CHAPTER 1

Introduction

If you can fix a hardware bug in firmware, it's not a bug but a documentation issue.

—An anonymous hardware manager

What Is Embedded Firmware?

Since you are reading this book, you must have some understanding of what the words *embedded* and *firmware* mean in the context of computer technology. There are quite a few interesting discussions on the Web about the difference between firmware engineers and embedded engineers. Some say that an embedded engineer is a software engineer turning into a hardware engineer, and a firmware engineer is a hardware engineer turning into a software engineer. There is some truth to this because most firmware engineers learned how to design circuits in school, but there are definitely a lot of smart firmware engineers out there with a computer science or nonhardware degree. Regardless, writing firmware is a unique skill that deals with both hardware and software at the same time. You need to know something about signal strength, timing, voltage, and at the same time, data structure, algorithm, and modularity.

For the purposes covered in this book, let's define *firmware* as the layer of software between the hardware and the operating system (OS), with the main purpose to initialize and abstract enough hardware so that the operating systems and their drivers can further configure the hardware to its full functionality. To make embedded systems run faster and be more robust, the relationship between the firmware and the OS is transitioning from isolating themselves from each other to cooperating with each other. In the past, you might have seen the same hardware initialized by the firmware first, and then initialized again by a driver in the OSas shown in Figure 1-1; but in modern systems, you see more effort in trying to eliminate redundancy between the firmware and the OS.

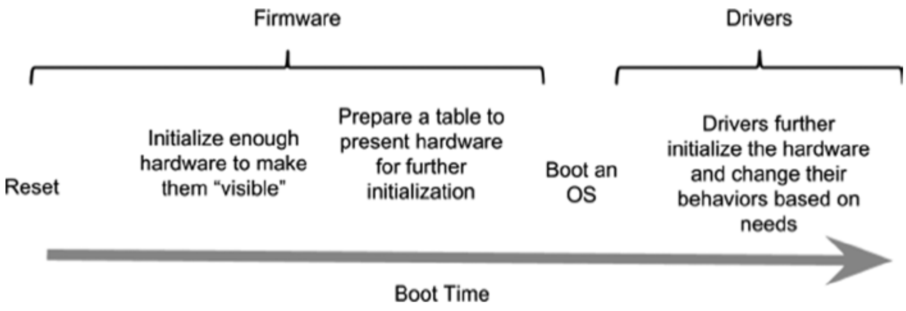

Firmware
Drivers
Reset
Initialize enough hardware to make them "visible"
Prepare a table to present hardware for further initialization
Boot an OS
Drivers further initialize the hardware and change their behaviors based on needs
Boot Time

Figure 1-1. *The roles of firmware and device drivers during the boot process*

Hardware design is also moving toward being more "software friendly" so that hardware takes less effort to program. Self-initializing chips and a built-in boot ROM are just two examples currently available in System-on-Chip (SoC). Due to the evolution toward software friendliness, the heavy-duty hardware initialization responsibility has been gradually shifted from the firmware to the drivers of the operating systems, and the operating systems are relying less and less on firmware to carry out the hardware initialization work. Some real-time operating systems (RTOS) and specialized operating systems, such as Android and Chrome OS, are carrying out a lot of hardware initialization functions with help from the hardware vendors. This is especially true when system designers compartmentalize their design to a set of standardized hardware components and constrain the hardware selection pool; in some cases, hardware configuration can be achieved with a GUI (graphical user interface) -based configuration tool instead of relying on rewriting programs in firmware or software.

The line between firmware and specialized operating systems is definitely blurring, if not disappearing completely. The very minimum things that firmware has to do in a well-coordinated environment are presenting a data structure of features that can be further processed by OS drivers, such as an ACPI (Advanced Configuration and Power Interface) table, and carrying out the tasks that can only be done more effectively by firmware, such as memory controller initialization. When a seamless cooperation between the firmware and the OS cannot be guaranteed, firmware still plays a significant role in the system to make sure all system features are utilized properly. Therefore, no matter what the design trend is, firmware remains a critical component in a system.

Let's face it: firmware does have a troublesome reputation in the x86 world. In the dawn of laptop computers, System Management Mode (SMM) was created to do many things behind the back of the OS. System Management Interrupt (SMI) was not only nonmaskable, but was also not controllable by the OS. Firmware would take its own initiative to blank the display to save power, put the system to sleep when no user activity was detected, slow down the clock when running on the battery, take care of battery warnings when they happened, and react to Fn hot keys when they are pressed. These functions were not coordinated with the OS, and they just happened, seemingly at random. Obviously, none of these are bad features, but the amount of time it took to go through the process, and the adverse effects (missing timer ticks, long latency in interrupt delivery, etc.) on the OS were too much for the OS to ignore. Therefore, ACPI was created to allow the OS and firmware to coordinate and cooperate on these features. The utilization of SMM has been gradually reduced over the years. In addition to some SMM firmware dealing with actions requested by ACPI and security features, chip vendors are the only entities still using SMM to work around chip issues when necessary.

Where Is Firmware?

Firmware is generally considered part of the hardware (rather than part of the software) because it resides inside a hardware component, which is typically a Flash storage device or ROM. However, when it comes to the programming language, the tools, and the methodology a firmware engineer follows, firmware is clearly a type of software, even though it is tightly coupled with hardware in most cases.

What Do Firmware Engineers Do?

Regardless of whether firmware engineers are hardware engineers turning into software engineers, or the other way around, firmware engineers have a lot of interesting work to do. One of the most important and challenging jobs that a firmware engineer does is to make a new circuit board work when it is first manufactured, especially when most components on the new board are also new. It is not a unique case to have many new components on a brand-new circuit board because most hardware evolves in similar cycles. In chip manufacturing companies, some firmware engineers' only job is to bring up and test a new chip on a new board. The combination of a new chip, new components, and a new board not only makes debugging work much more complicated, but also makes the preparation work much more challenging.

When a firmware engineer prepares for the bring-up of a new board, he or she does not just wait for the hardware to show up, and then write the code; he or she needs to read a lot of early specifications, such as datasheets, a couple of months before the new board shows up. Since these specifications are mostly evolving, the information may not be 100 percent correct. Firmware engineers need to help correct the information in datasheets and write programming guides as they go through the debug process. In a typical hardware-oriented company, firmware needs to be ready when the first circuit board shows up. It is not acceptable for hardware to wait for firmware because hardware is usually more difficult and more expensive to alter than firmware. Manufacturers want hardware bugs to be discovered and fixed as early as possible. In the beginning, firmware only needs to have enough functions to test the circuit board to determine if the new components can be manufactured, but it is a lot of hard work to do it right the first time.

Firmware Preparation for New Hardware

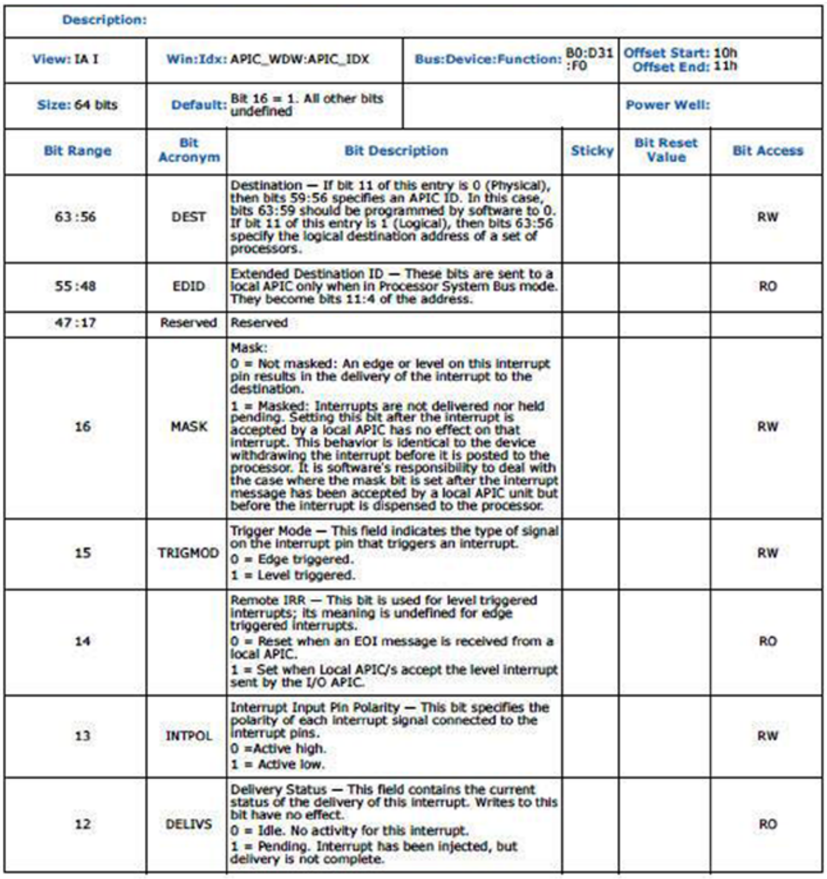
During the preparation, firmware engineers need to figure out what to program, how to program, and the sequence in programming the new components after they study the materials available to them. The obvious challenge after writing the program for the new components is to figure out how to test the new code before the new hardware shows up at the door of their lab. Many manufacturers have simulators designed to test early firmware and software, such as Intel's Simics and AMD's SimNow, but the usefulness of a simulator depends on the behavioral models written to simulate the hardware. The accuracy and fidelity of the models in the simulation tool decide whether or not you can find bugs and programming errors in your early firmware.

Besides simulators, FPGA-based emulators are also frequently used to test early firmware and the circuitry inside a new chip. Compared to simulators, FPGA-based emulators are much more accurate in representing the final hardware, but since they are running in a much slower clock speed, timing-related issues may not be discovered easily, and firmware is sometimes modified to accommodate the slower clock speed; therefore, some parts of firmware are not well tested. In most cases, based on our experience, these simulators and emulators actually deliver pretty solid results, and the simulated and emulated firmware usually works when it is put on the real hardware for the first time they integrate. Even with the help of simulators, it typically takes a couple of days, weeks, or even months of effort to iron out all the hardware issues.

The Mystery of Bits

In the process of preparing a bring-up firmware, a firmware engineer spends a lot of time figuring out what needs to be programmed into the microprocessor and chipset by studying datasheets and specifications. In this case, literally, every bit matters. Even though many bits will work in their default states, a single mistake in misinterpreting the definition of a bit in a chip can turn the circuit board into a brick, and there are many bits to be programmed from their default states in order to work properly.

A datasheet is like the Bible for firmware engineers (see Figure 1-2); it has almost everything a firmware developer needs to know about the chip. Using a published datasheet from Intel as an example, the datasheet for Intel® Communications Chipset 89xx Series, published in October 2012, contains 1,682 pages of useful register data. The list of tables that are used to describe the registers span across 30 pages, with about 60 entries on each page.



Description:					
View: IA I	Win:Idx: APIC_WDW:APIC_IDX	Bus:Device:Function: B0:D31 :F0		Offset Start: 10h Offset End: 11h	
Size: 64 bits	Default: Bit 16 = 1. All other bits undefined			Power Well:	
Bit Range	**Bit Acronym**	**Bit Description**	**Sticky**	**Bit Reset Value**	**Bit Access**
63:56	DEST	Destination — If bit 11 of this entry is 0 (Physical), then bits 59:56 specifies an APIC ID. In this case, bits 63:59 should be programmed by software to 0. If bit 11 of this entry is 1 (Logical), then bits 63:56 specify the logical destination address of a set of processors.			RW
55:48	EDID	Extended Destination ID — These bits are sent to a local APIC only when in Processor System Bus mode. They become bits 11:4 of the address.			RO
47:17	Reserved	Reserved			
16	MASK	Mask: 0 = Not masked: An edge or level on this interrupt pin results in the delivery of the interrupt to the destination. 1 = Masked: Interrupts are not delivered nor held pending. Setting this bit after the interrupt is accepted by a local APIC has no effect on that interrupt. This behavior is identical to the device withdrawing the interrupt before it is posted to the processor. It is software's responsibility to deal with the case where the mask bit is set after the interrupt message has been accepted by a local APIC unit but before the interrupt is dispensed to the processor.			RW
15	TRIGMOD	Trigger Mode — This field indicates the type of signal on the interrupt pin that triggers an interrupt. 0 = Edge triggered. 1 = Level triggered.			RW
14		Remote IRR — This bit is used for level triggered interrupts; its meaning is undefined for edge triggered interrupts. 0 = Reset when an EOI message is received from a local APIC. 1 = Set when Local APIC/s accept the level interrupt sent by the I/O APIC.			RO
13	INTPOL	Interrupt Input Pin Polarity — This bit specifies the polarity of each interrupt signal connected to the interrupt pins. 0 =Active high. 1 = Active low.			RW
12	DELIVS	Delivery Status — This field contains the current status of the delivery of this interrupt. Writes to this bit have no effect. 0 = Idle. No activity for this interrupt. 1 = Pending. Interrupt has been injected, but delivery is not complete.			RO

Figure 1-2. *An example of a datasheet page*

Learning and understanding what each bit does and does not do is as tedious as sorting sand on a beach, especially when some of the data documented in the datasheet does not provide as much detailed information as needed to understand how to use it. Sometimes, it takes a lot of trial and error in the process. There are also many undocumented bits that are there either for internal testing or for tuning purposes; these bits are usually not documented inside a datasheet. Therefore, missing critical information could be another challenge for firmware developers.

From time to time, a mysterious problem can stall the debug and development effort for a long time, and the final solution is sometimes a mysterious bit that was somehow discovered after scrubbing the design data. With a stabilizing feature set and better tools, these kinds of problems are not happening as frequently in modern chips.

It is the purpose of early-stage firmware to find the problems of a newly designed chip. Even though there are chip bugs that cannot be resolved without fixing the chip itself, more often than not, a chip problem can be resolved with fixes in firmware; chip vendors frequently call these kinds of fixes “work-arounds.” If there is a work-around for a chip bug, it usually involves a bit or a set of bits that need to be programmed or changed. Or, there will be an algorithm developed to work around a problem only when certain conditions are met. To apply a work-around with as little impact as possible to the existing software, designers frequently suggest the fixes to be implemented in the SMM. Since SMM code takes away operating cycles and time, these kinds of fixes are sometimes intrusive and problematic. It is ultimately up to the designers to reveal the fixes after studying the original design of the chip. Most of the time, the fixes are not obvious; the designers need to analyze and figure out if there exists a setting of bits that could fix the problem, or check if they should turn off certain new features that are not working properly. Even though a firmware engineer may accidentally find a fix to resolve an issue through a trial-and-error process, this is very rare these days—especially when the work-around involves an undocumented bit or bits. Designers ultimately hold the key to resolving a chip issue.

How do undocumented bits exist? As stated earlier, when designers design a chip, they put many configurable bits in a chip to help tune the chip or control features that are supposed to be hidden from the programmers. They keep some of these bits undocumented and locked so that no one can accidentally program them to cause unintentional damage or adverse effects. There is a nickname for these bits, called “chicken bits.” The origin of this phrase is unknown, but it may have something to do with the fact that these bits are scattered everywhere in the chip like the food for chickens; or it may imply that the designers were too “chicken” to show these bits to others, therefore hiding them.

There could be as many as 60,000 chicken bits in a chip, depending upon the complexity, the functionality, and the size of the chip. This also explains why a chip vendor cannot possibly document every programmable bit of a chip in a datasheet or a programming guide, even if they wanted to try. As a matter of fact, most of these chicken bits will never be documented, and a small portion of the chicken bits will be documented only when they are needed to work around a chip problem. Many of you may have heard of or even read an errata sheet from chip vendors; this errata sheet frequently contains information for bits that were not documented before.

Programming Guides

When designers design a new chip, they will compile of a list of registers to support various chip features. During the chip design phase, hardware engineers and firmware engineers work with designers to design, simulate, and validate the chip. By doing this together, the function of bits and bytes in registers are defined, refined, and documented so that a comprehensive programming guide can be available at the same time that the first chip shows up at a customer’s door, typically a manufacturer of a product using the chip. In Intel, this programming resource is named the *BIOS Writer’s Guide* (BWG) because it is designed to help BIOS developers write a BIOS for a PC to begin with. For a modern chip like Quark, the name of the document has been changed to the *UEFI Firmware Writer’s Guide* to distant itself from the term *BIOS*. Regardless, this programming guide has all the information a firmware engineer needs to know beyond programming a PC and for every embedded system as well.

In 1996, the BWG for the Pentium Pro was 73 pages long; but today, the BWG for the BayTrail SoC is 440 pages divided in two volumes; it grew more than six times in 18 years. Not only the amount of information, but the complexity has increased as well.

For example, the Pentium Pro BWG describes very basic programming information that most people can probably figure out themselves after reading a few standard specifications from Intel, like SMM, BIOS INT functions, and how to handle multiprocessor initialization and so forth. In comparison, besides basic SMM multiprocessor topics, BayTrail BWG uses 34 pages just to talk about MSR (model-specific register); other information includes CPUID handling, the Microcode Update, SpeedStep, C-State Control, thermal management, and more. This list is just the information contained in volume one. In volume two, almost all the subjects require a domain expertise to understand, such as HD audio, graphics, the HPET timer, xHCI, EHCI, DDR3, ISP, P-Unit, SIO, PCIe, PCU, TCO, and so forth. (If you don't recognize any of these acronyms, you get the point: modern-day BWGs have become very specialized repositories of information.)

The programming guide not only specifies the features that can be customized by the customer via programming the bits and bytes, it also methodically covers the many bits and bytes that must be programmed with particular values in a fixed order to support certain features.

Even if you have not been involved in debugging a new circuit board, you can imagine that finding a bit-setting error in an ocean of bits is pretty painful. Does a chip vendor need to put firmware engineers through a painful experience every time it produces a new chip? If the programming guide is done right, firmware engineers should not have to read through 440 pages of BWG to study what needs to be programmed just to get the chip up to the point of providing its features. Firmware engineers have better things to do than look at the programming guide to figure out which bits to flip and which bytes to write; they should spend their valuable time developing value-added features to help deliver a product with differentiating features.

The Intel® Firmware Support Package

Intel has taken the initiative to provide a way to encapsulate tedious chip initialization code into a package: the Intel® Firmware Support Package (Intel® FSP, for short; we will use *Intel FSP* and *FSP* interchangeably in this book). Obviously, there is more than one way to ease the programming pain associated with chip initialization. For example, releasing the full source code to allow people to view how it is done, or putting all the chip initialization code in one binary to hide the complexity, or anything in between. Intel has decided to go with the option to put all the chip initialization code in one place with the hope that, once it is used in the firmware stack of your choice, the developers will be able to quickly get over the chip initialization hump and move on to value-added features development work.

Why would Intel produce Intel FSP for the embedded designs and the Internet of Things (discussed shortly) in the first place? After all, Intel is already providing a comprehensive reference code for each reference platform, and there is also an open source EDK II codebase under Tianocore.org that allows people to study the UEFI implementation. The problem is not about having reference code out there or not; it is portability, scalability, and flexibility that developers are looking for, especially in the embedded and IoT space where UEFI and PC architecture are not playing a major role as things stand today.

Intel recognizes that many developers have a hard time extracting chip initialization code out of an EDK II codebase or from a BIOS to port to a different firmware stack or to a different platform design. Going to IBV (Independent BIOS Vendors) to ask for help is not always an option for some customers. Intel believes that there are a lot of smart firmware engineers out there, and once these engineers get hold of a technical specification, an industry standard, or reference code, they can produce a firmware implementation without too much difficulty. The only thing that is missing for them is the chip programming information.

That said, it does not matter how smart the firmware engineer is: he or she cannot and will not be able to figure out how to program a new chip without help from chip vendors. Over the years, some smart developers have tried to reverse-engineer what has been done in an existing platform, but the process is long, hard, and error-prone.

In this book, we talk about Intel's FSP solution; other chip vendors have also provided similar packages or mechanisms to reduce the programming burden for initializing basic chip functions, such as AMD's AGESA (AMD Generic and Encapsulated Software Architecture), and ARM's boot ROM concept. This book will not discuss these implementations, but they are efforts that chip vendors put out to ease the programming challenges.

Since its launch in October of 2012, Intel FSP has been widely used in many customers' embedded designs, including customers who chose to convert from a competing architecture to Intel Architecture. As you will discover in later chapters of this book, Intel FSP can be easily integrated into an existing firmware stack to save you time and energy in figuring out the information needed to program an Intel chip.

■ **Note** Keep in mind that Intel FSP is not a stand-alone firmware stack. It does not have all the ingredients to boot to an OS on its own; therefore, it must be integrated with a firmware stack, such as BIOS, coreboot, RTOS, or other proprietary bootloader solutions.

The Uniqueness of Embedded Firmware

It is one thing to develop firmware for a general-purpose and open system such as a PC; it is quite another thing to develop a firmware stack for a closed system with dedicated functions. Over the last three decades, firmware engineers have almost perfected BIOS (including UEFI) for the PC. The PC BIOS has the ability to deal with devices that come and go anytime (plug-and-play) and to boot to any general-purpose OS (Linux, Windows), and it is smart enough to learn about its environment (ambient light, battery status, and user inactivity) to adjust itself to save energy. Arguably, it could be the most intelligent firmware stack ever created. Over the last three decades, we have definitely seen the evolution of BIOS, and we have witnessed a great improvement of quality in the BIOS realm.

Can the experience learned from the PC BIOS be applied to an embedded firmware design? The answer is "yes" because there are many useful industry standards created and implemented in the code.

Can we use a PC BIOS stack on an embedded firmware design? The answer is "yes, but…" because it depends on the purpose of the design. Many embedded designs leverage PC architecture for cost and ecosystem support reasons. Since the designs inherit all the characteristics of a PC, the PC BIOS and similar technologies could still be the best choice for an open and general-purpose system if boot speed and the size of the firmware stack are not issues of concern. There are also a lot of embedded designs that are not based on PC architecture. In these cases, the PC BIOS and similar technologies can be used, but will need a lot of effort to fit into the design; there are better firmware solutions out there to choose from. In some cases, a PC BIOS just won't work because the special need for boot speed, a small footprint, real-time performance, and so forth, are required.

Many mission-critical and time-sensitive designs require the system (hardware, firmware, and software) to be deterministic and predictable among other special constraints; many intelligent and dynamic configuration capabilities of a PC BIOS can be prohibitors for those deterministic and predictable considerations. Some closed devices designed without any upgradability do not need the flexibility of plug-and-play, heuristic training algorithm, and bus enumeration techniques that are typical in a PC BIOS; therefore, embedded firmware can sometimes be dramatically simplified. Finally, there are more embedded designs asking for faster boot time and quicker response time, which a typical PC BIOS cannot achieve. The rear-view camera in a vehicle, for example, needs to be turned on within 2 seconds after the car engine is ignited; this is for safety concerns and also a regulation requirement in many countries. Even though it is not impossible for a PC BIOS to achieve faster boot time, the hardware and firmware stack used in an IVI (in-vehicle infotainment) system is heavily customized to achieve this stringent boot-time requirement. All of these unique requirements make a PC BIOS harder to fit into embedded applications.

The Choice of Firmware Stacks

There are quite a few firmware options in the market, such as PC BIOS (including various UEFI implementations), RTOS, open source stacks, and proprietary solutions. Each of these firmware stacks has a specific purpose to fulfill, and we will discuss their special usage models later in the book. We'll also discuss how to work with different firmware stacks using a common chip initialization module (in the next chapter, we'll take a look at a few of the options), and we will also talk about their pros and cons and how to make a choice based on your needs.

Welcome to the Era of the Internet of Things

Due to the fast evolution of microprocessors, embedded systems are not only becoming more intelligent, but also becoming more ubiquitous. Some of the devices we use today will start to make decisions for us and even protect us from danger someday. There are refrigerators that can order food for us when they are empty. There are thermostats that can call the police when they detect an intruder in our house. There are cars that can steer themselves out of danger when they are about to collide with another car. There are a lot of more new ideas and new devices in the works to make you safer and make your life more convenient. The next wave of the technology revolution has arrived, and it comes in the form of built-in connectivity and intelligence. People are calling these intelligent and connected devices the "Internet of Things" or IoT. It takes a lot of creative minds to conceive, develop, and refine such an IoT, and no doubt much of the work needs to be done at the firmware and software level. The demand for the skill sets covered in this book will increase in the foreseeable future.

Technical Coverage in This Book

This book covers topics related to embedded firmware stacks and an important ingredient that enables them—Intel FSP. Since this book uses Intel Architecture as the centerpiece, we focus on how Intel FSP works and demonstrate how it can be integrated into popular firmware stacks; coreboot and EDK II examples are used in this book. We cover detailed information about each of these two firmware stacks, including the internals and how to work with them. The reader can pick and choose a particular subject and do a deep dive, or the reader can choose to learn both of the firmware stacks in a holistic way. If you are among the readers who are not using either the coreboot or the EDK II codebase, the same principles and practices also apply to your own firmware stack once you have a basic understanding.

This book does not cover the details of implementing specific features, such as power management, device enumeration, graphics, audio/video, and other non-chip related features. However, this book will touch upon the new Intel Quark family products and talk about their firmware stack and detailed firmware architecture. Even though the firmware strategies of some Intel products are not yet using FSP as the building block, you can compare and understand the differences and the rationale behind the decisions.

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 on its own, and there have been books about UEFI, BIOS, Fast Boot, and so forth, and many system requirements and constraints 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 prepare a firmware stack for it regardless which firmware stack you choose. There might be areas that are not covered in great detail in this book, such as integrating Intel FSP into a RTOS codebase, but hopefully the same practices still apply.

The Future of Firmware

We have been witnessing an interesting phenomenon since the beginning of this century: open source projects are gaining momentum, led by companies such as Google and Facebook. Many legacy and proprietary software solutions are either disappearing or losing steam very quickly; open source solutions are becoming a primary interest of technologists at an amazing speed.

Even though this century is still young, we are riding on a fascinating wave that will make the 21st century a distinctly different century than any other. The phrase "open source" clearly connotes sharing and collaboration, in contrast to the waning business philosophy of protecting intellectual properties so that "we can win as an individual company." What has emerged is a new concept to tie business success to a collaborative ecosystem effort. Under this new model, everyone has a chance to thrive in the ecosystem because innovation is multiplied with the participation of many intelligent scientists, hobbyists, and engineers.

Will embedded firmware solutions be moving toward the same model that operating systems and computing systems are currently going through? Yes, we can tell that the open source model is impacting the firmware world in equal force in the last few years. Even some of the traditionally closed solutions are adapting themselves to the open source model as we write this book. For example, even though Tianocore was created as an open source project by Intel years ago, the community did not thrive due to lack of crucial components in the source tree. Until recently, many ARM developers have tapped into the Tianocore source code to make it workable with ARM designs. Intel has also ramped up its effort to provide an FSP component so that it can be successfully built for a platform in distribution.

However, due to its many unique characteristics, firmware is a source of intense debate because it is so tightly coupled with the hardware underneath, and hardware still has IP that companies want to protect. The questions for chip vendors will always be how much firmware can be benefited by the collaborative model, and how much firmware is so chip-specific that there is nothing that needs to be collaborated on? For example, if some of the chip code is fragile enough that only the designers can program it right, does it make a difference if the code is open or not? If it is open, does it do more harm than good? How about the code inside a boot ROM that is masked? Should that code be open as well? What about the firmware that is currently outside an SoC? Where is the boundary? What makes sense to be open and what does not? These questions may be rhetorical, but the binary distribution format will gradually move away from being a mechanism to hide features, and instead be a way to simplify its integration.

Putting these rhetorical questions aside for the moment, the open source model is a trend that all chip vendors, BIOS vendors, and software vendors will continue to adapt to. In the spirit of open source communities, the best of the best will emerge after all the debates, arguments, and brainstorming. This book was written with the aim to enable the open source community to thrive on its own and alongside proprietary solutions. It will undoubtedly fall short of some readers' expectations, and it may disappoint a few hard-core open source enthusiasts. However, it is still a step toward the right direction to make chip initialization a non-issue (or less of an issue) for open source communities.

We are all aware that the state of the art in chip engineering will continue to rapidly evolve. New methodologies will emerge to make firmware development easier and faster for the developers trying to bring innovative IoT devices to the world, and perhaps subsequent editions of this book will cover those developments. In the meantime, we welcome you to this discussion. Please read on.