CHAPTER 2

Firmware Stacks for Embedded Systems

Computers themselves, and software yet to be developed, will revolutionize the way we learn.

—Steve Jobs

If you have been doing BIOS work or read a book about it, you probably have an idea about what firmware is doing for a PC. Firmware needs to do the following:

- It needs to discover devices that are connected to a system bus
- It needs to deal with devices disappearing and appearing anytime during runtime
- It needs to be prepared to boot any operating systems that are written for the PC
- It needs to wake up if an external stimulus occurs
- It needs to adjust its backlight, clocks, and speed based on the temperature, the power source, and the status of the user (inactive or active)

However, embedded systems and IoT devices have some unique firmware requirements from PC BIOS:

- *Timeliness in responding to external stimuli (real-time behavior).* The examples are industrial precision machines. These machines cannot tolerate any deviation from their specified error margins during operation.
- *Determinism during execution*. The examples are missiles and rockets. You probably don't want to see these things taking different software paths every time they fly out of their launch pad; who knows what will happen if they act differently every time they execute their software after launch? Results might be deadly if the software is nondeterministic.

- *Predictability of the outcome*. The examples are industrial machines and manufacturing robots. They need to deliver the same result repeatedly without deviating from its programmed behavior; otherwise, the produced items may not be usable.
- *Closed system with fixed function*. The examples are the set-top box and GPS. The software running inside these things are doing one function (or a few functions) very well without errors.
- *Closed system with limited expansion*. The examples are the Mars Rover and home appliances. Once these things are delivered (departed Earth or sent home), the software inside should not need to deal with components coming and going in the systems, because a closed system can safely assume no expansion after certain points.
- *Fast boot time*. The examples are rearview cameras of cars, mobile phones, and home appliances. These things need to boot fast for either safety reasons or for a better user experience. It was very annoying when some early Blu-ray DVD players took a long time to be ready to play DVDs, because no consumer electronic devices should take more than a few seconds to be ready (at least this is not what consumers are used to).
- *Small footprint*. The examples are wearable devices, sensors, and software that needs to be certified. Big software frequently needs bigger storage devices, and they tend to be error-prone and difficult to certify.
- *Security and reliability*. The examples are point-of-sale (POS) terminals and in-vehicle-infotainment systems (IVI). In IVI, you probably don't want a software crash in the DVD player of the system to bring down the GPS, radio, and phone connection, and you do not want your credit card information stolen during checkout at a store; isolation of mission-critical and nonessential elements are, therefore, very important for some applications.
- *Fixed boot target*. The examples are Chromebook and IVI. These devices only have one boot target in mind (ChromeOS or embedded Linux); therefore, the firmware does not need to worry about having a general-purpose boot loader interface ready to boot any OS. A direct-boot interface can not only save code space, but also boots faster because shortcuts can be taken.

These are just the common requirements for embedded firmware. As you can see, they are very different from PC firmware. Still, each embedded system design can have its own requirements that may or may not come from the preceding list, or it may have multiples of these characteristics.

In addition to these requirements, embedded systems tend to have a longer life cycle than PCs; therefore, the cost of extensive and deep customization can be justified. Why is this important? Because a highly customized solution can deliberately violate some commonly known design principles to achieve the best and optimized outcome—a perfect product for the very niche market. The reusability and the ability to leverage a horizontal technology may suffer, and the sustaining effort may be expensive, but some products are never meant for service after being sold in the market. Therefore, a lot of firmware stacks, especially commercial off-the-shelf (COTS) products, focus more on ease of customization than anything else. Some software companies even have products designed for specialized markets with special certification requirements, such as avionics and military applications where safety and security are the main concerns.

When we examine the list of unique firmware requirements for embedded systems, we know some of these requirements can be achieved by specialized OS or RTOS. These specialized systems sometimes consist of a well-designed microkernel instead of a monolithic kernel; they also may have a customized firmware stack that is designed for ease of adding and subtracting components. Some of the requirements need the involvement of a virtual machine monitor (VMM); therefore, picking, integrating, customizing, and designing components in these firmware stacks can be an art sometimes.

Despite all of these unique requirements, there are still embedded designs that can be based on a familiar and ubiquitous PC architecture, such as industrial controllers, vending machines, POS, switches, and routers. Not necessarily because PC architecture is the best for these applications, but because the advantages of using a well-established and well-tested architecture sometimes outweigh the cost and disadvantages.

This is particularly true when developers want to reduce the design cost: they can simply buy off-the-shelf PC motherboards. It is easier to get ecosystem support, and may shorten the development cycle as well. The developers who have decided to leverage PC motherboards can still decide on whether or not to use PC BIOS. If the developers want to boot nothing but Linux and would like to replace the existing BIOS on the motherboard, they can certainly do that. There are many pros and cons to consider when picking the right firmware stack for your product.

Is a One-Size-Fits-All Solution Possible?

Ever since computer science emerged as a popular subject in the academic world, people have been trying to invent solutions that are as flexible as possible and to cover as much ground as possible. It is a challenge to come up with a smart design that works in all scenarios and meets all the requirements in the world. Unfortunately, no matter how hard we try, we have not succeeded in creating something that can deal with extreme requirements effectively. For example:

- Can we have a firmware stack that is optimized for a closed system while keeping the capability to boot a general-purpose OS and run general-purpose applications in an open system?
- Can we have a firmware stack that is not only optimized for a dedicated function, but also to support a computing environment that is capable of expanding into other functions?

- Can we have a firmware stack that is optimized for speed while maintaining the capability for exhaustive device discovery mechanisms and heuristic self-adjustment capability?
- Can we have a firmware stack that is optimized for size with the smartness to resolve device dependencies on the fly?
- Can a firmware stack be smart and intelligent, but at the same time deterministic and predictable?

Although it is not impossible, it is extremely hard to support extreme requirements using the same firmware stack. And even if this difficult design goal was achieved, the source code would probably be too complicated to read, or the build process would be too convoluted to comprehend. Regardless, a few people are trying to make a one-size-fits-all solution; one of the attempts is the UEFI firmware stack. Supported by Intel, HP, AMD, Dell, and other partners, there is still much research and development needed in making the UEFI firmware stack fit into fast and small applications beyond the PC paradigm. We shall see the results in a few years.

The motivation of promoting a one-size-fits-all solution could be based on the desire to reduce the support cost, but there are many alternatives available to customers. The following are some firmware solutions that typical customers of embedded systems are looking for.

Microkernel

Microkernel is frequently associated with RTOS, but it can be designed without real-time elements. The difference between a microkernel and a monolithic kernel is the responsibilities carried out inside the kernel. A monolithic kernel typically handles all the privileged tasks and system services in the kernel (such as file systems, interprocess communication, I/O, and a scheduler), while a microkernel does only basic process communication and I/O control, leaving the user space applications to provide services such as a pager, file systems, a virtual machine monitor, and so forth. Linux and Windows have monolithic kernels, and microkernel is frequently used in the operating systems designed for embedded applications in a closed system. The advantage of a microkernel is its size, flexibility, and the ability to handle mission-critical tasks in a tightly controlled manner. The main drawback of a microkernel is its lack of common features, and the fact that adding extra features can be costly and time-consuming.

Real-Time Operating System (RTOS)

For our daily computing needs, we seldom need RTOS because the tasks we typically carry out on a computer are either *transformational* or *interactive*.

Transformational tasks are the tasks we submit to a computer, and then the computer executes the task and gives us the result. When there is no task, there is no result. For example, if you calculate 2 + 3 on a soft calculator on the computer, a result of 5 is generated after you submit the task for calculation. When you submit a software source code for compilation, the software source code is the input, and compiled results (binaries, symbols, and executables) are the output after transformation.

Interactive tasks are those that occur when your computer responds to your requests, and vice versa; when there is no request, there is no action. When you type a character in a Word document or play a game on a computer, it is considered to be an interactive task between you and the computer. Every key you type on the keyboard becomes a task that the computer needs to handle, and then displays the text or image on the screen to give you the feedback you are looking for. The computer will also request that you take actions when dialog boxes are displayed, and keyboard commands are sometimes needed during games and with other interactive software.

The only time you will need to consider using RTOS is when you have a *reactive* system, which needs to respond to external stimuli in a timely manner. Although with many tasks the device handles could still be transformational, the actions are secondary to the main requirement. It is most important to respond to the stimuli and finish the tasks associated with the stimuli in a timely manner. Therefore, the major design goal is not about how fast or how many tasks the device can handle, but the ability to deal with a peak load when many stimuli are happening at the same time and to complete all the tasks invoked by the stimuli within the time specified by the design requirement.

The design of the kernel of a RTOS is mostly based on a preemptive scheduler, which responds to interrupts according to preset priority, and won't let go of the task until it is finished. In comparison, a GPOS (general-purpose operating system) typically uses a time-based, round-robin scheduler to ensure fairness. There are exceptions to both camps, but scheduler design is the main difference between the two categories.

There are many RTOS vendors owning a great deal of embedded markets, such as Wind Rivers, Green Hills, QNX, and so forth. There are also many open source and proprietary RTOS projects, such as FreeRTOS, eCos, and so forth. At the time this book is written, there are more than 160 RTOS offered for various platforms and usage cases under different licensing terms. Among them, at least 120 of them are still active. This means that the needs are so diversified and so customized that any niche market can spawn a RTOS of its kind. For this very reason, many big RTOS vendors also offer a few RTOS products for the specialized markets that they serve, most notably, avionics and military.

Legacy BIOS

Even though legacy BIOS is becoming less and less popular, it has transformed itself into a "payload" or a "compatibility module" for a very specific reason: to boot and run legacy software. There are still designs offered by many companies that require legacy BIOS to boot DOS, Linux, and 32-bit Windows (such as Vista, XP, and Windows 7). A legacy BIOS can be a stand-alone BIOS, or it can piggyback on a host firmware stack as a payload, such as SeaBIOS for coreboot, or it can be a module in a firmware stack to perform the compatibility boot, like compatibility support module (CSM) for UEFI firmware stack. Legacy BIOS is not necessarily old or obsolete, because some of the BIOS vendors and developers are still updating their code base to support newer hardware and CPUs. Most users of legacy BIOS are self-sufficient because updates are rare and support is scarce; a few IBV (independent BIOS vendors) are still doing business with legacy BIOS, but the revenue stream from legacy BIOS is quickly dwindling to a minuscule level compared to its mainstream business—UEFI-compatible products.

Implementations of the UEFI Framework

A firmware implementation of UEFI Framework is a firmware stack primarily for booting a 64-bit OS. It is modular at every stage of the boot process; therefore, supporting a new device can be added individually with flexibility. During the debug phase, a new device driver can be loaded and tested separately without reprogramming the whole flash device. There is also a Dependency Expression Grammar in the core to allow modules to be loaded in the order decided by a dependency expression, instead of an order that is statically programmed. There are many other clever designs in the UEFI firmware stack.

This UEFI firmware stack has been widely adopted by BIOS vendors, OEMs, and ODMs. It has become a de facto code base for booting 64-bit OS from the Microsoft and the Linux communities. The ARM community has also started to use the open source tianocore.org for 64-bit ARM support.

Open Source Firmware Stacks

Even though various Linux communities have been in existence for a long time, and they have established a very healthy and prosperous ecosystem, open source communities for firmware projects had not been as successful in terms of participants, number of platforms, and ecosystem support. It is changing because more and more silicon, hardware, and software companies, such as AMD, Google, Sage Electronic Engineering, and Intel, are getting involved.

The major difference between the communities for Linux and the communities for firmware is their dependency on hardware vendors and silicon vendors, and this factor also contributed to their progress in recent years. Most of the Linux development and collaboration work can be done without thinking too much of the hardware or firmware. The exception is the drivers, but even with drivers, they rely less on firmware, and more on direct hardware manipulation. In the past, the abstraction of hardware was mostly provided by a layer of firmware using tables and runtime services, but this model is changing to minimize the dependency on firmware beyond basic hardware abstraction tables. Thanks to the involvement of more hardware companies that are willing to contribute directly to the Linux development communities and are committed to write device drivers for their own devices, the overall performance and reliability of the systems has improved over the years. As a result, the user experience of Linux has also improved, and Linux has gained a lot of popularity from developers who did not treat Linux seriously in the past; like an upward spiral, when Linux gains popularity, more hardware companies are willing to be involved in contributing to the Linux source code. Even if there is a lag in the contribution from a key hardware vendor, it will not stall a Linux development project completely; missing a device driver may cause inconvenience when the corresponding device is not functioning correctly or is not accessible, but the other parts of the system will remain functional, and development work associated with the kernel or applications will not be completely blocked by the missing device support.

In contrast, lacking support from a key hardware or silicon vendor could be a death sentence for an open source firmware project. Even if a hardware initialization sequence can be reverse engineered, it may take a long time for developers to come up with an implementation to satisfy the needs of a community. The dependency on hardware vendors can determine the health of an open source firmware community; for example, U-Boot developers are having more success than coreboot because ARM and PowerPC architectures are considered more “friendly” and easier to program from the firmware perspective, and they have less dependencies on hardware vendors. That is also why board-level projects based on Intel Architecture are lagging: it has been hard to get support from Intel in the past, since Intel has not been very active in open source firmware until recently. With the introduction of Intel FSP and Google’s usage of coreboot in the Chromebook products, the situation is quickly changing, and Intel has become actively involved in contributing silicon code to these communities. Although Intel FSP is distributed in binary format as the book is written, Intel has also been releasing complete source code for Quark families. The debate on whether or not a binary solution is acceptable to an open source community remains, but for the purpose of helping these communities to quickly get the key components they need, regardless of which format the support code is distributed, it is a good starting point.

■ **Note** A quote from `http://www.coreboot.org/pipermail/coreboot/2014-November/078930.html`: “My personal take on this is that I would rather see coreboot run 75% free on a billion machines out there than 100% free on 1000 machines. Because that will ultimately leave the project in a better place of negotiating future ports and leaves the whole ecosystem with a higher percentage of free software total.” —by Stefan Rainaure in answering the debate of binary modules in coreboot.

As mentioned in the previous paragraph, the two better-known open source firmware communities are coreboot and U-Boot; tianocore or EDK II open source communities are also getting more attention due to ARM’s 64-bit movement. Among these open source firmware communities, coreboot, tianocore, and EDK II have been traditionally focusing on x86 platforms, and U-Boot has been used widely in the ARM and PowerPC designs. In later chapters, we will dive deeper into the coreboot and EDK II sides of development and practices.

This book will not discuss the different licensing agreements and terms offered by open source communities, but it is nonetheless a very important topic to study before you are fully committed to an open source project. Licensing agreement topics can usually be found in the open source community web sites.

Proprietary Firmware Stacks

Per the survey done by *EE Times* in 2013, about 24% of the companies surveyed are using in-house or homegrown software and firmware stacks for their embedded designs. We won’t be able to show an example of these firmware stacks in this book, but this is certainly a viable option, and the same principles and examples shown in this book will apply to these in-house designs, especially when industry standards, specifications, and sample code are easily obtainable for the designs that require them. Regardless what choice a company makes, the design principles and methodologies for developing a robust firmware stack to meet design requirements are common to all. You can go for an extremely customized proprietary solution, or an open source solution with reusability in mind, or even something in between. Hopefully, silicon companies such as Intel are no longer a roadblock for your development work. Read on and learn how you can take advantage of Intel FSP and Intel's full source code releases for Intel products for the IoT markets.

Make or Buy

Even though a PC ecosystem and an embedded ecosystem will not overlap in most cases, many Independent BIOS Vendors (IBVs) and Independent Software Vendors (ISVs) provide solutions and services to customers in both ecosystems. However, the business models in the two ecosystems are quite different: some IBVs and ISVs need to take a different approach when facing customers from different segments. Customers have a choice to buy the service from these IBVs and ISVs, or to develop a firmware stack on their own, and the decision to make or buy a firmware stack/solution can be complicated, and it depends on many questions, such as:

- *Do we have ongoing products that we need a firmware stack for?* If the products are ongoing and last for many years, either building a long-term relationship with a vendor, or working in-house with dedicated resource should work better. Buying a short-term service contract or changing vendors along the way could be problematic because the switching costs could be higher.
- *Is there a turnkey solution out there?* If the product has become a commodity and there are turnkey solutions available in the market, it might make sense to take advantage of an existing solution to save time and cost. In this case, it might make more sense to put as few resources as possible on the firmware creation, or pay someone to customize an existing solution.
- *Do we need source code to modify after the project is done?* If there is no engineering resource in the company, source code availability is not too much of an issue. But, some companies with a low-touch development model may have one or two engineers helping resolve customers' issues, and then they need to have access to the source code in order to do some modifications. If this is the case, the company can decide on hiring a vendor for everything from cradle-to-grave, including support, or pay for a contract that allows them to own the source code after it is done with all the features. Some vendors may not be willing to give the source code as part of the deal. Be aware.

- *Who can do it faster with the domain knowledge?* Many vendors will try to convince you that they can do it faster with lower cost, but you might want to check their domain knowledge in the field you intend them to work in. Sometimes a blocking issue is not necessarily a technical or programming issue, but rather that the domain knowledge is not easily obtainable. If you know a domain inside out, you might want to assess the risk involved in teaching your vendor to learn everything from you. After you teach the vendors to obtain the domain knowledge, they may use it to serve other customers, and these customers of theirs might happen to be your competitors.
- *Who needs to maintain the code after it is done?* Anyone who has developed or implemented firmware can attest that shipping a product is just the beginning of a headache, no matter how good your quality control process is. Who has the source code and who can provide support are the issues to think about early in the decision-making process. If you don't like going back to your vendors after the contract is over, you might want to pay for a long-term contract, or maintaining the code in-house from the beginning.
- *Are there special intellectual properties (IP) and differentiators in the firmware?* This is an important question to ask because any noncommodity product may have a few areas that you can differentiate on; once the differentiators are in place, they may be considered as IP. Do you feel comfortable sharing this knowledge with a vendor? We have seen a lot of cases where the vendors have become the main competitors, who turned around and outsold the original owner of the IP.
- *Related to the previous question, do we need to control our own destiny (to protect IP and differentiators)?* One way to protect IP and differentiators is to not share the code with anyone, but keep it to yourself; however, this does not mean that vendors cannot protect your secrets for you. In most cases, carefully designed non-disclosure agreements, once in place, can adequately protect your IP. But, you cannot protect IPs forever. The intangible experience the vendors accumulate during the projects cannot only serve you better over time, but could also potentially be used to serve others.
- *Can we get the attention from our vendors when we need their help during emergencies?* Most vendors are very reliable in dealing with emergencies, but if your company is small in size, or the volume of your product shipment is smaller than that of other customers of your vendor, you might want to make sure that you have a plan to deal with emergencies if you don't plan to do in-house design.

- *Which way is more cost effective: NRE (Non-recurring engineering), royalty, or internal resource?* This is a pure financial assessment that you have to make to determine whether you prefer to pay for a one-time contract or to save upfront costs by sharing your future profit. In-house engineering is an investment that you can make if it makes financial sense after comparing different business models.

The Advantages of Outsourcing

The following are the obvious advantages of outsourcing:

- IBVs and ISVs usually have early access to programming information and silicon samples before most customers do. Silicon vendors treat ecosystem vendors with a higher priority to give them more privileges so that they can help enable the industry.
- IBVs and ISVs have more experience because they frequently deal with many customers at once; if there is a tough blocking issue facing you, mostly likely they have dealt with it or are working on it— unless you are the first customer to discover it. You may be able to leverage what they have learned before you do.
- IBVs and ISVs have built their business model around services; therefore, there will be fewer surprises when issues arise. For example, they know how to get help and where to get help when encountering issues, and it could be hard for you to do the same effectively if you haven't dealt with issues regularly.
- Why do it in-house if the product is already a commodity? Outsourcing is a better option when there is no differentiation to be done for the product. Examples are tablets and phones: they have become such commodities that the hardware selection is limited and turnkey solutions are widely available. One cannot say that there is no innovation left, but the hardware and firmware are pretty much set in most cases.

The Disadvantages of Outsourcing

The following are the disadvantages of outsourcing:

- Lack of knowledge of what is implemented. If you are curious about what has been done for you by a third-party vendor, be prepared to get lost unless you also invest your own resources to monitor the design.
- Lack of capability to sustain a product. If you plan to sustain a product after a release done by a vendor, you may not have the knowledge and experience to carry it out effectively.

- Sometimes vendors don't share the source code or the knowledge they learned from making the product work. In this case, the next project is going to look new to you even if it shares common components.
- Sometimes vendors charge money for additional support, even though they picked up the knowledge while working on your product.

In-House Development

After analyzing the pros and cons of the outsourcing model, let's take a look at the following advantages of developing in-house:

- Despite many tedious technical challenges, the knowledge belongs to the company after the project is done.
- It is easier to maintain and sustain a product if you own the resulting source code and technology. Some vendors provide the source code to you after the project is done, but you will still need to understand the codebase if you are not familiar with it.
- It is easier to add differentiating features if the product is not yet a commodity. Although it is also possible through vendors, it is more reliable if it is done by internal resources, and it is easier to protect the IP in the product.

In recent years, the reason for not doing in-house development has been cost concerns. At the beginning of the twenty-first century, it was trendy for companies to eliminate their internal resources to rely on external vendors. Initially, it seemed to provide a cost advantage, but those who blindly followed this model were hurt not only financially, but also from a technology leadership perspective. Many "sold the farm" (gave away their know-how) just for the sake of doing outsourcing.

Whether or not to leverage an external vendor is sometimes a complicated decision, but not always; many companies have tried different models in their practices. Sometimes the short-term cost benefits of using external vendors bring long-term disadvantages; sometimes internal resources investment is wasted when insurmountable obstacles cannot be overcome in a reasonable time.

It is hard to tell if you can save money in the long run if you outsource, and it is also hard to tell if you will be better off financially if you do things in-house, because the trade-offs can go either way. You, being an expert of firmware in your company, need to make the recommendation carefully.

Summary

There is a lot of common-sense discussion related to “choices” in this chapter. The purpose is to help readers gain a general understanding of the choices they can make when facing a decision that has short-term and long-term implications on a product. In the following chapters, there are in-depth examples and programming steps and results that will equip you with practical knowledge to make intelligent design decisions for your products.