CHAPTER 6

Intel FSP and UEFI Integration

"In theory, there is no difference between theory and practice. But, in practice, there is."

—Yogi Berra

This chapter will provide a bridge from the earlier chapters on FSP and coreboot to Unified Extensible Interface (UEFI)-based technologies. These UEFI technologies include the UEFI Platform Initialization (PI)–based modules in the EFI Developer Kit II (EDK II) found on tianocore.org. EDK II elements can "consume" an Intel FSP binary to build a complete platform. In addition, EDK II can be used as a payload for coreboot-style systems. Finally, there are examples of EDK II–based open hardware platforms, such as Minnow and MinnowMax, which demonstrate the Intel FSP and EDK II platform elements.

Introduction to EFI

EFI (Extensible Firmware Interface) and coreboot began around the same time in the late 1990s. They started with two different purposes. One of the objectives of EFI was moving legacy BIOS to a modern interface with a modular driver model that allowed components to be added and removed with ease.coreboot, on the other hand, was mostly created from scratch to keep a minimum set of hardware initialization to boot Linux, and it was designed to be an open source project from the beginning. EFI was later adopted by many industry leaders, and turned into UEFI (Unified Extensible Firmware Interface). Along with the UEFI evolution, the EDK (EFI Development Kit) was created to help firmware development projects based on UEFI. Today's EDK II is the second generation of the original EDK, as the name suggests. You may also find UDK (UEFI Development Kit) as the name representing the same codebase, which designates specific validated instances of the EDK II tree.

We have talked about coreboot and Intel FSP in the last few chapters, and we will focus on EDK II and Intel FSP integration in this chapter. Figure 6-1 shows an overview of the development flow of different firmware ingredients in route to make the final target ROM. As stated before, the goal of this book is to teach you how to put these ingredients together so that you have the freedom to explore and develop your value-added features.

The life cycle of firmware creation is shown in Figure 6-1. The flow starts from the left, with the silicon reference code formatted as PEI Modules (PEIMs). These PEIMs are, in turn, built into a firmware volume with the appropriate FSP interfaces via the FSP SDK. This SDK includes elements of the MDE Module Package (https://svn.code.Sf.net/p/edk2/code/trunk/edk2/MdeModulePkg) and the Intel FSP Package (https://svn.code.Sf.net/p/edk2/code/trunk/edk2/IntelFspPkg/). The latter package encapsulates the interfaces that a FSP binary needs to publish, as defined in the FSP External Architecture Specification at http://www.intel.com/fsp.

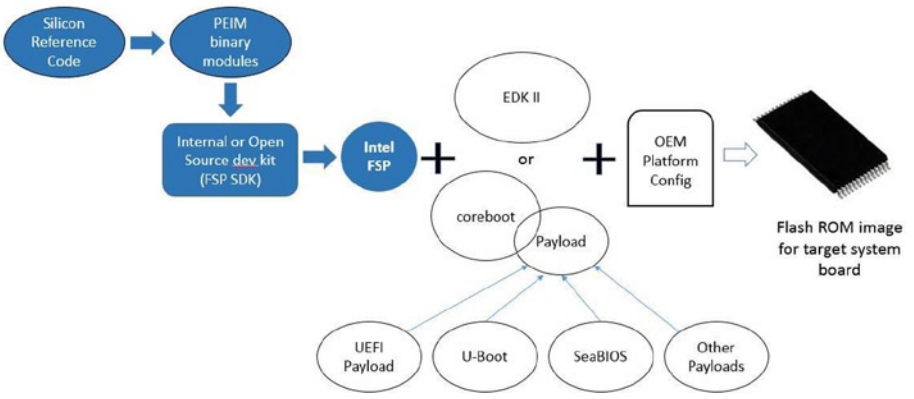

Figure 6-1. *Work flow of FSP and EDK II*

Even though UEFI/PI–based EDK II and core boot are distinctly different code bases, conceptually, they share similar boot phases. Table 6-1 compares core boot and UEFI/PI–based EDK II from a boot phase and terminology perspective. The left most column describes the common capability, the middle column lists the core boot mapping, and finally, the right most column contains the UEFI PI–based firmware mapping.

Table 6-1. *Comparison of coreboot and UEFI PI*

Capability	coreboot	UEFI PI
The reset vector and pre cache-as-RAM setup.	boot block	Security Phase (SEC)
Cache-as-RAM setup, early silicon initialization, memory setup. Covered largely by Intel FSP.	rom stage	Pre-EFI Initialization (PEI) Create HOBs
Normal device setup and main board configuration. Publishes SMBIOS/ACPI tables.	ram stage	Early Driver Execution Environment (DXE)
Memory map hand-off.	CBMEM	UEFI Memory Map
The OS or application boot loader.	pay load	DXE BDS and UEFI Drivers

The reality is that EDK II is necessarily rich and expansive because it supports the specifications at www.uefi.org and a large set of capability sets to meet the various business demands. To that end, EDK2 and UEFI have been successful in the broad market.

Even UEFI/EDK II stalwarts have expressed complexity, including anecdotal commentary at http://uefi.blogspot.com/2014/04/the-tale-of-three-conferences.html.

Many discussions around UEFI have to do with complexity. And there is something to these discussions, since the very power and flexibility of UEFI has led to implementations (like that on TianoCore) that are broken into hundreds of pieces, where assembling the right one requires the appropriate recipes. Most embedded vendors don't need their firmware distribution to be as complicated as their Linux distribution (see yoctoproject.org).

Alternate firmware communities like core boot (www.coreboot.org) have Source Control Management (SCM), a simpler workflow and number of files, and many public main boards with full source. This poses an opportunity to embrace some of those properties in our extant EDK II community. Specifically, the challenge for this effort is to balance the richness of the present EDK II source, standards, and http://tianocore.org, but provide simplified views or "recipes."

The Firmware Support Package (FSP) is a recipe for aggregating a series of PEI Modules (www.uefi.org) into a firmware volume. More information on the interface to the FSP, including the hand-off state and additional APIs, can be found at http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/fsp-architecture-spec.html in the FSP EAS. We refer to this internally as "FSP 1.0" since the evolution of FSP to include the migration of code to the reset vector and other state management will be provided on the path toward the vision of "FSP2.0."

FSP allows for the reuse of the validated EDK II code from the Intel product groups providing silicon-initialization reference code. So the FSP + coreboot, or the FSP + EDK II boot loader allow for amortizing the validation listed earlier.

There are presently efforts underway to assess open sourcing the closed-source silicon reference code (SiRC) in Figure 6-1. If that happens, does that mean that the FSP is no longer of value? No. Instead, FSP provides a simple means by which to segregate the CPU and chipset-specific initialization code from the platform initialization (board-specific elements) and the boot loader (OS-interface specific). This provides a clean separation of duty from the ecosystem.

Introduction to FSP

The Intel® Firmware Support Package (Intel® FSP) provides key programming information for initializing Intel® silicon. It can be easily integrated into a firmware boot environment of the developer's choice.

Different Intel hardware devices may have different Intel FSP binary instances, so a platform user needs to choose the right Intel FSP binary release. The FSP binary should be independent of the platform design, but specific to the Intel CPU and chipset complex. We refer to the entities that create the FSP binary as the *FSP producer* and the developer who integrates the FSP into some platform firmware as the *FSP consumer*.

Despite the variability of the FSP binaries, the FSP API caller (a.k. A. FSP consumer) could be a generic module to invoke the three APIs defined in FSP EAS. This allows for rich differentiation in the platform silicon, but it provides for a single set of interface code in the consuming firmware base, whether it be coreboot or EDK II based. In other words, the interface to the FSP binary can be like a class-driver or generic code, written for any FSP binary. This allows for the ease of maintaining generic open source firmware code that can absorb FSPs from different lines of business within a given manufacturer or other manufacturers.

Figure 6-2 describes the FSP architecture, with the FSP binary from the FSP producer in the center of the figure, and the platform code that integrates the binary, or the FSP consumer, in blue. The fixed APIs published by the center FSP producer code, which is the center of the figure, allow for a consistent set of invoking "consumer" code in the adjacent boxes on the left and right.

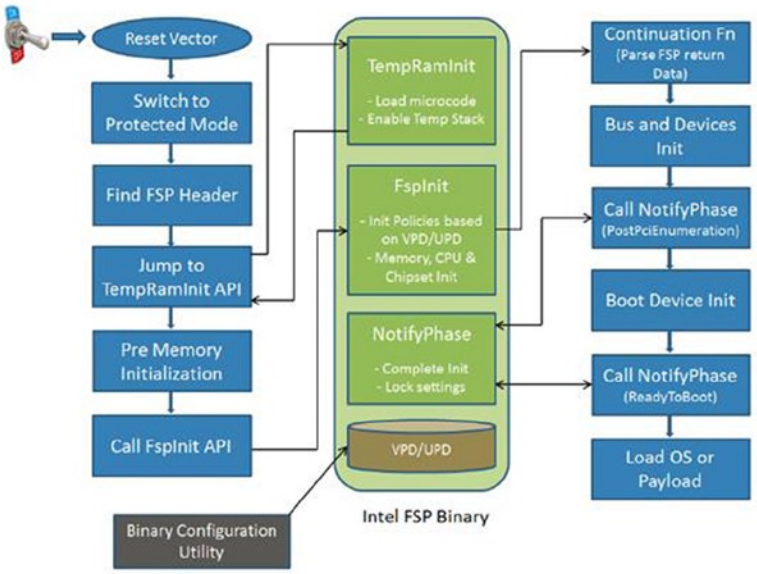

Figure 6-2. *Platform flow with FSP*

The FSP EAS describes the API interface to the FSP binary that the consumer code will invoke, but it also describes the hand-off state from the execution of the FSP binary. The latter information is conveyed in Hand-Off Blocks, or HOBs. The Hand-Off Block generic definition can be found in the UEFI Platform Initialization (PI) specification at `www.uefi.org`, and the generic FSP EAS contains additional definitions of the HOB, as do silicon-specific integration guides. Both the HOB definition and the binary layout of the FSP.bin, namely as a Firmware Volume (FV), are the same as that defined in the UEFI PI specification. Both the reuse of the PI specification artifacts and the EDK II open source are used in the FSP production.

The FSP consumption, which is the topic of this chapter, can be a plurality of firmware environments, of which an EDK II-style consumer will be described in more detail. This EDK II consume will be a mixture of generic code from Tiano Core and platform-specific PEI and DXE modules.

Introduction to EDK II

EDK II is the open source implementation for UEFI firmware, which can boot multiple UEFI OS. This document will introduce how to use EDK II as a FSP consumer module to build a platform BIOS.

Summary

This section provided an overview of Intel FSP and EDK II. Specifically, the use of open source http://www.Tianocore.org/ elements that provide UEFI and UEFI PI interfaces—along with consumer code of the FSP APIs—will be treated. TianoCore/ is a package-based firmware architecture that provides an open source implementation of industry-standard APIs in order to provide a "BIOS Core" that can be used in the construction of platforms. The more generic, open source packages of TianoCore/, such as the Module Development Environment (MDE) and the associated modules of the MDE Module Package, should be reusable across a broad class of architectures, from ARM Ltd to AMD to Intel. Chapter 2 of the UEFI 2.4 specification describes the various bindings for which UEFI can be built.

For the Intel FSP, the https://svn.code.Sf.net/p/edk2/code/trunk/edk2/IntelFspWrapperPkg/ is the package that provides an abstraction into an FSP.bin file. The FSP.bin is included in a firmware build by importing the binary into the directory and including a reference to the DSC file, along with the other necessary packages from TianoCore/.

FSP Components

In EDK II, there are two different FSP-related packages. One is the producer, IntelFspPkg, which is used to produce FSP.bin together with other EDK II packages and silicon packages. The other is consumer, IntelFspWrapperPkg, which consumes the API exposed by FSP.bin.

This chapter only focuses on IntelFspWrapperPkg and how it consumes FSP.bin. This chapter will not describe IntelFspPkg or how it produces FSP.bin because the upcoming chapter on Tiny Quark describes the construction of a full initialization binary very much like a FSP binary. The core boot consumption of FSP was described in Chapter 4. Figure 6-3 shows the EDK II work flow for the production of an FSP (on the left) and the resultant paths to consume the FSP (on the right).

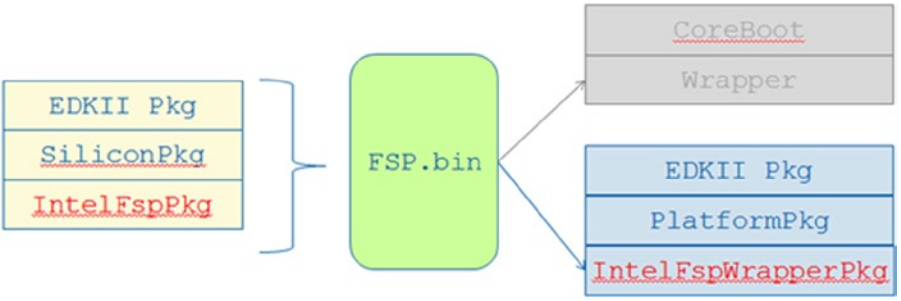

Figure 6-3. *FSP components*

FSP Wrapper Boot Flow

According to the FSP EAS, an FSP.bin exposes three APIs: TempRamInitApi, FspInitApi, and FspNotifyApi (PciEnumerationDone and ReadyToBoot).

When should they be invoked in EDK II BIOS? There are three options for integrating the FSP into an EDK II-style firmware. Details on the three options are described next, with a grading that can be found in Table 6-2.

There are many architectural choices. The choices of FSP integration include having a SEC core that directly invokes the FSP binary, and a second implementation that includes a full PEI firmware volume with a PEI core. The former would be for a simple system without other features in the PEI firmware volume (FV), such as Capsule Update or Recovery.

With the SEC integration of FSP, which is shown in Figure 6-4, the SecCore can call TempRamInitApi and FspInitApi immediately, and then skip the entire PEI phase and jump to DxeLoad. DxeLoad can consume the FspHob, produce HOBs for DXE, and then enter DxeCore directly. Afterward, FspNotifyDxe will register for a notification on the PciEnumerationDone and ReadyToBoot callback function.

Finally, the FspNotifyApi is called in the callback function.

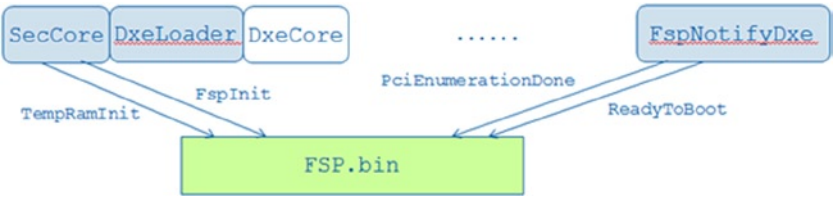

Figure 6-4. *FSP wrapper boot flow option 1*

For the option described in Figure 6-5, SecCore calls TempRamInitApi and FspInitApi immediately, and then enters PeiCore as normal. One PEIM will consume FspHob and produce the HOBneeded by DXE. At the end of PEI, DxeIpl is launched and enters DxeCore. The FspNotifiyDxe is the same as the example shown in Figure 6-4.

Figure 6-5. *FSP wrapper boot flow option 2*

In the final integration option shown in Figure 6-6, SecCore calls TempRamInitApi only, and then enters the PeiCore. The FspInitPei module calls FspInitApi. However, once FspInitApi is back, all PEI context saved in CAR is destroyed. So FspInitPei has to enter PeiCore again to continue the PEI phase boot. Then, the rest of the initialization activities will be same as a normal UEFI PI firmware boot flow. In addition, FspNotifyDxe is the same as option 1 listed above in Figure 6-4.

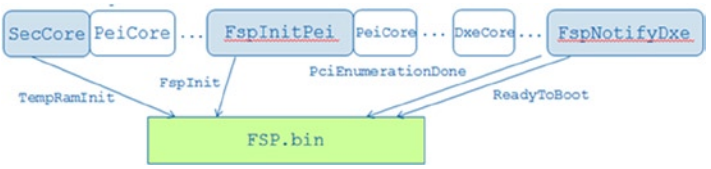

Figure 6-6. *FSP wrapper boot flow option 3*

Table 6-2 compares the pros and the cons for each solution.

Table 6-2. *Comparison of the FSP Integration Options*

FSP Wrapper Flow	Pros	Cons
Option 1	Small firmware size.	No generic Dxe Loader. Hard to support a different PI boot mode.
Option 2	All generic code.	Hard to support a different PI boot mode.
Option 3	All generic code. Supports all PI boot modes.	Complex. Needs to enter PEI Core twice.

In EDK II, the default option is the last one. That means the IntelFspWrapperPkg can support multiple PI boot modes, like Normal boot, S3 resume, Capsule Update, and Recovery. Boot modes are describes in the UEFI PI Specification.

However, an EDK II developer can use option 2 if the platform is so simple that there is no need to support multiple boot modes. Or, a developer can use option 1 if the platform is simple enough to skip the PEI phase. And in this case, "simple" means PEI-phase features like platform Recovery, S3 Resume, Capsule Detection, and other platform-based PEI phase actions can be omitted or deferred to a later phase, such as the Driver Execution Environment (DXE).

Generic FSP Wrapper Boot Flow

Next is a deeper dive into the FSP wrapper boot flow. The detailed boot flow in each of the UEFI PI boot modes will be described, too.

Normal Boot

A Normal Boot is typically a restart of the system that passes control from the reset vector to the ensuing OS loader or payload. It is typically an ACPI S5 restart from a hardware perspective, as opposed to a wake event like ACPI S3 or a restart event that is intended to support a firmware update, such as the UEFI Capsule Update.

Boot Flow

In Normal boot mode, SecCore will firstcall theFSP API—TempRamInitApi, and then transfer control to the PeiCore. One platform PEIM will be responsible to detect the current boot mode and find some variable to finalize the boot mode selection. The Normal boot flow is shown in Figure 6-7.

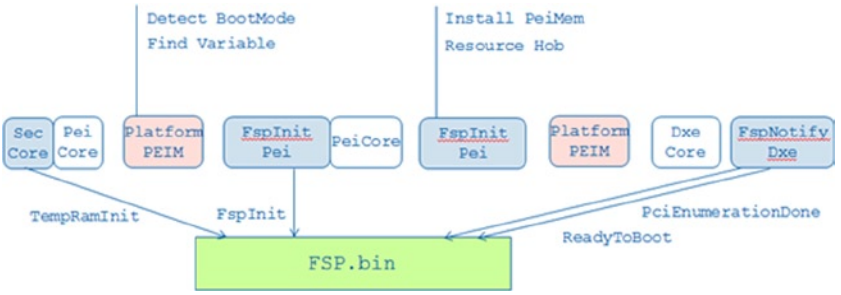

Figure 6-7. *FSP Normal boot flow*

FspInitPei has a dependency on MasterBootModePpi, so after the boot mode is determined, FspInitPei is invoked for the first time, and it will call the second FSP API—FspInitApi. In FSP.bin, the cache is torn down, so all previous PeiCore context is lost. In the FspInit continuation function, it will emulate SecCore to launch the PeiCore again, with a special PEIM-to-PEIM Interface (PPI)—FspInitDonePpi—as a parameter

for the PeiCore. Then FspInitPei will be invoked for a second time. At that moment, since FspInitDonePpi is installed, FspInitPei will run into another path to parse the FspHob and install PEI memory.

PeiCore will continue dispatching the final PEIMs and jump into the DXE core. Then, DXE core will launch FspNotifyDxe, which registers a callback function for the last FSP API—FspNotifyApi, for both PciEnumerationDone and ReadyToBoot.

Memory Layout for a Normal Boot Flow

The memory layout for FSP Normal boot is shown in Figure 6-8. The left-hand side demonstrates the component on flash and the temporary memory, such as cache as RAM. On the right-hand side is the DRAM layout. The green part is for FSP.BIN. The blue part is for EDK II BIOS.

When SecCore calls TempRamInitApi, FSP binary sets up the temporary memory using the processor cache-as-RAM (CAR), uses part of them, and leaves the rest of these activities to the EDK II BIOS. This CAR information is reported as a return parameter of TempRamInitApi (see bottom left of Figure 6-8).

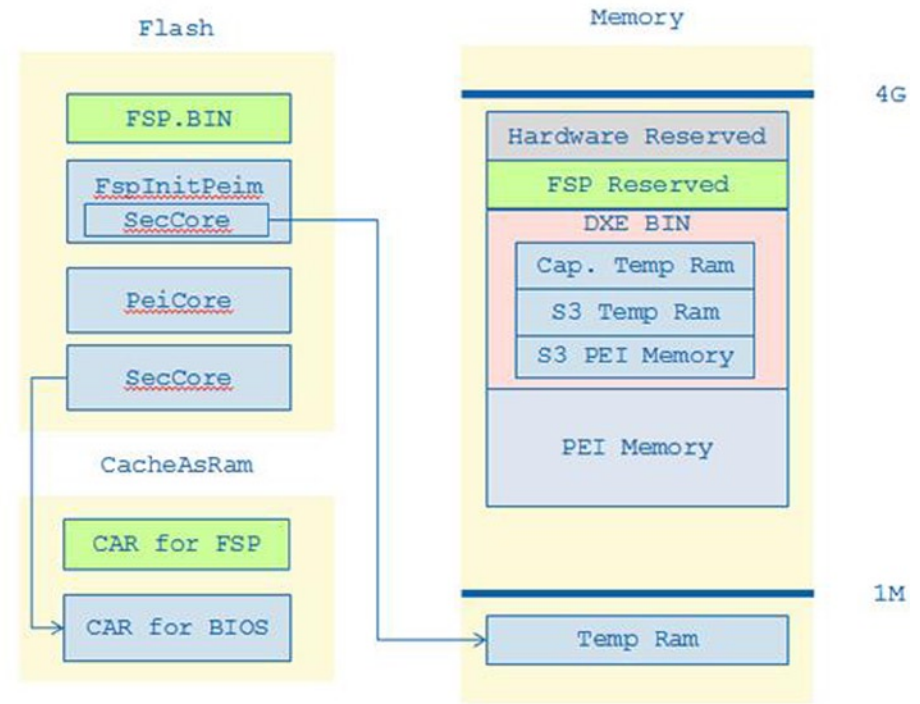

Figure 6-8. *FSP Normal boot flow memory layout*

Then FspInitPei calls FspInitApi, wherein the FSP binary will initialize silicon including DRAM and reserved portions of DRAM. The full memory layout, including full DRAM size, reserved DRAM location, and SMRAM location will be reported by the FspHob. After FspInitApi, it will return back to the Continue Function provided by FspInitPei, with the stack pointed to DRAM (because CAR is destroyed; see the bottom right of Figure 6-8).

In FspInitPei, the Continue Function will launch the second SecCore, with the temp ram pointed to DRAM. The second SecCore will launch the same PeiCore and continue dispatching the PEI firmware volume (see the top left of Figure 6-8 for more information). The purpose of the second PEI core is so that the FSP binary has its own small PI initialization flow. Regions in the figure are broken out by "who allocates the memory—FSP reserved at the top from FSP.bin, DXE code and data with DXE allocations in the middle, and PEI allocated at the bottom."

Finally, the system enters the DXE phase, and a platform module may allocate temporary ram for the S3 boot path and a capsule boot path to save the information in a tamper-proof, safe location.

FSP Normal Boot Data Structure

According to Figure 6-8, there are two SecCores involved. The first one is the normal SecCore and the second one is a small SecCore inside FspInitPei. Given the two SecCores, there needs to be a scheme where in the first SecCore passes information to the second one, like the Built-In Self-Test (BIST) data and the initial reading of the performance counter, or 'boot time ticker' needed for subsequent construction of the Firmware Performance Data Table (FPDT).

In IntelFspWrapperPkg, the first SecCore saves the BIST and ticker in CAR. Before FspInitApi is called, the platform may choose to save them in some special registers not touched by FSP.bin. One example could be IA CPU multimedia registers like MMX, or a PCI scratch register. In the implementations listed later, the MMX option is used, as shown in Figure 6-9, using MMX0 and MMX5/6 for BIST and performance ticker information, respectively. After FspInitApi, the FspInitPei launches the second SecCore, which will restore the information from special registers to a new stack in temp ram. The second SecCore also registers a special TopOfTemporaryRam PPI (a.k. A., TopOfCarPPI, as shown in Figure 6-9), which has pointer to the top of temp ram.

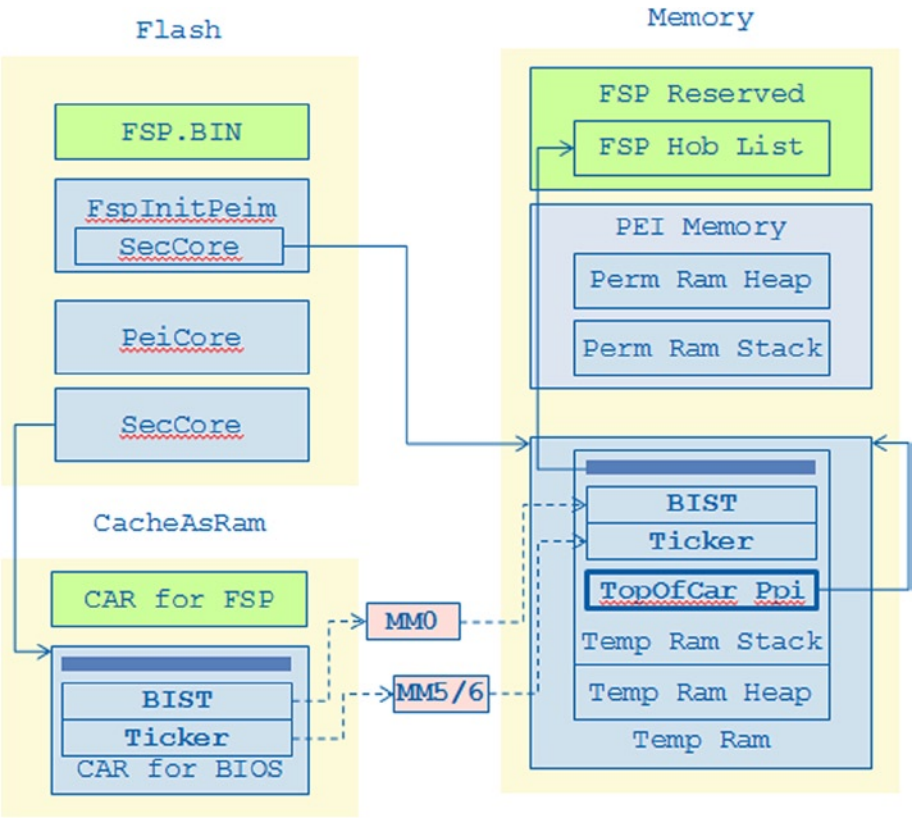

Figure 6-9. *FspInitPei data structure*

The reason to introduce TopOfTemporaryRam PPI is because the FspInitPei needs a way to ascertain the beginning of the FSP's HOB list, while the FspHobList pointer itself is saved to the top of temp ram. Also, the BIST and boot time ticker information are saved at the top of temp ram. It becomes easy to know the information by having a PPI to tell the temp ram location.

This section described the FSP wrapper boot flow in Normal boot mode. As part of enabling the Normal boot, several options were reviewed, with details on the present open source implementation (option 3), that supports all the UEFI PI PEI phase related features. The next section goes into the specific implementation of those features, such as S3, Recovery, and Capsule Update.

S3 Boot

TheS3 Boot represents one of the more involved restart mechanism. S3 is a wake event wherein most of the hardware it put into a low-power state and the main memory needs to be taken out of a self-refresh mode. The system firmware orchestrates this restoration by replaying a set of the configuration settings applied during a Normal Boot as one embodiment.

Boot Flow

In S3 boot, the difference from the Normal boot entails when to call FspNotifyApi. In Normal boot mode, this invocation occurs in the DXE phase, but in S3 boot mode, there is no DXE. Details of the FSP S3 boot flow are shown in Figure 6-10.

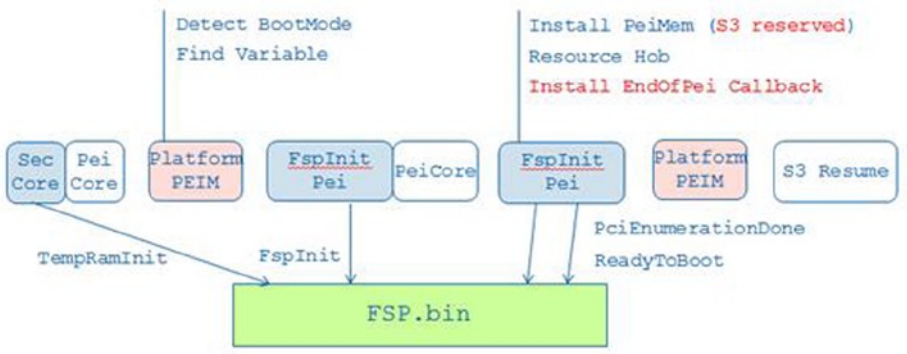

Figure 6-10. *FSP S3 boot flow*

In the IntelFspWrapperPkg, the FspInitPei routine will register EndOfPei callback in S3 boot mode. So when the boot script finishes execution, FspNotifyApi is invoked, and then the system invokes the OS waking vector.

S3 Memory Layout

In an S3 boot, the difference in the memory layout entails the temp ram location. In Normal boot mode, the temporary ram is at some low DRAM address, configured by Platform Configuration Database (PCD) settings, which is used by no one in PEI phase. In the S3 boot, usable DRAM is owned by OS, expected the one reported as ACPI reserved or ACPI NVS. The details of this memory layout are shown in Figure 6-11.

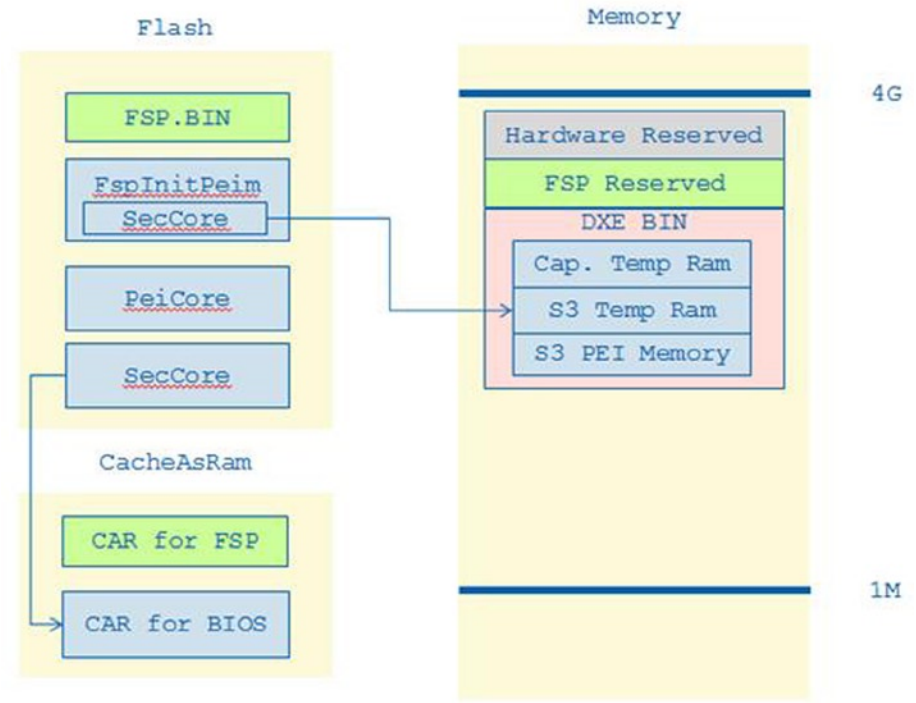

Figure 6-11. *FSP S3 memory layout*

In a Normal boot DXE phase, a platform driver should allocate S3 temp ram, marking it as reserved to OS. Then in the S3 phase, the FspInitPei can use it as temp ram for purposes of invoking the continuation function.

S3 NV Data Passing

In some platforms, S3 phase initialization needs the configuration saved in a Normal boot. Figure 6-12 is an example of how memory configuration data is passed from the MRC module in Normal boot to the MRC module in S3.

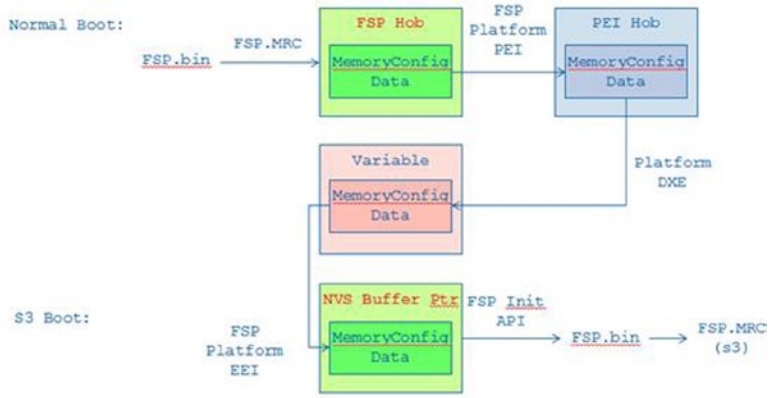

Figure 6-12. *FSP S3 NVS data passing*

In a Normal boot, the FSP Memory Reference Code (MRC) PEI module produces a MemoryConfigData HOB and saves it in the FSP HOB list. The resultant FSPlist is published after FspInitApi. Then an FSP platform PEI parses the FSP HOB, gets the MemoryConfigData, and saves it into the normal PEI HOB list. In the DXE phase, a platform module parses the PEI HOB list and saves MemoryConfigData into an NV variable.

In S3 boot, the FSP PEI module finds the MemoryConfigData from an NV variable region, constructs NvsBufferPtr as an FspInitApi parameter, and calls the FSP binary. Then, the FSP binary has the NvsBufferPtr, and the MRC module can get the MemoryConfigData from NvsBufferPtr and do the memory initialization in S3 phase.

This section describes the FSP wrapper boot flow in S3 boot mode. This section treated concerns about having memory reservation for purposes of invoking the OS wake vector and securely storing the configuration information necessary to restore the platform state prior to the hand-off to the OS.

Capsule Flash Update

The Normal Boot and S3 boot typically refer to hardware restarts, namely S5 and S3, respectively. The Capsule Flash Update is a logical boot flow that leverages an S5 or S3 hardware restart but applies some underlying logic to orchestrate receiving and applying a firmware update.

Boot Flow

In the Capsule Update boot, there is only a small difference with the Normal boot flow. In the Capsule Update flow, the FspInitPei needs to call CapsuleCoalesce before installing PEI memory, and it needs to install PEI memory for Capsule Update mode. Other differences from the Normal boot flow include the size and location of the PEI memory. The reason that the Capsule Update flow may need to sequester more memory is that the Capsule FV in memory is loaded by the PEI phase and thus consumes additional resources from the PEI usages of DRAM. Details of the FSP Capsule Update boot flow can be found in Figure 6-13.

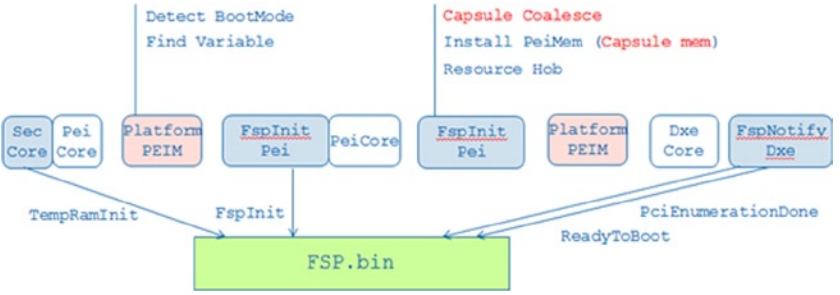

Figure 6-13. *FSP Capsule Update boot flow*

Capsule Update Memory Layout

In Capsule Update boot mode, the difference in memory layout is the temp ram location. In Normal boot mode, the temp ram is at some low DRAM, configured by a PCD, which is used by no one at PEI phase. In Capsule Update boot, usable DRAM is owned by OS, and one can expect this to be reported as ACPI reserved or ACPI NVS. The OS might put the capsule image into any usable DRAM. Specifically, the UpdateCapsule() runtime call from the UEFI runtime service table has a pointer to a scatter gather list of memory allocated by the OS kernel. As such, the firmware, such as the PEI phase and FSP, cannot predict "where" the OS will put the capsule. The PEI phase's responsibility is to discover the capsule and then coalesce or merge the fragments into a continuous run of memory, such that the following DXE phase can ascertain data or code from the Capsule Firmware volume. Additional details on UpdateCapsule can be found in the UEFI Specification at www.uefi.org. Details of the flash and memory during the Capsule flow can be found in Figure 6-14.

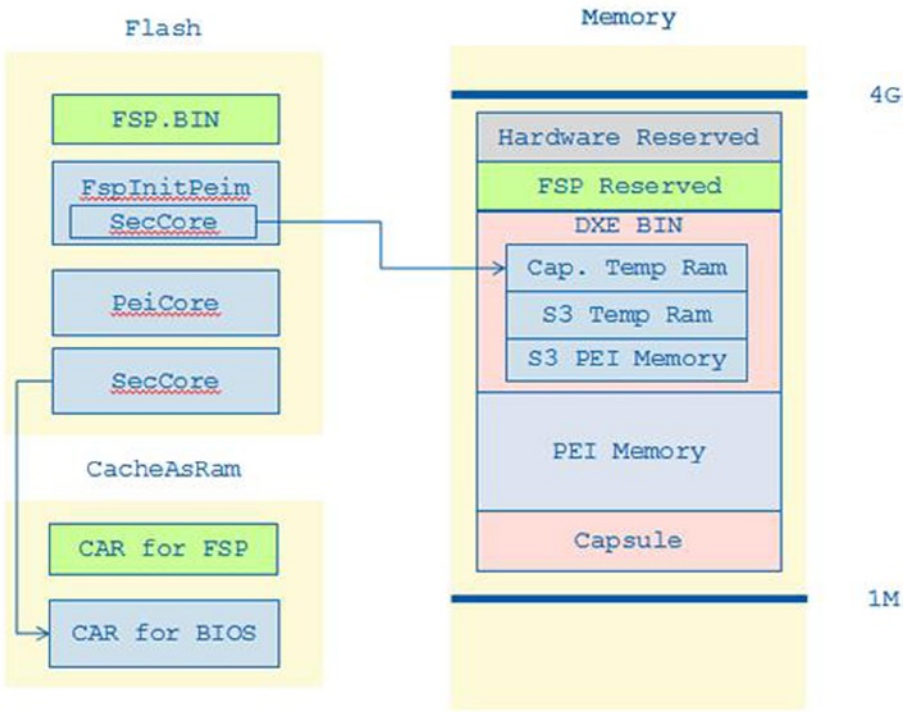

Figure 6-14. *FSP Capsule Update boot memory layout*

In a normal boot DXE phase, a platform driver should allocate capsule temp ram, marking it as reserved to the OS. Then in the Capsule Update phase, the FspInitPei can use it as temp ram for continued functioning.

This section has provided an overview of the Capsule Update boot flow. The flow is very similar to the Normal boot flow, with the exception of additional memory reservation for storing the capsule firmware volume.

Recovery Boot Flow

In Recovery boot, there is only a small difference from the earlier flow: FspInitPei needs to install PEI memory for Recovery Mode. The size might be different with Normal boot mode because the recovery firmware volume will be loaded from some recovery media, such as a USB thumb drive, and put into memory reserved by the PEI phase. This is similar to the Capsule Update in that the PEI phase will pass up a firmware volume not contained in the system board flash: OS-reserved memory for the capsule and on-recovery-media data for recovery. Details of the recovery flow are shown in Figure 6-15.

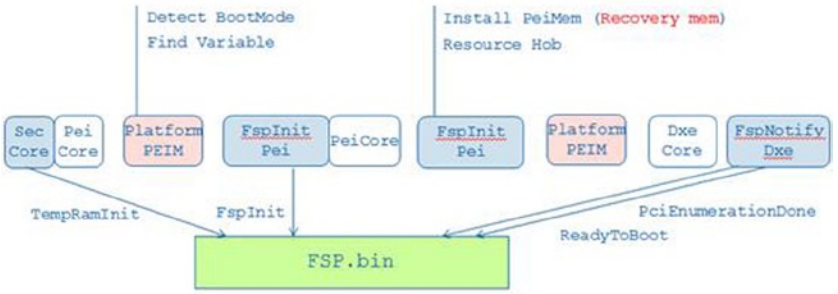

Figure 6-15. *FSP Recovery boot flow*

FSP Recovery Memory Layout

In Recovery boot, the memory layout is the same as Normal boot mode. Details are shown in Figure 6-16.

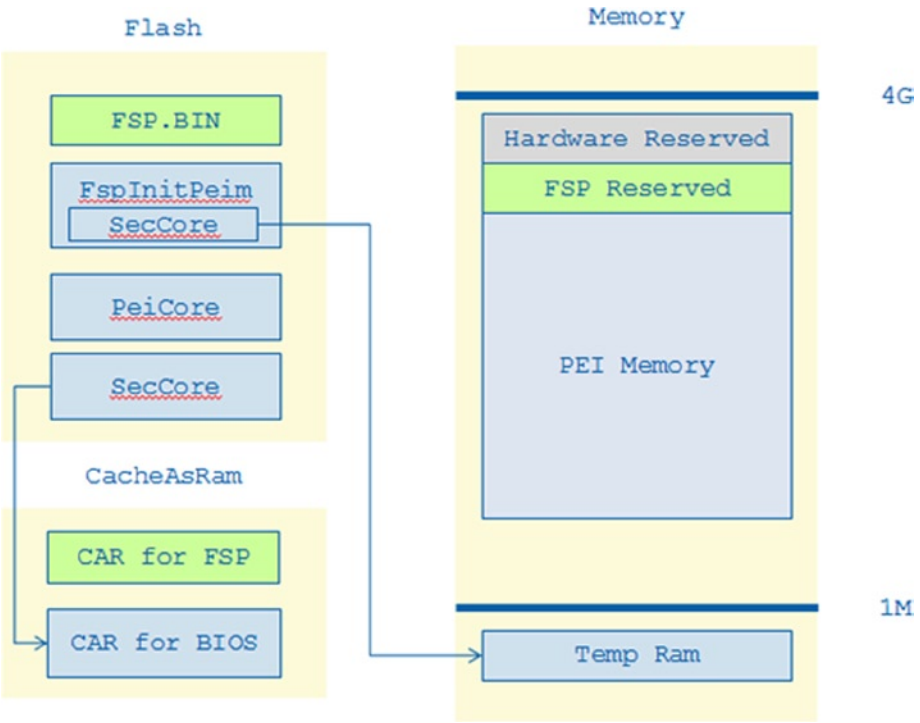

Figure 6-16. *FSP Recovery boot memory layout*

This section has shown how a full set of UEFI PI features can be implemented with the Intel FSP Wrapper Package. These features include Capsule Update, Recovery, and S3. These discussions included the design considerations and memory layout for the different scenarios.

The packages on TianoCOre are prefixed with "Intel" since the FSP EAS is an Intel-published document. This Intel-qualified moniker is very much akin to the Intel Platform Innovation for the Extensible Firmware Interface, or "Framework" specifications, lifecycle. These specifications started viaan Intel self-published matter that Intel subsequently contributed to the UEFI Forum to become the UEFI PI specifications.

coreboot Payload Based upon EDK II

Beyond using EDK II to consume an FSP, a payload for coreboot can be built upon EDK II. Chapter 4 talked about the overall concept of payloads in coreboot. As a quick recap, recall that the coreboot romstage and ramstage do platform initialization, but the set of interfaces exposed to the operating system hand-off can vary. As an old friend once told me, "Compatibility is the software you choose to run," (see http://vzimmer.blogspot.com/2013/02/what-is-compatibility.html), so the payloads vary based upon that criteria, namely the operating system "software" that you wish to run. For example, if you want to launch a PC/AT BIOS–based OS, then Sea BIOS is a choice. Correspondingly, if you want to use coreboot for platform initialization but launch a UEFI-style OS, then the UEFI payload is an option.

There have been many UEFI payloads built in the past, including EDK-based. For this discussion, we build upon earlier EDK II review to show, in addition to EDK II being one means by which to construct an FSP, the generic EDK II code can be used to create a coreboot payload, too. A variant of this payload can be found at http://www.uefidk.com. It consists of a DSC file that includes many generic drivers from TianoCore and a few purpose-built modules that take the coreboot state and adapt it into UEFI. The firmware device (FD) that contains these modules is shown in Figure 6-17.

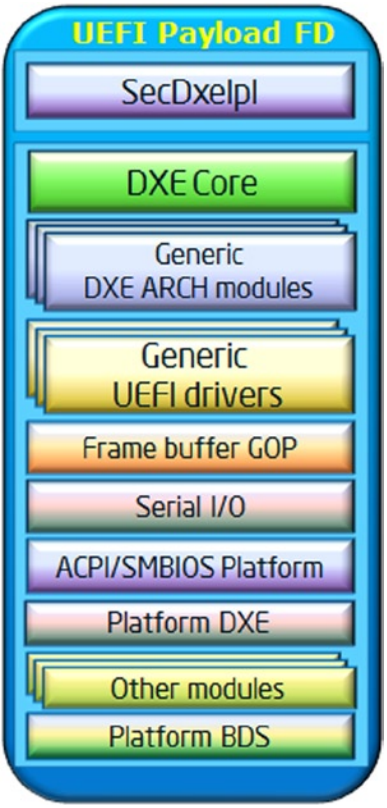

Figure 6-17. *UEFI payload based upon EDK II*

Figure 6-17 shows the spatial view of the code; the UEFI payload in Figure 6-18 describes the temporal flow of the constituent elements. As described in a generic UEFI flow, there is a miniature variant of the same in the payload, namely a SEC module that takes control from coreboot ramstage and passes CBMEM from coreboot into the following payload PEI and DXE phase. One module in the payload is responsible for transforming the CBMEM description of the memory and addressing space topology into UEFI PI HOBs and the UEFI DXE internal tracking elements, like Global Coherency Domain (GCD) entries, for purposes of publishing the resultant descriptions from the UEFI service GetMemoryMap(). The HOBs and GCD can be found in the UEFI PI specifications, and the GetMemoryMap API can be found in the UEFI specification. All of these documents can be found at www.uefi.org.

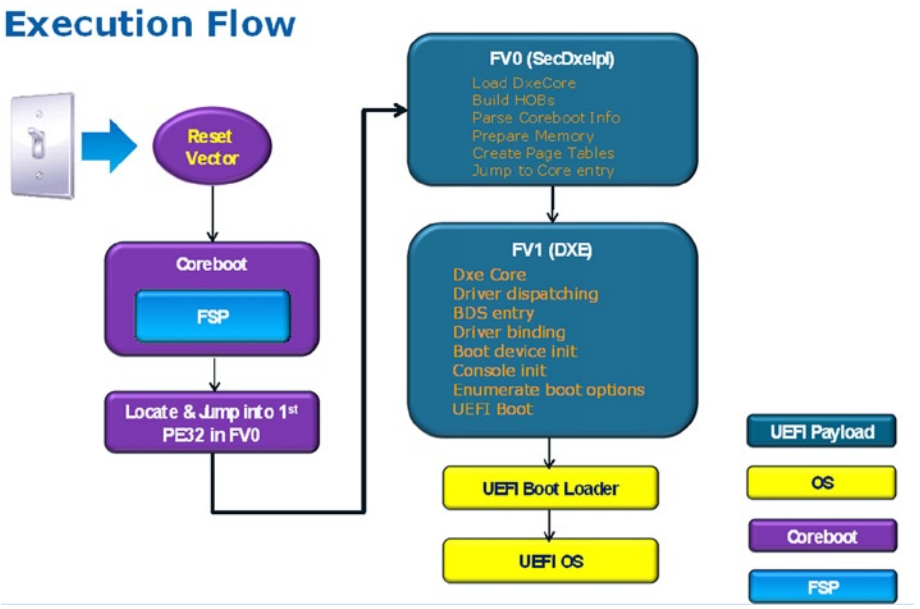

Figure 6-18. *UEFI Payload boot flow*

Building Minnow and MinnowMax with FSP

The preceding discussions talk about infrastructure code, but not a complete platform. Intel FSP provides a simple method to integrate a solution that reduces time-to-market, and it is economical to build. IntelFspWrapperPkg is the FSP consumer in EDK II to support building out a UEFI BIOS. This section describes in detail the work flow and data structure in IntelFspWrapperPkg.

The source trees for Minnow and MinnowMax can be found at http://www.uefidk.com. These distributions include the elements necessary to build a full UEFI-based firmware image, including a build option based upon the Intel FSP. The locations are http://www.uefidk.com/content/minnowboard-uefi-firmware and https://uefidk.com/content/minnowboard-max, respectively.

This section features a simple work flow to adapt an Intel® Firmware Support Package (FSP) (http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/intel-fsp-overview.html.html) –based binary module and an operating system boot loader layer built upon the EFI Development Kit 2 (EDK II) (http://tianocore.Sourceforge.net/wiki/EDK2). The work flow entails retrieving the Intel FSP binary from the Intel website. The FSP binary will be unique for a given CPU, chipset, and memory controller.

The FSP binary can then be adapted to an EDK II style boot environment with the https://svn.code.Sf.net/p/edk2/code/trunk/edk2/IntelFspWrapperPkg/. The FSP binary itself is formatted as a UEFI Platform Initialization (PI) firmware volume and conveys the output results of the FSP initialization in UEFI PI Hand-Off Blocks. These HOBs include details such as the initialized set of main memory, the graphics frame

buffer in the case of integrated FSP graphics, and other initialization resources, like TSEG. Beyond the Intel FSP Wrapper, a small complement of EDK II code can be used to provide a minimal set of architectural protocols, DXE core, and a Boot Device Selection (BDS) driver to boot an embedded operating system.

The BDS can provide a standards-based UEFI boot, or alternate BDS implementations can feature direct kernel loads from flash, as in a coreboot payload approach, or direct kernel load from disk, as found in U-Boot.

Three FSP wrapper modules are provided in the Intel FSP Wrapper Package to help EDK II firmware make calls to the FSP binary. FspSecCore will prepare the running environment for FSP binary, search the FSP information header, and then call the two basic FSP APIs (TempRamInit and FspInit). After these two APIs are executed, both the processor and the chipset have been initialized, and memory is also ready for use. At last, FspSecCore will transfer the control to PeiCore. FspWrapperPei will be executed in the PEI phase. It will parse the HOBs produced by FSP and report them to the EDK II BIOS if they are required. Also, it will install memory according to the memory resource information from the FSP. FspWrapperDxe should be run in DXE phase to notify the FSP about the different phases in the boot process. This allows the FSP to take appropriate actions as needed during different initialization phases.

Details on the workflow are as follows.

Before we start the integration, let's first figure out what you need to have.

- *FSP binary*. The binary file that contains the basic CPU and chipset init code; you also get a FSP usage guide.
- *IntelFspWrapperPkg*. The EDK II wrappers we created and put together into a single package; this is available at TianoCore. This package includes the components FspDxeIpl and FspSecCore.
- *Your current EDK II platform code base*.

Now, let's go through following steps to prepare your codebase to incorporate the FSP binary. For the Minnow and MinnowMax projects, these files can be found on http://www.uefidk.com.

1. Change the BIOS flash map file and put the FSP binary at the expected address.

During the build of FSP binary, the base address of FSP binary will be placed at the predefined address; if your flash map layout is different with those predefined addresses, please change them accordingly. Next is the list of the six PCDs used by FSP. For example, in Platform.fdf:

```
// Base address of bios flash device
SET gEfiFspTokenSpaceGuid.PcdFlashAreaBaseAddress = 0xFFC00000
// Size of bios flash device
SET gEfiFspTokenSpaceGuid.PcdFlashAreaSize = 0x400000
// Base address of cpu microcode
SET gEfiFspTokenSpaceGuid.PcdFlashMicroCodeAddress = 0xFFFB8000
// Size of cpu microcode
SET gEfiFspTokenSpaceGuid.PcdFlashMicroCodeSize = 0x00004000
0x003c0000|0x00020000
gEfiFspTokenSpaceGuid.PcdFlashFvFspBase|gEfiFspTokenSpaceGuid.PcdFlashFvFspSize
FILE = $(WORKSPACE)/MinnowPkg/FspBinary/FvFsp.bin
```

2. Replace the original SecCore Module with FspPkg's SecCore Module.

In addition, you need to set values for these two PCDs (PcdTemporaryRamBase and PcdTemporaryRamSize). These two PCDs define the base address and size of the temporary memory, which is used in the PEI phase before memory is installed. As in SecCore, FSP has already initialized the memory. You can use any physical memory address, or you can remove any definitions on these two PCDs in platform.dsc and use the default values.

For example, in Platform.dsc:

```
[PcdsFixedAtBuild]
gEfiCpuTokenSpaceGuid.PcdTemporaryRamBase|0x00080000
gEfiCpuTokenSpaceGuid.PcdTemporaryRamSize|0x00010000
[Components.IA32]
FspPkg/SecCore/SecCore.inf
In Platform.fdf:
// INFIA32FamilyCpuPkg/SecCore/SecCore.inf
INFFspPkg/SecCore/SecCore.inf
```

3. Add FSP support PEIM and DXE modules from FspPkg.

For example, in Platform.dsc:

```
[Components.IA32]
FspPkg/FspPei/FspPei.inf
FspPkg/FspDxe/FspDxe.inf
In Platform.fdf:
INF FspPkg/FspPei/FspPei.inf
INF FspPkg/FspDxe/FspDxe.inf
```

For the open hardware, the website http://www.uefidk.com/projects has the landing page (https://uefidk.com/content/minnowboard-max) with files to build a full MinnowMax (http://www.minnowboard.org/meet-minnowboard-max/) tree.

Presently, these distributions are online as a zip archive. In the future, these sources should evolve to the live open source tree at TianoCore.

This tree supports both a native EDK II build from individual binary and sources, along with the FSP build. The latter is keyed off of MINNOW2_FSP_BUILD in the platform DSC file.

Specifically, as noted in the release notes, you need to open Vlv2TbltDevicePkg\PlatformPkgConfig.dsc.

Modify "DEFINE MINNOW2_FSP_BUILD" macro from "FALSE" to "TRUE". DEFINE MINNOW2_FSP_BUILD = TRUE

The preceding definition tells the build tools to select the modules that work with the Intel FSP Wrapper Package, as opposed to building the firmware wholly from separate .PEI and .EFI files for the PEI and DXE phases of execution, respectively.

Future of the Intel FSP

The Intel FSP is a binary-enabling model that works in tandem with open or closed source IA firmware platform code. The work flow of binaries can include additional tools in the future, such as the Intel® Firmware Engine (http://www.uefidk.com/sites/default/files/resources/SF14_STTS002_100f.pdf). These tools allow for automating the creation of the FSP, or configuring the FSP, such as the PCD-as-VPD for platform adaptation, from a user interface.

In addition to automation tools, the present FSP has the UEFI EDK II platform code SecCore or coreboot rom stage for the hardware reset, but a future evolution of Intel FSP can include moving some of the critical silicon initialization to reset itself. This would entail having a CPU manufacturer code that might even ship with the hardware element with hand-off to platform IA firmware via today's FSP or EFI PI PEI hand-off, such as HOBs.

The work flow, including binary manipulation tools and the migration for the FSP binary to reset, are shown in Figure 6-19.

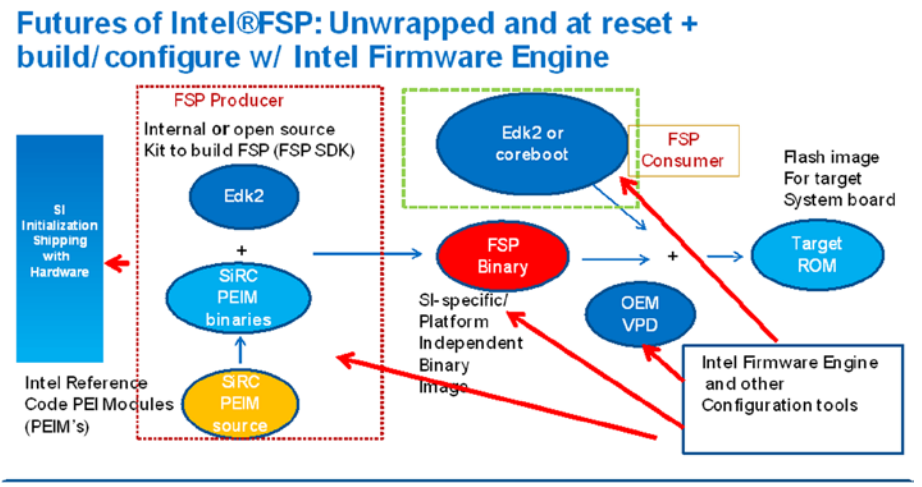

Figure 6-19. *Evolution of Intel FSP*

And as the Intel Framework Specification went to UEFI PI, the Intel FSP EAS could also follow the same pattern and appear in a future UEFI PI specification.

Conclusion

This chapter described several aspects of building firmware using UEFI class technology. The chapter began by building upon the Chapter 4 coreboot section, which is leveraged to compare coreboot phases with UEFI PI flows. This was followed by describing EDK II on FSP via the various boot flows of the Intel FSP Wrapper Package. After this overview, a description of the wrapper enablement of a Normal, Capsule, Recovery, and S3 boot followed. Building upon the Intel FSP Wrapper Package, the construction of EDK II on FSP for Minnow and MinnowMax open hardware platforms followed. After instantiating FSP on these two platforms, a glimpse is provided into possible evolution of the Intel FSP construction workflows with tool assistance and the migration of the Intel FSP binary to the reset vector.