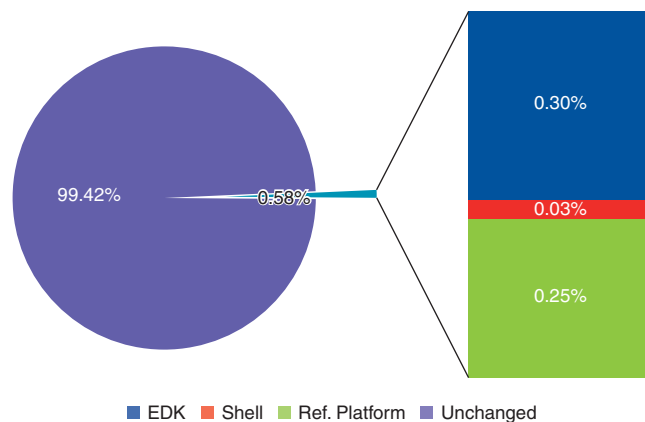


Figure 9: Lines added/changed for ARM port
 (Source: Copyright 2011 Hewlett-Packard Development Company, L.P.)

The modularity of the UEFI technology also enables the silicon vendors to deliver drivers for their own hardware. This gives OEM/ODMs the flexibility to use modules from different suppliers, widening the options.

UEFI for ARM provides a truly OS-independent boot solution whereas most other existing ARM boot solutions (such as UBoot and Little Kernel) are coupled to the OS they support.

Standardizing preboot firmware on UEFI also enables potential independent firmware vendors to enter the market to provide support. It would make it possible for these vendors to provide support to multiple customers, deploy modules efficiently, and provide common scalable solutions.

This standardization would also reduce the development cost for the silicon vendors, since they would only need to support one type of driver that could be integrated by all OEM/ODMs. This could improve validation efficiency and debug-ability.

For the OSVs, this standardization would allow them to focus their investment in a single bootloader. Such standardization would also give new opportunities to the ISVs for innovations.

Future Direction: Scratching the Surface Today?

With more and more systems transitioned or transitioning to UEFI, the foundation is established to realize the benefit of code reuse across product segments and across architectures. Done well, this can be a significant cost savings to the company. This converged firmware infrastructure builds on an industry-standard technology that the industry is collectively relying on to support modern features like secure boot and booting from an IPv6 network. But more importantly, as the E in the UEFI name indicates, HP is in the best position to fully take advantage of the extensibility of this technology. Now that the foundation is established, it is time for us to shift more resources toward differentiation and innovation. We have shown a few examples in this article, but opportunities are there everywhere. Our task is to recognize, capture and make it happen.

“Now that the foundation is established, it is time for us to shift more resources toward differentiation and innovation.”

References

- [1] UEFI Industry Communications Working Group (ICWG), Evaluating UEFI using Commercially Available Platforms and Solutions, http://www.uefi.org/news/uefi_industry/UEFIEvaluationPlatforms.pdf
- [2] <http://www.denx.de/wiki/U-Boot>

Authors' Biographies

Dong Wei is an executive-level Distinguished Technologist/Strategist at HP. An IEEE Senior Member with extensive experience in leading industry innovations and standardization, he is HP's chief architect for the UEFI and

ACPI technologies. He is Chief Executive at the UEFI Forum, Inc., Secretary of the ACPI SIG, and chairs the PCI SIG Firmware Working Group. He participated in defining platform interfaces between hardware, BIOS firmware, and the operating systems for x64, x86, ARM, ia64 and PA-RISC-based systems, has deep expertise in the platform architecture, embedded firmware, OS loader and hardware abstraction layer/machdep supporting system boot, power management, dynamic system configuration, system manageability, virtualization, PCI/USB I/O technologies and RAS features. He has an MBA as well as MS degrees in electrical engineering and physics. He also works on the China strategy for Business Critical Systems.

Kimon Berlin is the Master BIOS Engineer for HP Workstations. He has worked on BIOS and firmware development since 1994. He holds an “Ingénieur centralien” engineering degree from Ecole Centrale Paris, majoring in real-time computing.

Eugene Cohen is a Master Firmware Architect for HP LaserJet. He has worked on boot and platform firmware at HP since 1999 on numerous embedded systems architectures based on PowerPC, X86, MIPS, and ARM running various embedded and mainstream operating systems.