CHAPTER 8

Putting It All Together

> *"The future depends on what you do today."*
>
> —Mahatma Gandhi

Embedded firmware has a lot of facets beyond the topics we could cover in this book. Our goals have been to provide enough knowledge and tools to get you started quickly when you deal with an embedded design based on Intel Architecture. The contents in this book may not have anticipated all the changes in the specifications and APIs that were under development when we were writing this book, but the framework of the book can represent the essence of the ideas and examples shown in the book. Regardless of whether you have read all the chapters in this book, you should have learned about the key ingredients mentioned in it: Intel FSP, coreboot, EDK II, Chromebook firmware, and the unique firmware features that embedded industries are looking for.

If we include automotive, industrial, medical, retail, smart home and building, avionics, military, print imaging, gaming, and other broad market sectors in the picture, there are many versatile design requirements—ranging from using completely open standards to using only proprietary closed specifications; and from performing multiple general-purpose functions to possessing only a single dedicated function; and from having a long life cycle that spans a couple of decades to a short life cycle that lasts just long enough until the next coolest thing comes out. Some sectors require a rigorous certification process to examine every line of code shipped in the unit to ensure safety and security, and it is costly if any line of the code needs to be updated from both a time and a monetary perspective. That explains why some crucial components, such as the recording technology embedded in the black box of an airplane, do not seem to evolve as quickly as other sectors are experiencing. Each vertical sector has to develop its own methodology to optimize its own firmware development approach to maximize its value.

From the perspective of firmware development methodology in these vertical sectors, developers can choose a high-touch design model using an army of developers to write every line of code, or choose to use a low-touch turnkey model with only a handful of staff to turn control knobs provided by a software utility that cranks out a firmware binary at the other end. It is the decision of the business development managers, not only based on the customers they are targeting, but also on a long-term view of the business.

From the ecosystem perspective, some vertical sectors have a large well-established ecosystem; some have only a small ecosystem with only a few players in it. The availability of the either high-touch or low-touch solutions may not be consistently available at the same time.

Therefore, the spectrum of supporting all these vertical sectors is wide open and full of variables. The thing we need to keep in mind is that there is no one-size-fits-all solution out there. Even within the same company selling the same source code to their customers, some customize the common source code and turn them into two or more products to serve different market sectors. For example, it is a very typical process for a RTOS vendor to provide one RTOS product for avionic customers to satisfy safety-related certification, and yet another RTOS product for military customers to use for security-specific certification. Safety standards—such as ARINC 653, ISO 26262, and IEC 61508—define critical requirements that are very different from security-focused standards that regulate data integrity and protection. The most important standard, MILS (Multiple Independent Levels of Security) architecture, defines four conceptual layers of separation in separation kernel and hardware, middleware services, trusted applications, and distributed communications. The RTOS or firmware vendors in these spaces frequently provide several products to serve the customers that need certain certification done. It is very costly and tedious to run through the line-by-line source code certification process; some said that it can be $50 or more per line to cover the certification. That is why sometimes a smaller code size is critical to some market sectors.

RTOS and Intel FSP

As discussed in Chapter 2, there were more than 120 active RTOS projects being developed for various applications and vertical markets at the time this book was written. Intel FSP can be a useful ingredient if these RTOS vendors are looking for a reliable way to support Intel Architecture.

Traditionally, RTOS products have been mainly designed for microcontrollers and low-end microprocessors rather than feature-rich microprocessors; therefore, there has not been a lot of demand to support Intel's Atom and Core products until recently, as IoT devices are becoming important and Intel pushes Quark SoC and other low-power products to the market. As Moore's Law continues to affect microprocessors' size and power reduction, it is more and more feasible to use high-performance Intel processors in IoT and embedded devices. Intel has put in a considerable amount of effort to make embedded-friendly products. As a result, more and more developers are looking for RTOS solutions to support Intel microprocessors; this is especially true since Quark SoC was introduced to the market in 2014.

There are currently two ways for a RTOS to work with a microprocessor that requires some silicon-level programming:

- An RTOS vendor can choose to put a firmware stack or bootloader underneath. Just like a PC, the target system can first boot a firmware stack, such as BIOS, and then boot the RTOS kernel using a GRUB or similar OS bootloader. This is a viable solution as long as boot speed and code size are not the critical design requirements. A size- and speed-optimized firmware stack can make the option more appealing too.

- RTOS can also have a native silicon initialization code built inside the RTOS stack. With Intel FSP, RTOS vendors can choose to boot natively by integrating Intel FSP into the RTOS boot code.

In 2012, Wind River Systems did a demo at the Intel Developer Forum to boot to a VxWorks kernel on an Intel platform in 350 milliseconds using native Intel silicon initialization code. Obviously, the demo was done without graphics and PCI devices. RTOS vendors often deal with a system without graphics and PCI devices; therefore, it is a viable solution for RTOS vendors to integrate a small silicon initialization code, such as Intel FSP, and add other components on top of it, if necessary.

Intel FSP and Open Source Philosophy

When the Intel® Quark™ SoC X1000 Series was announced, its EDK II source code was fully released as an open source project on Intel's web site. It is actually a source code without the involvement of Intel FSP. If an open source code base has all the source code available, do we still need Intel FSP? This is a good question, and the answer is: it depends. As we discussed in this chapter, many embedded firmware developers do not have the time or the expertise to look for components inside an open source codebase, and they are frequently looking for a way to quickly enable their own firmware stack to boot to their value-added features. Giving them all the source code that contains the silicon initialization code is useful, but it does not help if they need to quickly find the components inside the source code and port it over to another firmware stack. The EDK II source code uses a diver model for device initialization, and the documentation provided has comprehensive data to guide a developer to extract the source, if needed. Therefore, it is very feasible for a seasoned firmware engineer to find all the necessary source code to port it over to a different codebase.

Even though many hard-core open source advocates always love fully opened source code, as a compromise, many open source developers don't mind picking up some code that other people have written, tested, and verified, and then wrapped in a binary file. Because these developers don't have too much value to add to the code (serving a very specific hardware enablement mission), the code can be treated as part of the "hardware" as necessary. For example, a chip needing rigid and sensitive hardware initialization may not have too much room for innovation. Obviously, this is debatable; many developers can still add value by optimizing and reducing the code size if they want to. Silicon vendors may not have done the best job in optimizing the code to initialize their chips.

That said, the majority of the developers do not have as much time to look for firmware code in the open source repository to construct a firmware stack, or simply do not have the expertise to experiment open source projects.

Even though Intel FSP is far from a turnkey solution, it encapsulates enough silicon initialization code to provide the flexibility that people need to pick and choose a different firmware stack to work with. One day, Intel may decide to make all the components inside Intel FSP open source; but even if that happens, the concept of having all the silicon code in one place, such as Intel FSP, should still be the best option for people who do not want to invest time in figuring out how to initialize Intel's silicon.

Since the goal of Intel (and potentially other silicon vendors as well) is to simplify the silicon initialization work inside a firmware stack, it is predictable that there will be many types of firmware solutions available to the market from Intel to help developers—from completely open source to completely turnkey. Some solutions may require more work than the others; and some may be no work at all beyond turning a few knobs on a utility-based GUI user interface. Intel FSP will be a key component of all. As shown in Figure 8-1, the options can be an integration of FSP inside a host firmware, or a turnkey Boot ROM that allows customization by an OEM and ODM, or to the solution where the designers can pick and choose components from a set of prebuilt initialization modules.

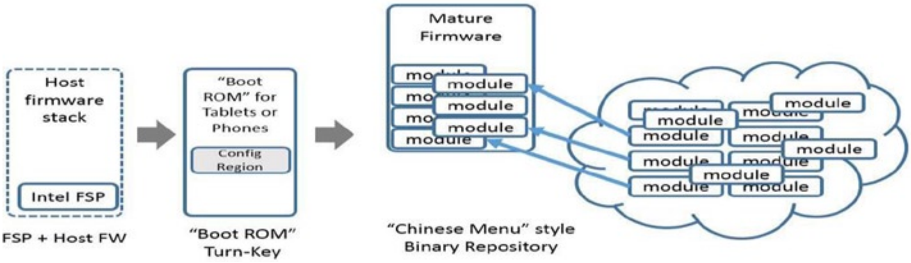

Figure 8-1. *Prebuilt silicon and hardware module options*

For example, in some markets that involve standardized platforms with limited hardware selections, "Boot ROM"-type turnkey solutions can be made available to simplify the development work. In some mature markets with many known hardware components, bus topology, and standards, there can even be a binary repository for constructing a complete ROM by using "Chinese Menu"-style of configuration tool recently introduced at the 2014 Intel Developer Forum.

Customization and Production of Intel FSP

Some advanced developers might have gone beyond just consuming Intel FSP in their firmware projects; they may be thinking about customizing and producing FSP for themselves or their downstream customers. For example, some IBVs and ISVs are in the business of creating firmware solutions—such as Board Support Package (BSP), services, and tools, and they might already have an intimate knowledge of Intel silicon. These vendors can consider producing a customized FSP to a special market that they want to serve. The other possibilities are when an OEM or ODM customizes Intel FSP for a particular family of their products, or when an OEM/ODM wants to distribute a FSP to its own vendors after customization. All these scenarios can be made possible in the future.

It Is a Community Effort After All

Regardless of whether you have read the book thoroughly, or picked and chose the chapters you were interested in, this book is written to help you develop a firmware stack for Intel Architecture quickly and effectively. During the process of developing Intel FSP, and while writing this book, we received a lot of good feedback and suggestions on how to make Intel FSP better, and how to make this book more useful. We are in an era of openness; as part of the process in embracing openness, we encourage you to contribute your constructive ideas as well.

The four authors and countless reviewers of this book are coming from different companies (Intel, Google, Sage, and Huawei). We are extremely proud that we are able to pull this book together to introduce to you the firmware solutions for different embedded and IoT devices. We know that there are still many stones unturned and many interesting topics not yet discussed related to this topic, but this is hopefully a good starting point to get everyone excited and involved.

Together, we can make embedded firmware easier and better, so that we can have more intelligent and reliable products; this is why we wrote this book. Regardless of whether the source code is UEFI-based or coreboot, or U-Boot, or RTOS, we hope that everyone can benefit from this book.

Now, go write some firmware.