CHAPTER 3

Intel® Firmware Support Package (Intel® FSP)

"Simple can be harder than complex: you have to work hard to get your thinking clean to make it simple. But it's worth it in the end because once you get there, you can move mountains."

—Steve Jobs

Over the years, Intel has made several firmware products to support the embedded industry; most of them are either BIOS, UEFI Firmware, or something with the flavor of these two.

Intel enables its customers to design products around its chips by (like every other silicon vendor) making a Customer Reference Board (CRB) to showcase its new chips and its features. Intel, like other silicon vendors, expects customers to either copy or leverage what Intel has done with hardware, firmware, and software on the board, so that their designs can quickly go to the market for sale.

After three decades of perfecting the PC ecosystem and supply chain, the PC system development process for planning and designing chip and board features has become extremely efficient. The efficiency covers every link from top to bottom in terms of parts selection, vendor roadmap, software development, technology interception, distribution channels, marketing practices, to final sales online or in the brick-and-mortar shops. Everything is done in a well-tuned rhythm, and very few companies would miss a beat or have a hiccup in the process.

The efficiency of the PC industry can be a blessing as well as a curse. Due to the maturity of the ecosystem, the selection-choices of hardware components, software, and peripherals are gravitating toward a standard set of choices. These choices become more and more limited, and products are becoming more and more homogeneous even though the number of vendors is increasing. At the same time, the hardware, firmware, and software interfaces and architectures are also gravitating toward a few comprehensive standards, such as Wi-Fi, PCI, ACPI, UEFI, USB, Display Port, HDMI, Bluetooth, and so forth. Innovation and breakthroughs can still happen in these confined areas, but it is harder and harder for innovators to stand out in the crowd. Consumers may have a preference on brands, but there is hardly any reason to be loyal to a brand. The "cool" factor of Apple Mac products is one of the few remaining examples of strong product differentiation in an increasingly homogenized market.

Compared to the PC industry, the embedded market and the emerging IoT industry are more vibrant and versatile. They are not as mature as the PC industry, and are not as well-regulated by standards and specifications. However, the embedded market does not live in chaos. Each vertical segment of the embedded market still has its own regulations, standards, supply chain, and practices; they are just not as organized or homogeneous as the PC industry.

Therefore, trying to provide a horizontal solution across all segments and verticals is harder to achieve in the embedded market. Since the embedded market is about customization and diversification, and the life cycle of the devices is typically longer than a typical PC device, solutions for the industry need to be tailored to the needs of the designs for the vertical segments instead of the other way around.

This was the challenge that Intel was facing with firmware enablement strategies before Intel FSP was introduced. Since not all embedded market customers are leveraging PC architecture, they struggled with the firmware choices—or the lack of choices.

In 2010, Intel produced the Intel Boot Loader Development Kit (BLDK) to help developers customize an existing CRB and its BIOS. Intel BLDK was a good idea, but fell short of expectations, because even though it provided most of the source code for customers to customize, it was not very useful when customers wanted to alter their designs or use a completely different firmware stack for their boards.

After a few years of gathering feedback and prototyping, Intel announced the Intel Firmware Support Package in October 2012. It was not an answer to all the challenges, but it was a simple solution to support versatile needs of designs and choices. Intel FSP contains only the silicon initialization code that can be integrated into any firmware stack. Once integrated, developers can design the rest of the firmware stack for value-added features.

If you are confused by BIOS, BLDK, and FSP, let's compare these three offerings using automobile metaphors. The reference BIOS on a CRB is like an 8-cylinder, fully loaded SUV: powerful, impressive, and with all the features you can dream of. BLDK, on the other hand, is like a customizable VW Beetle with a lot of accessories and parts to alter the design and to customize its look. But, the Beetle is still a car in a fixed frame. If you plan to build a different kind of car, whether it is a race car, a sand buggy, or a military tank, it is not impossible, but it is hard to pick the parts you need from an SUV or a Beetle to build the car you want.

Using the same analogy, metaphorically, Intel FSP is like an engine, which is needed by every car. Obviously, in reality, cars of different models use different engines, but in the firmware case, the engine for initializing a silicon is identical whether you are building a server, a desktop, a laptop, a set-top box, or a wearable device. If a silicon vendor does it right, it is very possible to share the same silicon code for all market segments.

Intel FSP is the engine that provides the silicon initialization code for Intel chips. Each Intel silicon and its companion chip will have their own Intel FSP. There is no super FSP that encapsulates multiple silicon families, simply because it is not necessary. The goal is to keep FSP small enough that it can be utilized in the most compact and size-constrained environment. In Intel's case, the programming information is mostly documented in a BIOS Writer's Guide (BWG) or UEFI Firmware Writer's Guide, which are confidential documents that only privileged customers can obtain (after signing a non-disclosure agreement with Intel). If you are a developer who does not want to understand or debug Intel's code, and you have no interest in reading hundreds of pages of a BWG, you can simply take the Intel FSP, integrate it into your firmware stack, and let the magic of Intel FSP work for you.

Once executed, Intel FSP initializes the memory, programs the system address space, and initializes the input/output (I/O) controllers of the microprocessor and its companion chip. You will be able to run the next stage firmware stack to do whatever you like to do from this point on. Obviously, you still have a lot of work to do to initialize the rest of the platform, but the good news is that the rest of the system depends less on silicon vendors, but more on industry standards, publicly available specifications, and your own design recipes. There is even reference code and sample code to help you finish the features that are commonly available in a system.

The Intel FSP Philosophy

As discussed in Chapter 1, Intel believes that there are a lot of smart firmware engineers out there in the embedded industry. Intel knows that some of them are looking for turnkey solutions, with which they don't have to do anything besides turning knobs on a GUI-based tool. Intel also knows that some of them are looking for completely open source solutions that bring total freedom in creating customer-specific firmware stacks. Besides the 100% GUI-tool users and 100% open source users, there are many engineers who need to work with hybrid solutions because their code bases are either proprietary or have a lot of customized features to be added beyond standard features offered by the reference designs.

No doubt in people's mind, silicon vendors know their chips inside out because they design them; therefore, providing Intel FSP to encapsulate the chip initialization code that Intel knows the best makes sense. Developers who choose to leverage Intel FSP will have a better starting point with a new Intel chip than relying on a self-learning process from reading programming manuals.

Intel FSP does not have features that are typically available in reference BIOS from Intel (such as secure boot, PCI Bus enumeration, power management, etc.). The reason why these features are not included is because Intel does not want to burden everyone with code they may not need. Intel wants to keep the FSP footprint as small as possible. For this reason, a developer may feel lost if he or she has been depending upon Intel to deliver the additional features in the reference BIOS, but the good news is that these features have publicly available specifications, standards, and even sample code to follow or copy. Obviously, the bad news is that the developer will have to somehow implement these features in the firmware stack if they need the features. If the developer is implementing them from scratch, it could be hard, but this is the price the developer has to pay when switching away from the reference BIOS model. Whichever way the developer chooses to acquire the features (do it himself/herself or pay someone to do it), the developer has a better starting point with Intel FSP than with a reference BIOS because of the additional freedom provided by FSP. Hopefully, as the ecosystem for supporting Intel FSP grows, there will be more and more ISVs involved to provide firmware solutions different than BIOS. Additionally, once the code for these peripheral features is done, most of the code is reusable and can last for a long time. For example, PCI bus enumeration code, even though is complicated, does not change between generations of chips. Most of the power management and secure boot features do not need an overhaul once the foundation is built.

It is really all about additional freedom provided by Intel FSP; it is not an answer to all the firmware needs associated with Intel Architecture. If you have a design that heavily leverages Intel CRB and standard features, you are probably better off with a standard BIOS solution; it does not likely make sense to start a new firmware stack to replace your existing BIOS. However, if you are starting from scratch, or if your design is significantly different from a traditional PC, you might want to consider the options involving Intel FSP. Intel FSP addresses a niche market where the developer doesn't use PC architecture or when the developer does not plan to use BIOS. Therefore, don't jump onto the Intel FSP bandwagon until you know your ultimate goal.

What Is in Intel FSP?

Later in this book, we will show you where to download Intel FSP. At the FSP download site on the Web, you will find self-extracting executable packages (in `.exe` format for Windows or in `.tgz` files if you are exploring Linux options).

After you download the file in `.exe` or `.tgz` format for your favorite OS, you can extract the contents from the file by executing the `.exe` file or by using your favorite Linux unpacking utility. And, you can find a couple of items in your expanded folder:

- A subfolder called Documentation, which contains an integration guide and release notes.
- A subfolder called FSP, which contains an FSP binary file (with an `.fd` file extension), a Boot Setting File (BSF, with a `.bsf` file extension), and two subfolders with sample header files and source code.
- A subfolder called Graphics. This folder is optional. When it exists, it contains graphics components that you might need, such as a VGA BIOS image.
- A subfolder called Microcode, which contains the microcode patches available at the time of the release of FSP. The microcode patches can be updated independently of FSP, as well as when you must download the latest ones to make sure that they match with the microprocessor used on your board.

There are additional files, such as the licensing agreement, a ReadMe (describing the contents of FSP), and so forth. These are not important to the discussion of this book. There is an important tool available—the Binary Configuration Tool—on the same download page, but it is a separate download file. We will discuss this tool later in the book.

We will explain each item in more detail, but a hard-core open source software engineer may already have a question about why Intel ships only a binary file, not the source code. Currently, there are a couple of reasons. Intel may change the design and deliverables in the future.

- Intel FSP still has some IP-protected code inside; therefore, until this code can be properly protected or cleared for protection, Intel does not have a near-term plan to release it in an open source repository. However, you may have discovered that Intel has already released 100% of the source code for the Intel® Quark™ SoC X1000 Series; therefore, Intel is gradually removing the barrier of IP protection. It may take a few years to see Intel completely open all the source code at this level.

- Intel wants to eventually move basic silicon initialization code inside the silicon one day; therefore, a binary file is isolated enough that it can pave the road for future silicon inclusion. Many silicon vendors have already included a Boot ROM inside their SoC. Since the code inside the Boot ROM is for the purpose of initializing the SoC, it should not be a concern if silicon initialization code is isolated and protected somehow. Intel FSP does offer a way to customize its internal configuration with the BCT tool; thus it further eliminates the need to change what is inside Intel FSP.

- Many vendors release binary files to abstract silicon and hardware code. This practice has been in existence for many decades; for example, the Option ROM on plug-in cards contains a binary code that initializes the hardware on the card. VGA BIOS still exists in the form of Option ROM, even though there is no longer a plug-in card. This abstraction has been working out nicely in the PC industry because it provides a simple interface and isolation so that the cards can be sold separately from the motherboards.

- Being a binary, it can be interfaced with a firmware stack without being dynamically or statically linked into the host firmware. In this way, it can help eliminate the concerns of open source codebase using General Public License (GPL) when integrating. Because this book does not provide legal advice and there are a few different versions of GPL, you need to double-check with your legal department to decide what is OK and what is not OK.

That said, Intel does realize that many hardware and firmware engineers need source code to bring up or power-on a brand-new board in order to debug a problem when the board is not functioning after the power is applied. Intel offers source code to customers under a special software license agreement (SLA). It is beyond the scope of this book to discuss how to obtain source code from Intel; therefore, you should contact Intel's sales or field engineers to ask for details.

As previously discussed, even though Intel FSP is a binary file, it needs a provision to customize its internal states and features; therefore, it has reserved a data region inside the binary for customization. The data area also contains a couple of platform-specific parameters that Intel FSP would otherwise have no knowledge about, or would initialize the board with default values. The Boot Setting File (BSF) plays an important role for this purpose. It is basically a text file that contains firmware internal settings associated with the board; for example, the SMBUS (System Management Bus) address of a SPD (Serial Presence Detect) ROM on a DIMM (Dual Inline Memory Module) is one of the data in the BSF.

```
$gPlatformFspPkgTokenSpaceGuid_PcdMrcInitSPDAddr1  1 byte $_DEFAULT_ = 0xA0
$gPlatformFspPkgTokenSpaceGuid_PcdMrcInitSPDAddr2  1 byte $_DEFAULT_ = 0xA2
```

A sample BSF can be found in Appendix A.

The data in BSF is represented in a GUI-based tool, which allows developers to visualize the meaning of each component in BSF. With the GUI and BSF, it is collectively called a Binary Configuration Tool (BCT). There are three versions of BCT: one runs under Windows, one runs under Linux, and the third is a command-line option under Linux. Figure 3-1 shows a BCT tool for Windows.

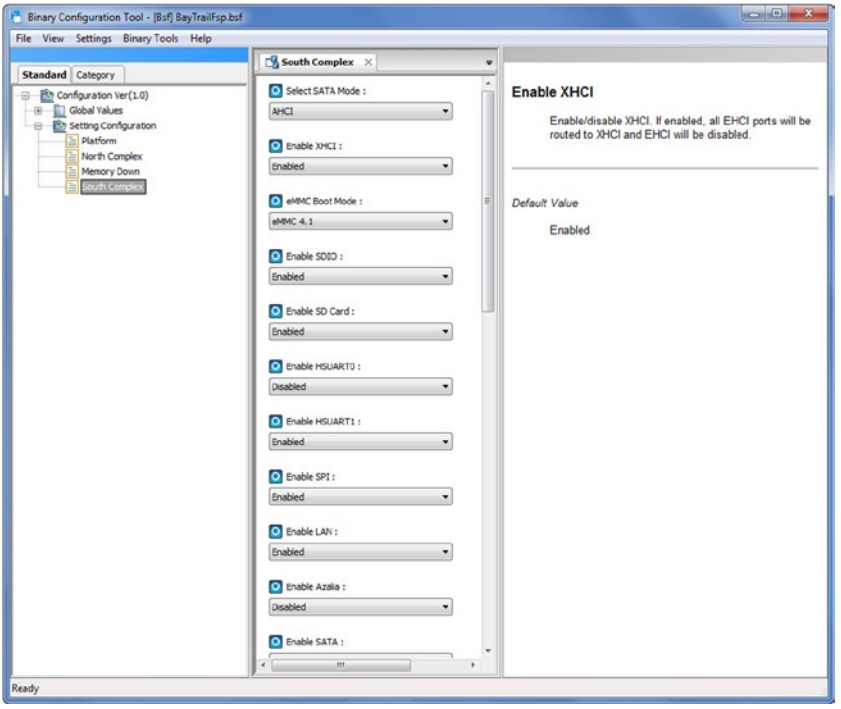

Figure 3-1. *Binary Configuration Tool (BCT) GUI*

The last part of Intel FSP is the header files and sample code. These files are for developers to include in their firmware stack to develop the interface code with Intel FSP; therefore, developers can include these files in the firmware stack or re-create the files based on these files. For example, `fsp.h` under the `include` directory, looks like this:

```
#include <stdint.h>
#include "fsptypes.h"
#include "fspfv.h"
#include "fspffs.h"
#include "fsphob.h"
#include "fspapi.h"
#include "fspplatform.h"
#include "fspinfoheader.h"
#include "fspvpd.h"

#define FSP_HOB_RESOURCE_OWNER_FSP_GUID \
{ 0x69a79759, 0x1373, 0x4367, { 0xa6, 0xc4, 0xc7, 0xf5, 0x9e, 0xfd, 0x98, 0x6e } }
#define FSP_NON_VOLATILE_STORAGE_HOB_GUID \
{ 0x721acf02, 0x4d77, 0x4c2a, { 0xb3, 0xdc, 0x27, 0xb, 0x7b, 0xa9, 0xe4, 0xb0 } }
#define FSP_HOB_RESOURCE_OWNER_TSEG_GUID \
{ 0xd038747c, 0xd00c, 0x4980, { 0xb3, 0x19, 0x49, 0x01, 0x99, 0xa4, 0x7d, 0x55 } }
#define FSP_HOB_RESOURCE_OWNER_GRAPHICS_GUID \
{ 0x9c7c3aa7, 0x5332, 0x4917, { 0x82, 0xb9, 0x56, 0xa5, 0xf3, 0xe6, 0x2a, 0x07 } }
#define FSP_BOOTLOADER_TEMPORARY_MEMORY_HOB_GUID \
{ 0xbbcff46c, 0xc8d3, 0x4113, { 0x89, 0x85, 0xb9, 0xd4, 0xf3, 0xb3, 0xf6, 0x4e } }

//
// 0x21 - 0xf..f are reserved.
//
#define BOOT_WITH_FULL_CONFIGURATION       0x00
#define BOOT_ON_S3_RESUME                  0x11
```

Intel FSP Binary Format

Since Intel FSP is a binary file, it needs to be organized in a standard way that the data inside can be easily found and retrieved. What are things you need to find inside the Intel FSP? The two most important interface points are the application programming interface (API) and the configuration region.

Intel FSP chose to follow UEFI's Firmware Volume (FV) layout format. Don't be alarmed if you are not familiar with Platform Initialization (PI), UEFI, or FV. FV is just a layout format that dictates how the data inside is organized. If you want to learn more about FV, please find more information in the Appendix or look it up on the UEFI web site. The diagram in Figure 3-2 shows the FV layout format.

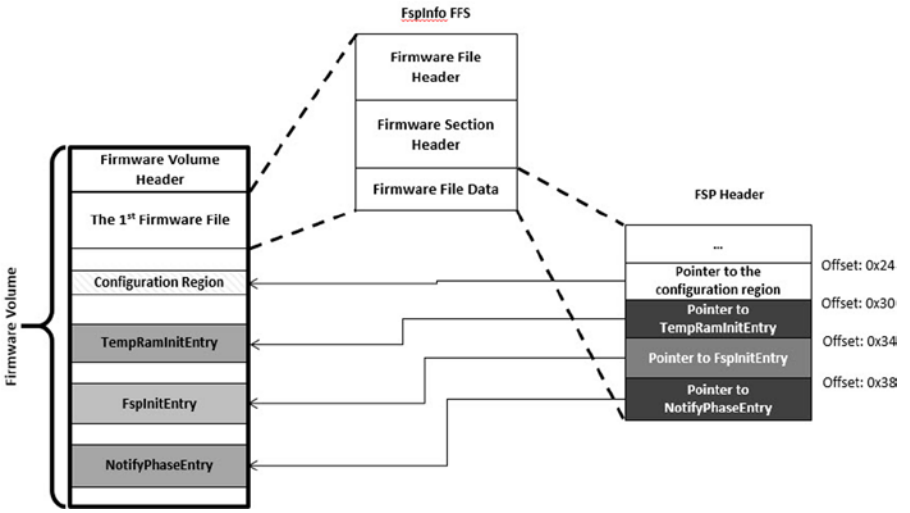

Figure 3-2. *Intel FSP binary Firmware Volume (FV) layout*

Figure 3-2 shows an FV where Intel FSP resides, but Intel FSP can be split into multiple FVs due to the needs of secure boot and other features. For example, there can be one FV that contains part of the code that needs to be certified and signed, and there can be another part of the code that is customizable for developers; therefore, the latter can exist in a different FV so that the second part can be certified or signed separately from the first one. The other case that warrants the separation of the FVs is where one part can be masked after production and the other part can be updated in the field. Intel FSP provides the flexibility to be used in different ways.

A few key ingredients are in the Intel FSP binary:

- FSP information header region for configuration data
- Offsets pointing to API entry points and the configuration data region
- The APIs themselves

The FSP information header provides a basic FSP image signature, a base, size information, and the offsets pointing to the API entry points and the configuration region.

The following shows the FSP information header:

```
typedef struct  {

        UINT32  Signature;      // Off 0x94
        UINT32  HeaderLength;
        UINT8   Reserved1[3];
        UINT8   HeaderRevision;
        UINT32  ImageRevision;
```

```
        CHAR8   ImageId[8];      // Off 0xA4
        UINT32  ImageSize;
        UINT32  ImageBase;

        UINT32  ImageAttribute;  // Off 0xB4
        UINT32  CfgRegionOffset;
        UINT32  CfgRegionSize;
        UINT32  ApiEntryNum;

        UINT32  NemInitEntry;    // Off 0xC4
        UINT32  FspInitEntry;
        UINT32  NotifyPhaseEntry;
        UINT32  Reserved2;

} FSP_INFO_HEADER;
```

As you can see, offset numbers are provided here in the sample header structure: 148 bytes (0x94 is the hexadecimal number) from the top of the file is the starting point of the FSP_INFO_HEADER; 164 bytes (0xA4 is the hexadecimal number) from the top is the identification string of the FSP binary; 180 bytes (0xB4 is the hexadecimal number) from the top is the location that contains the offset to the configuration region; and 196 bytes (0xC4 is the hexadecimal number) from the top is where the three pointers to the three APIs are located. Later on, we will talk about why the offset values are important to know. These numbers may change based on the platform. Please check the documentation that comes with each Intel FSP release—particularly the Intel FSP Integration Guide—to obtain the latest offset values.

The configuration data region stores platform-specific information; some parameters are customizable during runtime by the firmware stack, and some are fixed once configured—thus static once they are manipulated by the BCT. Details will be discussed later. The pointers to the APIs and the APIs themselves are part of the FV, and they will be discussed in later sections as well.

Sample Boot Flow

The boot process starts from the Reset Vector after power is applied. In modern computers, many things have already happened before the first fetch of the opcode located at the Reset Vector of the CPU, such as microcontroller firmware and management engine firmware. In a secure boot implementation, the management engine in the system can hold the reset signal to the CPU until it deems that the firmware running the CPU is trustworthy and safe to run.

As shown in Figure 3-3, once the firmware stack is allowed to execute, the CPU will fetch its first instruction from the Reset Vector, which is located at top of the 4-gigabyte physical memory space, minus 16 bytes (or FFFFFFF0, in hexadecimal value). This address is hardwired in the Intel microprocessors during reset, and the location should fall inside the Flash ROM if you have designed your board correctly. In the design of future microprocessors, this physical address of FFFFFFF0 may change to allow more flexibility.

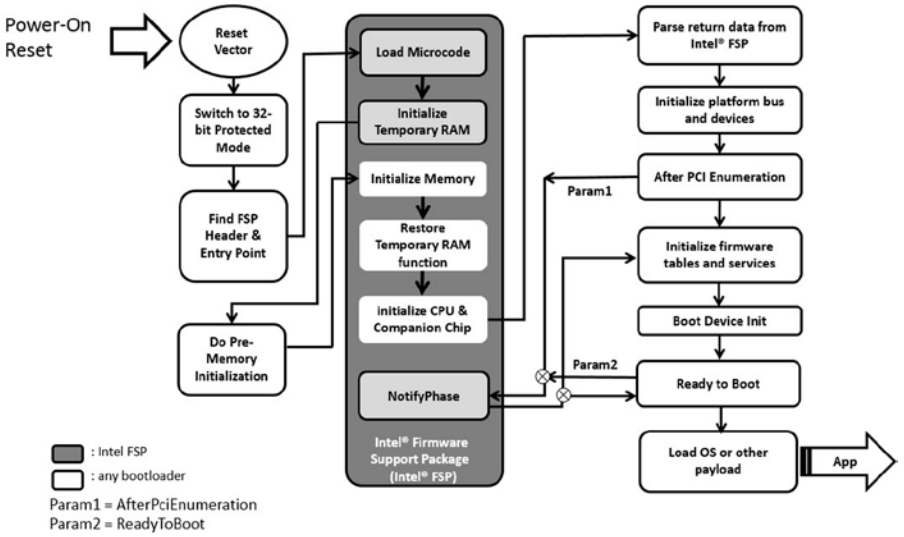

Figure 3-3. *A sample boot flow involving Intel FSP*

At this point, the firmware stack inside the Flash ROM will begin to execute; the first thing it needs to do is quickly exit the "reset mode" and enter a more permanent 32-bit Protected Flat Mode. This "32-bit Protected Flat Mode" is a loaded term, but it is basically a mode that allows you to access 4 gigabytes of "flat" physical address space with the existing segment register and selector settings. After entering this 32-bit Protected Flat mode, the firmware stack has a powerful execution environment to access code and memory mapped I/O anywhere within the 4-gigabyte range.

However, even with the power to access the 4-gigabyte address range, you still cannot do too much without memory and I/O devices; therefore, the firmware stack should be initializing the memory controller and the companion I/O controllers as soon as possible so that it can actually run its code inside the faster DRAM (instead of the slow Flash ROM) and access I/O devices in the system. This is where you will run into the first headache in dealing with a new silicon from a silicon vendor. In Intel's case, you may use Intel's Memory Reference Code (MRC) directly if you can get one, or you can download Intel FSP that contains the MRC.

Since MRC is very complicated, we recommend you not to touch MRC. If you understand DDR (Double Data Rate) memory technology, you know that MRC contains the memory training algorithm to try to find the sweet spot of the operating speed, signal strength, and other parameters for the DRAMs in the system; whatever algorithm Intel (or any other silicon vendor) has developed inside MRC, it is robust, optimized, and validated. You should not touch the code if you don't have a reason to do so. However, you might want to customize your design; for example, bypassing memory training when you have a soldered-down or fixed memory configuration. But this so-called "Fast Boot" path is also provided inside MRC; it is typically an option provided by the Intel FSP BCT configuration tool. You can achieve this "Fast Boot" by simply turning a bit on and off

inside the tool. FSP will automatically apply the fast boot option when a valid NVS (Non-Volatile Storage) data region is provided by the firmware stack. The NVS data region contains the memory parameters that can be used repeatedly by Intel FSP.

■ **Note** DDR technology is quite delicate; the training algorithm exists for a reason. The "sweet spot" of operation may change over time due to temperature, aging, and other factors in the environment; therefore, the parameters stored in the NVS may not work all the time unless careful error margins are built in, and a failover mechanism is provided when the parameters in NVS no longer work. The memory training may need to be executed again to acquire new parameters when old parameters do not function.

As shown in Figure 3-3, Intel FSP does more than just memory initialization; you should use it rather than dealing with MRC directly. Intel FSP also loads microcode patches and initializes companion I/O controllers for you. In addition, it sets up a small amount of temporary memory using cache or SRAM before DRAM is initialized, so that you can execute code more efficiently before the system memory is fully initialized.

Therefore, once the host firmware stack transitions to the 32-bit Protected Flat Mode, it can look for the first Intel FSP entry point and jump into it. The first thing that Intel FSP does inside the first FSP API is load microcode patches, and then initialize a small region of temporary memory. Once the temporary memory is set up, Intel FSP returns control back to the host firmware stack. Why? Because the next step Intel FSP is about to do is to initialize DDR memory, where a memory training algorithm will be conducted, and it can take 100 milliseconds to 300 milliseconds to finish, depending upon the board layout, the amount of memory, and which DDR technology is used. If you have something important and urgent or devices that can be turned on early to allow the electronics to be stabilized before the OS starts (e.g., the LED backlight, HDD spin-up, etc.), you can do it before memory is initialized. Some firmware stacks take this opportunity to verify the integrity of the next stage of boot code to ensure that the code running next is secure and trust worthy.

Once pre-memory initialization work is done by the host firmware, it can once again call into the second entry point of Intel FSP. This time, Intel FSP is going to finish the memory initialization and the companion chip initialization altogether. Once this is done, upon return, the host firmware will be able to see 4 gigabytes of memory available for them to use. There are many techniques in firmware to look beyond 4 gigabytes, such as memory hoisting (moving memory above 4 gigabyte to fill the memory holes created by Memory Mapped I/O); but the best way to access memory above 4 gigabyte is to transition to 64-bit long mode, although most embedded firmware stacks don't do that unless they are designed to boot a 64-bit OS. Some firmware stacks will remain in 32-bit mode until the last minute before booting a 64-bit OS; it is up to the firmware developer to decide when to switch to 64-bit mode if it is necessary.

At this point, Intel FSP has done its job and leaves the rest of the platform initialization work to the host firmware. The following chapters will be walking you through two firmware stacks as examples for carrying out the platform initialization work.

There is another entry point of Intel FSP that we have not talked about—NotifyPhase. NotifyPhase is an entry point for the host firmware to call back to Intel FSP after PCI bus enumeration and before booting an OS, because there are things to be adjusted, locked down, and cleaned up; as an example, the TSEG register will be locked so that no one can mess it up after booting an OS.

Once NotifyPhase returns the control to the host firmware, the firmware is ready to transition control to a payload or an interface to boot an OS.

Is this the only way to go through the boot process? Of course not; this is a typical boot process for a system based on PC architecture. For a system without PC bus or graphics, or closed without expansion, the boot process can be different or simpler, and Intel FSP can still be part of the boot process to initialize silicon features.

One thing should be very clear to you now: Intel FSP is not a firmware stack itself. It needs to be integrated into one. And, Intel FSP contains only the basic silicon initialization code; more sophisticated features need to be carried out by the host firmware stack; for example, power management, bus enumeration, ACPI, and so forth.

A rough estimation of the Intel FSP size is between 100 and 300 kilobytes, and it can finish all the tasks within 100 to 300 milliseconds. These size and performance estimations are all tied to the CPU type, and the size and performance of MRC dominate the overall estimation because it is the biggest and slowest code inside Intel FSP.

The boot flow shown in Figure 3-3 may change in the future, when new requirements for Intel FSP show up; for example, Intel FSP may provide more hooks to allow host firmware to carry out actions in between various Initialization tasks.

Locating the Entries of Intel FSP

Since Intel FSP is a binary image, the host firmware stack needs to find the image of Intel FSP soon after it starts running from the reset vector. Once Intel FSP is found, it needs to find the entry points pointing to the three APIs that we mentioned in the section about Boot Flow. There are two ways to find the pointers to these APIs.

The Hard Way to Find Intel FSP APIs: Use Data Structure

A developer can certainly follow the predefined and standard FV data structure to parse the data in an FV binary. All you have to do is trace through the data structure and look for header and data regions that contain relevant FSP data.

The first step is to find the base address of the FV where Intel FSP resides. This is where you physically put the FSP binary image. Many Intel FSP binary images are designed to exist at a physical address, such as 0xFFFF8000. A developer can customize the location by rebasing Intel FSP to a different location by running the BCT, which will be discussed later. As you read the data out of the FV and match it against the data structure of EFI_FIRMWARE_VOLUME_HEADER, defined in a standard UEFI header file for FV, you can basically follow the hierarchy of the data structure and look for the data you need.

```
typedef struct {
        UINT8 ZeroVector[16];
        EFI_GUID FileSystemGuid;
        UINT64 FvLength;
        UINT32 Signature;
        EFI_FVB_ATTRIBUTES_2 Attributes;
        UINT16 HeaderLength;
        UINT16 Checksum;
        UINT16 ExtHeaderOffset;
        UINT8 Reserved[1];
        UINT8 Revision;
        EFI_FV_BLOCK_MAP BlockMap[];
} EFI_FIRMWARE_VOLUME_HEADER;
```

FSP_INFORMATION_HEADER is a firmware file that is placed as the first firmware file within the FV. All firmware files have a GUID that can be used to identify the files, including the FSP header file. The FSP header firmware file GUID is defined as 912740BE-2284-4734-B971-84B027353F0C.

The host firmware stack can find the offset of the FSP header within the FSP binary by following these steps:

1. Use EFI_FIRMWARE_VOLUME_HEADER to parse the FSP FV header and skip the standard and extended FV header.
2. The EFI_FFS_FILE_HEADER with the FSP_FFS_INFORMATION_FILE_GUID is located at the 8-byte aligned offset following the FV header.
3. The EFI_RAW_SECTION header follows the FFS File Header.
4. Immediately following the EFI_RAW_SECTION header is the raw data. The format of this data is defined in the FSP_INFORMATION_HEADER structure.

The following is a sample code snippet that does the previous steps in a stackless environment. Since the code will be executed in a stackless environment, assembly code is preferred. In this example, we use the C language for easier readability.

```
//
// Validate FV signature _FVH
//
        if (((EFI_FIRMWARE_VOLUME_HEADER *)ptr)-> Signature != 0x4856465F) {
              ptr = 0;
               goto NotFound;
        }
//
// Add the Ext Header size to the Ext Header base to go to the
// end of FV header
```

```
//
        ptr += ((EFI_FIRMWARE_VOLUME_HEADER *)ptr)->ExtHeaderOffset;
        ptr += ((EFI_FIRMWARE_VOLUME_EXT_HEADER *)ptr)->ExtHeaderSize;
//
// Align the end of FV header address to 8 bytes
//
        ptr = (UINT8 *)(((UINT32)ptr + 7) & 0xFFFFFFF8);
//
// Now ptr is pointing to thr FFS Header. Verify if the GUID
// matches the FSP_INFO_HEADER GUID
//
         if ( (((UINT32 *)&(((EFI_FFS_FILE_HEADER *)ptr)->Name))[0] !=
0x912740BE) ||
         (((UINT32 *)&(((EFI_FFS_FILE_HEADER *)ptr)->Name))[1] != 0x47342284) ||
         (((UINT32 *)&(((EFI_FFS_FILE_HEADER *)ptr)->Name))[2] != 0xB08471B9) ||
         (((UINT32 *)&(((EFI_FFS_FILE_HEADER *)ptr)->Name))[3] != 0x0C3F3527) ) {
                ptr = 0;
                goto NotFound;
         }
//
// Add the FFS Header size to the base to find the Raw section
// Header
//
        ptr += sizeof(EFI_FFS_FILE_HEADER);
        if (((EFI_RAW_SECTION *)ptr)->Type != EFI_SECTION_RAW) {
               ptr = 0;
                goto NotFound;
        }
//
// Add the Raw Header size to the base to find the FSP INFO
// Header
//
         ptr += sizeof(EFI_RAW_SECTION);
NotFound:
        __asm__ __volatile__ ("ret");
```

All of these data structures are provided in FSP header files in the package; therefore, you can develop a generic solution to locate FSP APIs.

The Easy Way to Find FSP APIs: Use Hard-Coded Constants

If you are more concerned more about saving a few bytes here and there, or a few nanoseconds or microseconds here and there, and you are not worrying about changing code for every platform you build, you may consider using hard-coded constants to find the FSP APIs.

For example, the offset constant OFFSET_OF(FSP_INFO_HEADER) is defined as 0x94 for BayTrail platforms. It is the offset from the beginning of the FSP binary. The next important offset constant is OFFSET_OF(FSP_BASE), which is defined as 0x1C (28 bytes from the beginning of FSP_INFO_HEADER). The FSP_BASE must match the location where you put the FSP image in Flash. Then we have three pointers pointing to Intel FSP's three APIs located at offset constants from the beginning of FSP_INFO_HEADER: 0x30, 0x34, and 0x38, respectively. They are 4 bytes apart because these offsets are 32 bits long.

Using these offset constants, you may find the following:

- The first API (TempRAMInit) entry point at the physical location of (FSP_BASE + *(UINT32 *)(FSP_BASE + OFFSET_OF(FSP_INFO_HEADER) + 0x30))
- The second API (FspInit) entry point at the physical location of (FSP_BASE + *(UINT32 *)(FSP_BASE + OFFSET_OF(FSP_INFO_HEADER) + 0x34))
- The last API (NotifyPhase) entry point at the physical location of (FSP_BASE + *(UINT32 *)(FSP_BASE + OFFSET_OF(FSP_INFO_HEADER) + 0x38))

Programming Interface: The APIs of Intel FSP

Now you know how to find the three APIs inside of Intel FSP. It is time to understand what they are and how you should interface with them.

TempRamInit

As described in the "Sample Boot Flow" section, this FSP API will load the microcode patches, enable the code cache region specified by the host firmware, and set up a temporary stack to be used until the main memory is initialized. There are a couple of input parameters for passing in: the microcode patch base address and its size, and the host firmware code region base address and its size. The microcode patch mechanism built in FSP will try to load the correct microcode patches for the silicon by matching its CPUID and the ID of the patches; if no matching one is found, the API will return an error code. This is a typical error that the developer will face when he or she is not familiar with Intel microprocessors and the microcode patching mechanism.

The other input parameters are firmware code region base and size. They are used to enable the code cache to speed up the execution of the code in Flash. Here is the data structure of the input parameters:

```
typedef struct {
        UINT32 MicrocodeRegionBase,
        UINT32 MicrocodeRegionLength,
        UINT32 CodeRegionBase,
        UINT32 CodeRegionLength
} FSP_TEMP_RAM_INIT_PARAMS;
```

Since the host firmware is supposed to blindly jump into TempRamInit after finding its address, how does Intel FSP know where to return to after TempRamInit is done? A trick called "ROM-based stack" is used here. It is basically a trick to point the ESP (Stack Pointer) register to a location in ROM where a return address is stored. When the TempRamInit code is done, all it has to do is execute a "ret" instruction, which will pop the return address from the location pointed by the ESP register, and the Instruction Pointer will point to the next instruction that the host firmware specifies. Besides the return address, the input parameters will also be stored there so that TempRamInit can simply look at the "stack" and retrieve the input parameter as if it is running a subroutine written in the C language.

The following is an example of what the ROM-based stack looks like. ESP will be loaded with the address `tempRamInitStack`, where it contains the return address as `temp_RamInit_done`, and also the pointer to the input parameters—`tempRamInitParams`. The microcode patches–based address and length are passed in here.

```
tempRamInitParams:
        .long _ucode_base # Microcode base address
        .long _ucode_size # Microcode size
        .long 0xfff00000 # Code Region Base
        .long 0x00100000 # Code Region Length
tempRamInitStack:
        .long temp_RamInit_done # return address
        .long tempRamInitParams # pointer to parameters
```

This API should be called only once after the system comes out of the reset, and it must be called before any other FSP APIs. The system needs to go through a reset cycle before this API can be called again; otherwise, unexpected results may occur. The sample implementation of `find_fsp_info_header` and `tempRamInit` are listed next as an example:

```
# prepare to find FSP header
        lea findFspHeaderStack, %esp
        lea _fsp_rom_start, %eax
        jmp find_fsp_info_header
findFspHeaderDone:
        mov %eax, %ebp # save fsp header address in ebp
        mov 0x30(%ebp), %eax # TempRamInit offset in the header
        add 0x1c(%ebp), %eax # add FSP base to get the API address
        lea tempRamInitStack, %esp # initialize to a rom stack
#
# call FSP PEI to setup temporary Stack
#
        jmp *%eax
        temp_RamInit_done:
        addl $4, %esp
        cmp $0, %eax
        jz continue
```

```
#
# TempRamInit failed, dead loop
#
        jmp .
continue:
#
# Save FSP_INFO_HEADER in ebx
#
        mov %ebp, %ebx
#
# setup bootloader stack
# ecx: stack base
# edx: stack top
#
        lea -4(%edx), %esp
#
# call C based early_init to initialize meomry and chipset.
# pass the FSP INFO Header address as a paramater
#
        push %ebx
        call early_init
#
# should never return here
#
        jmp .
.align 4
findFspHeaderStack:
        .long findFspHeaderDone
tempRamInitParams:
        .long _ucode_base # Microcode base address
        .long _ucode_size # Microcode size
        .long 0xfff00000  # Code Region Base
        .long 0x00100000  # Code Region Length
tempRamInitStack:
        .long temp_RamInit_done # return address
        .long tempRamInitParams # pointer to parameters
```

If this function is successful, ECX and EDX registers will be returned to point to a temporary but writeable memory range available to the host firmware with FSP_SUCCESS stored in register EAX. Register ECX points to the start of this temporary memory range and EDX points to the end of the range. At this point, the host firmware is free to use the whole range described. Typically, the host firmware can reload the ESP register to point to the end of this returned range so that it can be used as a standard stack now, and the C-style function call can now be used; in other words, the programming language can be switch to the C language after this API is called. Since there is only limited space, the host firmware should not try to do crazy things using this memory range. Stack is probably the only function that the host firmware should consider using until the main memory is initialized by the next API.

Developers should also be aware that this returned range is just a subregion of the whole temporary memory initialized by the TempRamInit function; Intel FSP maintains and consumes the remaining temporary memory. It is important for the host firmware not to access the temporary memory beyond the returned boundary.

FspInitEntry

As shown in the sample boot flow diagram in the previous section, once Intel FSP hands over control back to the host firmware after TempRamInit is executed, the host firmware can choose to do a few things before calling the next API—FspInitEntry. Since this upcoming API initializes the memory, the CPU, and the companion chips, it may take a few hundred milliseconds before the host firmware gains control again; therefore, the host firmware can utilize this opportunity to carry out some pre-memory initialization work; for example, turn on the LED backlight, spin up the hard drive, or turn on other hardware components that need time to stabilize their electronic and mechanical states. The host firmware can also choose to verify components for secure boot purposes before continuing.

Since FspInitEntry deals with CPU, memory controller, and companion chips, it is highly dependent on the silicon it is associated with. Therefore, even though the input parameter is consistent among all Intel FSP as a pointer to a data structure, the contents of the data structure will be defined differently by each FSP release. They will be documented in the Integration Guide. The prototype of the input parameter is as follows:

```
typedef
FSP_STATUS
(FSPAPI *FSP_FSP_INIT) (
        INOUT FSP_INIT_PARAMS *FspInitParamPtr
);

And, the data structure is shown as the following:

typedef struct {
        VOID *NvsBufferPtr;
        VOID *RtBufferPtr;
        CONTINUATION_PROC ContinuationFunc;
} FSP_INIT_PARAMS;
```

Within the data structure, *NvsBufferPtr* points to the data buffer with data that needs to be stored into and retrieved from a non-volatile storage device. In the very beginning—the first time the host firmware calls the API—this parameter is NULL, and Intel FSP will return the data in the Hand-Off Block (HOB) after this API is executed. The data can then be stored in the non-volatile storage device for later usage by the host firmware. For subsequent boots, the data can be passed in by the host firmware to the FSP to handle special cases, such as S3 resume or fast boot.

RtBufferPtr points to the data buffer used for runtime configuration. For example, the "StackTop" pointer pointing to the top of the host firmware stack after memory is available; the pointer pointing to the configuration data region that contains customized configuration settings; and the boot mode, which is the flag to tell Intel FSP to optimize

its initialization flow. This S3 resume path (i.e., resume from a sleep state using existing memory contents) is used to optimize the boot speed by using the memory parameters passed in (instead of going through the memory training code). It can also be used for soldered-down memory configuration, or after the first boot when there is no change in memory configuration and when memory training parameters remain valid. There is not a "Fast Boot" flag to reflect the need to skip memory training, but FSP will look at the input parameter, NvsBufferPtr, to determine if it should skip memory training or not. If it is NULL, it will not skip, and if there is a valid pointer stored in NvsBufferPtr, it will skip the memory training code. This can save potential boot time up to 100 milliseconds.

```
typedef struct {
        UINT32 *StackTop;
        UINT32 BootMode;
        VOID *UpdDataRegPtr;
        UINT32 Reserved[7];
} FSP_INIT_RT_COMMON_BUFFER;
```

ContinuationFunc is a pointer to the "continuation function" address, where Intel FSP will jump back to after the execution of this API is done. After the FspInitEntry API completes its execution, it does not return to the host firmware from where it was called, but instead returns control to the host firmware by jumping to the continuation function. The jump is accompanied by two parameters: the Status and the Pointer of a HOB list that contains the hand-off data from Intel FSP to the host firmware.

```
typedef VOID (* CONTINUATION_PROC)(
        IN FSP_STATUS Status,
        IN VOID *HobListPtr
);
```

Why are we using an input parameter, continuation function's address, as the return address? The reason why a simple return is not used after FspInitEntry is done in this case is because after memory is initialized, the old stack using temporary memory will be destroyed, and a new stack will be set up in its place. The transfer of contents in a stack can be done, but this is very risky due to compiler compatibility issues, and the old stack may no longer have valid contents after the transition; for example, a pointer to a local variable in the old stack will still point to the old stack. Even after migration, it is more reliable if we simply jump to an address passed in as a parameter.

Like the previous API, this API should be called only once.

NotifyPhase

This API is used by the host firmware to notify the FSP after finishing certain phases during the boot process, so that Intel FSP can take appropriate actions as needed for these phases. The actions and phases are platform dependent and will be documented with each FSP release. Examples of boot phases include "post pci enumeration" and "ready to boot."

The FSP will lock the configuration registers to ensure system security and reliability.

```
#define FSPAPI __attribute__((cdecl))
typedef UINT32 FSP_STATUS;
typedef FSP_STATUS (FSPAPI *FSP_NOTFY_PHASE)
(NOTIFY_PHASE_PARAMS *NotifyPhaseParamPtr);
typedef enum {
        EnumInitPhaseAfterPciEnumeration = 0x20,
        EnumInitPhaseReadyToBoot = 0x40
} FSP_INIT_PHASE;
typedef struct {
        FSP_INIT_PHASE Phase;
} NOTIFY_PHASE_PARAMS;

void FspNotifyPhase (UINT32 Phase)
{
        FSP_NOTFY_PHASE NotifyPhaseProc;
        NOTIFY_PHASE_PARAMS NotifyPhaseParams;
        FSP_STATUS Status;

        /* call FSP PEI to Notify PostPciEnumeration */
        NotifyPhaseProc = (FSP_NOTFY_PHASE)(fsp_info_header->ImageBase +
fsp_info_header->NotifyPhaseEntry);
        NotifyPhaseParams.Phase = Phase;
        Status = NotifyPhaseProc (&NotifyPhaseParams);
        if (Status != 0) {
               printf("FSP API NotifyPhase failed for phase %d!\n",Phase);
        }
}
```

Intel FSP Output

As initialization progresses, a lot of system information and configuration data are collected in the process. The data needs to be passed to the host firmware so that it does not need to rediscover this data on its own. While Intel FSP is discovering the data, it builds a series of data structures called Hand-Off Blocks, or HOBs, and fills them with useful information, such as total memory size and so forth. The data structures conform to the HOB format, as described in the PI specification Volume 3: Shared Architectural Elements, which can be downloaded from www.uefi.org/specifications/ under the latest specification listed on the page. HOBs are the only conduits between Intel FSP and the host firmware once Intel FSP is done with its work. Therefore, the user of the FSP binary is strongly encouraged to go through the specification mentioned earlier in order to understand the HOB design details and create a simple infrastructure to parse the HOBs, because the same infrastructure can be reused with different FSPs across different platforms.

The specification mentioned earlier describes about nine different HOBs; most of the information may not be relevant to a particular host firmware. It's up to the host firmware to decide how to consume the information passed through the HOBs produced by the FSP.

Regarding how you know which information is important and which information is not important, you need to examine the data structures and decide what is needed by the platform initialization code, and then retrieve the data from the data structures, as needed. The data structures can be different from platform to platform, and new data can be available as new features and new requirements are developed.

API Execution Status

The host firmware can check the status after each API call. The following are the statuses for TempRamInit:

- FSP_SUCCESS: Temp RAM was initialized successfully.
- FSP_INVALID_PARAMETER: Input parameters are invalid.
- FSP_NOT_FOUND: No valid microcode was found in the microcode region.
- FSP_UNSUPPORTED: The FSP calling conditions were not met.
- FSP_DEVICE_ERROR: Temp RAM initialization failed.

The following are the statuses for FspInit:

- FSP_SUCCESS: The FSP execution environment was initialized successfully.
- FSP_INVALID_PARAMETER: Input parameters are invalid.
- FSP_UNSUPPORTED: The FSP calling conditions were not met.
- FSP_DEVICE_ERROR: FSP initialization failed.

The following are the statuses for NotifyPhase:

- FSP_SUCCESS: The notification was handled successfully.
- FSP_UNSUPPORTED: The notification was not called in the proper order.
- FSP_INVALID_PARAMETER: The notification code is invalid.

Temporary Memory Data HOB

A few paragraphs ago, we talked about the temporary stack being destroyed after main memory initialization, but there could be data that interests you that is still in the stack. Intel FSP will save the subregion where the host firmware data is stored, and pass it back to the host firmware using a HOB data structure with a unique GUID, defined as follows:

```
#define FSP_BOOTLOADER_TEMPORARY_MEMORY_HOB_GUID \
{ 0xbbcff46c, 0xc8d3, 0x4113, { 0x89, 0x85, 0xb9, 0xd4, 0xf3,
0xb3, 0xf6, 0x4e } };
```

Non-Volatile Storage HOB

Another HOB worth mentioning is the Non-Volatile Storage HOB, which is used to pass data to the host firmware to save for S3 resume or fast boot. The host firmware needs to parse the HOB list to see if such a GUID HOB exists once the continuation function is regaining control from Intel FSP. If so, the host firmware should extract the data portion from the HOB and save it into a platform-specific NVS device, such as Flash, EEPROM, and so forth.

During the following boot process, the host firmware should load the data block back from the NVS device to temporary memory and populate the buffer pointer into the FSP_INIT_PARAMS.NvsBufferPtr field before calling into the FspInit() API. If the NVS device is memory mapped, the host firmware can initialize the buffer pointer directly to the buffer.

Sample Code for Parsing HOBs

This is sample code that parses HOBs to look for memory size below the 4 gigabyte boundary:

```
VOID
GetLowMemorySize (
UINT32 *LowMemoryLength
)
{
        EFI_PEI_HOB_POINTERS Hob;
        *LowMemoryLength = 0x100000;
        //
        // Get the HOB list for processing
       //
        Hob.Raw = GetHobList();
        //
        // Collect memory ranges
        //
        while (!END_OF_HOB_LIST (Hob)) {
                if (Hob.Header->HobType == EFI_HOB_TYPE_RESOURCE_DESCRIPTOR)
{
                        if (Hob.ResourceDescriptor->ResourceType ==
                        EFI_RESOURCE_SYSTEM_MEMORY) {
                         //
                         // Need memory above 1MB to be collected here
                         //
```

```
                                if (Hob.ResourceDescriptor->PhysicalStart >=
                                0x100000 &&
                                Hob.ResourceDescriptor->PhysicalStart <
                                (EFI_PHYSICAL_ADDRESS) 0x100000000) {
                                       *LowMemoryLength += (UINT32)
                                       (Hob.ResourceDescriptor-
                                       >ResourceLength);
                                }
                        }
                }
           Hob.Raw = GET_NEXT_HOB (Hob);
        }
        return;
}
For memory above 4 gigabytes, the following code is used in place of the
code looking a PhysicalStart element:
if (Hob.ResourceDescriptor->ResourceType == EFI_RESOURCE_SYSTEM_MEMORY) {
//
// Need memory above 4GB to be collected here
//
        if (Hob.ResourceDescriptor->PhysicalStart >=
        (EFI_PHYSICAL_ADDRESS) 0x100000000) {
                *HighMemoryLength += (UINT64) (Hob.ResourceDescriptor-
                >ResourceLength);
        }
}
```

Customization of Intel FSP

Even though Intel FSP can carry out CPU, memory, and companion chip initialization, it needs the cooperation from the host firmware to pass in crucial information about the platform. And developers can also customize Intel FSP by changing the hardware configuration settings when necessary.

The way Intel FSP organizes the configurable region is to separate it into two areas: a static area and a dynamic area. The static area is called VPD (Vital Product Data) and the dynamic area is called UPD (Updatable Product Data). Both areas can be customized using the BCT; however only the UPD area can be overridden by the host firmware during runtime.

The way to override the UPD data is to copy the UPD data structure to the temporary memory (remember that this is before FspInit is called), modify the data you want to alter, and then pass the pointer in as an input parameter. The FspInit API parameter includes a pointer that can be initialized to point to the UPD data structure. If this pointer is initialized to NULL when calling the FspInit API, the FSP will use the default UPD data that is available in the FSP configuration region. If it is not NULL, the FSP will use the data structure in temporary memory. Whatever data you have modified will be used to configure the hardware when FspInit executes.

The following shows an example of the VPD data structure:

```
typedef struct _UPD_DATA_REGION {
        UINT64 Signature; /* Offset 0x0000 */
        UINT32 RESERVED1; /* Offset 0x0008 */
        UINT8 Padding0[20]; /* Offset 0x000C */
        UINT16 PcdMrcInitTsegSize; /* Offset 0x0014 */
        UINT16 PcdMrcInitMmioSize; /* Offset 0x0016 */
        UINT8 PcdMrcInitSPDAddr1; /* Offset 0x0018 */
        UINT8 PcdMrcInitSPDAddr2; /* Offset 0x0019 */
        UINT8 PcdeMMCBootMode; /* Offset 0x001B */
        UINT8 PcdEnableSdio; /* Offset 0x001C */
        UINT8 PcdEnableSdcard; /* Offset 0x001D */
        UINT8 PcdEnableHsuart0; /* Offset 0x001E */
        UINT8 PcdEnableHsuart1; /* Offset 0x001F */
        UINT8 PcdEnableSpi; /* Offset 0x0020 */
        UINT8 PcdEnableLan; /* Offset 0x0021 */
        UINT8 PcdEnableSata; /* Offset 0x0023 */
        UINT8 PcdSataMode; /* Offset 0x002E */
        UINT8 PcdEnableAzalia; /* Offset 0x002F */
        UINT32 AzaliaConfigPtr; /* Offset 0x0030 */
        UINT8 PcdEnableXhci; /* Offset 0x0034 */
        UINT8 PcdEnableLpe; /* Offset 0x0029 */
        UINT8 PcdLpssSioEnablePciMode; /* Offset 0x002A */
        UINT8 PcdEnableDma0; /* Offset 0x002B */
        UINT8 PcdEnableDma1; /* Offset 0x002C */
        UINT8 PcdEnableI2C0; /* Offset 0x002D */
        UINT8 PcdEnableI2C1; /* Offset 0x002E */
        UINT8 PcdEnableI2C2; /* Offset 0x002F */
        UINT8 PcdEnableI2C3; /* Offset 0x0030 */
        UINT8 PcdEnableI2C4; /* Offset 0x0031 */
        UINT8 PcdEnableI2C5; /* Offset 0x0032 */
        UINT8 PcdEnableI2C6; /* Offset 0x0033 */
        UINT8 PcdEnablePwm0; /* Offset 0x0034 */
        UINT8 PcdEnablePwm1; /* Offset 0x0035 */
        UINT8 PcdEnableHsi; /* Offset 0x0036 */
        UINT8 PcdIgdDvmt50PreAlloc; /* Offset 0x0043 */
        UINT8 PcdApertureSize; /* Offset 0x0044 */
        UINT8 PcdGttSize; /* Offset 0x0045 */
        UINT16 PcdRegionTerminator; /* Offset 0x003A */
} UPD_DATA_REGION;
```

Whereas a sample VPD data structure is shown here:

```
typedef struct _VPD_DATA_REGION {
        UINT64 PcdVpdRegionSign; /* Offset 0x0000 */
        UINT32 PcdImageRevision; /* Offset 0x0008 */
        UINT32 PcdUpdRegionOffset; /* Offset 0x000C */
```

```
        UINT8 Padding0[16]; /* Offset 0x0010 */
        UINT32 RESERVED1; /* Offset 0x0020 */
        UINT8 PcdPlatformType; /* Offset 0x0024 */
        UINT8 PcdEnableSecureBoot; /* Offset 0x0025 */
        UINT8 PcdMemoryParameters[16]; /* Offset 0x0026 */
} VPD_DATA_REGION;
```

These two examples are from the Intel FSP for Intel Atom E3800 Product family. In this VPD data structure, each element has a matching member provided in a Binary Setting Flag file, which specifies the layout of the configuration region.

Downloading Intel FSP

Now you understand the basic plumbing and interface of Intel FSP. More practice exercises will be provided in later chapters using coreboot and EDK II codebase to show how they integrate Intel FSP, including the handling of input parameters and output parameters. During these exercises, you will need to download a copy of Intel FSP. Here are the steps:

1. Figure out which CPU and Chipset combo you are designing your firmware for. If you are working on a SoC, you need to figure out which SoC family you are using.
2. Go to `www.intel.com/fsp/` and select the "Download an Intel FSP" link, as shown in Figure 3-4.

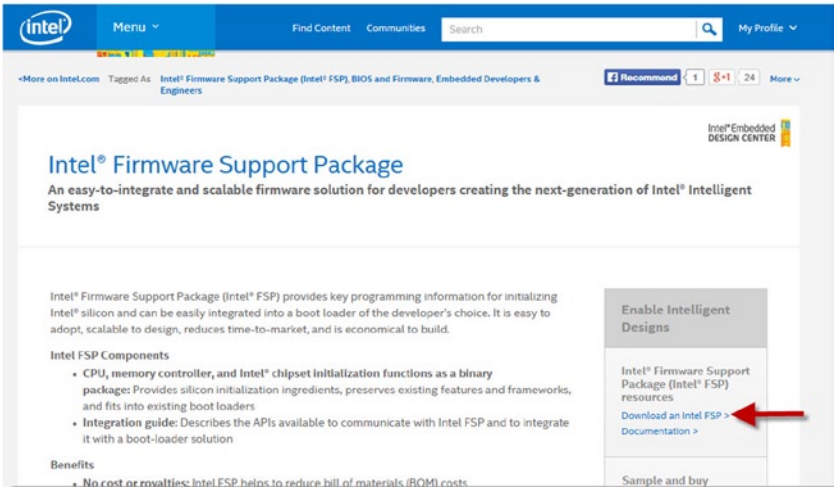

Figure 3-4. *Intel FSP download web site*

3. Once you click the "Download an Intel FSP" link, it takes you to the next screen, as shown in Figure 3-5. Select the kind of package you want: Windows or Linux.

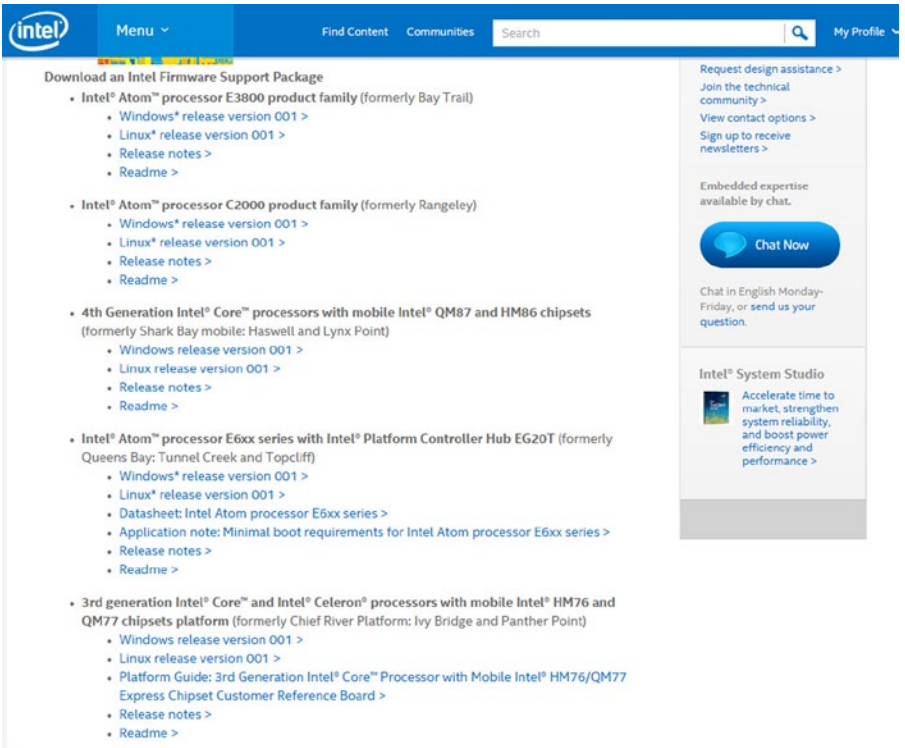

Figure 3-5. *Intel FSP download selection*

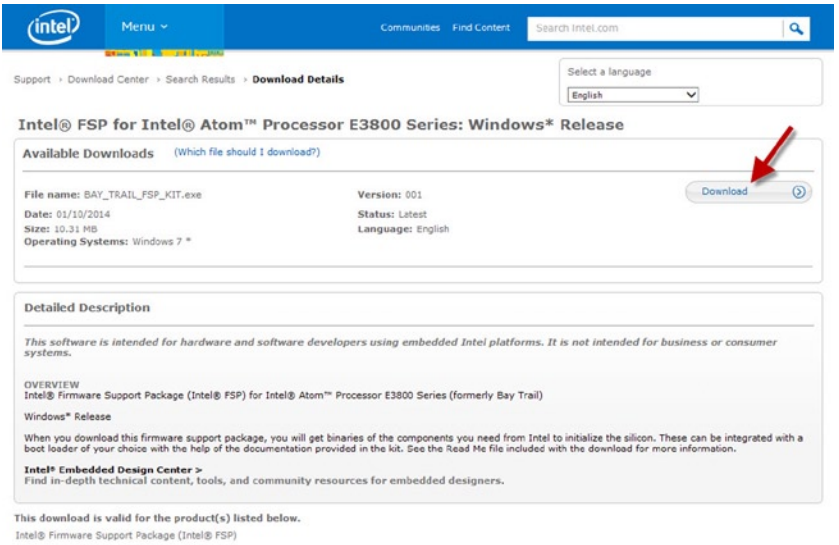

Figure 3-6. *Intel FSP download action*

4. After accepting the licensing agreement, the download will start. After downloading, unpack it.

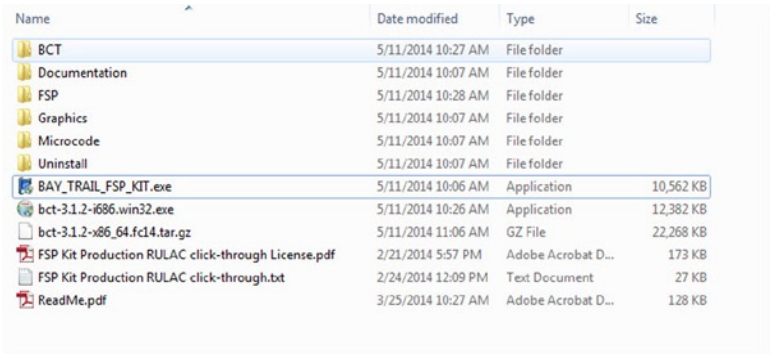

Figure 3-7. *Intel FSP package layout and contents*

5. Depending upon which source code codebase you are using, you can start copying them to the appropriate folders under the host firmware directory trees.

Now, you are ready to create code, change the code, and build the code.

Microcode Patches

Microcode is a unique machine code that runs inside a CPU and many other silicon chips to initialize its internal states and features. Since the year 2000, all modern CPUs have been using microcode updates to fix issues that are frequently documented in erratum by silicon vendors.

The microcode update format is not documented and it cannot be easily read or understood by engineers who are not specialized in microprogramming. For example, one microinstruction could carry out a "Connect this register to that side of the ALU" task, and the other one might "Set ALU's carry input to zero". Each microinstruction may be doing something completely tied to the silicon at the logic level, and will only make sense to the silicon designers who actually designed the chip.

As shown in Figure 3-8, one of the Intel FSP folders is called Microcode, and it has the microcode updates that are available at the time Intel FSP was released. Intel FSP has the microcode update mechanism already incorporated. During the microcode update step, it will check the CPUID of the microprocessor on the circuit board and update the appropriate microcode update file into the CPU. There is the possibility that you are using a newer board with a newer stepping of the CPU, and thus the microcode updates inside Intel FSP do not match the stepping of your CPU; therefore, it will not be able to find the appropriate microcode update to update your CPU, and the system will hang as a result. If this is the case, you will need to go to Intel's support web site and look for the latest microcode updates for the CPU you have on board, convert it from a raw text file to an `.h` or an `.inc` text file, and copy it to the corresponding microcode folder in the host firmware directory tree.

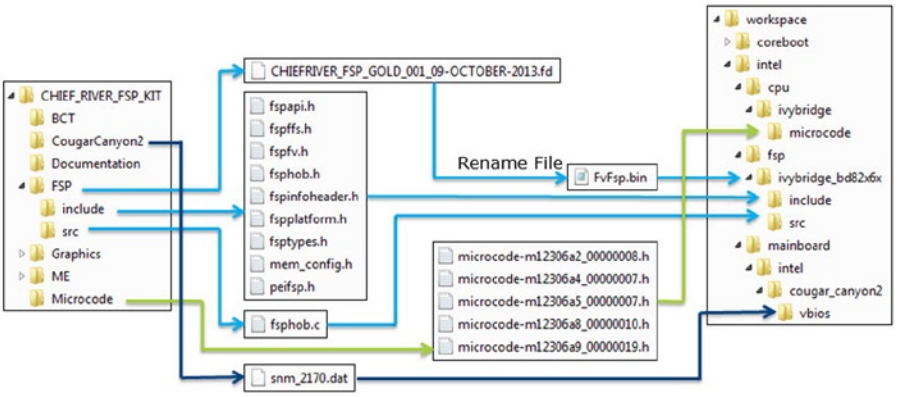

Figure 3-8. *Intel FSP file movement to the host firmware folders*

If you are familiar with Linux commands, here are two commands to convert from a `.txt` file to `.h` or `.inc` files:

```
cat name.txt | awk ' { print $1 ; print $2 ; print $3; print $4 } ' > name.h
cat name.txt | sed 's/,//'| awk ' { print " .long " $1 } '
```

There is also a utility in the coreboot tree to convert the microcode file for you; there may be other tools available to do the same thing.

The microcode updates–based address and the length are passed into the TempRamInit API as an input parameter, which is stored in ROM and is part of the ROM-based stack that we described earlier. In a multiprocessor environment, FSP supports BSP (Boot Strap Processor) and AP (Application Processor) microcode loading. The microcode range can contain multiple back-to-back microcode binaries, and the FSP will check them one by one and load all applicable patches. The microcode code patches for BSP will be loaded inside the TempRamInit API, and the patches for AP will be loaded in the FspInit API.

Relocating Intel FSP

After you download and start building your host firmware, you are told to put the Intel FSP binary at the physical location described in the FSP integration guide. However, if you decide to move Intel FSP, you will need to run BCT and change the base address of FSP to the location you prefer to use. This step is needed because many internal references to the base address need to be corrected by a tool, since they do not dynamically resolve address assignment at runtime.

Integration and Build

At this point, you should have all the elements from Intel in place to do an integration with your host firmware stack. Once you have done the integration, the build process and testing will begin. Hopefully, you will spend most of time developing your value-added features, rather than debugging Intel silicon code. Intel FSP should have taken care of your basic silicon initialization needs at this point.

The Future of Intel FSP

Intel FSP is only the first step in the direction that Intel wants to take to help embedded and IoT developers to adopt Intel Architecture easily and quickly, so that they aren't spending precious time studying, implementing, and debugging silicon-related configurations and issues. Going forward, Intel FSP will continue its evolution toward a more flexible, scalable, and customizable silicon initialization module, and will cover more silicon on the Intel roadmap. We do realize that there are different needs in the developer community. Unfortunately, these needs are distributed on a scale with two opposite extremes on the spectrum. On one end are the developers who want completely

open source code available to them so that they can play with the source code with total freedom and total control. On the other end are the developers who just want a turnkey solution so that they don't have to know anything about firmware; "knobs" for tuning parameters and for turning on and off features are all that they need.

The best way to satisfy everyone is to keep opening up source code for collaboration and customization and to provide different levels of turnkey solutions where it makes sense.

What Is Coming in the Following Chapters

By now, if you have been reading this chapter carefully, you should know what Intel FSP is, why it was created, what it can do, and how it carries out the work, as well as how to customize Intel FSP and how to integrate it into the host firmware stack of your choice.

Even though there are only three APIs (this number may change in a future Intel FSP) that you need to interface with in your firmware stack, you still need to prepare input parameters and deal with output parameters carefully, and learn how to build the firmware with the right tools. It is best if you continue by reading the following hands-on chapters to get familiar with the practical knowledge about building successful firmware stacks.