White Paper

A Tour Beyond BIOS Creating the Intel® Firmware Support Package Version 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 details on how to create an Intel® Firmware Support Package (FSP)-conformant [FSP EAS] binary by using in EDKII [EDK2]. After this "FSP Production", then the resultant Intel FSP binary can be integrated into any boot loader [FSP Consumer].

Prerequisite
This paper assumes that the audience has EDKII/UEFI firmware development experience. He or she should be familiar with UEFI/PI firmware infrastructure (e.g., SEC, PEI), 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.

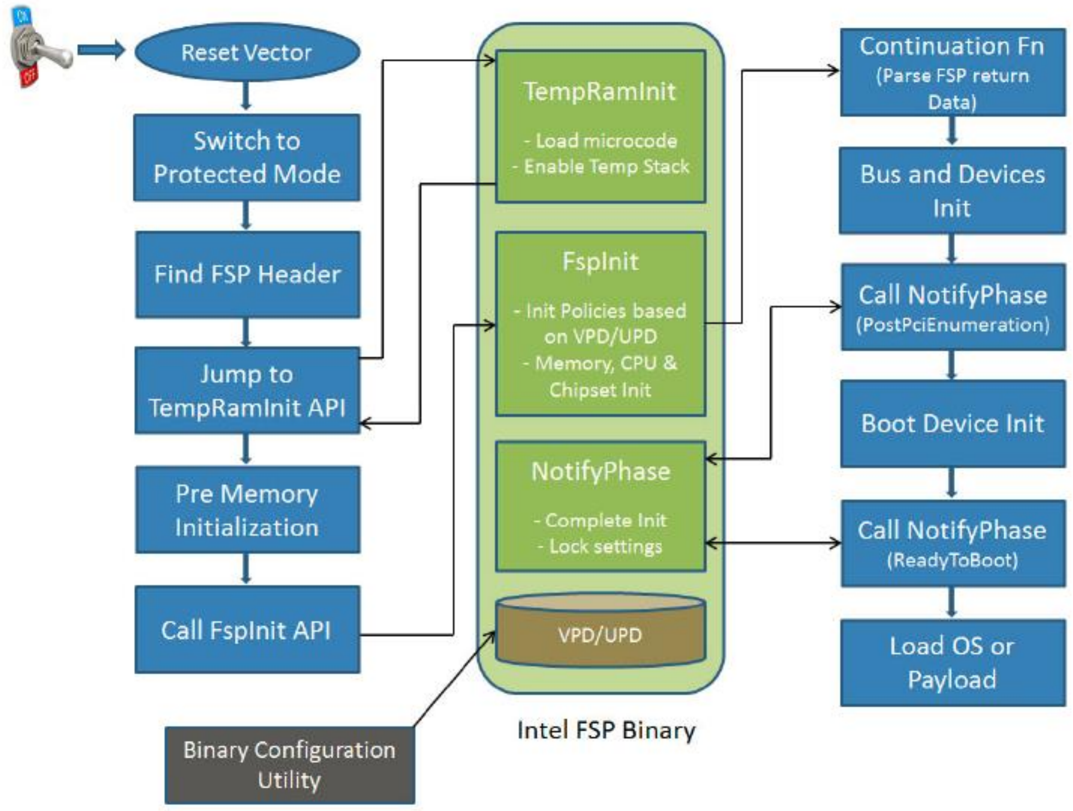

Figure 1 FSP architecture

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 topic of this paper is Intel FSP binary. We will discuss how to create Intel FSP binary (the producer) by using EDKII environment 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.

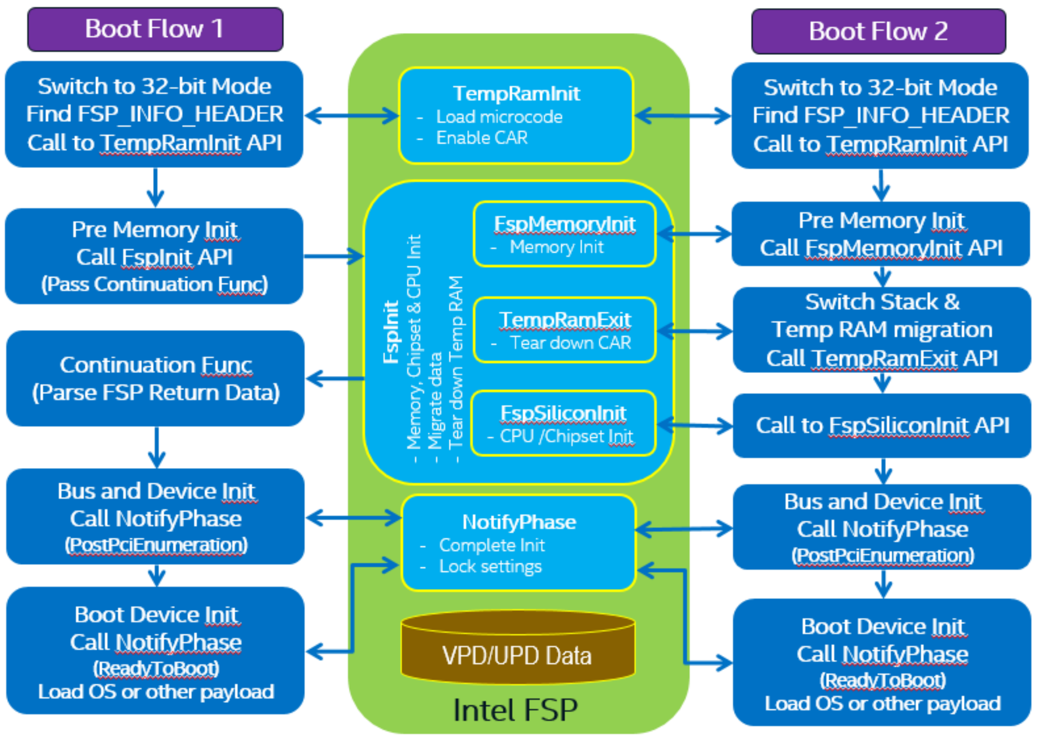

Figure 2.1 FSP 1.1 architecture

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 a 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 producer module, to build an FSP binary.

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 IntelFspPkg on how to use IntelFspPkg to create FSP.bin. This paper will not describe IntelFspWrapperPkg on how it consumes FSP.bin, which is described in [FSP Consumer].

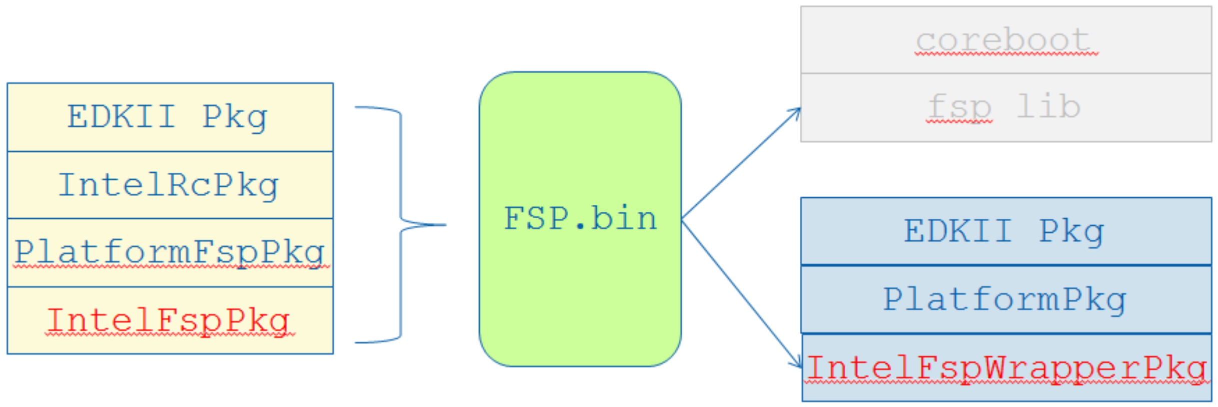

Figure 2 FSP component

Summary
This section describes the FSP component in EDKII.

FSP infrastructure

Binary layout

According to the FSP EAS, an FSP binary contains:

1) FSP_INFO_HEADER structure providing information about FSP.
   It can be found in firmware section data of 1st firmware file in the FSP firmware volume.

2) Initialization code and data needed by the Intel silicon supported.
   The silicon initialization processes are exposed by APIs defined in FSP_INFO_HEADER. For example, FSP_INFO_HEADER has offset of FspInit API. The caller can find the API offset and use C-style function to call FspInit, then after return the silicon initialization work is done.

3) Configuration region that allows the bootloader developer to customize some of the settings.
   The configuration region is exposed by CfgRegionOffset in FSP_INFO_HEADER.

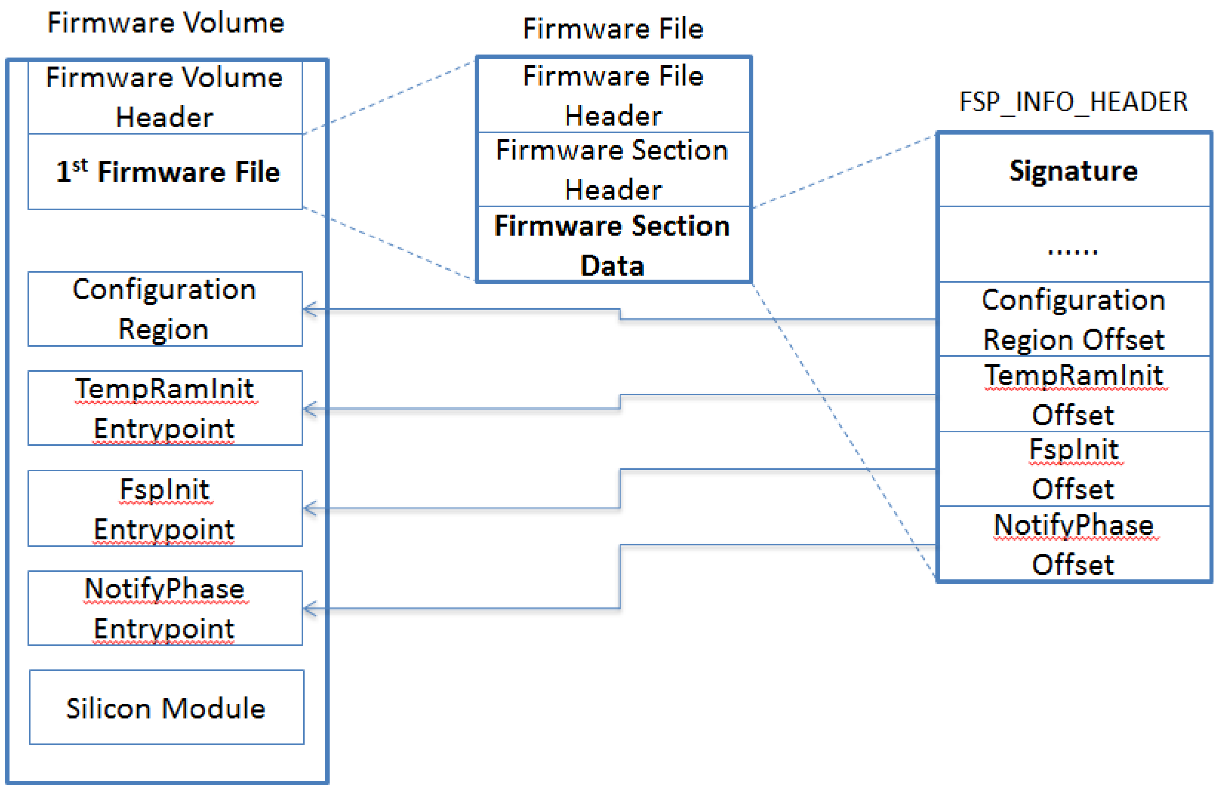

Figure 3 FSP binary layout

FSP runtime data structure

Current FSP binary is built in flash region and can be executed in flash region. On SPI NOR flash, those data can be accessed directly via MMIO.

In FSP_INFO_HEADER, (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Include/FspInfoHeader.h) ImageBase is a pointer to FSP binary base address. CfgRegion is a pointer to configuration region. And there are 6 offset of API entrypoint in FSP header. (TempRamInit, FspInit, NotifyPhase, FspMemoryInit, TempRamExit, FspSiliconInit). Those entrypoint will be invoked during boot. (We will discuss detail API flow in next section, and we will discuss how to build FSP_INFO_HEADER in "step 4 – Build with IntelFspPkg")

The configurable data region has two sets of data:

1) VPD – Vital Product Data, which can only be configured statically.
2) UPD – Updatable Product Data, which can be configured statically for default values, but also can be overwritten during boot at runtime.

Both the VPD and the UPD parameters can be statically customized using a separate tool. There will be a Boot Setting File (BSF) provided along with FSP binary to describe the configuration options within the FSP. (We will discuss detail on how to expose silicon configuration in "step 3 – Expose silicon configuration")

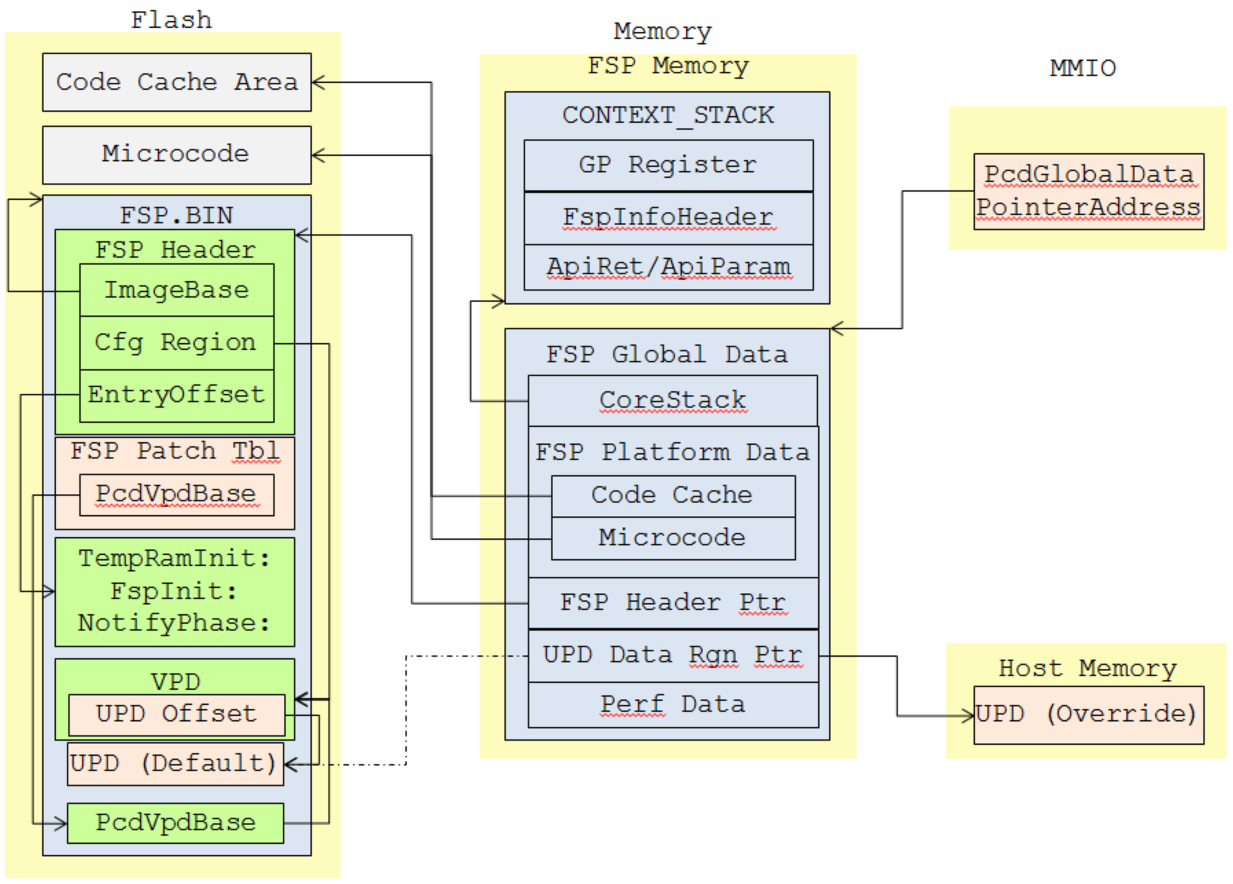

Figure 4 FSP runtime data structure

During runtime, FspSecCore (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/FspSecCore) will setup FSP Global Data. The Global Data area contains below important data structure:

1) **CoreStack** – This is context stack used by FSP. The context includes all general purpose register. So that the FSP API can do SetJump/LongJump-like context switch to original FSP API caller provided stack, from FSP internal stack.
2) **PlatformData** – This area record the Microcode and CodeCache region passed from TempRamInit API. Also an FSP specific implementation can save its own private data pointer to “DataPtr” field of PlatformData.
3) **FspInfoHeader** – This is pointer to FSP binary on flash.
4) **UpdDataRgnPtr** – This is pointer to user provided UPD region to override the default UPD data region in VPD. If caller provides valid value for UpdDataRgnPtr of FSP_INIT_RT_COMMON_BUFFER, this new UPD will be used. If UpdDataRgnPtr is NULL, the FSP will assume default UPD at UpdRegionOffset (0x0C) of VPD CfgRegion. Every FSP implementation should follow this way to put UPD data pointer there.
5) **API Mode** – 0 means FSP1.0 flow. 1 means FSP1.1 flow.
6) **PerfData** – Record the performance data. Totally 32 entry available.

The location of this Global Data area is recorded into PcdGlobalDataPointerAddress. (defined in https://github.com/tianocore/edk2-IntelFspPkg/tree/master/IntelFspPkg.dec) In most current platforms, it is an MMIO based scratch register.

FSP API Flow

After system power on, bootloader will call FSP **TempRamInit** API in FspSecCore (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/FspSecCore/Ia32/FspApiEntry.asm), before memory and stack are available. This FSP API loads the microcode update, enables code caching for a region specified by the bootloader and sets up a temporary stack to be used prior to main memory being initialized.

Then bootloader setups stack in temporary ram, run C code, and call **FspInit** API. This FSP API initializes the memory, the processor and the chipset to enable normal operation of these devices. It starts from FspSecCore, then call PeiCore in FSP binary. (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/FspSecCore/SecMain.c) Then PeiCore dispatches all silicon PEI modules one by one. After all modules are dispatched, PeiCore will invoke EFI_DXE_IPL_PPI.Entry.

The FspDxeIpl (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/FspDxeIpl) is different with normal DxeIpl. After FspDxeIpl installs EndOfPei, it means FspInit is done. Then FspDxeIpl will use switch stack to 1) save current execution context, 2) jump back to bootloader stack, and return to bootloader.

Then bootloader continues device and bus initialization then notify the **NotifyPhase** FSP API on “AfterPciEnumeration” phase, or “ReadyToBoot” phase. The entrypoint of NotifyPhase API is still in FspSecCore, but it will use switch stack again, to 1) save caller context, 2) jump to previous context – FspDxeIpl. Then FspDxeIpl can continue running code to notify silicon module on “AfterPciEnumeration” or “ReadyToBoot”.

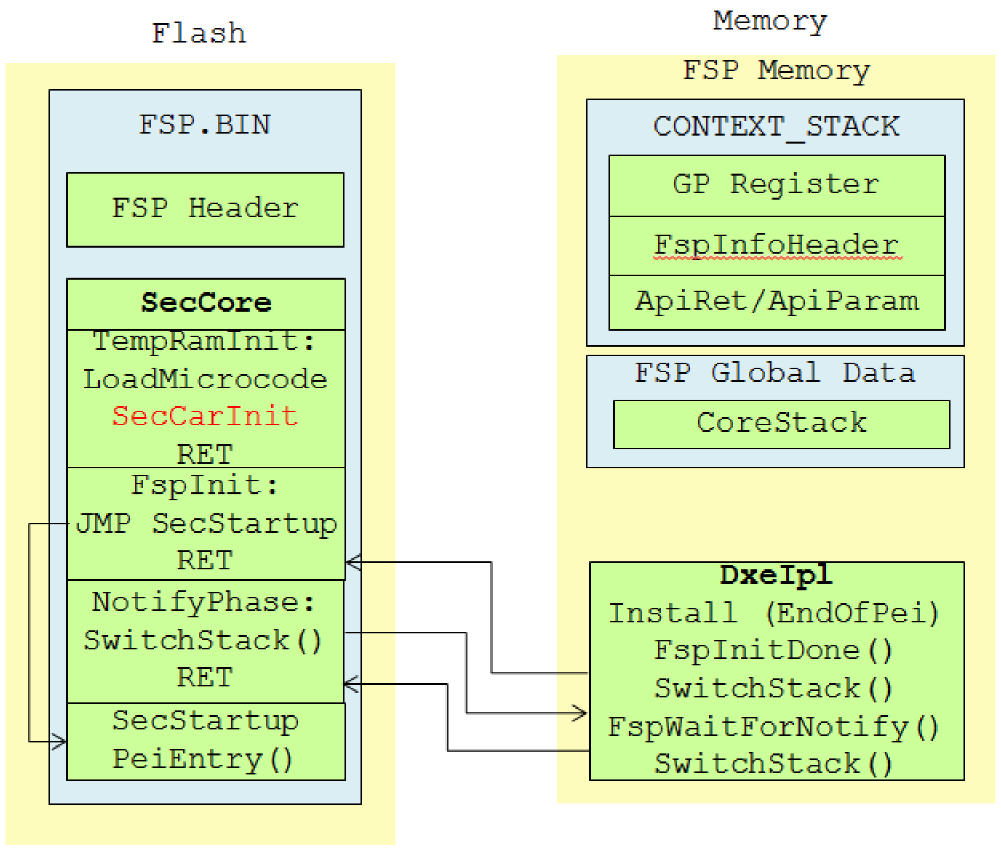

Figure 5 FSP API Flow

FSP API Flow in FSP 1.1

Between FSP1.0 and FSP1.1, the major difference of API flow is that FspInit API is split to 3 – FspMemoryInit, TempRamExit, and FspSiliconInit. One potential problem of FSP1.0 is that FspInit will teardown temporary ram unconditionally, so the previous context in temporary ram cannot be preserved.

FSP1.1 resolved the problem. After **FspMemoryInit** API initializes the memory subsystem, it gives control back to bootloader and let bootloader migrate its stack and heap data from temporary ram to system memory. Then bootloader can call **TempRamExit** API to teardown temporary ram setup by TempRamInit API.
Finally, bootloader need call **FspSiliconInit** API to initialize the processor and the chipset including the IO controllers in the chipset to enable normal operation of these devices.

Since FSP need return to caller after FspMemoryInit, this work must be done in a platform defined **MemoryDiscoveredPpiNotifyCallback()** function. This function does switch stack back to bootloader after memory initialization done. After bootloader calls 2nd API - TempRamExit in SecCore, this API switches back to MemoryDiscoveredPpiNotifyCallback, run next instruction to reset cache, and switches back to bootloader again. Then bootloader call 3rd

API – FspSiliconInit in SecCore, it switches back to MemoryDiscoveredPpiNotifyCallback, run rest code of system initialization.

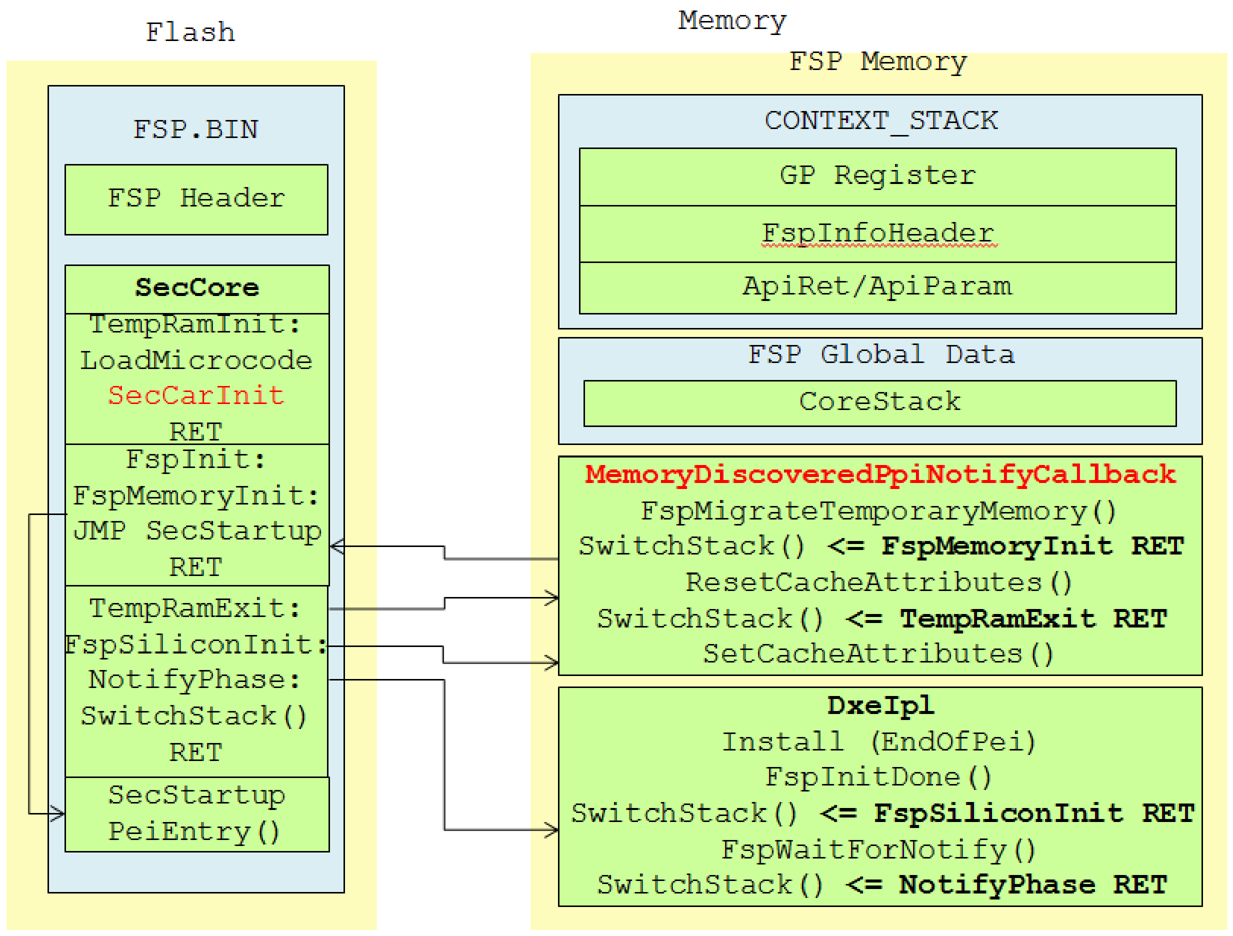

Figure 6 FSP API Flow in FSP 1.1

Summary

This section has a generic overview of FSP binary infrastructure, including FSP binary layout, runtime data structure, and API flow for both 1.0 and 1.1.

In next sections, we will discuss detail step by step on how to create a FSP binary and release it.

Step 1 – Put silicon initialization module to PI PEI

Silicon code V.S. Generic code V.S. Platform code

Since FSP is for silicon initialization, so the first step is to find what and where is silicon code.

There is one way to identify **silicon code**. If the code is written according to Intel silicon external design specification, most likely it is silicon code. For example, MemoryInit, CpuInit, SmartTimer, and Smbus are silicon code. The silicon code might need rewritten if silicon hardware is upgraded. But it should be same if only platform or board is changed.

Generic code means the code can be used on almost silicon or platform. For example, PEI Core, PCI Bus, and USB bus are generic code.

Platform code is the code binding to platform board hardware. For example, GPIO setting and ACPI table are platform code. Platform code can also be generic code used for some platforms, but not for all platforms. For example, EfiVariable and SetupBrowser driver are also platform code.

Here for FSP, we only focus on first category "**silicon code**".

Silicon Initialization Module V.S. Silicon Function Module

There are 2 types of silicon code – initialization module and function module.

Initialization module means the module to initialize memory subsystem, processor and the chipset including the IO controllers in the chipset to enable normal operation of these devices. For example, MemoryInit and CpuInit are initialization modules. The code runs once. After initialization, the code is never used any more.

Function module means the module to provide functionality of chipset. For example, SmartTimer provides EFI_TIMER_ARCH_PROTOCOL service abstraction, Smbus provides EFI_PEI_SMBUS2_PPI service abstraction. These services are always available in memory after they are provided.

Here for FSP, we only focus on first category "**silicon initialization module**".

Put silicon initialization module to PEI

After we find out silicon initialization modules, we need put these modules into PEI phase. For UEFI/PI BIOS, there might be some cases that the original initialization code is in DXE phase. For example, part of Platform Control Hub (PCH) initialization and System Agent (SA) initialization are in DXE phase. If so, these module need to be ported from PEI to DXE.

Summary
This section describes how to find silicon initialization modules and put them into PEI phase.

Step 2 – Expose silicon configuration

Expose configuration - VPD/UPD

Silicon initialization might need some configuration or policy. For example, SMRAM TSEG size, or UART debug enable/disable. These data can be set in FSP configuration region.

The configurable data region has two sets of data:

1) VPD – Vital Product Data, which can only be configured statically.
2) UPD – Updatable Product Data, which can be configured statically for default values, but also can be overwritten during boot at runtime.

The silicon vendor can create VPD or UPD based on the need. In most case, if a configuration can be set by end user via bootloader setup page, it should be in UPD region. For example, SMRAM TSEG size can be changed from 8M to 16M by user for validation purpose.
If a configuration does not need to be changed by end user, but bootloader maker, it can be in VPD region. For example, UART debug enable/disable can be chosen by bootloader maker, but not end user. See below sample:

```
PlatformFsp.dsc:
======================================
[PcdsDynamicVpd]
# VPD Region Signature "FSB_VPDS"
# Need to be in sync with FspHeader.ImageId
# The first 7 bytes should match with ImageId, the last byte should be
# different to avoid multiple instance matches
#
# !BSF FIND:{$SI_VPD$}
gSiFspPkgTokenSpaceGuid.PcdVpdRegionSign   | 0x0000

# VPD Region Revision
# !BSF NAME:{PcdImageRevision}  TYPE:{None}
gSiFspPkgTokenSpaceGuid.PcdImageRevision   | 0x0008

# This is a offset pointer to the UCD regions used by FSP
# The offset will be patched to point to the actual region during the
build process
#
gSiFspPkgTokenSpaceGuid.PcdUpdRegionOffset | 0x000C

# !BSF NAME:{PcdSerialIoUartDebugEnable} TYPE:{Combo}
# !BSF OPTION:{$EN_DIS}
# !BSF HELP:{Enable SerialIo Uart Debug.}
gSiFspPkgTokenSpaceGuid.PcdSerialIoUartDebugEnable|0x0030

[PcdsDynamicVpd.Upd]
# !BSF FIND:{$SI_UPD$}
gSiFspPkgTokenSpaceGuid.Signature|0x0000|0x08|0x245053464C4B5324
gSiFspPkgTokenSpaceGuid.Reserved |0x0008|0x08|0x0000000000000000
```

```
# !BSF NAME:{Tseg Size} TYPE:{Combo}
# !BSF OPTION:{$EN_DIS}
# !BSF HELP:{Size of SMRAM memory reserved.}
gSiFspPkgTokenSpaceGuid.TsegSize|0x0100|0x04|0x01000000
======================================
```

The first section defines VPD data, and the second section defines UPD data.
NOTE: It is required to put PcdUpdRegionOffset at offset 0x000C for all FSPs.

```
PlatformFsp.fdf:
======================================
FILE RAW = 12345678-1234-1234-1234-1234567890AB {
    SECTION RAW = $(OUTPUT_DIRECTORY)/$(TARGET)_$(TOOL_CHAIN_TAG)/FV/$(VPD_TOOL_GUID).bin
    SECTION RAW = $(OUTPUT_DIRECTORY)/$(TARGET)_$(TOOL_CHAIN_TAG)/FV/$(UPD_TOOL_GUID).bin
}
======================================
```

In FDF file, we put VPD binary and UPD binary together into one file section.

Prebuild - GenCfgOpt

After VPD/UPD is defined in dsc file, GenCfgOpt.py tool (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Tools/GenCfgOpt.py) will create corresponding VPD/UDP header file for bootloader developer.

The content in VpdHeader file is like below:

```
VpdHeader.h:
======================================
typedef struct _VPD_DATA_REGION {
  UINT64    PcdVpdRegionSign;              /* Offset 0x0000 */
  UINT32    PcdImageRevision;              /* Offset 0x0008 */
  UINT32    PcdUpdRegionOffset;            /* Offset 0x000C */
  ......
  UINT8     PcdSerialIoUartDebugEnable;    /* Offset 0x0030 */
  ......
} VPD_DATA_REGION;

typedef struct _UPD_DATA_REGION {
  UINT64    Signature;                     /* Offset 0x0000 */
  UINT64    Reserved;                      /* Offset 0x0008 */
  ......
  UINT32    TsegSize;                      /* Offset 0x0100 */
  ......
} UPD_DATA_REGION;
======================================
```

This GenCfgOpt.py tool runs before formal EDKII build. That is why we call it as prebuild tool. For more detail, please refer to user manual (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Tools/UserManuals/GenCfgOptUserManual.docx)

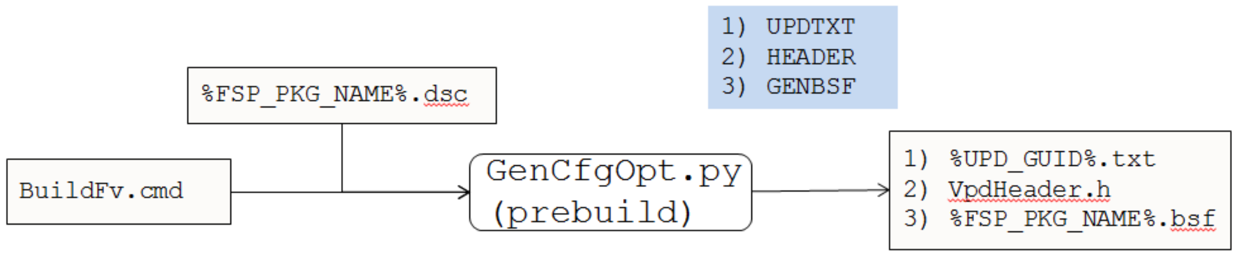

Figure 7 Prebuild tool – GenCfgOpt.py

User configuration - BSF/BCT

GenCfgOpt.py too also creates BSF configuration file, user can use BCT tool (www.intel.com/fsp) to do configuration after FSP binary is generated.

The content in BSF file is like below.

```
FSP.bsf:
==========================================
/* VPD data region*/
Find "$SI_VPD$"
  $gPlatformFspPkgTokenSpaceGuid_PcdImageRevision 4 bytes $_DEFAULT_ =
0x00070100
  Skip XX bytes
  $gSiFspPkgTokenSpaceGuid_PcdSerialIoUartDebugEnable 1 bytes $_DEFAULT_ = 0
  ......

/* UPD data region*/
Find "$SI_UPD$"
  Skip XX bytes
  $gSiFspPkgTokenSpaceGuid_TsegSize 4 bytes $_DEFAULT_ = 0x01000000
  ......

/* Show configuration */
Page "Platform"
  Combo $gSiFspPkgTokenSpaceGuid_PcdSerialIoUartDebugEnable,
       "UART Debug",
       &gSiPkgTokenSpaceGuid_PcdSerialIoUartDebugEnable,
       Help "Select UART Debug enable/disable"
  ......
==========================================
```

Summary

This section describes how to user VPD/UPD to expose silicon configuration in FSP package.

Step 3 – Build with IntelFspPkg

FSP binary includes silicon modules and FspSecCore and FspDxeIpl in IntelFspPkg. (https://github.com/tianocore/edk2-IntelFspPkg). But they are not enough. We should include below components to make a full solution.

FSP Description File

The FSP binary may optionally include an FSP description file. This file will provide information about the FSP including information about different silicon revisions the FSP supports. See below sample in FDF file.

```
PlatformFspPkg.fdf:
=========================================
#
# Description file
#
FILE RAW = D9093578-08EB-44DF-B9D8-D0C1D3D55D96 {
  SECTION RAW = FspDescription/FspDescription.txt
}
=========================================
```

FSP_INFO_HEADER

FSP_INFO_HEADER must be the 1st firmware file within the FSP firmware volume. We can use below way to ensure that.

First, we define a FSP_INFO_HEADER structure in C style file and use .ASLC as file name extension. We must define ReferenceAcpiTable() function referring to this global data structure, or this global data will be optimized at link phase. We also need FSP_INFO_EXTENDED_HEADER for FSP1.1.
FSP_PATCH_TABLE is FSP implementation specific table. It is put here, because we want to reserve space in FSP binary to let platform refer patch information.

```
FspHeader.aslc:
=========================================
typedef struct{
  FSP_INFO_HEADER          FspInfoHeader;
  FSP_INFO_EXTENDED_HEADER FspInfoExtendedHeader;
  FSP_PATCH_TABLE          FspPatchTable;
}TABLES;

TABLES mTable =
{
  { // FspInfoHeader
  0x48505346,                         // UINT32  Signature  (FSPH)
  sizeof(FSP_INFO_HEADER),            // UINT32  HeaderLength;
  {0x00, 0x00, 0x00},                 // UINT8   Reserved1[3];
  FixedPcdGet8(PcdFspHeaderRevision),    // UINT8   HeaderRevision;
  FixedPcdGet32(PcdFspImageRevision),    // UINT32  ImageRevision;
  UINT64_TO_BYTE_ARRAY(`
  FixedPcdGet64(PcdFspImageIdString)),   // CHAR8   ImageId[8];

  FixedPcdGet32(PcdFspAreaSize),         // UINT32  ImageSize;
  FixedPcdGet32(PcdFspAreaBaseAddress),  // UINT32  ImageBase;

  0x00000000,                            // UINT32  ImageAttribute;
  0x12345678,                            // UINT32  CfgRegionOffset;
  0x12345678,                            // UINT32  CfgRegionSize;
  0x00000006,                            // UINT32  ApiEntryNum;

  0x12345678,                            // UINT32  NemInitEntry;
  0x12345678,                            // UINT32  FspInitEntry;
  0x12345678,                            // UINT32  NotifyPhaseEntry;
  0x12345678,                            // UINT32  FspMemoryInitEntry;
  0x12345678,                            // UINT32  TempRamExitEntry;
  0x12345678,                            // UINT32  FspSiliconInitEntry;
  },
  { // FspExtendedHeader
  0x45505346,                            // UINT32  Signature  (FSPE)
  sizeof(FSP_INFO_EXTENTED_HEADER),      // UINT32  Length;
  FSPE_HEADER_REVISION_1,                // UINT8   Revision;
  0x00,                                  // UINT8   Reserved;
  {OEM_ID},                              // CHAR8   FspProducerId[6];
  0x00000001,                            // UINT32  FspProducerRevision;
  0x00000000,                            // UINT32  FspProducerDataSize;
  },
  { // FspPatchTable
  0x50505346,                 // UINT32  Signature  (FSPP)
  sizeof(FSP_PATCH_TABLE),    // UINT16  HeaderLength;
  FSPP_HEADER_REVISION_1,     // UINT8   HeaderRevision;
  0x00,                       // UINT8   Reserved;
  2,                          // UINT32  PatchEntryNum;
  {0xFFFFFFFC,                // UINT32  Patch FVBASE at end of FV
   0x12345678}                // UINT32  Patch module patchable PCD VpdBase
  }
};


VOID*
ReferenceAcpiTable (
  VOID
  )
{
  //
  // Reference the table being generated to prevent the optimizer from
  // removing the data structure from the executable
  //
  return (VOID*)&mTable;
}
==========================================
```

After that we need create INF file and use USER_DEFINED as MODULE_TYPE.

```
FspHeader.inf
==========================================
  MODULE_TYPE                     = USER_DEFINED
======================================
```

Then we add this module in FDF file as first module of FSP FV, and use build rule override FSPHEADER to extract global data section only.

```
PlatformFspPkg.fdf:
======================================
[FV.SIFSP]
BlockSize          = 0x00001000
FvAlignment        = 16
ERASE_POLARITY     = 1
MEMORY_MAPPED      = TRUE
STICKY_WRITE       = TRUE
LOCK_CAP           = TRUE
LOCK_STATUS        = TRUE
WRITE_DISABLED_CAP = TRUE
WRITE_ENABLED_CAP  = TRUE
WRITE_STATUS       = TRUE
WRITE_LOCK_CAP     = TRUE
WRITE_LOCK_STATUS  = TRUE
READ_DISABLED_CAP  = TRUE
READ_ENABLED_CAP   = TRUE
READ_STATUS        = TRUE
READ_LOCK_CAP      = TRUE
READ_LOCK_STATUS   = TRUE
FvNameGuid         = 12345678-1234-1234-1234-1234567890CD

INF RuleOverride = FSPHEADER $(FSP_PACKAGE)/FspHeader/FspHeader.inf
......

[Rule.Common.USER_DEFINED.FSPHEADER]
  FILE RAW = $(NAMED_GUID)               {
     RAW BIN                  |.acpi
  }
======================================
```

Convert VPD/UPD to Policy

Then we need a platform module to convert VPD/UPD data to the silicon policy PPI. For example, to use GetFspUpdDataPointer API defined in FspCommonLib. (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Include/Library/FspCommonLib.h)

Some sample code to consume GetFspUpdDataPointer() below:

```
======================================
UPD_DATA_REGION *UpdDataRgnPtr;

UpdDataRgnPtr = (UPD_DATA_REGION *)GetFspUpdDataPointer ();
// Update policy based on UPD data.
======================================
```

MemoryDiscoveredPpiNotifyCallback for FSP 1.1

In FSP 1.1 mode, we need use Pei2LoaderSwitchStack API defined in FspSwitchStackLib. (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Include/Library/FspSwitchStackLib.h) to switch back to bootloader.

Some sample code for MemoryDiscoveredPpiNotifyCallback() below:

```
=========================================
UINT32
GetUsableLowMemTop (
  CONST VOID       *HobStart
  )
{
  EFI_PEI_HOB_POINTERS  Hob;
  UINT32                MemLen;
  Hob.Raw = (VOID *)HobStart;

  MemLen = 0x100000;
  while (!END_OF_HOB_LIST (Hob)) {
    if (Hob.Header->HobType == EFI_HOB_TYPE_RESOURCE_DESCRIPTOR) {
      if (Hob.ResourceDescriptor->ResourceType == EFI_RESOURCE_SYSTEM_MEMORY) {
        if (Hob.ResourceDescriptor->PhysicalStart >= 0x100000 &&
            Hob.ResourceDescriptor->PhysicalStart < (EFI_PHYSICAL_ADDRESS) 0x100000000) {
          MemLen += (UINT32) (Hob.ResourceDescriptor->ResourceLength);
        }
      }
    }
    Hob.Raw = GET_NEXT_HOB (Hob);
  }

  return MemLen;
}

EFI_STATUS
ReportAndInstallFspSiliconFv (
  VOID
  )
{
  EFI_FIRMWARE_VOLUME_EXT_HEADER *FwVolExtHeader;
  FSP_INFO_HEADER                *FspInfoHeader;
  EFI_FIRMWARE_VOLUME_HEADER     *FvHeader;
  UINT8                          *CurPtr;
  UINT8                          *EndPtr;

  FspInfoHeader = GetFspInfoHeaderFromApiContext ();
  if (FspInfoHeader->Signature != FSPH_SIGNATURE) {
    DEBUG ((DEBUG_ERROR, "The signature of FspInfoHeader is invalid.\n"));
  }

  CurPtr = (UINT8 *)FspInfoHeader->ImageBase;
  EndPtr = CurPtr + FspInfoHeader->ImageSize - 1;

  while (CurPtr < EndPtr) {
    FvHeader = (EFI_FIRMWARE_VOLUME_HEADER *)CurPtr;
    if (FvHeader->Signature != EFI_FVH_SIGNATURE) {
      break;
    }

    if (FvHeader->ExtHeaderOffset != 0) {
      //
      // Searching for the Silicon FV in the FSP image.
      //
      FwVolExtHeader = (EFI_FIRMWARE_VOLUME_EXT_HEADER *) ((UINT8 *) FvHeader + FvHeader-
>ExtHeaderOffset);
      if (CompareGuid(&FwVolExtHeader->FvName, &gFspSiliconFvGuid)) {
        PeiServicesInstallFvInfoPpi (
          NULL,
          (VOID *)FvHeader,
          (UINTN) FvHeader->FvLength,
          NULL,
          NULL
          );
      }
    }
    CurPtr += FvHeader->FvLength;
  }

  return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
MemoryDiscoveredPpiNotifyCallback (
  IN EFI_PEI_SERVICES           **PeiServices,
  IN EFI_PEI_NOTIFY_DESCRIPTOR  *NotifyDescriptor,
  IN VOID                       *Ppi
  )
{

  DEBUG ((DEBUG_INFO | DEBUG_INIT, "Memory Discovered Notify invoked ...\n"));

  ......
  //
  // Build PEI Reserve memory HOB
  //
  MemBase = GetUsableLowMemTop (GetHobList ());
  MemSize = PcdGet32 (PcdFspReservedMemoryLength);

  BuildResourceDescriptorWithOwnerHob (
    EFI_RESOURCE_MEMORY_RESERVED,
    (
    EFI_RESOURCE_ATTRIBUTE_PRESENT |
    EFI_RESOURCE_ATTRIBUTE_INITIALIZED |
    EFI_RESOURCE_ATTRIBUTE_TESTED |
    EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE |
    EFI_RESOURCE_ATTRIBUTE_WRITE_COMBINEABLE |
    EFI_RESOURCE_ATTRIBUTE_WRITE_THROUGH_CACHEABLE |
    EFI_RESOURCE_ATTRIBUTE_WRITE_BACK_CACHEABLE
    ),
    MemBase - MemSize,
    MemSize,
    &gFspReservedMemoryResourceHobGuid
    );

  //
  // Get system memory from HOB
  //
  FspGetSystemMemorySize (&LowMemoryLength, &HighMemoryLength);

  //
  // FSP reserved memory is immediately folowing all availabel system memory regions,
  // so we should add it back to ensure this reserved region is cached.
  //
  LowMemoryLength += PcdGet32 (PcdFspReservedMemoryLength);

  //
  // Migrate BootLoader data before destroying CAR
  //
  FspMigrateTemporaryMemory ();

  ApiMode = GetFspApiCallingMode ();
  if (ApiMode != 0) {
    //
    // Calling use FspMemoryInit API
    // Return the control directly
    //
    FspMemoryInitParams = (FSP_MEMORY_INIT_PARAMS *)GetFspApiParameter ();
    if ((FspMemoryInitParams->HobListPtr) != NULL) {
      *(FspMemoryInitParams->HobListPtr) = (VOID *)GetHobList ();
    }
    //
    // This is the end of the FspMemoryInit API
    // Give control back to the boot loader
    //
    DEBUG ((DEBUG_INFO | DEBUG_INIT, "FspMemoryInitApi() - End\n"));
    SetFspApiReturnStatus (EFI_SUCCESS);
    Pei2LoaderSwitchStack ();
  }

  //
  // Disable CAR
  //
  ResetCacheAttributes ();
  if (ApiMode != 0) {
    //
    // This is the end of the TempRamExit API
    // Give control back to the boot loader
    //
    DEBUG ((DEBUG_INFO | DEBUG_INIT, "TempRamExitApi() - End\n"));
    SetFspApiReturnStatus (EFI_SUCCESS);
    Pei2LoaderSwitchStack ();
  }

  //
  // Install FSP silicon FV (To Support Dual FSP Only)
  //
  if (NeedSupportDualFsp ()) {
    ReportAndInstallFspSiliconFv ();
  }

  //
  // Set the code region as cachable for performance
  //
  ......

  DEBUG ((DEBUG_INFO | DEBUG_INIT, "Memory Discovered Notify completed ...\n"));

}
=========================================
```

NOTE: In order to support dual FSP, this module will call ReportAndInstallFspSiliconFv() in the middle so that PEI Core will find Silicon FV location.

Postbuild - PatchFv

Finally, FSP.FD will be generated. But it is not enough. We need patch the pointer referred in binary.

1) Patch API offset.

```
=========================================
python IntelFspPkg\Tools\PatchFv.py ^
     %OUT_DIR%\%FSP_PKG_NAME%\%BD_TARGET%_%TOOL_CHAIN_TAG%\FV ^
     SIFSP  ^
     "0xFFFFFFFC, [0x000000B0],                                @FVBASE" ^
     "0xFFFFFFE0, <PeiCore:__ModuleEntryPoint>,                @PeiCore Entry" ^
     "0x000000C4, <FspSecCore:_TempRamInitApi>,                @TempRamInit API" ^
     "0x000000C8, <FspSecCore:_FspInitApi>,                    @FspInit API" ^
     "0x000000CC, <FspSecCore:_NotifyPhaseApi>,                @NotifyPhase API" ^
     "0x000000D0, <FspSecCore:_FspMemoryInitApi>,              @FspMemoryInit API" ^
     "0x000000D4, <FspSecCore:_TempRamExitApi>,                @TempRamExit API" ^
     "0x000000D8, <FspSecCore:_FspSiliconInitApi>,             @FspSiliconInit API" ^
     "0x000000B8, 12345678-1234-1234-1234-1234567890AB:0x1C,     @VPD Region offset" ^
     "0x000000BC, [12345678-1234-1234-1234-1234567890AB:0x14]  - 0xF800001C,  @VPD Region size" ^
     "0x00000100, PcdPeim:__gPcd_BinaryPatch_PcdVpdBaseAddress - [0x000000B0], @VPD PCD offset" ^
     "12345678-1234-1234-1234-1234567890AB:0x28, ([12345678-1234-1234-1234-1234567890AB:0x18] +
0x00000003) & 0x00FFFFFC + 12345678-1234-1234-1234-1234567890AB:0x1C,  @UPD Region offset"
==========================================
```

In VPD/UPD data are in file section, so final 4 entries patch VPD region offset and UPD region offset.
The FSP_PATCH_TABLE is also patched here. For example, offset 0x00000100 is 2nd FSP_PATCH_TABLE patch entry. It means VPD PCD offset.

2) Patch FspInfoHeader relative offset

```
==========================================
python IntelFspPkg\Tools\PatchFv.py ^
     %OUT_DIR%\%FSP_PKG_NAME%\%BD_TARGET%_%TOOL_CHAIN_TAG%\FV ^
     SIFSP  ^
     "FspSecCore:_FspInfoHeaderRelativeOff, FspSecCore:_GetFspBaseAddress - {912740BE-2284-4734-
B971-84B027353F0C:0x1C}, @FSP Header Offset"
==========================================
```

The FspInfoHeaderRelativeOff is the symbol in FspSecCore.
(https://github.com/tianocore/edk2-IntelFspPkg/tree/master/FspSecCore/Ia32/FspHelper.asm)
It is used to calculate FspBaseAddress.

3) Patch VPD base into the PcdPeim module patchable PCD

```
==========================================
python IntelFspPkg\Tools\PatchFv.py ^
     %OUT_DIR%\%FSP_PKG_NAME%\%BD_TARGET%_%TOOL_CHAIN_TAG%\FV ^
     SIFSP  ^
     "PcdPeim:__gPcd_BinaryPatch_PcdVpdBaseAddress, {[0x000000B8]}, @VPD PCD base"
==========================================
```

The offset of VPD region data is at FSP_INFO_HEADER, so we need patch the value into PcdVpdBaseAddress of PcdPeim.

This PatchFv.py tool runs after formal EDKII build and it will patch final FD binary, that is why we call it as postbuild tool.
For more detail, please refer to user manual (https://github.com/tianocore/edk2-IntelFspPkg/tree/master/Tools/UserManuals/PatchFvUserManual.docx)

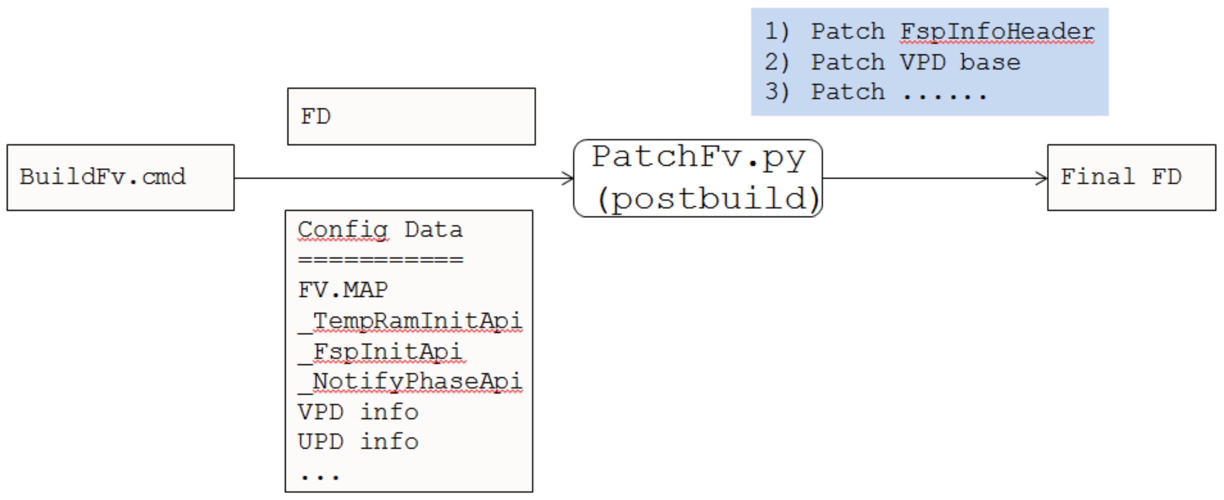

Figure 8 Postbuild tool – PatchFv.py

Dual FSP Support

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

In order to support Dual FSP, user need define 2 FV and combine those 2 FV into one FD in FDF file. One BFV contains VPD region, SecCore, PeiCore and MemoryInit module. The other silicon FV contains FspInfoHeader, DxeIpl and SiliconInit module. This FD could be factory FSP binary. For updatable FSP binary, only silicon FV can be updated there.

See example below:

```
PlatformFspPkg.fdf:
==============================================
[FD.SIFSP]
BaseAddress   = $(FLASH_BASE) | gIntelFspPkgTokenSpaceGuid.PcdFspAreaBaseAddress
Size          = $(FLASH_SIZE) | gIntelFspPkgTokenSpaceGuid.PcdFspAreaSize
ErasePolarity = 1
BlockSize     = $(FLASH_BLOCK_SIZE)
NumBlocks     = $(FLASH_NUM_BLOCKS)

SET gIntelFspPkgTokenSpaceGuid.PcdFspBootFirmwareVolumeBase = $(FLASH_BASE) +
$(FLASH_REGION_FSP_SILICON_FV_SIZE)

#FSP Silicon
$(FLASH_REGION_FSP_SILICON_FV_OFFSET)|$(FLASH_REGION_FSP_SILICON_FV_SIZE)
gSiFspPkgTokenSpaceGuid.PcdFlashFspSiliconFvBase|gSiFspPkgTokenSpaceGuid.PcdFlashFspSi
liconFvSize
FV = SIFSP_Silicon

#FSP BFV
$(FLASH_REGION_FSP_BFV_OFFSET)|$(FLASH_REGION_FV_FSP_BFV_SIZE)
gSiFspPkgTokenSpaceGuid.PcdFlashFspBfvBase|gSiFspPkgTokenSpaceGuid.PcdFlashFspBfvSize
FV = SIFSP_BFV

[FV.SIFSP_BFV]
BlockSize          = 0x00001000
FvAlignment        = 16
ERASE_POLARITY     = 1
MEMORY_MAPPED      = TRUE
STICKY_WRITE       = TRUE
LOCK_CAP           = TRUE
LOCK_STATUS        = TRUE
WRITE_DISABLED_CAP = TRUE
WRITE_ENABLED_CAP  = TRUE
WRITE_STATUS       = TRUE
WRITE_LOCK_CAP     = TRUE
WRITE_LOCK_STATUS  = TRUE
READ_DISABLED_CAP  = TRUE
READ_ENABLED_CAP   = TRUE
READ_STATUS        = TRUE
READ_LOCK_CAP      = TRUE
READ_LOCK_STATUS   = TRUE
FvNameGuid         = 12345678-1234-1234-1234-1234567890CD
```

```
# Core components
INF IntelFspPkg/FspSecCore/FspSecCore.inf
INF MdeModulePkg/Core/Pei/PeiMain.inf
INF MdeModulePkg/Universal/PCD/Pei/Pcd.inf

# VPD/UPD region
FILE RAW = 12345678-1234-1234-1234-1234567890AB {
    SECTION RAW =
$(OUTPUT_DIRECTORY)/$(TARGET)_$(TOOL_CHAIN_TAG)/FV/$(VPD_TOOL_GUID).bin
    SECTION RAW =
$(OUTPUT_DIRECTORY)/$(TARGET)_$(TOOL_CHAIN_TAG)/FV/$(UPD_TOOL_GUID).bin
}

# Memory Init module
INF <MemoryInit>.inf

# Description file
FILE RAW = D9093578-08EB-44DF-B9D8-D0C1D3D55D96 {
    SECTION RAW = $(FSP_PACKAGE)/FspDescription/FspDescription.txt
}

[FV.SIFSP_Silicon]
BlockSize          = 0x00001000
FvAlignment        = 16
ERASE_POLARITY     = 1
MEMORY_MAPPED      = TRUE
STICKY_WRITE       = TRUE
LOCK_CAP           = TRUE
LOCK_STATUS        = TRUE
WRITE_DISABLED_CAP = TRUE
WRITE_ENABLED_CAP  = TRUE
WRITE_STATUS       = TRUE
WRITE_LOCK_CAP     = TRUE
WRITE_LOCK_STATUS  = TRUE
READ_DISABLED_CAP  = TRUE
READ_ENABLED_CAP   = TRUE
READ_STATUS        = TRUE
READ_LOCK_CAP      = TRUE
READ_LOCK_STATUS   = TRUE
FvNameGuid         = 12345678-1234-1234-1234-1234567890EF

# FSP Info header
INF RuleOverride = FSPHEADER $(FSP_PACKAGE)/FspHeader/FspHeader.inf

INF IntelFspPkg/FspDxeIpl/FspDxeIpl.inf

# Silicon Init module
INF <SiliconInit>.inf
==============================================
```

Then in MemoryDiscoveredPpiNotifyCallback(), user can call ReportAndInstallFspSiliconFv() to install the silicon FV, then all the modules in silicon FV will be dispatched by PEI Core.

Summary

This section describes the additional components and steps needed to build FSP binary.

Step 4 – Release it

Package content

Finally, the release package should include, but not limit to:

1) Binary file – FSP.fd
2) Source header file – FspUpdVpd.h
3) Tool configuration file – FSP.bsf
4) Document - <Silicon>_FSP_Integration_Guide

Congratulations!!!

Conclusion

FSP provides a simple to integrate solution that reduces time-to-market, and it is economical to build. IntelFspPkg is the core infrastructure of FSP producer in EDKII to support building FSP binary. This paper describes detail work flow and how to use IntelFspPkg to build FSP binary.

Glossary

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

BCT – Binary Configuration Tool. The tool to patch FSP binary.

BSF – Boot Setting File. The configuration file used by BCT tool.

BFV – Boot Firmware Volume. See [UEFI PI Specification].

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

DEC – EDKII Package Declaration File. See [EDKII specification]

DSC – EDKII Platform Description File. See [EDKII specification]

FD – EDKII Flash Device binary image, defined in FDF.

FDF – EDKII Flash Descirption File. See [EDKII specification]

FFS – firmware file system, describes the organization of files and (optionally) free space within the firmware volume. See [UEFI PI 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]

FV – Firmware Volume, a logical firmware device. See [UEFI PI Specification].

INF – EDKII Module Information File. See [EDKII specification]

PCD – Platform Configuration Database. See [UEFI PI Specification].

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

SMM – System Management Mode. x86 CPU operational mode that is isolated from and transparent to the operating system runtime.

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

UPD – Updatable Product Data. Configuration data region in FSP binary, which can only be configured statically for default value, but also can be overwritten during boot at runtime. See [FSP EAS]

VPD – Vital Product Data. Configuration data region in FSP binary, which can only be configured statically. See [EDKII specification] DSC format and [FSP EAS].

VTF – Volume Top File, a file that must be located such that the last byte of the file is also the last byte of the firmware volume. See [UEFI PI Specification]

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

[EDKII specification] A set of specification describe EDKII DEC/INF/DSC/FDF file format, as well as EDKII BUILD. http://tianocore.sourceforge.net/wiki/EDK_II_Specifications

[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 http://www.intel.com/content/www/us/en/embedded/software/fsp/fsp-architecture-spec-v1-1.html

[FSP Consumer] Yao, Zimmer, Rangarajan, Ma, Estrada, Mudusuru, "A_Tour_Beyond_BIOS_Using_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.