White Paper

A Tour Beyond BIOS Using the Intel® Firmware Support Package 1.1 with the EFI Developer Kit II

Jiewen Yao
Intel Corporation

Vincent J. Zimmer
Intel Corporation

Ravi Rangarajan
Intel Corporation

Maurice Ma
Intel Corporation

David Estrada
Intel Corporation

Giri Mudusuru
Intel Corporation

April 2015

Executive Summary

This paper presents the internal structure and boot flow of Intel® Firmware Support Package (FSP) conformant binary [FSP EAS] wrapper package in EDKII [EDK2]. This wrapper will consume an Intel FSP binary and be combined with EDKII-based platform code and core modules to support UEFI OS boots.

Prerequisite

This paper assumes that audience has EDKII/UEFI firmware development experience. He or she should be familiar with UEFI/PI firmware infrastructure (e.g., SEC, PEI, DXE, runtime phase), and know the UEFI/PI firmware boot flow (e.g., normal boot, S3, Capsule update, recovery) [UEFI][UEFI Book].

Table of Contents

Overview

Introduction to FSP

The Intel® Firmware Support Package (Intel® FSP) [FSP] provides key programming information for initializing Intel® silicon and 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 (aka FSP consumer) could be a generic module to invoke the three APIs defined in FSP EAS (External Architecture Specification) to finish silicon initialization [FSP EAS].

The flow below describes the FSP, with the FSP binary from the "FSP Producer" in green and the platform code that integrates the binary, or the "FSP Consumer", in blue.

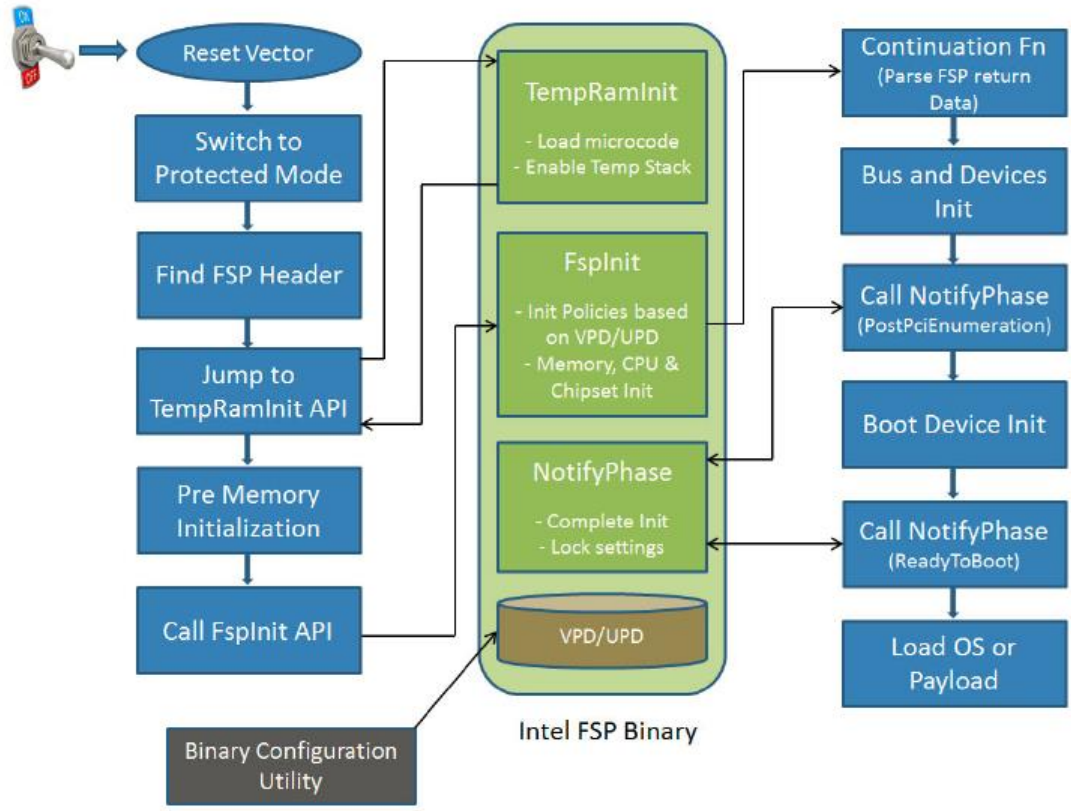

Figure 1 FSP architecture

The FSP EAS describes both 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 HOB's. 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 EDKII open source are using in the FSP production.

The FSP consumption, which is the topic of this paper, can be a plurality of firmware environments, of which an EDKII-style consumer will be described in more detail.

Introduction to FSP 1.1

FSP version 1.1 supports two boot flows. See below figure. Boot Flow 1 is simpler for the boot loader. It provides the compatible support as FSP version 1.0. Boot Flow 2 increases flexibility and control for the boot loader. It splits FspInit API to 3 different APIs (FspMemoryInit, TempRamExit, and FspSiliconInit), so that boot loader always keeps control of the boot flow. The two boot flows are mutually exclusive.

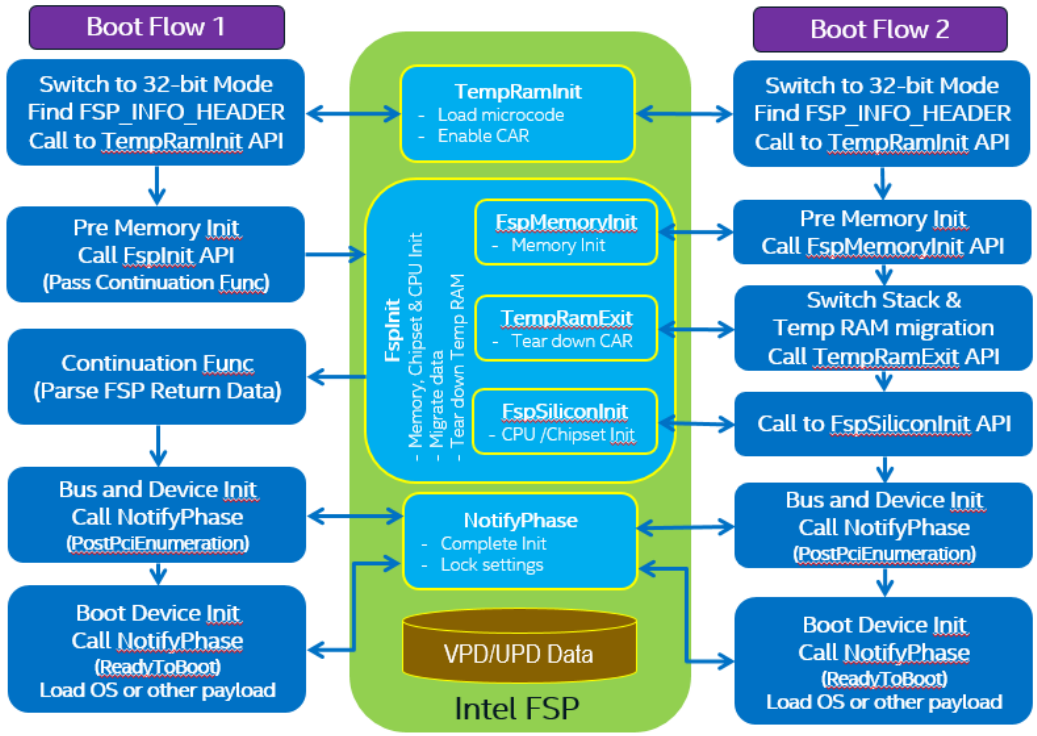

Figure 2.1 FSP 1.1 architecture

FSP 1.1 also provides graphic support. If BIT0 (GRAPHICS_SUPPORT) of the ImageAttribute field in the FSP_INFO_HEADER is set, the FSP include graphics initialization capabilities. When graphics capability is included in FSP and enabled, FSP produces an EFI_PEI_GRAPHICS_INFO_HOB as described in the PI Specification which provides information about the graphics mode and framebuffer. So that the boot loader may have a

generic driver to produce EFI_GRAPHICS_OUTPUT_PROTOCOL defined in UEFI specification.

Introduction to EDKII

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

Summary

This section provided an overview of Intel FSP and EDKII.

FSP Component

In EDKII, there are 2 different FSP related packages. One is producer – IntelFspPkg, it is used to produce FSP.bin together with other EDKII package and silicon package. The other is consumer - IntelFspWrapperPkg, it will consume the API exposed by FSP.bin.

This paper only focuses on IntelFspWrapperPkg on how IntelFspWrapperPkg consume FSP.bin. This paper will not describe IntelFspPkg on how it produces FSP.bin.[FSP Producer] This paper will not describe other way to consume FSP.bin, like coreboot [COREBOOT].

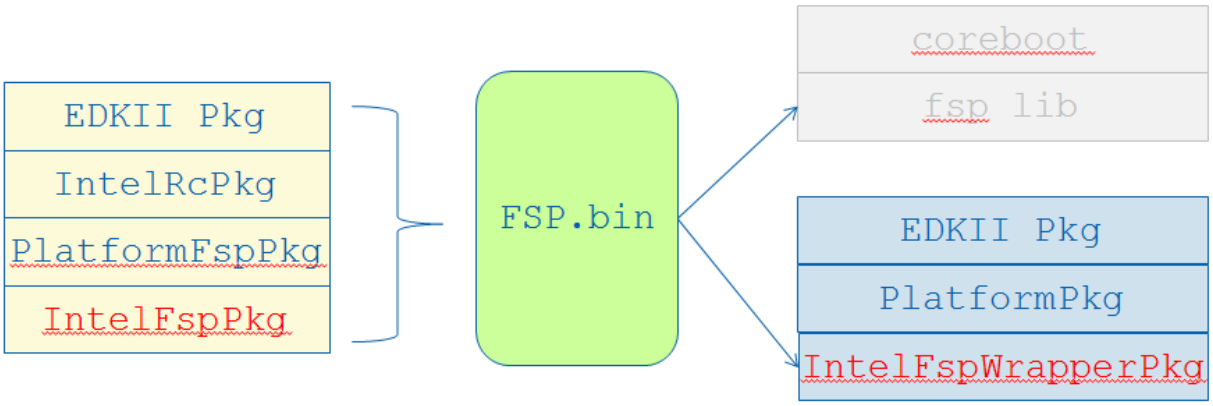

Figure 2 FSP component

Summary
This section describes the FSP component in EDKII.

FSP Wrapper Boot Flow

According to the FSP EAS, an FSP.bin exposes 3 API's - TempRamInitApi, FspInitApi, FspNotifyApi (PciEnumerationDone and ReadyToBoot) in boot flow 1. In boot flow 2, FspInitApi split to FspMemoryInitApi, TempRamExitApi, and FspSiliconInitApi, while TempRamInitApi and FspNotifyApi are same.

So when they should be invoked in EDKII BIOS?

There are many architectural choices. See below example:

1) SecCore can call TempRamInitApi and FspInitApi (or FspMemoryInit, TempRamExitApi, and FspSiliconInit) immediately, then skip the entire PEI phase, jump to DxeLoad. DxeLoad can consume the FspHob, produce Hob's for DXE and then enter DxeCore directly. Afterward FspNotifyDxe will register for a notification on PciEnumerationDone and ReadyToBoot callback function. Finally, the FspNotifyApi will be called in the callback function.

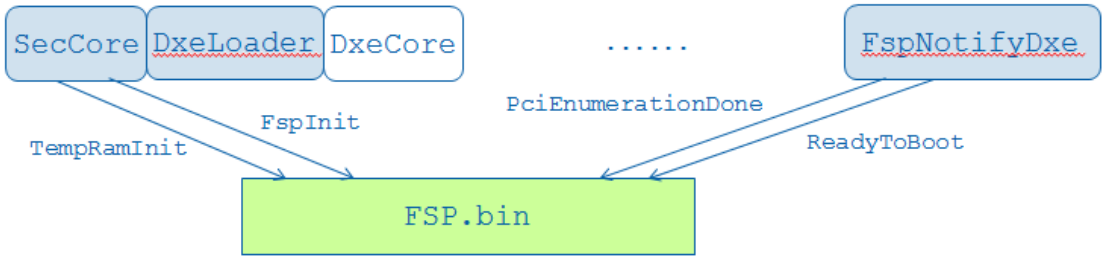

Figure 3 FSP wrapper boot flow #1

2) SecCore calls TempRamInitApi and FspInitApi (or FspMemoryInit, TempRamExitApi, and FspSiliconInit) immediately, and then enters PeiCore as normal. One PEIM will consume FspHob and produce Hob needed by DXE. At the end of PEI, DxeIpl will be launched and enter DxeCore. The FspNotifiyDxe is same as 1).

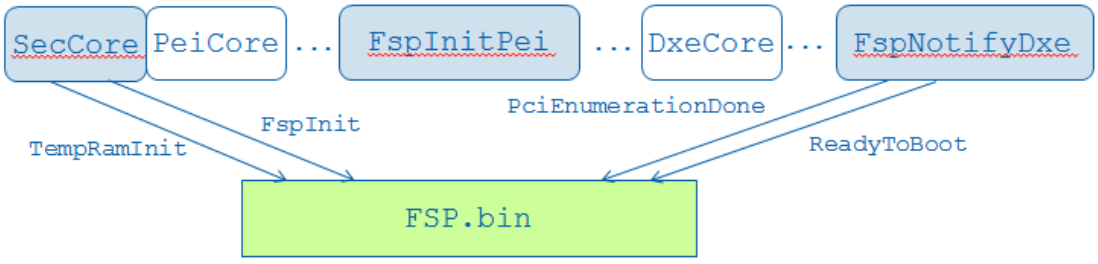

Figure 4 FSP wrapper boot flow #2

3) SecCore calls TempRamInitApi only, and then enters the PeiCore. FspInitPei module will call FspInitApi in boot flow 1. However, after FspInitApi is back, all PEI context saved in CAR is destroyed. So FspInitPei has to enter PeiCore again to continue PEI phase boot. Then the rest of the initialization activities will be same as normal UEFI PI firmware boot flow.

And FspNotifyDxe is the same as 1).

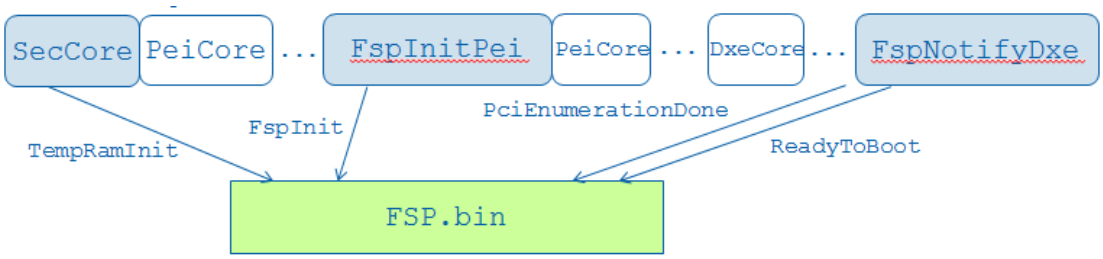

Figure 5 FSP wrapper boot flow #3

In FSP 1.1 boot flow 2, FspInitPei module will call FspMemoryInit, TempRamExit and FspSiliconInit. Since CAR teardown in TempRamExit is invoked by FspInitPei, all PEI context saved in CAR is migrated by PEI core. As a result, there is no need to let FspInitPei enter PEI Core again.

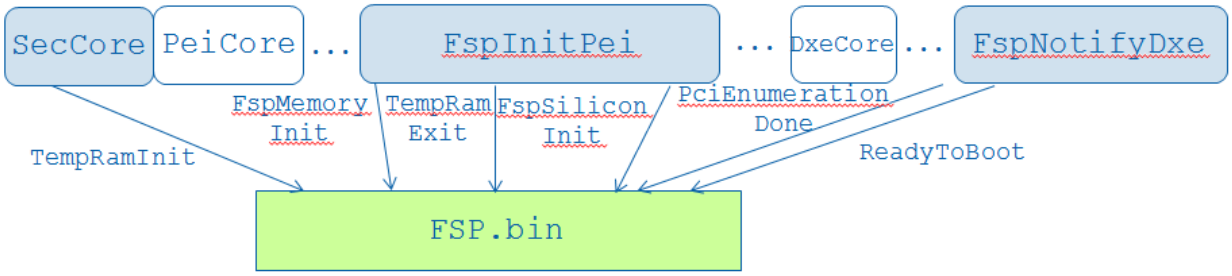

Figure 5.1 FSP1.1 wrapper boot flow #3

See the below table in order to compare the PROs and CONs for each solution.

Table 1 FSP wrapper boot flow summary

	PROs	CONs
Option 1	● Small firmware size	● No generic DxeLoader ● Hard to support different PI boot mode.
Option 2	● All generic code	● Hard to support different PI boot mode.
Option 3	● All generic code ● Support all PI boot modes.	● Complex; need enter PEI Core twice in boot flow 1. There is no such need in boot flow 2.

In EDKII, the default option is the last one. That means the IntelFspWrapperPkg (https://github.com/tianocore/edk2-IntelFspWrapperPkg) can support multiple PI boot modes, like normal boot, S3 [ACPI] resume, capsule update, as well as recovery. Boot modes are describes in the UEFI PI Specification [UEFI PI Specification].

However, an EDKII developer can use option 2 if the platform is so simple that there is no need to support multiple boot modes. Or he or she can use option 1, if the platform is simple enough to skip PEI phase.

Summary

This section has a generic overview of FSP wrapper boot flow. The detail boot flow in each boot mode will be described in next several sections.

If developer owns a platform which is so simple that it does not support advanced boot modes like S3, capsule update and recovery, he or she can selectively skip S3 boot mode section, capsule update section or recovery section.

Normal Boot

Boot Flow

In normal boot, FspWrapperSecCore (https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/FspWrapperSecCore) will call first FSP 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.

FspInitPei (https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/FspInitPei) has a dependency on MasterBootModePpi, so after the boot mode is determined, FspInitPei (https://github.com/tianocore/edk2-IntelFspWrapperPkg/blob/master/FspInitPei/FspInitPeiV1.c) will be invoked at first time, and it will call second FSP API – FspInitApi. In FSP.bin, the cache will be torn down, so all previous PeiCore context will be lost. In the FspInit continuation function, it will emulate SecCore (https://github.com/tianocore/edk2-IntelFspWrapperPkg/blob/master/FspInitPei/SecMain.c) to launch the PeiCore again, with a special PPI – FspInitDonePpi as a parameter for the PeiCore. Then FspInitPei will be invoked at second time. At that moment, since FspInitDonePpi is installed, FspInitPei will run into another path to parse the FspHob and install PEI memory.

Then PeiCore will continue dispatching the final PEIMs and jump into the DXE core. Then DXE core will launch FspNotifyDxe (https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/FspNotifyDxe). FspNotifyDxe registers a callback function for the last FSP API – FspNotifyApi, for both PciEnumerationDone and ReadyToBoot.

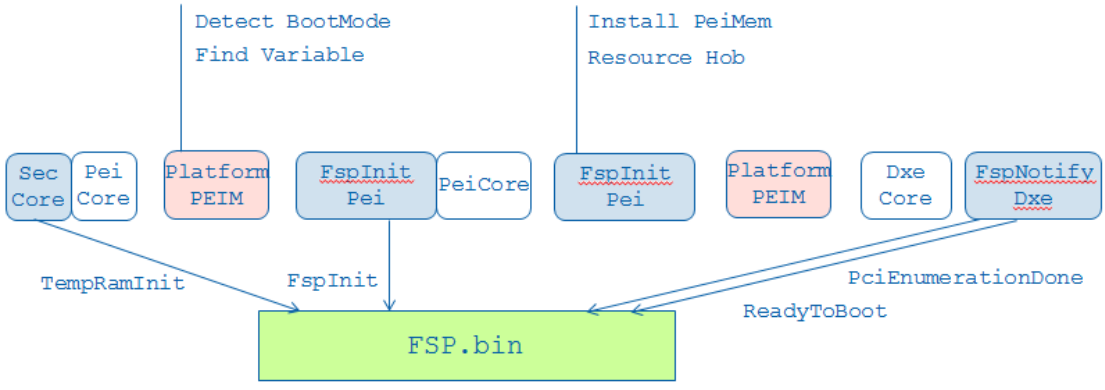

Figure 6 FSP normal boot flow

Boot Flow in FSP 1.1

In FSP 1.1 boot flow, FspInitPei module (https://github.com/tianocore/edk2-IntelFspWrapperPkg/blob/master/FspInitPei/FspInitPeiV2.c) will call FspMemoryInit, install PeiTemporaryRamDonePpi, and register PeiMemoryDiscoveredNotify. Once PEI core gets permanent memory, it will do TemporaryRam migration and call PeiTemporaryRamDonePpi, where TempRamExitApi is called. Then PeiCore installs PeiMemoryDiscovered, so that

FspSiliconInitApi is called in PeiMemoryDiscovedNotify. There is no need to let FspInitPei enter PEI Core again, because all PEI context saved in CAR is migrated by PEI core and CAR teardown in TempRamExitApi is invoked by FspInitPei.

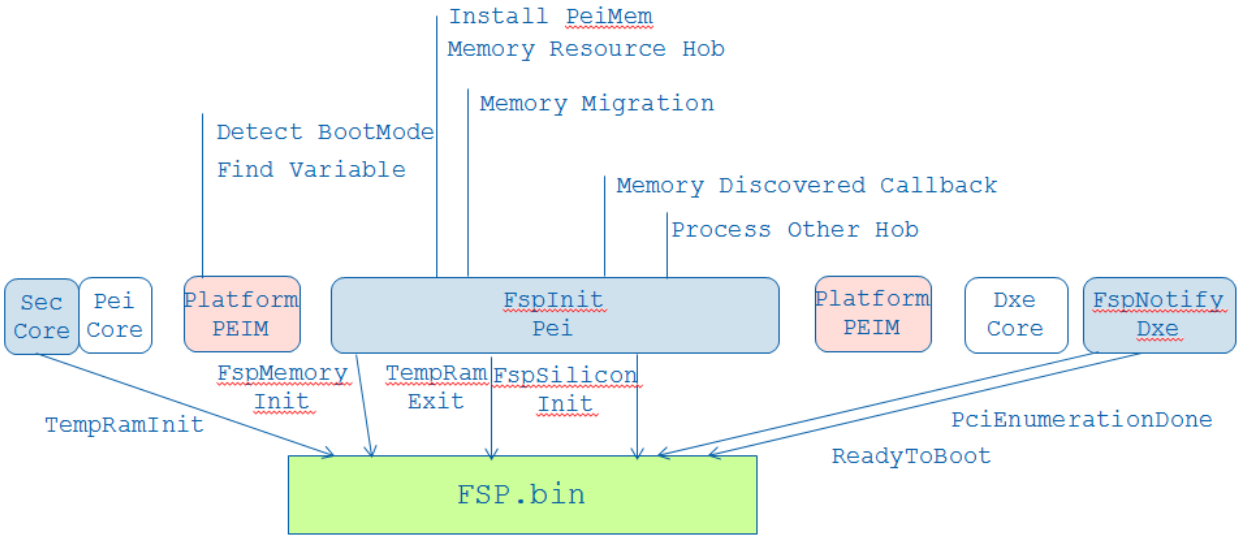

Figure 6.1 FSP normal boot flow 2

Memory Layout

The memory layout for FSP normal boot is below. The left hand side is component on flash and the temporary memory, such as cache as RAM. The right hand side is the DRAM layout. The GREEN part is for FSP.BIN. The BLUE part is for EDKII BIOS.

When SecCore calls TempRamInitApi, FSP binary will setup CAR, and use part of them, and leave rest of these activities to the EDKII BIOS. This CAR information will be reported as a return parameter of TempRamInitApi. (See left bottom)

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 ContinueFunction provided by FspInitPei, with the stack pointed to DRAM (Because CAR is destroyed). (See right bottom)

In FspInitPei, the ContinueFunction will launch second SecCore, with temp ram pointed to DRAM. The second SecCore will launch same PeiCore and continue dispatch PEI firmware volume (See left top)

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

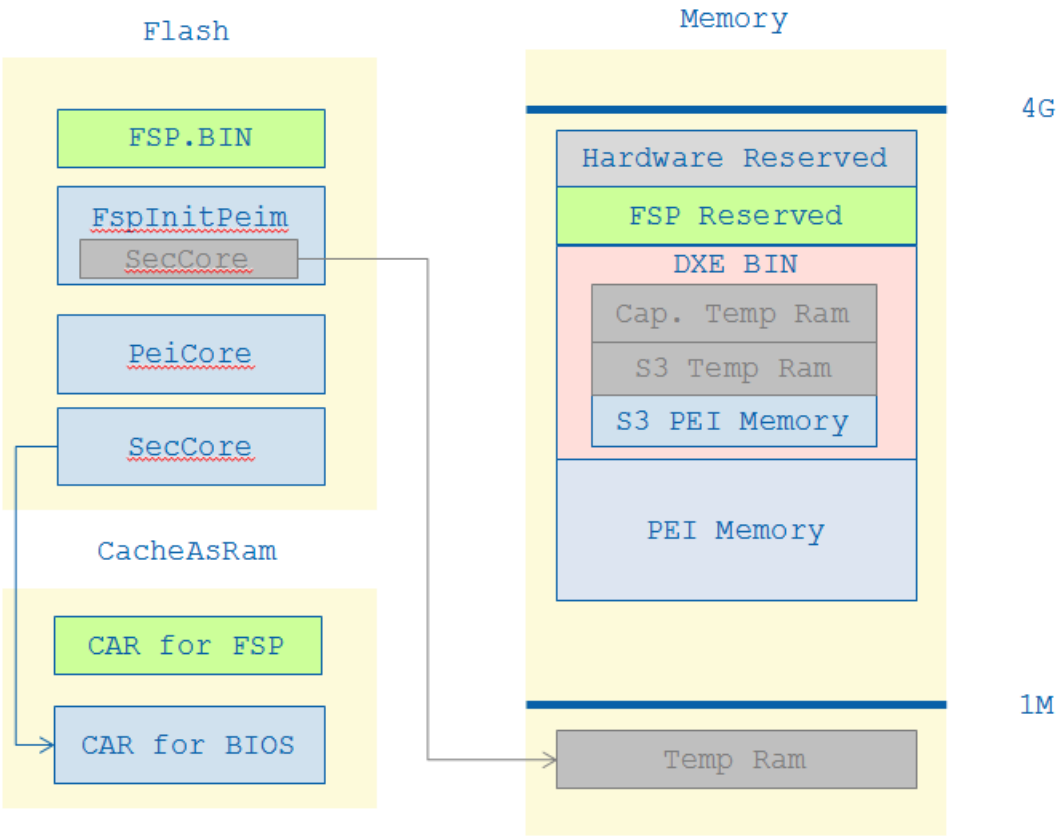

Figure 7 FSP normal boot memory layout

Memory Layout in FSP 1.1

In FSP 1.1 boot flow 2, there is no need to run SecCore in FspInitPeim, so temp ram is not needed. Gray part in Figure 7 does not exist. Because PEI Core will handle temporary RAM migration, the stack and head in temporary RAM will be copied to PEI permanent memory. (See below figure 7.1). PI specification defines 2 ways on memory migration. 1) If a platform provides PeiTemporaryRamSupportPpi, the migration will be done by TemporaryRamMigration() API defined in this PPI. 2) If a platform does not provide PeiTemporaryRamSupportPpi, the migration will be done by PEI Core. Even if there are some memory regions in CAR not covered by temporary RAM stack or heap, these regions (named as hole) will still be copied during migration. The reason is that there might be some data referred as PPI, produced in platform SEC phase. The PPI data pointer to the hole will be converted to new hole region after migration.

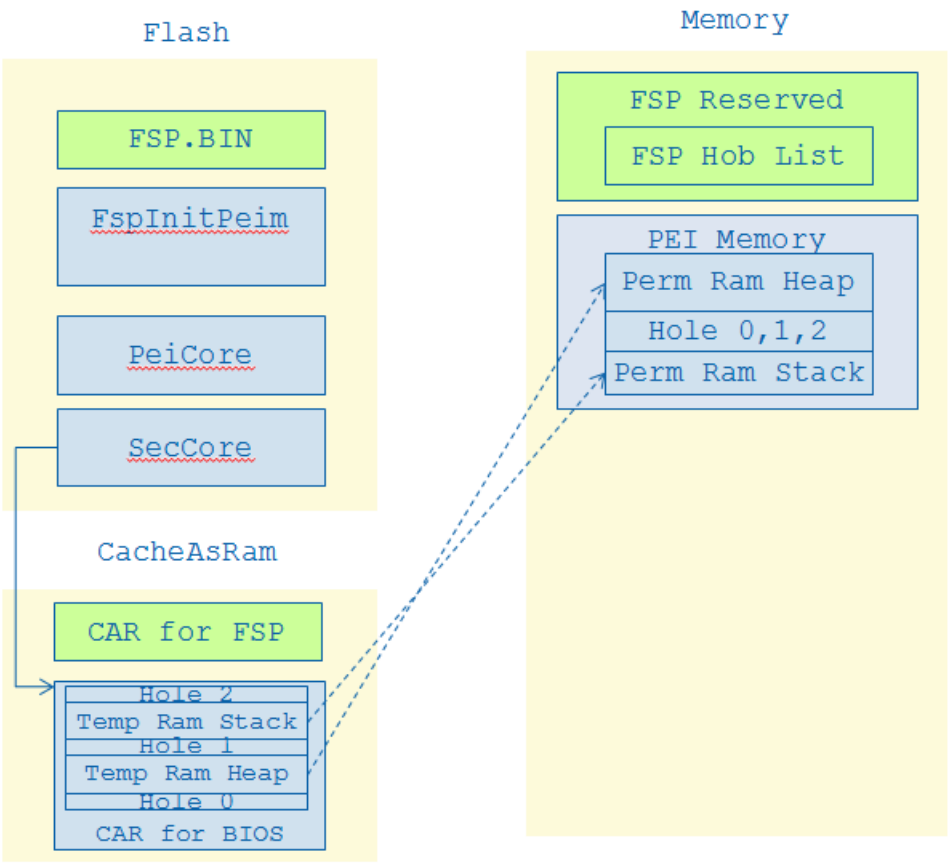

Figure 7.1 FSP 1.1 normal boot memory layout

Data Structure

According to the above description, there are 2 SecCore's involved. The first one is the normal SecCore, and the second one is a small SecCore inside FspInitPei. So how the first SecCore pass information to second one, like BIST data, boot time ticker needed in FPDT [ACPI]?

In IntelFspWrapperPkg, the first SecCore can save BIST and ticker in CAR (https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/Library/SecPeiFspPlatformSecLibSample). Before FspInitApi called, the platform may choose to save them in some special registers not touched by FSP.bin. Example could be IA CPU MM register, or PCI scratch register. After FspInitApi, the FspInitPei launch second SecCore, which will restore the information from special registers to new stack in temp ram. The second SecCore also register a special TopOfTemporaryRam PPI (aka, TopOfCar Ppi in below picture), which has pointer to top of temp ram. (See below bottom)

The reason to introduce TopOfTemporaryRam PPI is that the FspInitPei need a way to get FSP Hob List, while the FspHobList is saved to the top of temp Ram. Also, BIST and Ticker are saved on the top of temp ram. It becomes easy to know the information by having a PPI to tell the temp ram location.

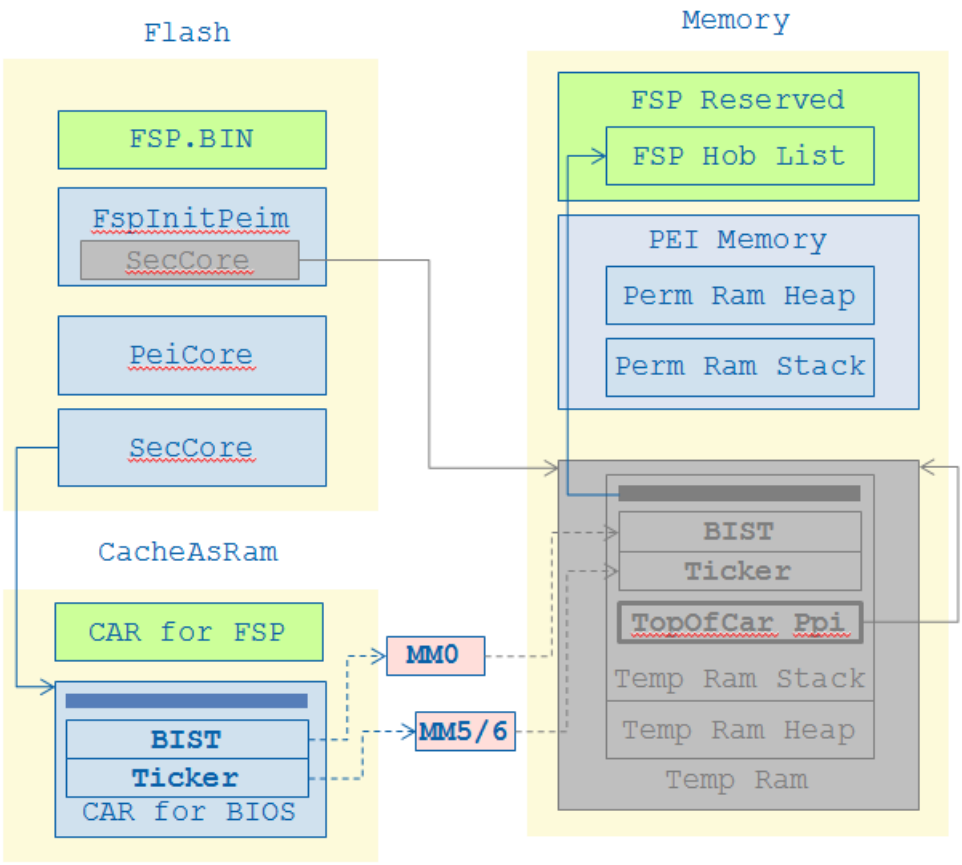

Figure 8 FspInitPei data structure

Data Structure in FSP 1.1

In FSP 1.1 boot flow 2, the data structure is much simpler. Gray part in Figure 8 does not exist. The memory migration is controlled by PEI Core as normal UEFI/PI BIOS, which is described in "Memory Layout in FSP 1.1"

Dual FSP Support

Dual FSP means there could be two FSP images at different locations in a flash - one factory version (default) and an updatable version (updatable). TempRamInit, FspMemoryInit and TempRamExit are always executed from the factory version. FspSiliconInit and NotifyPhase can be executed from the updatable version if it is available; FspSiliconInit and NotifyPhase are executed from the factory version if there is no updateable version.

In order to support Dual FSP, the user just needs to define PcdFlashFvSecondFspBase and PcdFlashFvSecondFspSize to updatable FSP version. The previous PcdFlashFvFspBase and PcdFlashFvFspSize designates the factory FSP version.
(https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/IntelFspWrapperPkg.dec).
Then FspInitPei and FspNotifyDxe will use the PcdFlashFvSecondFspBase to call FspSiliconInit and NotifyPhase.

Summary
This section describes the FSP wrapper boot flow in normal boot mode.

S3 Boot

Boot Flow

In S3 boot, the difference is when to call FspNotifyApi. In normal boot mode, it happens in DXE phase, but in the S3 boot mode there is no DXE.

In IntelFspWrapperPkg, FspInitPei (https://github.com/tianocore/edk2-IntelFspWrapperPkg/blob/master/FspInitPei/FspNotifyS3.c) will register EndOfPei callback in S3 boot mode. So when boot script finishes execution, FspNotifyApi will be invoked, then system enter OS waking vector.

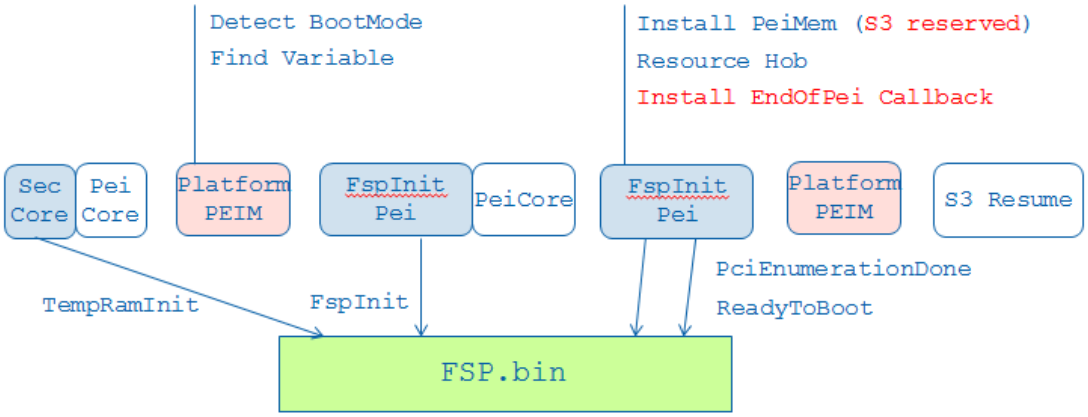

Figure 9 FSP S3 boot flow

Boot Flow in FSP 1.1

Boot flow 2 in FSP 1.1 is same as boot flow 1 in S3 phase. FspInitPei registers EndOfPei callback in S3 boot mode, and FspNotifyApi will be invoked there.

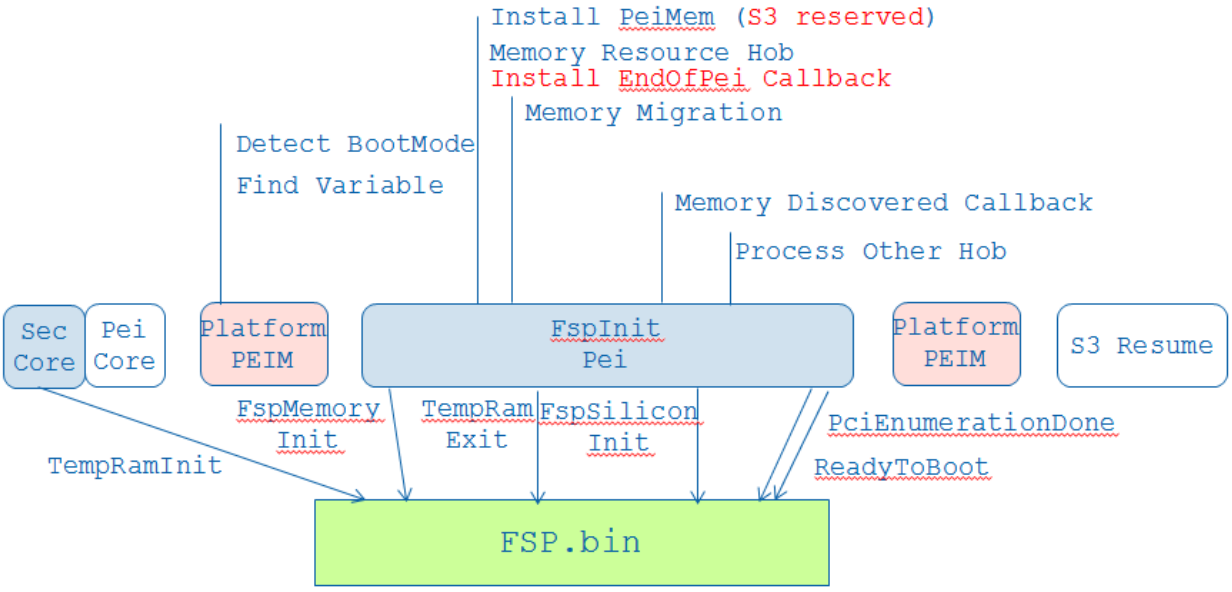

Figure 9.1 FSP 1.1 S3 boot flow

Memory Layout

In S3 boot, the difference memory layout is temp ram location. In normal boot mode, it is at some low DRAM, configured by PCD, which is used by no one at PEI phase. In S3 boot, usable DRAM is owned by OS, expected the one reported as ACPI reserved or ACPI NVS.

In normal boot DXE phase, a platform driver should allocate S3 temp ram, mark it as reserved to OS. Then in S3 phase, the FspInitPei can use it as temp ram for continue function.

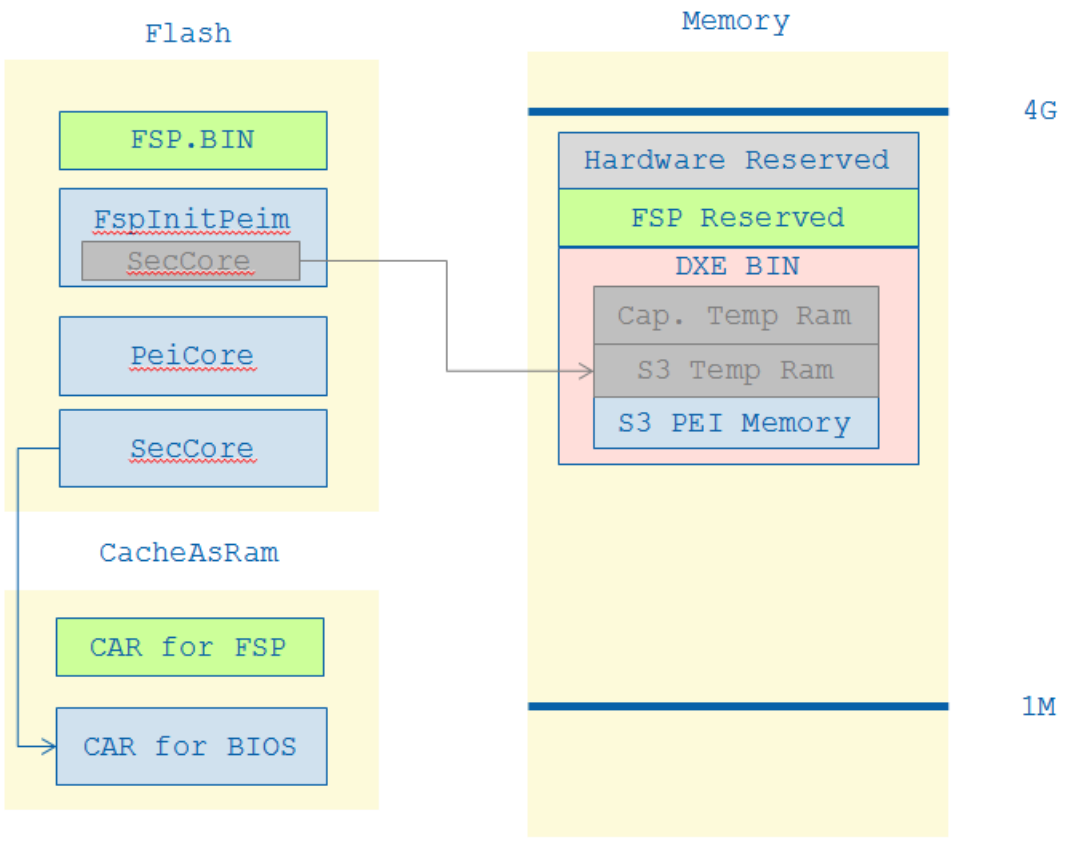

Figure 10 FSP S3 boot memory layout

Memory Layout in FSP 1.1

In FSP 1.1 boot flow 2, there is no need to run SecCore in FspInitPeim, so S3 temp ram is not needed. Gray part in Figure 10 does not exist.

S3 NV Data Passing

In some platforms, S3 phase initialization needs configuration saved in a normal boot. Below is an example on how memory configuration data is passed from the MRC module in normal boot to MRC module in S3.

In a normal boot, the FSP MRC module produces a MemoryConfigData hob and saves it in the FSP hob list, and the FSP hob list 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 save MemoryConfigData into NV variable.

In S3 boot, the FSP PEI module finds the MemoryConfigData from a 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.

The place to get UPD data is at FspPlatformInfoLib. The template is at https://github.com/tianocore/edk2-IntelFspWrapperPkg/tree/master/Library/BaseFspPlatformInfoLibSample

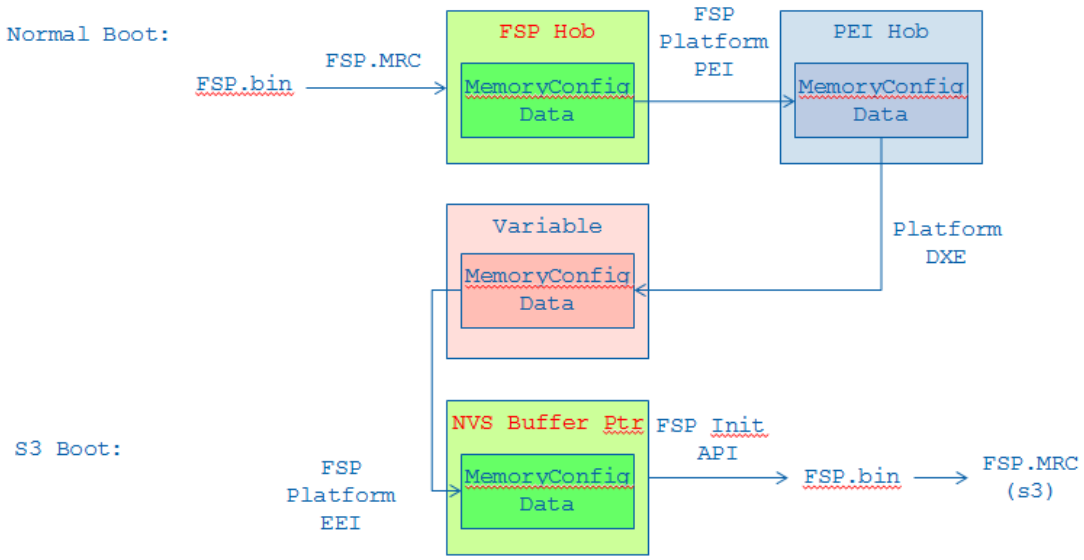

Figure 11 FSP S3 NVS data passing

Summary

This section describes the FSP wrapper boot flow in S3 boot mode.

Capsule Flash Update

Boot Flow

In capsule update boot, there is only small difference: FspInitPei need call CapsuleCoalesce before install PEI memory, and it need install PEI memory for capsule update mode. (The size and location might be different with normal boot mode)

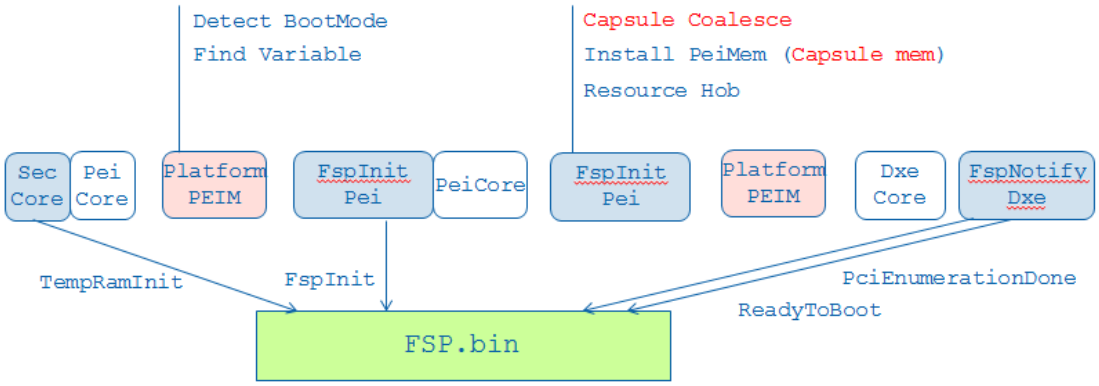

Figure 12 FSP capsule update boot flow

Boot Flow in FSP 1.1

Boot flow 2 in FSP 1.1 is same as boot flow 1 in capsule update. FspInitPei need call CapsuleCoalesce before install PEI memory, and it need install PEI memory for capsule update mode.

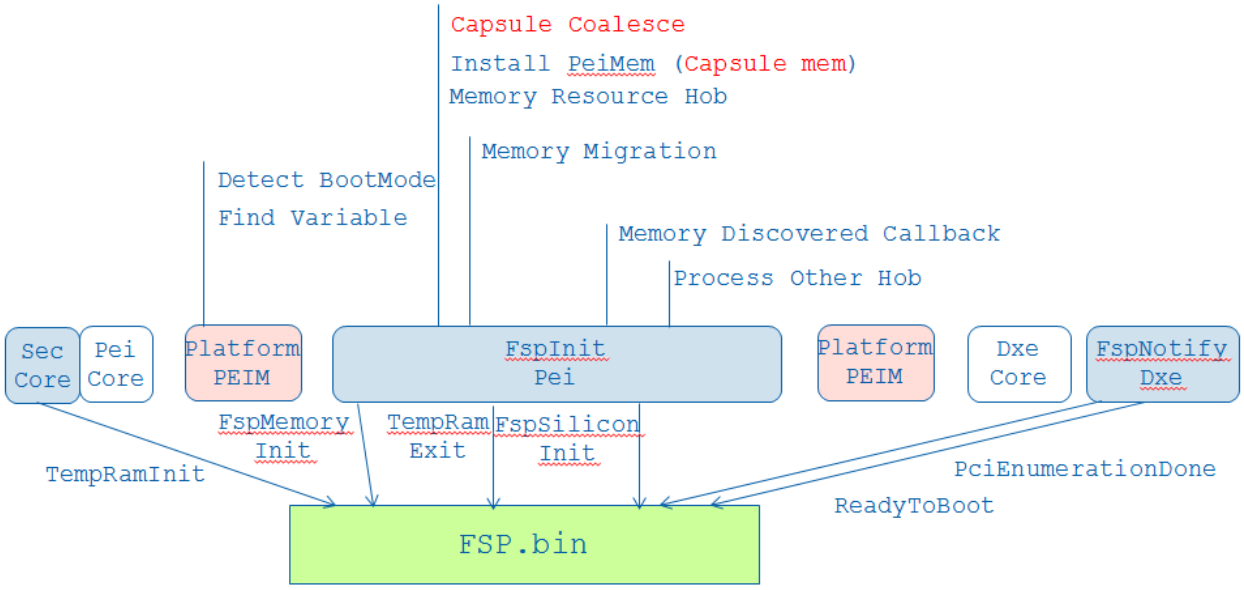

Figure 12.1 FSP 1.1 capsule update boot flow

Memory Layout

In capsule update boot, the difference memory layout is the temp ram location. In normal boot mode, it is at some low DRAM, configured by 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 to any usable DRAM.

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

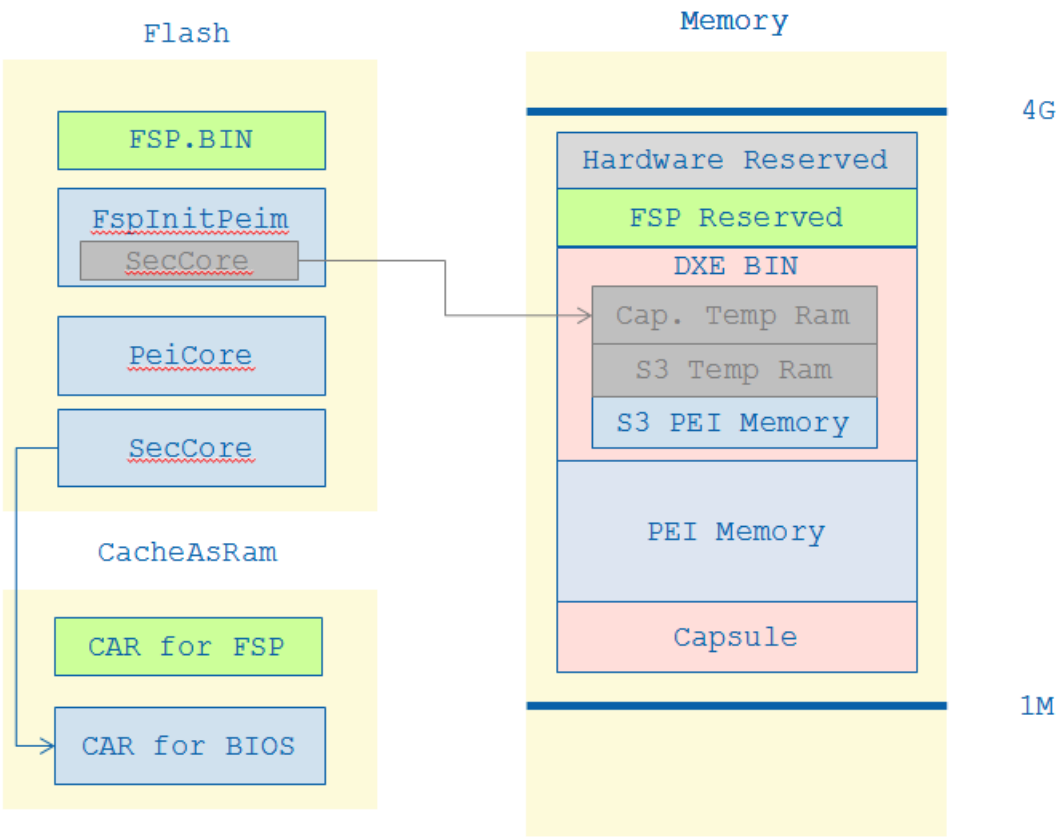

Figure 13 FSP capsule update boot memory layout

Memory Layout in FSP 1.1

In FSP 1.1 boot flow 2, there is no need to run SecCore in FspInitPeim, so capsule temp ram is not needed. Gray part in Figure 13 does not exist.

Summary

This section describes the FSP wrapper boot flow in capsule update boot mode.

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)

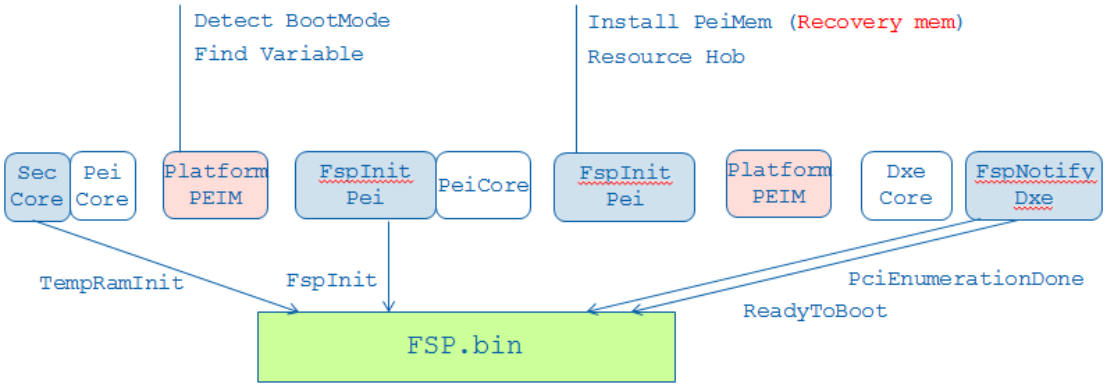

Figure 14 FSP recovery boot flow

Boot Flow in FSP 1.1

Boot flow 2 in FSP 1.1 is same as boot flow 1 in recovery. FspInitPei needs to install PEI memory for recovery mode.

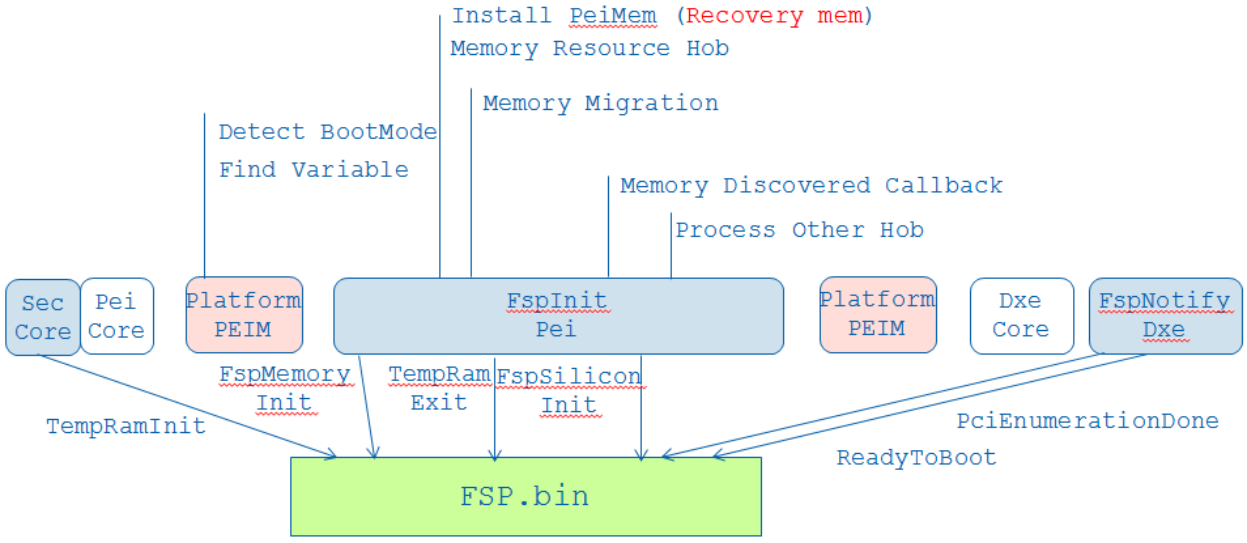

Figure 14.1 FSP 1.1 recovery boot flow

Memory Layout

In recovery boot, the memory layout is the same as normal boot mode.

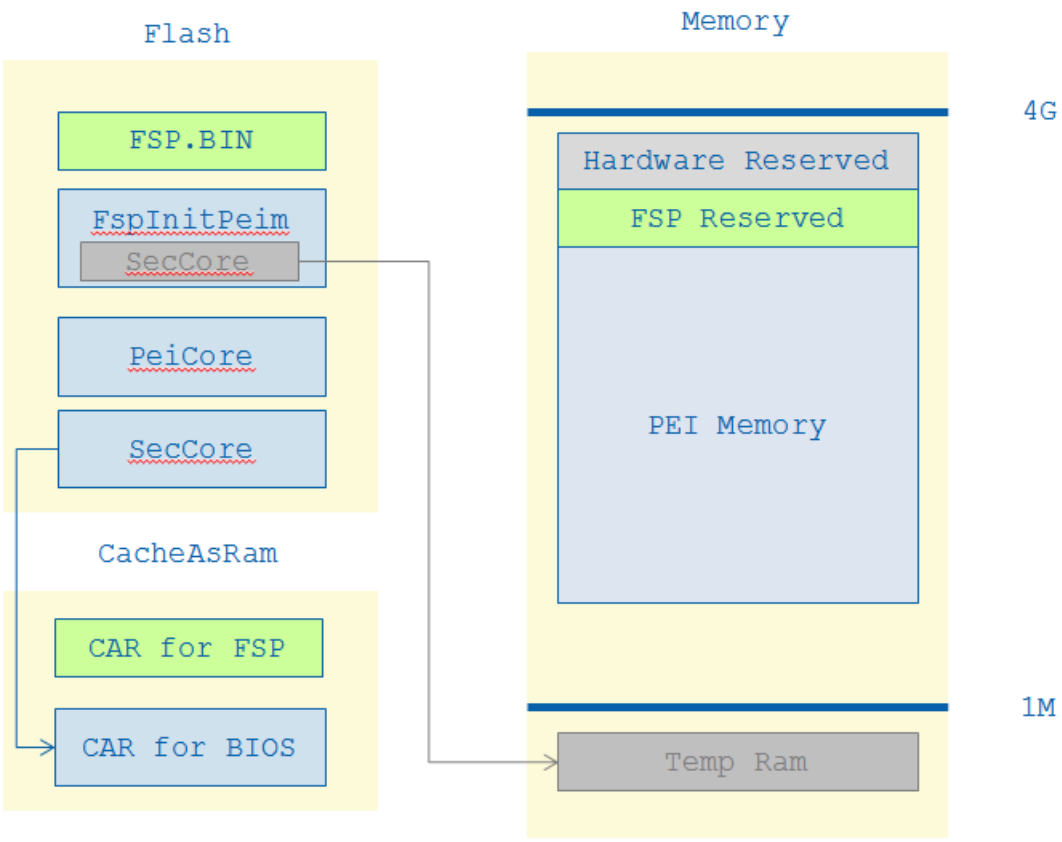

Figure 15 FSP recovery boot memory layout

Memory Layout in FSP 1.1

In FSP 1.1 boot flow 2, there is no need to run SecCore in FspInitPeim, so temp ram is not needed. Gray part in Figure 15 does not exist.

Summary

This section describes the FSP wrapper boot flow in recovery boot mode.

Conclusion

FSP provides a simple to integrate solution that reduces time-to-market, and it is economical to build. IntelFspWrapperPkg is the FSP consumer in EDKII to support building out a UEFI BIOS. This paper describes detail work flow and data structure in IntelFspWrapperPkg.

Glossary

ACPI – Advanced Configuration and Power Interface. Describe system configuration that is not discoverable and provide runtime interpreted capabilities

CAR – Cache-As-RAM. Use of the processor cache as a temporary memory / stack store

FPDT –Firmware Performance Data Table defined in ACPI specification.

FSP –Intel Firmware Support Package

FSP Consumer – the entity that integrates the FSP.bin, such as EDKII or other firmware like coreboot

FSP Producer – the entity that creates the FSP binary, such as the CPU and chipset manufacturer (e.g., “Silicon Vendor”).

Bootloader – another name for an “FSP Consumer”, as distinct from a MBR-based loader for PC/AT BIOS or the OS loader as a UEFI Executable for UEFI [UEFI Overview]

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system.

References

[ACPI] Advanced Configuration and Power Interface, vesion 6.0, www.uefi.org

[COREBOOT] coreboot firmware www.coreboot.org

[EDK2] UEFI Developer Kit www.tianocore.org

[FSP] Intel Firmware Support Package http://www.intel.com/content/www/us/en/intelligent-systems/intel-firmware-support-package/intel-fsp-overview.html

[FSP EAS] FSP External Architecture Specification, Version 1.1, April 2015 http://www.intel.com/content/www/us/en/embedded/software/fsp/fsp-architecture-spec-v1-1.html

[FSP Producer] Yao, Zimmer, Rangarajan, Ma, Estrada, Mudusuru, "A_Tour_Beyond_BIOS_Creating_the_Intel_Firmware_Support_Package_with_the_EFI_Developer_Kit_II_(FSP1.1)" http://firmware.intel.com

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.5 www.uefi.org

[UEFI Book] Zimmer,, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specifications, volumes 1-5, Version 1.4 www.uefi.org

Authors

Jiewen Yao (jiewen.yao@intel.com) is EDKII BIOS architect, EDKII FSP package maintainer with Software and Services Group (SSG) at Intel Corporation.

Vincent J. Zimmer (vincent.zimmer@intel.com) is a Senior Principal Engineer with the Software and Services Group (SSG) at Intel Corporation based in Seattle, WA.

Ravi P. Rangarajan (ravi.p.rangarajan@intel.com) is BIOS architect in the Internet of Things (IOT) Group (IOTG) at Intel Corporation.

Maurice Ma (maurice.ma@intel.com) is BIOS architect in the Internet of Things (IOT) IOT Group (IOTG) at Intel Corporation.

David Estrada (david.c.estrada@intel.com) is BIOS architect in the Client Components Group (CCG) at Intel Corporation.

Giri Mudusuru (giri.p.mudusuru@intel.com) is BIOS architect and Principal Engineer in the Client Components Group (CCG) at Intel Corporation.