Embedded Firmware Solutions

Development Best Practices for the Internet of Things

Jiming Sun
Marc Jones
Stefan Reinauer
Vincent Zimmer

Apress open

Embedded Firmware Solutions: Development Best Practices for the Internet of Things

Jiming Sun, Marc Jones, Stefan Reinauer, and Vincent Zimmer

ISBN-13 (pbk): 978-1-4842-0071-1

ISBN-13 (electronic): 978-1-4842-0070-4

Managing Director: Welmoed Spahr
Lead Editors: Steve Weiss (Apress); Stuart Douglas (Intel)
Coordinating Editor: Melissa Maldonado
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

About ApressOpen

What Is ApressOpen?

- ApressOpen is an open access book program that publishes high-quality technical and business information.
- ApressOpen eBooks are available for global, free, noncommercial use.
- ApressOpen eBooks are available in PDF, ePub, and Mobi formats.
- The user-friendly ApressOpen free eBook license is presented on the copyright page of this book.

Contents at a Glance

Contents

About the Authors

Jiming Sun is a firmware and BIOS industry veteran who started to write RTOS kernel code (pSOS) for Bell Labs in 1986. After changing career paths from telecom to PC, he was involved in the early laptop PC evolution, and was among the first batch of firmware engineers to implement System Management Mode (SMM) code for 386SL in the early 1990s. After joining Intel in 1993, Jiming did early APM (Advanced Power Management) and ACPI (Advanced Configuration and Power Interface) implementation. Jiming is one of the creators of Tiano, which turned into UEFI, at Intel, and he is the major contributor of AMD's AGESA (AMD Generic and Encapsulated Software Architecture). Besides his experience with Intel, Bell Labs, and AMD, Jiming has also worked for Zenith Data Systems, HP (Compaq), Insyde Software, Dell, and Apple. Jiming recently grandfathered Intel Firmware Support Package (Intel FSP) and was instrumental in launching the product in October of 2010. Jiming has master's degrees in Electrical Engineering and Management of Science and Technology, and he has 19 granted and one pending US patents. He currently lives in the Bay Area of California with his wife and two sons.

Vincent Zimmer is a senior principal engineer in the Software and Services Group at Intel Corporation. With over 23 years' experience in embedded software development and design, Vincent holds more than 310 US patents and was awarded two Intel Achievement Awards for his development of firmware architecture and security. 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.

Marc Jones is an accomplished firmware developer with over 18 years' experience in x86 embedded systems development. Marc has been a vital, active member of the coreboot community since 2007. As a lead firmware developer at Sage Electronic Engineering, his most recent focus has been on Intel FSP coreboot integration, Google Chromebook development, and AMD embedded APU solutions. Marc got started with coreboot as the lead developer and coreboot project liaison at AMD. As a senior software engineer, he developed coreboot source code for the AMD Barcelona Family10 processor, AMD Geode processor, and AMD CS5536 chipset. In addition, he contributed to the development of support for the AMD RS690 and SB600 chipsets along with reference mainboards. Prior to coreboot, Marc was one of the primary architects of the AMD GeodeROM BIOS, the basis of most Geode systems. He has also developed firmware and BIOS for Cyrix and National Semiconductor. Marc has been a proponent of open source development for years and has written papers and blog posts that spotlight its merits. He has presented coreboot at the Southern California Linux Expo (SCaLE) 2013, 2012, 2010, the Free Software Foundation (FSF) Libre Planet 2009, the 2008 High Performance Computer Science Week Conference, and the 2007 Ottawa Linux Symposium. For the past five years, Marc has been the coreboot administrator and a mentor for Google Summer of Code, which provides students summer internships with open source software projects.

Stefan Reinauer is a staff engineer/manager in the Chrome OS Group at Google Inc. He has been working on open source firmware solutions ever since he started the OpenBIOS project in 1997. Stefan joined the LinuxBIOS project in 1999, and worked on the first x64 port for LinuxBIOS back in 2003. In 2005, Stefan founded coresystems GmbH, the first company to ever provide commercial support and development around the coreboot project, working on ports to new chipsets and mainboards. In 2008, Stefan took over maintainership of the LinuxBIOS project and renamed it "coreboot". He was the original implementer of the project's ACPI and SMM implementations. Since 2010, Stefan is leading the coreboot efforts at Google and contributed significantly to what is the largest coreboot deployment in the history of the project. Stefan currently lives in the San Francisco Bay Area.

About the Technical Reviewers

Xiang (Maurice) Ma is an Intel software architect on IA firmware, BIOS, and bootloader. He has more than 15 years' extensive experience in the legacy BIOS, UEFI firmware, bootloader, and embedded OS development for various Intel IA platforms, including embedded systems and workstation/servers, focusing on the core architecture, silicon reference code design and prototyping, as well as platform enabling and porting. Xiang Ma now works in the Intel IOTG group on the Intel firmware and bootloader initiatives. He is the primary architect who defined the Intel FSP design specification and prototyped the initial Intel FSP solution on Intel Haswell and Bay Trail platforms. He holds a master's degree in Control Theory and Control Engineering from Huazhong University of Science & Technology in China.

Ravi Rangarajan is a firmware architect with 15 years' experience in a range of areas, from embedded systems to server firmware development. He has a bachelor's degree in electronics and a master's degree in computer applications. He is currently working for the Intel Corporation. Ravi's interests include computer architecture, firmware, and operating systems.

Ravi was one of the original authors of the FSP Architecture Specification and was part of the team that prototyped, designed, and developed the Intel Firmware Support Package. His involvement continues in the evolution of the Intel FSP.

Edward Roache graduated from Dublin City University in 1993 with a B. Eng in Electronic Engineering. He has been working with BIOS for 15 years. He joined Stratus Technologies, Inc. in 1999 and worked on BIOS for their ftServer Fault-Tolerant servers. From 2006, he worked for Ircona providing BIOS services for companies such as Fujitsu-Siemens and NettApp. He joined Intel in 2011 as the BIOS technical lead for Quark X1000. He lives with his wife, Ann, and their son, Darragh. He's a keen GAA follower and helps coaching of the underage teams in Ratoath, Co. Meath where he lives.

Kangkang Shen is the chief architect for BIOS in Huawei. As a BIOS industry veteran, he has experienced the development of the PC industry since its early days. After joining Award software in 1993, he became one of the key developers for BIOS boot manager, PCI BIOS, and many other key BIOS features. In 1998, he joined Phoenix Technologies as engineering manager and director responsible for Phoenix and Award BIOS kernels. In 2003, he was assigned to lead the Phoenix R&D center in China. In 2006, he cofounded Nanjing Byosoft, an Intel-authorized BIOS vendor. In addition to his industry experience, he worked in Nanjing University of Technology as a professor from 2008-2011. He has a Bachelor's of Science in Optical Engineering from Zhejiang University, China and a Ph. D. from Georgia Institute of Technology, Atlanta.

Acknowledgments

We thank our families first for their support of us in writing this book. Since all of us have a demanding job to focus on during the day, we frequently spent our precious family time, evenings, weekends, and holidays, writing this book. Without our families' support, this book would not be possible.

We accepted the challenge of a tight publishing schedule because, like everything else in the high-tech world, the contents are actually "perishable". We had just about enough time to catch up with the latest development in the space; as soon as we thought we were done, we found areas that needed to be updated. This book is useful only if we can make it accurate and up-to-date so that developers can benefit from the information in this book. Thankfully, we have many high-caliber reviewers and alpha book readers to help us to correct the information in the book.

Without a particular order and with no implication of the importance of their contributions to the book, these people include:

- Aaron Durbin
- Edward Roache
- Maurice Ma
- Ravi P Rangarajan
- Martin Roth
- Kangkang Shen
- Bob Hart
- Ron Minnich

We thank you for your efforts in making this book useful to many more people like you.

Foreword

By Ron Minnich
Creator of LinuxBIOS (later renamed coreboot)
Software Engineer at Google, Inc.

I started the coreboot project at Los Alamos National Lab in 1999. At the time, it was seen by hardware vendors as an impractical idea that would soon vanish. Now, 15 years later, it is mainstream: millions of x86- and ARM-based Chromebooks and Chromeboxes run coreboot, as its speed and reliability are an essential part of the Chrome OS user experience. coreboot is now a key component of the fastest-growing consumer laptop segment.

It might come as a surprise to embedded programmers that the initial goal of coreboot was to make very large supercomputing clusters manageable. We had a 128-node VA Linux cluster at Los Alamos in 1999 that had no keyboards or displays. BIOS upgrades required that we wheel around a "crash cart" with a keyboard and monitor; boot DOS on a floppy, which in turn started an autoexec.bat script; and wait 5 to 10 minutes for the process to complete for each node. If anything went wrong, it got more fun: we had to crack open the case, move a jumper, and do the recovery with no working graphics. As if this were not bad enough, the vendor BIOS had a habit of coming up displaying "No keyboard present—hit F1 to continue" on a nonexistent monitor, asking us to hit F1 on a keyboard that it had already discovered was not there!

Could this possibly get any worse? It could, and did, on the Thunderbird cluster at Sandia National Labs: 4400 nodes, none having a keyboard or a monitor, came up one day with that same vendor BIOS message. The fix? Dispatch 20 people with 20 keyboards to 220 machines each; they had to plug in the keyboard, hit F1, and hope it all went well.

By 2002, we had a 1024-node Linux cluster using coreboot. The reflash process for all 1024 nodes took 30 seconds total, not five days. If something went wrong, coreboot would figure it out on the next boot, switch to a backup BIOS image, and boot up: the nodes could not be put into an unrecoverable state. There was no longer a need to open the nodes and move a jumper. coreboot represented a huge jump in the manageability of cluster nodes.

coreboot has had many uses since its inception: everything from the smallest systems (Apache Military Modem II) to some of the largest supercomputers. While there is wide adoption in Chrome OS systems, coreboot's earliest and continuing use is in embedded systems such as televisions, network switches, and robotic systems. In fact, about the same time we deployed a supercomputer using coreboot, iRobot had ported coreboot to its Packbot robot.

Embedded systems used to be very simple: a low-power CPU connected to low-performance memory and peripherals, used in low-performance and limited applications such as digital clocks and automobile computers. But in the last ten years, we can see low-power embedded CPUs used in unexpected places. The highest-end systems—such as IBM Blue Gene supercomputers, which were the fastest in the world for many years—used 65,536 embedded PowerPC CPUs with 18 cores each. We now see higher-power CPUs used in small embedded systems such as Chromebox videoconferencing systems—an inexpensive system with a very powerful Intel CPU.

These embedded Intel CPUs have memory bandwidth much higher than classic Cray vector supercomputers and hence are fiendishly complex to design. Once designed, this high-performance hardware is quite difficult to initialize, and even should we wish to write the code to manage the initialization, the programming information is not public.

This has led to a dilemma: How can we enable coreboot on complex systems that are not fully open?

This book shows one path. Intel has released in binary form a basic set of functions to initialize the messiest—and hardest to program—bits of the Intel chipset. The calling conventions and behavior of the binary are completely and clearly documented. The developer is freed from having to deal with very difficult chipset setup. Upgrades of this software are simple: just replace the old binary with a new one. This code is called the Firmware Support Package, a.k.a. FSP.

The result is that high-performance Intel chipsets can be used with coreboot in all kinds of systems, including embedded ones, with a binary supplied by Intel that removes much of the porting difficulty.

Wearable embedded systems are a growing area right now, and many use the CPU described in this book. This book is an ideal companion for those wishing to be current with current and future embedded technology.

coreboot has succeeded because of the efforts of the many talented people involved in the project for the last 15 years. The reader is fortunate that this book is written by four of the best minds in the business. There is a lot to learn here and it will stand you in good stead if you continue to work in embedded systems.

I'd like to thank Jiming Sun and the team for conceiving FSP and bringing it to fruition. Without their tireless efforts and diplomacy, we would not have FSP or this book.

Introduction

We consider ourselves lucky enough to live in an era when new things and new ideas seem to come out every few years, if not every few days. We are not only experiencing an explosion of new ideas, but also witnessing some existing technologies being completely maxed out in our lifetime, including the semiconductor technology. Since Brattain and H. R. Moore made a demonstration of the first transistor at Bell Labs on December 23, 1947, the semiconductor, as we know it today, is reaching its physical limit, even though we are still trying very hard to shrink it below 10 nanometers. For the sake of argument, even if we can still shrink a couple of nanometers below 10 nanometers, how much further can we really go without changing the fundamental theory the technology is based on?

In the meantime, there are many other technologies that are approaching the limits of our sense and sensibility. Do we need more than 12 bits of color depth that shows more than billions of colors? Do we need a frame rate that is beyond what our eyes and brain can process? Do we need cars that go faster than our own response time? We now have display devices, media playback technologies, and transportation vehicles that achieve the best that they need to be.

Even though that is the case, there are still unlimited opportunities to make devices smarter and more connected to make our lives easier and safer. People are calling these devices the Internet of Things, or IoT for short. The explosive cycle of the IoT has just begun: cars will be talking to cars in the near future, thermostats and sprinkler systems can adjust themselves based on current weather forecast, buildings can manage lighting and air circulation based on where people are, and the list goes on and on.

Yes, this book is related to the explosion of the Internet of Things. We are addressing a technical area that is rarely talked about—the firmware inside of the Internet of Things. Firmware is the first piece of software that runs after silicon, coming out from the power-on reset state. Sometimes it is mysterious to people why building a firmware stack is hard and why firmware can be problematic. Considering the fact that the time it takes to run a piece of firmware is only between subseconds to a few minutes at most, why are we writing a book about it? After all, there are already books that talk about BIOS, UEFI (beyond BIOS), and techniques to optimize the firmware to boot faster. Why do we need yet another book to talk about firmware for the Internet of Things and the embedded system in general? There is one important reason: the firmware for IoT is different from the firmware running on a PC (BIOS or UEFI-based firmware), and there are many unique requirements for IoT firmware, and we will talk about them in the second chapter of this book.

This is what this book is about. We are going to examine the uniqueness of firmware requirements in embedded systems and IoT devices, and then we are going to introduce the technique Intel introduced to help IoT system firmware developers overcome the steep learning curve in developing a firmware stack for their versatile IoT products.

In this book, we are going to use two open source firmware stacks—coreboot and UEFI—to demonstrate the concept and show the steps to develop a workable firmware stack using widely available platforms from Intel. We are also going to show how the firmware works in a Chromebook, and what it does in a Chromebook, and we will also discuss the firmware for Intel® Quark family.

The targeted audience for this book are firmware engineers, hardware engineers, software engineers, and other professionals curious about IoT firmware. This is a good book for students who are learning about firmware, because we are going to give step-by-step instructions about how to build a workable firmware stack using commercially available platforms. For developers who have been involved in PC firmware, this can be a good reference book to understand the differences between PC and IoT, and the alternative solutions available. For people who have been struggling with Intel® Architecture (IA) and its firmware stack due to a lack of technical information from Intel in the past, this book reveals an opportunity for you to quickly get over the silicon initialization hump, and you will be able to quickly develop an effective firmware stack using the techniques learned from this book.

This book uses a lot of pages to describe the Intel® Firmware Support Package (Intel FSP) because it is a way to encapsulate the complexity of silicon initialization to make firmware development work easier. Since its launch in October, 2012, many developers and designers of alternative architectures have benefited from this product.

Why Should You Read this Book?

There are not many books out there talking about firmware because it is not a standard discipline that can be talked about generically. Every subject in the realm of firmware can be a book of its own, and there have been books about UEFI, BIOS, Fast Boot, RTOS, assembly languages, and so forth. There are also many system requirements and constraints that can dictate how a firmware is chosen and written; therefore, it is a topic that cannot be easily addressed holistically without an objective. Our objective is to show you how you can take advantage of Intel Architecture, and how to prepare a firmware stack for Intel microprocessors regardless of the firmware stack that you choose. There will be areas that are not covered in this book, such as power management and secure boot features, but readers can certainly find in-depth discussion of those topics in other technical books in the market. This book is written to help you build a workable firmware stack for Intel Architecture.

What Chapters Should You Read?

Since there are many interesting but distinctly different topics surrounding IoT device firmware, busy readers can pick and choose the chapters to read and skip if needed.

If you are just curious about what firmware options you may have for IoT devices, you may read Chapters 1 and 2 before diving too deeply into actual implementations.

If you are interested in developing a coreboot-based firmware solution for Intel Architecture, you can get a complete picture of the process by reading Chapters 1, 3, and 4.

If you are more interested in developing an EDK II–based firmware solution for Intel Architecture, you can get a complete picture of the process by reading Chapters 1, 3, and 6.

If you are more curious about what Chromebook is about and how the firmware for Chromebook works, you can read Chapters 1, 3, 4, and 5.

If you have heard about Quark and you are interested in building firmware for Quark, you can read Chapters 1, 2, 3, 4, 6, and 7. Why do you need to read more chapters for Quark? It is not because it is complicated, it is because it can be used in many varieties of applications using different firmware stacks. If you want a complete picture about firmware solutions for embedded applications and IoT devices, you should take your time and read all of the chapters in this book. After all, this is the purpose of the book: to give you a complete picture of the firmware solutions for IoT devices.

Hobbyists should be able to obtain a platform mentioned in the chapters, follow the instructions to download the source trees and tools, build a firmware image to try on a real platform, and enjoy the accomplishments.

Every firmware stack has its advantages and disadvantages; there will be situations when a developer needs to pick a different and unfamiliar firmware stack for the applications at hand. From time to time throughout the book, you might find some unfamiliar terminologies. We will list them here for reference. If you are still puzzled by a specific terminology, Wikipedia is probably the best resource to check. Internet search engines may be the second best source, but careful filtering of information is needed.

- *Bootloader*: This term might be confusing from time to time. In coreboot, bootloader is identified as the payload, which loads the OS, but in some cases, bootloader is used to represent the code from the reset vector to the hand-off point to an OS. The definition changes based on context. Also, this term is mostly used outside of the Intel Architecture (x86) world, where hardware initialization is not as complicated. In this book, we will not use bootloader to represent the complete firmware stack. When you see this term in this book or outside this book, you need to read the context to see which part of the firmware stack it is referring to.

- *Firmware stack*: In this book, we use the term to represent all the components in a firmware solution; there might be phases in the boot process of a firmware solution, but the term firmware stack will cover them all. We will also refer the firmware stack used to integrate Intel FSP as the "host firmware".

- *PI and UEFI*: Platform Initialization and Unified Extensible Firmware Interface. These are two major standards governed by the UEFI Forum. People are frequently using UEFI to represent the modern firmware stack that boots 64-bit OS in a PC. "UEFI BIOS" is frequently used to represent the firmware stack developed based on UEFI and PI specifications. PI specifications is a set of specifications that focuses on platform and silicon initialization.

- *BIOS*: Basic Input/Output Systems. This is a term that is used "conveniently" to represent the firmware stack of a PC, even though it is no longer the same 16-bit hardware abstraction layer to interface with a 16-bit OS. Today, as a habit, people are still using this term to call the firmware stack of a PC, even though the firmware stack has become more powerful, more dynamic, and has more features. You will see "legacy BIOS" and "UEFI BIOS" terms in the book when we describe the implementations of PC firmware stacks today. Some companies might still use "BIOS" in the names of their products, but the purpose is to associate their products to a more familiar terminology so that PC developers understand the products better.
- *FSP*: Firmware Support Package. Intel FSP is the silicon initialization module that Intel produces to encapsulate basic silicon initialization code.
- *Microprocessors, CPU (Central Processing Unit), chips*: These three terminologies are used interchangeably to represent the silicon that does more of the general computing and control tasks.
- *I/O*: Input and output.
- *SoC, SOC, SIP*: Silicon-on-Chip, Silicon-in-Package. This represents silicon designed to include more than one component on a die or in a package; typically these components are CPU cores, northbridge(s), I/O components, and other glue logic. From the outside, they function as an integral unit.
- *Southbridge, northbridge, and companion chips*: Today's SoC still contains components that we used to call *northbridge* and *southbridge* for two distinct functions that used be on different sides of a front-side bus (or a high-speed point-to-point bus) that connects all the components internal to the chip. Even though internal buses have evolved in modern SoC designs, the names northbridge and southbridge remain in many code bases to represent the functions that used to be there: northbridge deals with CPU, memory controllers, and other related features, and southbridge deals with I/O–related features.

Obviously, this book cannot cover all of the peripheral knowledge that you might be interested in. Here are online resources and links for further reading and research:

- http://www.intel.com/fsp
- http://www.tianocore.org
- http://www.uefi.org
- http://www.coreboot.org