Intel® Firmware Support Package

External Architecture Specification

Revision 2.5

July 2024

Document Number: 736809

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

Altering clock frequency, voltage, or memory interface speeds may void any product warranties and reduce stability, security, performance, and life of the processor and other components. Intel has not validated processor running memory above Plan-Of-Record (POR) speed. DRAM/DIMM devices should support desired speed, check with DRAM/DIMM vendors for details. System manufacturers are responsible for all validation and assume the risk of any stability, security, performance, or other functional issues resulting from such alterations.

Contents

Figures

Tables

Revision History

Revision Date	Revision Number	Description
July 2024	2.5	• FSP_INFO_HEADER changes — Updated SpecVersion from 0x24 to 0x25 — Updated HeaderRevision from 7 to 8 — Defined bit 4 in ImageAttribute to indicate support for Configurable FSP TemporaryRamSize. • Updated FSPT_ARCH2_UPD — Updated Revision from 2 to 3 — Added FspTemporaryRamSize
December 2022	2.4 Errata A	• Fixed HeaderLength field in FSP_INFO_HEADER
August 2022	2.4	• Based on FSP EAS v2.3. • FSP_INFO_HEADER changes — Updated SpecVersion from 0x23 to 0x24 — Updated HeaderRevision from 6 to 7 — Defined bit 2 in ImageAttribute to indicate support for 64-bit interfaces. • Extended FSP API calling convention to support both 32-bit and 64-bit interfaces. • Updated FSP status code and OEM status code definition to support both 32-bit and 64-bit interfaces. • Added FSP-I SMM component • Added FspMultiPhaseMemInit() API • Added Variable Services interface
July 2021	2.3	• Based on FSP EAS v2.2 – Backward compatibility is retained. • FSP_INFO_HEADER changes — Updated SpecVersion from 0x22 to 0x23 — Updated HeaderRevision from 5 to 6 — Added ExtendedImageRevision • Added FSP_NON_VOLATILE_STORAGE_HOB2
May 2020	2.2	• Based on FSP EAS v2.1 – Backward compatibility is retained. • Added multi-phase silicon initialization to increase the modularity of the FspSiliconInit() API. • Added FSP event handlers. • Added FspMultiPhaseSiInit() API • FSP_INFO_HEADER changes — Updated SpecVersion from 0x21 to 0x22 — Updated HeaderRevision from 4 to 5 — Added FspMultiPhaseSiInitEntryOffset • Added FSPT_ARCH_UPD — Added FspDebugHandler • FSPM_ARCH_UPD changes — Added FspEventHandler

Revision Date	Revision Number	Description
		• Added FSPS_ARCH_UPD

§§

1.0 Introduction

1.1 Purpose

The purpose of this document is to describe the external architecture and interfaces provided in the Intel® Firmware Support Package (FSP). Implementation specific details are outside the scope of this document. Refer to Integration Guide for details.

1.2 Intended Audience

This document is targeted at all platform and system developers who need to generateor consume FSP binaries in their bootloader solutions. This includes but is not limited to: System firmware or UEFI firmware or BIOS developers, bootloader developers, system integrators, as well as end users.

1.3 Related Documents

- Intel® FSP EAS version 2.4: https://cdrdv2-public.intel.com/736809/736809_FSP_EAS_v2.4_Errata_A.pdf
- Boot Specification File (BSF) Specification: https://software.intel.com/en-us/download/boot-setting-file-specification-release-10
- Unified Extensible Firmware Interface (UEFI) Specification: http://www.uefi.org/specifications
- Platform Initialization (PI) Specification v1.7 (Errata A): https://uefi.org/sites/default/files/resources/PI_Spec_1_7_A_final_May1.pdf
- Binary Configuration Tool (BCT) for Intel® Firmware Support Package — available at: http://www.intel.com/fsp
- Intel® Firmware Module Management Tool (Intel® FMMT) — available at: https://software.intel.com/en-us/download/intel-firmware-module-management-tool-intel-fmmt-r22
- A Tour Beyond BIOS Launching Standalone SMM drivers in the PEI Phase using theEFI Developer Kit II (May 2015): https://www.intel.com/content/dam/develop/public/us/en/documents/a-tour-beyond-bios-launching-standalone-smm-drivers-in-pei-using-the-efi-developer-kit-ii.pdf

§§

2.0 FSP Overview

2.1 Design Philosophy

Intel recognizes that it holds the key programming information that is crucial for initializing Intel silicon. Some key programming information is treated as proprietaryinformation and may only be available with legal agreements.

Intel® Firmware Support Package (Intel® FSP) is a binary distribution of necessary Intel silicon initialization code. The first design goal of FSP is to provide ready access to thekey programming information that is not publicly available. The second design goal is to abstract the complexities of Intel Silicon initialization and expose a limited number of well-defined interfaces.

A fundamental design philosophy is to provide the ubiquitously required silicon initialization code. As such, FSP will often provide only a subset of the product's features.

2.2 Technical Overview

The FSP provides chipset and processor initialization in a format that can easily be incorporated into many existing bootloaders.

The FSP performs the necessary initialization steps as documented in the BIOS WritersGuide (BWG) / BIOS Specification including initialization of the processor, memory controller, chipset, and certain bus interfaces, if necessary.

FSP is not a stand-alone bootloader; therefore, it needs to be integrated into a bootloader to carry out other functions such as:

- Initializing non-Intel components
- Bus enumeration and device discovery
- Industry standards

2.2.1 Data Structure Descriptions

All data structures defined in this specification conform to the "little endian" byte order(i.e., the low-order byte of a multibyte data items in memory is at the lowest address), while the high-order byte is at the highest address.

All reserved fields defined in this specification must be zero unless stated otherwise.

§§

3.0 FSP Integration

The FSP binary can be integrated into many different bootloaders and embedded operating systems.

Below are some required steps for the integration:

- **Customizing:**

  The FSP has configuration parameters that can be customized to meet the needs of the target platform.

- **Rebasing:**

  The FSP is not Position Independent Code (PIC) and each FSP component has to be rebased if it is placed at a location which is different from the preferred base address specified during the FSP build.

- **Placing:**

  Once the FSP binary is ready for integration, the bootloader needs to be modified to place this FSP binary at the specific base address identified above.

- **Interfacing:**

  The bootloader needs to add code to setup the operating environment for the FSP, call the FSP with the correct parameters, and parse the FSP output to retrieve the necessary information returned by the FSP.

3.1 FSP Distribution Package

The FSP distribution package contains the following:

- FSP Binary
- Integration Guide
- Data structure definitions
- Boot Settings File (BSF)

The Binary Configuration Tool (BCT) can be used to configure the FSP. BCT is available as a separate package.

§§

4.0 FSP Binary Format

The FSP binary follows the UEFI Platform Initialization Firmware Volume Specification format. The Firmware Volume (FV) format is described in the Platform Initialization (PI) *Specification - Volume 3*: Shared Architectural Elements specification as referenced in *Section 1.3 Related Documents*.

Firmware Volume (FV) is a way to organize/structure binary **components** and enables a standardized way to parse the binary and handle the individual binary components that make up the Firmware Volume (FV).

4.1 FSP Components

The FSP will have several components each containing one or more firmware volumes (FV). Each component provides a phase of initialization as below

4.1.1 FSP-T: Temporary RAM Initialization Phase

Primary purpose of this phase is to initialize the Temporary RAM along with any other early initialization.

This phase consists of below FSP API

- TempRamInit()

4.1.2 FSP-M: Memory Initialization Phase

Primary purpose of this phase is to initialize the permanent memory along with any other early silicon initialization.

This phase consists of below FSP API

- FspMemoryInit()
- FspMultiPhaseMemInit()
- TempRamExit()

4.1.3 FSP-S: Silicon Initialization Phase

Primary purpose of this phase is to complete the silicon initialization including CPU andIO controller initialization.

This phase consists of below FSP API

- FspSiliconInit()
- FspMultiPhaseSiInit()
- **NotifyPhase()**-Post PCI bus enumeration, Ready To Boot and End of Firmware.

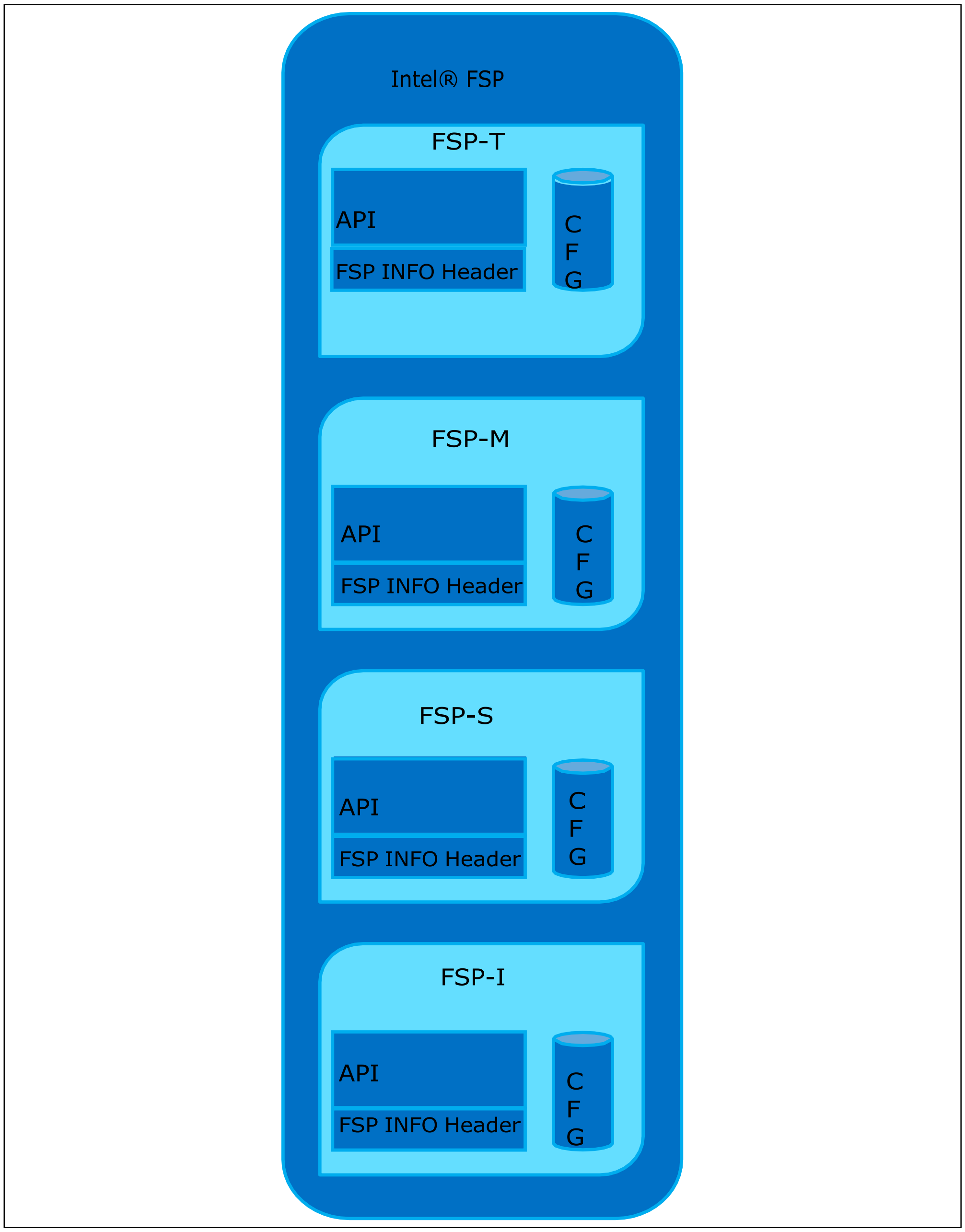
4.1.4 FSP-I: SMM Initialization Phase

An FSP may include an FSP-I component. This phase will initialize SMM and provide OSruntime silicon services; including Reliability, Availability, and Serviceability (RAS) features implemented by the CPU.

This phase consists of below FSP API

- FspSmmInit()

Figure 1. FSP Component Logical View



4.1.5 OEM Components (FSP-O)

An FSP may include optional OEM components that provide OEM extensibility. This component shall have an FSP_INFO_HEADER with component type in Image attributefield set to FSP-O.

4.2 FSP Component Identification

Each FSP component will have an **FSP_INFO_HEADER** as the first FFS file in the firstFirmware Volume (FV). The **FSP_INFO_HEADER** will have an attribute field that can beused to identify that component as an FSP-T / FSP-M / FSP-S / FSP-I / FSP-O component.

There can be only one instance of the FSP-T / FSP-M / FSP-S / FSP-I in an FSP binary,while multiple instances of the FSP-O component are valid.

4.2.1 FSP Image ID and Revision

The **FSP_INFO_HEADER** structure inside each FSP component also contains an ImageIdentifier field and an Image Revision field that provide the identification and revision information for the FSP binary. It is important to verify these fields while integrating theFSP as the FSP configuration data could change over different FSP Image identifiers and revisions.

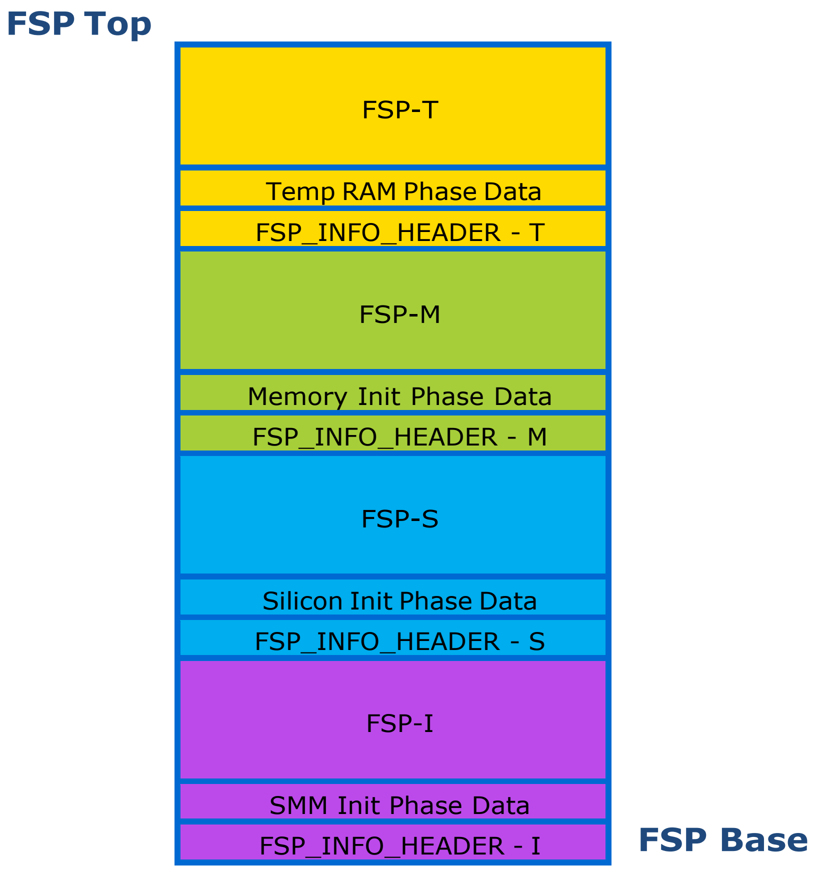
The FSP Image Identifier field should be the same for all the FSP components within the same FSP binary.

4.2.2 FSP Component Layout

All the FSP components are packaged back-to-back within the FSP and the size of eachcomponent is available in the component’s **FSP_INFO_HEADER** structure.

Furthermore, if there are multiple Firmware Volume(s) inside the FSP component, theyare also packaged back-to-back. These components can be packaged in any order inside the FSP binary.

Figure 2. FSP Component Layout View



§§

5.0 FSP Information Tables

Each FSP component has an **FSP_INFO_HEADER** table and may optionally have additional tables as described below.

All FSP tables must have a 4 bytes aligned base address and a size that is a multiple of4 bytes.

All FSP tables must be placed back-to-back.

All FSP tables must begin with a DWORD signature followed by a DWORD length field.

A generic table search algorithm for additional tables can be implemented with a signature search algorithm until a terminator signature 'FSPP' is found.

5.1 FSP_INFO_HEADER

The **FSP_INFO_HEADER** structure conveys the information required by the bootloader to interface with the FSP binary.

Table 1. FSP_INFO_HEADER

Byte Offset	Size in Bytes	Field	Description
0	4	Signature	'FSPH'. Signature for the FSP_INFO_HEADER.
4	4	HeaderLength	Length of the header in bytes. The current valuefor this field is 88.
8	2	Reserved1	Reserved bytes for future.
10	1	SpecVersion	Indicates compliance with a revision of this specification in the BCD format. 3 : 0 - Minor Version 7 : 4 - Major Version For revision v2.5 the value will be 0x25.
11	1	HeaderRevision	Revision of the header. The current value for this field is 8.
12	4	ImageRevision	Revision of the FSP binary. Major.Minor.Revision.Build If FSP HeaderRevision is <= 5, the ImageRevision can be decoded as follows: 7 : 0 - Build Number 15 : 8 - Revision 23 : 16 - Minor Version 31 : 24 - Major Version

Byte Offset	Size in Bytes	Field	Description
			If FSP HeaderRevision is >= 6, ImageRevision specifies the low-order bytes of the build number and revision while ExtendedImageRevision specifies the high-order bytes of the build number and revision. 7 : 0 - Low Byte of Build Number 15 : 8 - Low Byte of Revision 23 : 16 - Minor Version 31 : 24 - Major Version
16	8	ImageId	8 ASCII character byte signature string that willhelp match the FSP binary to a supported hardware configuration. Bootloader should not assume null-terminated.
24	4	ImageSize	Size of this component in bytes.
28	4	ImageBase	Preferred base address for this component. If the FSP component is located at the address different from the preferred address, the FSP component needs to be rebased.
32	2	ImageAttribute	Attributes of the FSP binary. The value of this field must be consistent across the FSP-T, FSP- M and FSP-S components within a FSP image. • Bit 0: Graphics Support – Set to 1 when FSP supports enabling Graphics Display. • Bit 1: Dispatch Mode Support – Set to 1 when FSP supports the optional Dispatch Mode API defined in *Section 7.2* and *10.0*. This bit is onlyvalid if FSP HeaderRevision is >= 4. • Bit 2: 64-bit Mode Support – Set to 1 to indicate FSP supports 64-bit long mode interfaces. Set to 0 to indicate FSP supports 32-bit mode interfaces. This bit is only valid if FSP HeaderRevision is >= 7. • Bit 3: FSP Variable Services Support – Set to 1to indicate FSP utilizes the FSP Variable Services defined in *Section 9.6* to store non-volatile data. This bit is only valid if FSP HeaderRevision is >= 7. • Bit 4: Configurable TemporaryRamSize support – Set to 1 to indicate FSP will support configurability of TempRamSize via FspTemporaryRamSize UPD in FSPT_ARCH2_UPD, defined in Section 6.1.1. Set to 0 indicates FSP will utilize the default TempRamSize that is part of FSP binary. This bit is only valid if FSP HeaderRevision is >= 8. • Bits 15:5 – Reserved
34	2	ComponentAttribute	Attributes of the FSP Component

Byte Offset	Size in Bytes	Field	Description
			• Bit 0 – Build Type 0 – Debug Build 1 - Release Build • Bit 1 – Release Type 0 –Test Release 1 -Official Release • Bit 11:2 - Reserved • Bits 15:12 – Component Type 0000 – Reserved 0001 – FSP-T 0010 – FSP-M 0011 – FSP-S 0100 - FSP-I (FSP SMM) 0101 to 0111 – Reserved 1000 – FSP-O 1001 to 1111 – Reserved
36	4	CfgRegionOffset	Offset of the UPD configuration region. This offset is relative to the respective FSP Component base address. Please refer to Section 6.0 for details.
40	4	CfgRegionSize	Size of the UPD configuration region. Please refer to Section 6.0 for details.
44	4	Reserved2	This value must be 0x00000000 if the FSP HeaderRevision is >=3.
48	4	TempRamInitEntryOffset	Offset for the API to setup a temporary stack till the memory is initialized. If the value is set to 0x00000000, then this API is not available in this component.
52	4	Reserved3	This value must be 0x00000000 if the FSP HeaderRevision is >=3.
56	4	NotifyPhaseEntryOffset	Offset for the API to inform the FSP about the different stages in the boot process. If the value is set to 0x00000000, then this API is not available in this component.
60	4	FspMemoryInitEntry Offset	Offset for the API to initialize the Memory. If the value is set to 0x00000000, then this API is not available in this component.
64	4	TempRamExitEntryOffset	Offset for the API to tear down the temporary memory. If the value is set to 0x00000000, then this API is not available in this component.
68	4	FspSiliconInitEntryOffset	Offset for the API to initialize the processor and chipset.

Byte Offset	Size in Bytes	Field	Description
			If the value is set to 0x00000000, then this API is not available in this component.
72	4	FspMultiPhaseSiInitEntryOffset	Offset for the API for the Multi-Phase processor and chipset initialization defined in Section 9.11. This value is only valid if FSP HeaderRevision is >= 5. If the value is set to 0x00000000, then this API is not available in this component.
76	2	ExtendedImageRevision	This value is only valid if FSP HeaderRevision is >= 6. ExtendedImageRevision specifies the high-order byte of the revision and build number in the FSP binary revision. 7 : 0 - High Byte of Build Number 15 : 8 - High Byte of Revision The FSP binary build number can be decoded as follows: Build Number = ExtendedImageRevision[7:0] << 8) | ImageRevision[7:0] Revision = (ExtendedImageRevision[15:8] << 8) | ImageRevision[15:8] Minor Version = ImageRevision[23:16] Major Version = ImageRevision[31:24]
78	2	Reserved4	
80	4	FspMultiPhaseMemInitEntryOffset	Offset for the API for the Multi-Phase memory initialization defined in Section 9.11. This value is only valid if FSP HeaderRevision is >= 7. If the value is set to 0x00000000, then this API is not available in this component.
84	4	FspSmmInitEntryOffset	Offset for the API to initialize SMM defined in Section 9.12. This value is only valid if FSP HeaderRevision is >= 7. If the value is set to 0x00000000, then this API is not available in this component.

5.2 FSP_INFO_EXTENDED_HEADER

The **FSP_INFO_EXTENDED_HEADER** structure conveys additional information aboutthe FSP binary component. This allows FSP producers to provide additional information about the FSP instantiation.

Table 2. FSP_INFO_EXTENDED_HEADER

Byte Offset	Size in Bytes	Field	Description
0	4	Signature	‘FSPE’. Signature for the FSP_INFO_EXTENDED_HEADER.
4	4	Length	Length of the table in bytes, including all additional FSP producer defined data.
8	1	Revision	FSP producer defined revision of the table.
9	1	Reserved	Reserved for future use.
10	6	FspProducerId	FSP producer identification string.
16	4	FspProducerRevision	FSP producer implementation revision number. Larger numbers are assumed to be newer revisions.
20	4	FspProducerDataSize	Size of the FSP producer defined data (n) in bytes.
24	n	…	FSP producer defined data of size (n) defined by FspProducerDataSize.

5.3 Locating FSP_INFO_HEADER

The **FSP_INFO_HEADER** structure is stored in a firmware file, called the **FSP_INFO_HEADER** file and is placed as the **first** firmware file within each of the FSPcomponent’s first Firmware Volume (FV). All firmware files will have a GUID that can beused to identify the files, including the **FSP_INFO_HEADER** file. The **FSP_INFO_HEADER** file GUID is **FSP_FFS_INFORMATION_FILE_GUID**

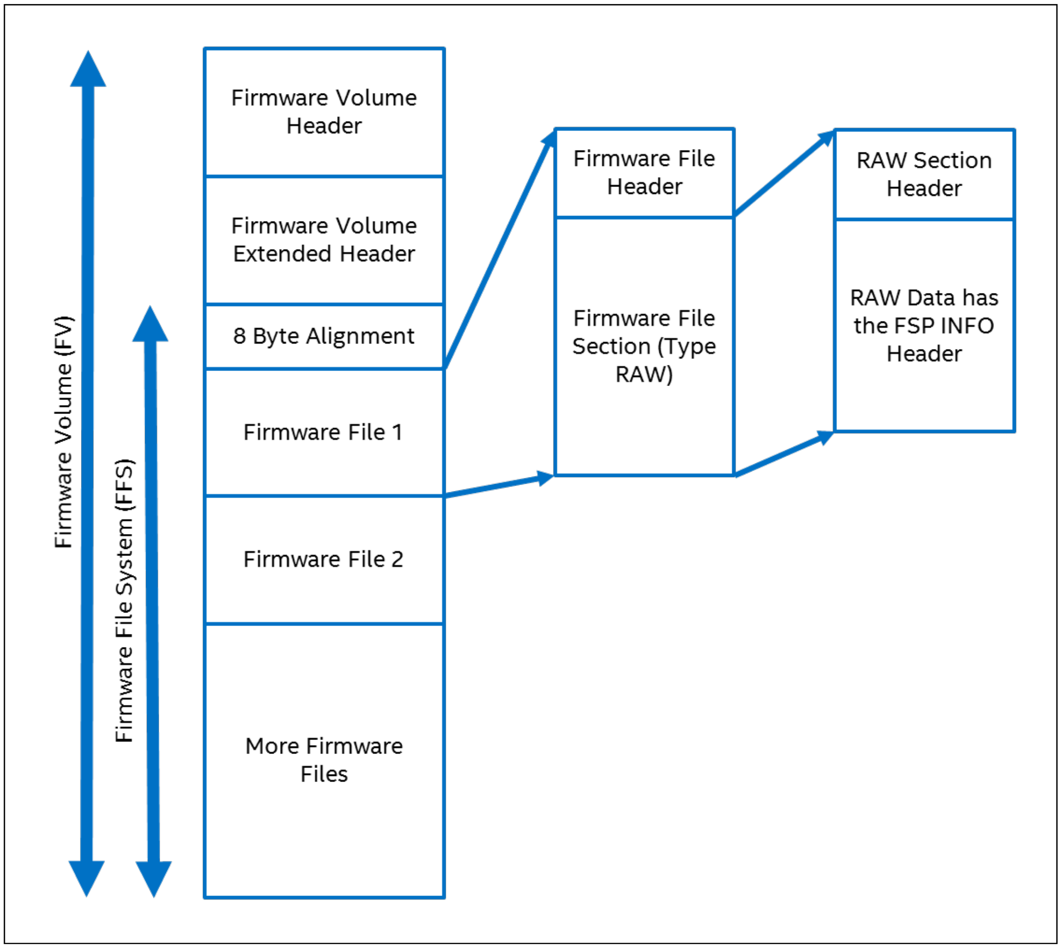
```
#define FSP_FFS_INFORMATION_FILE_GUID \
{ 0x912740be, 0x2284, 0x4734, { 0xb9, 0x71, 0x84, 0xb0,
0x27,0x35, 0x3f, 0x0c }};
```

The bootloader can find the offset of the **FSP_INFO_HEADER** within the FSP component’s first Firmware Volume (FV) by the following steps described below:

- Use **EFI_FIRMWARE_VOLUME_HEADER** to parse the FSP FV header and skip the standard and extended FV header.
- The **EFI_FFS_FILE_HEADER** with the **FSP_FFS_INFORMATION_FILE_GUID** islocated at the 8-byte aligned offset following the FV header.
- The **EFI_RAW_SECTION** header follows the FFS File Header.
- Immediately following the **EFI_RAW_SECTION** header is the raw data. The format of this data is defined in the **FSP_INFO_HEADER** and additional header structures.

A pictorial representation of the data structures that is parsed in the above flow is provided in below figure.

Figure 3. FSP Component Headers



5.4 FSP Description File

An FSP component may optionally include an FSP description file. This file will provideinformation about the FSP including information about different silicon revisions theFSP supports. The contents of the FSP description file must be an ASCII encoded text string.

The file, if present, must have the following file GUID and be included in the FDF file asshown below.

```
#define FSP_FFS_INFORMATION_FILE_GUID \
{ 0xd9093578, 0x08eb, 0x44df, { 0xb9, 0xd8, 0xd0, 0xc1,
0xd3,0xd5, 0x5d, 0x96 }};

#
# Description file
```

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

5.5 FSP Patch Table (FSPP)

FSP Patch Table contains offsets inside the FSP binary which store absolute addresses based on the FSP base. When the FSP is rebased the offsets listed in this table needs tobe patched accordingly.

A PatchEntryNum of 0 is valid and indicates that there are no entries in the patch tableand should be handled as a valid patch table by the rebasing software.

```
typedef struct {
 UINT32 Signature;   ///< FSP Patch Table Signature "FSPP"
 UINT16 Length;      ///< Size including the PatchData
 UINT8  Revision;    ///< Revision is set to 0x01
 UINT8  Reserved;
 UINT32 PatchEntryNum; ///< Number of entries to Patch
 UINT32 PatchData[];   ///< Patch Data
} FSP_PATCH_TABLE;
```

Table 3. FSPP – PatchData Encoding

BIT [23:00]	Image OFFSET to patch
BIT [27:24]	Patch type 0000: Patch DWORD at OFFSET with the delta of the new and old base. NewValue = OldValue + (NewBase - OldBase) 1111: Same as 0000 Others: Reserved
BIT [28:30]	Reserved
BIT [31]	0: The FSP image offset to patch is determined by Bits[23:0] 1: The FSP image offset to patch is calculated by (ImageSize – (0x1000000 – Bits[23:0])) If the FSP image offset to patch is greater than the ImageSize in the FSP_INFO_HEADER, then this patch entry should be ignored.

5.5.1 Example

Let's assume the FSP image size is 0x38000. And we need to rebase the FSP base from0xFFFC0000 to 0xFFF00000.

Below is an example of the typical implementation of the FSP_PATCH_TABLE:

```
FSP_PATCH_TABLE mFspPatchTable =
{
 0x50505346,   ///< Signature (FSPP)
 16,           ///< Length;
 0x01,         ///< Revision;
 0x00,         ///< Reserved;
 1,            ///< PatchEntryNum;
 {
  0xFFFFFFFC   ///< Patch FVBASE at end of FV
 }
};
```

Looking closer at the patch table entries:

```
0xFFFFFFFC,    ///< Patch FVBASE at end of FV
```

The image offset to patch in the FSP image is indicated by BIT[23:0], 0xFFFFFC. SinceBIT[31] is 1, the actual FSP image offset to patch should be:

> ImageSize – (0x1000000 – 0xFFFFFC) = 0x38000 – 4 = 0x37FFC

If the DWORD at offset 0x37FFC in the original FSP image is 0xFFFC0000, then the newvalue should be:

> OldValue + (NewBase - OldBase) = 0xFFFC0000 + (0xFFF00000 – 0xFFFC0000) =0xFFF00000

Thus, the DWORD at FSP image offset 0x37FFC should be patched to xFFF00000 afterthe rebasing.

§§

6.0 FSP Configuration Data

Each FSP module contains a configurable data region which can be used by the FSP during initialization. This configuration region is a data structure called the Updateable Product Data (UPD) and will contain the default parameters for FSP initialization. The UPD data structure is only used by the FSP when the FSP is being invoked using the APImode interface defined in *Section 8.0*.

When the FSP is invoked according to the dispatch mode interface defined in *Section 10.0*, the UPD configuration region and the UPD data structure are not used by the FSP.In dispatch mode, the PPI database and PCD database are shared between the boot loader and the FSP. Because they are shared, the UPD configuration region is not needed to provide a mechanism to pass configuration data from the bootloader to the FSP. Instead, configuration data is communicated to the FSP using PCD and PPI. The bootloader may utilize the UPD to influence PCD and PPI contents provided to the FSP in dispatch mode.

The UPD parameters can be statically customized using a separate Binary ConfigurationTool (BCT). There will be a Boot Setting File (BSF) provided along with FSP binary to describe the configuration options within the FSP. This file contains the detailed information on all configurable options, including description, help information, valid value range and the default value.

The UPD data can also be dynamically overridden by the bootloader during runtime inaddition to static configuration. Platform limitations like lack of updateable memory before calling *TempRamInit()* API may pose restrictions on the FSP-T data runtime update. Any such restrictions will be documented in the Integration Guide.

The UPD data is organized as a structure. The *TempRamInit()*, *FspMemoryInit()* and *FspSiliconInit()* API parameters include a pointer which can be initialized to point to the UPD data structure. If this pointer is initialized to NULL when calling these APIs', the FSP will use the default built-in UPD configuration data in the respective FSP components. However, if the bootloader needs to update any of the UPD parameters, it is recommended to copy the whole UPD structure from the FSP component to memory, update the parameters and initialize the UPD pointer to the address of the updated UPD structure. The FSP API will then use this data structure instead of the default configuration region data for platform initialization. The UPD data structure is a project specific structure. Please refer to the *Integration Guide* for the details of this structure.

The UPD structure has some standard fields followed by platform specific parametersand the UPD structure definition will be provided as part of the FSP distribution package.

6.1 UPD Standard Fields

The first few fields of the UPD Region are standard for all FSP implementations as documented below.

Table 4. UPD Standard Fields

Offset	Field
0x00 – 0x07	UPD Region Signature. The signature will be "XXXXXX_T" for FSP-T "XXXXXX_M" for FSP-M "XXXXXX_S" for FSP-S "XXXXXX_I" for FSP-I Where XXXXXX is a unique signature
0x08	Revision of the Data structure
0x09 – 0x1F	Reserved[23]
0x20 – n	Platform Specific Parameters, where the n is equal to (FSP_INFO_HEADER.CfgRegionSize – 1)

```
typedef struct {
 UINT64       Signature;
 UINT8        Revision;
 UINT8        Reserved[23];
} FSP_UPD_HEADER;
```

6.1.1 FSP-T UPD Structure

The UPD data structure definition for the FSP-T component will be provided as part ofthe FSP release package and documented in the integration guide as well.

```
typedef struct {
 FSP_UPD_HEADER     UpdHeader;
 FSPT_ARCH2_UPD     FsptArchUpd;

/**
Platform specific parameters
**/
...
} FSPT_UPD;

typedef struct {
 UINT8              Revision;
 UINT8              Reserved[3];
 UINT32             Length;
 FSP_DEBUG_HANDLER  FspDebugHandler;
 UINT8 Reserved1[20];
} FSPT_ARCH_UPD;

typedef struct {
 UINT8              Revision;
 UINT8              Reserved[3];
 UINT32             Length;
 EFI_PHYSICAL_ADDRESS FspDebugHandler;
 UINT32              FspTemporaryRamSize;
 UINT8 Reserved1[12];
} FSPT_ARCH2_UPD;
```

Revision	Revision of the structure. If this value is 1 then the structure definition shall be FSPT_ARCH_UPD. If this value is >=2 then the structure definition shall be **FSPT_ARCH2_UPD**. All FSP implementations compliant to v2.4 or newer version of this specification shall use FSPT_ARCH2_UPD regardless of whether 32-bit x86 or 64-bit x64 mode is used. FspTemporaryRamSize is added to the structure **FSPT_ARCH2_UPD** staring from v2.5 of this specification. The current value of Revision is 3 for this version of the specification.
Length	Length of the structure in bytes. The current value for thisfield is 32.
FspDebugHandler	Optional debug handler for the bootloader to receive debug messages occurring during FSP execution. This function shall have a signature matching **FSP_DEBUG_HANDLER**. Refer to *Section 9.5* for more details.
FspTemporaryRamSize	If BIT4 in the *ImageAttribute* field of the **FSP_INFO_HEADER** is set, bootloader can pass temporary RAM size as input to *FspTempRamInit*() API. Value passed by Bootloader shall not exceed the maximum temporary Ram size defined in the Integration Guide. If BIT4 in the ImageAttribute field of the FSP_INFO_HEADER is not set, bootloader must pass a value of 0 or FSP defined default value as FspTemporaryRamSize. If bootloader passes a value of 0 as FspTemporaryRamSize then FspTempRamInit () API will use FSP defined default value.

6.1.2 FSP-M UPD Structure

The UPD data structure definition for the FSP-M component will be provided as part ofthe FSP release package and documented in the integration guide as well.

```
typedef struct {
 FSP_UPD_HEADER      UpdHeader;
 FSPM_ARCH2_UPD      FspmArchUpd;

/**
Platform specific parameters
**/
...
} FSPM_UPD;
```

```
typedef struct {
 UINT8              Revision;
 UINT8              Reserved[3];
 VOID               *NvsBufferPtr;
 VOID               *StackBase;
 UINT32             StackSize;
 UINT32             BootLoaderTolumSize;
 UINT32             BootMode;
 FSP_EVENT_HANDLER  FspEventHandler;
 UINT8              Reserved1[4];
} FSPM_ARCH_UPD;

 typedef struct {
 UINT8                Revision;
 UINT8                Reserved[3];
 UINT32               Length;
 EFI_PHYSICAL_ADDRESS NvsBufferPtr;
 EFI_PHYSICAL_ADDRESS StackBase;
 UINT64               StackSize;
 UINT32               BootLoaderTolumSize;
 UINT32               BootMode;
 EFI_PHYSICAL_ADDRESS FspEventHandler;
 UINT8                Reserved1[16];
} FSPM_ARCH2_UPD;
```

Revision	Revision of the structure. If this value is 1 or 2 then thestructure definition shall be **FSPM_ARCH_UPD**. If this value is 3 then the structure definition shall be **FSPM_ARCH2_UPD**. The current value of Revision is 3 for this version of the specification. All FSP implementations compliant to v2.4 of this specification shall use **FSPM_ARCH2_UPD** regardless of whether 32-bit x86 or 64-bit x64 mode is used.
Length	Length of the structure in bytes. The current value for this field is 64. This value only exists if Revision >= 3.
NvsBufferPtr	This value is deprecated starting with v2.4 of this specification and will be removed in an upcoming version of this specification. If BIT3 (Variable Support) in the *ImageAttribute* field of the **FSP_INFO_HEADER** isset, then this value is unused and must be set to **NULL**. In this case, the FSP shall use the FSP variable services described in *Section 9.6* instead. Pointer to the non-volatile storage (NVS) data buffer. If it is **NULL** it indicates the NVS data is not available. Refer to *Section 11.2* and *11.3* for more details.

StackBase	Pointer to the temporary RAM base address to be consumed inside *FspMemoryInit()* API. For FSP implementations compliant to v2.0 or v2.4 of this specification, the temporary RAM is used to establish a stack and a HOB heap. For FSP implementations compliant to v2.1, v2.2, or v2.3 of this specification, the temporary RAM is only used for a HOB heap. FSP implementations compliant to v2.1 through v2.3 of this specification will run on top of the stack provided by the bootloader instead of establishing a separate stack. Starting with v2.4 of this specification, the behavior from v2.0 is restored and a separate stack will be established.
StackSize	For FSP implementations compliant to v2.0 or v2.4 of this specification, the temporary RAM size used to establish a stack and HOB heap. Consumed by the FspMemoryInit() API. For FSP implementations compliant to v2.1 through v2.3 of this specification, the temporary RAM size used to establish a HOB heap inside the FspMemoryInit() API. Starting with v2.4 of this specification, the behavior from v2.0 is restored and a separate stack will be established. Refer to the Integration Guide for the minimum required temporary RAM size.
BootloaderTolumSize	Size of memory to be reserved by FSP below "top of low usable memory" for bootloader usage. Refer to *Section 11.4* for more details.
BootMode	Current boot mode. Values are defined in *Section 13.1 Appendix A* – Data Structures. Refer to the Integration Guide for supported boot modes.
FspEventHandler	Optional event handler for the bootloader to be informed of events occurring during FSP execution. This function shall have a signature matching **FSP_EVENT_HANDLER**. Refer to *Section 9.5* for moredetails. This value is only valid if Revision is >= 2.

6.1.3 FSP-S UPD Structure

The UPD data structure definition for the FSP-S component will be provided as part ofthe FSP release package and documented in the integration guide as well.

```
typedef struct {
 FSP_UPD_HEADER      UpdHeader;
 FSPS_ARCH2_UPD      FspsArchUpd;

/**
Platform specific parameters
**/
...
} FSPS_UPD;
```

```
typedef struct {
 UINT8                Revision;
 UINT8                Reserved[3];
 UINT32               Length;
 FSP_EVENT_HANDLER    FspEventHandler;
 UINT8                EnableMultiPhaseSiliconInit;
 UINT8 Reserved1[19];
} FSPS_ARCH_UPD;

typedef struct {
 UINT8                  Revision;
 UINT8                  Reserved[3];
 UINT32                 Length;
 EFI_PHYSICAL_ADDRESS  FspEventHandler;
 UINT8                  Reserved1[16];
} FSPS_ARCH2_UPD;
```

Revision	Revision of the structure. If this value is 1 then the structuredefinition shall be **FSPS_ARCH_UPD**. If this value is 2 thenthe structure definition shall be **FSPS_ARCH2_UPD**. The current value of Revision is 2 for this version of the specification. All FSP implementations compliant to v2.4 ofthis specification shall use **FSPS_ARCH2_UPD** regardless of whether 32-bit x86 or 64-bit x64 mode is used.
Length	Length of the structure in bytes. The current value for thisfield is 32.
FspEventHandler	Optional event handler for the bootloader to be informed ofevents occurring during FSP execution. This function shall have a signature matching **FSP_EVENT_HANDLER**. Refer to *Section 9.5* for more details.
EnableMultiPhaseSiliconInit	This value is deprecated and has been removed starting with v2.4 of this specification. Multi-phase silicon initialization is mandatory for all FSP implementations compliant to v2.4 of this specification, see *Section 9.11* for further details. For FSP implementations compliant to v2.2 through v2.3 of this specification, an FSP binary may optionally implement multi-phase silicon initialization, see Section 9.11 for further details. This is only supported if the *FspMultiPhaseSiInitEntryOffse*t field in FSP_INFO_HEADER is non-zero, see Section 5.1.1 for further details. To enable multi-phase silicon initialization, the bootloader must set EnableMultiPhaseSiliconInit to a non- zero value.

6.1.4 FSP-I UPD Structure

If the FSP includes the FSP-I component, the UPD data structure definition for it will beprovided as part of the FSP release package and documented in the *Integration Guide*.

```
typedef struct {
 FSP_UPD_HEADER      UpdHeader;
 FSPI_ARCH_UPD       FspiArchUpd;

/**
Platform specific parameters
**/
...
} FSPI_UPD;

typedef struct {
 UINT8                Revision;
 UINT8                Reserved[3];
 UINT3               Length;
 EFI_PHYSICAL_ADDRESS BootloaderSmmFvBaseAddress;
 UINT64               BootloaderSmmFvLength;
 EFI_PHYSICAL_ADDRESS BootloaderSmmFvContextData;
 UINT16               BootloaderSmmFvContextDataLength;
 UINT8                Reserved1[30];
} FSPI_ARCH_UPD;
```

Revision	Revision of the structure is 1 for this version of the specification.
Length	Length of the structure in bytes. The current value for this field is 64.
BootloaderSmmFvBaseAddress	The physical memory-mapped base address of the bootloader SMM firmware volume (FV).
BootloaderSmmFvLength	The length in bytes of the bootloader SMM firmware volume (FV).
BootloaderSmmFvContextData	The physical memory-mapped base address of the bootloader SMM FV context data. This data is provided to bootloader SMM drivers through a HOB by the FSP MM Foundation. Please see Section 11.8 for details.
BootloaderSmmFvContextDataLength	The length in bytes of the bootloader SMM FV context data. This data is provided to bootloader SMM drivers through a HOB by the FSP MM Foundation. Please see Section 11.8 for details.

§§

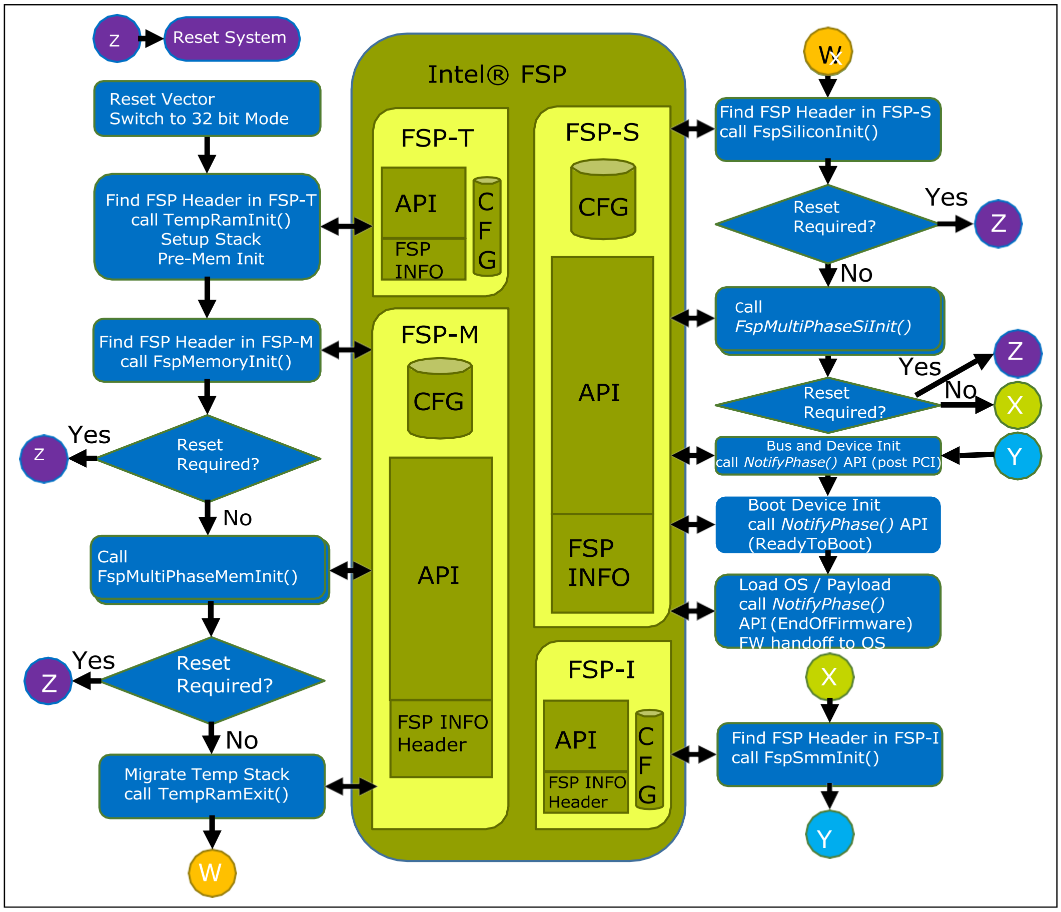
7.0 Boot Flow

The FSP v2.1 specification defines two possible FSP boot flows. The first boot flow isthe "API mode" boot flow. This boot flow is very similar to the boot flow defined in the FSP v2.0 specification. This specification also defines the "dispatch mode" boot flow. It is not required for a specific implementation of FSP to support the dispatch mode bootflow. The API mode boot flow is mandatory for all FSP implementations.

FSP_INFO_HEADER indicates if dispatch mode is supported by the FSP.

7.1 API Mode Boot Flow

Figure 4. API Mode Boot Flow



7.1.1 Boot Flow Description

- Bootloader starts executing from Reset Vector.
  - Switches the mode to 32-bit mode.

- If 64-bit mode support is indicated by FSP_INFO_HEADER.ImageAttribute[2], switch to x64 long mode and execute the remaining steps in this mode.
- Initializes the early platform as needed.
  - Finds FSP-T and calls the *TempRamInit()* API. The bootloader also has the option to initialize the temporary memory directly, in which case this step and step 2 are skipped.

- FSP initializes temporary memory and returns from TempRamInit() API.

- Bootloader initializes the stack in temporary memory.
  - Initializes the platform as needed.
    - Finds FSP-M and calls the *FspMemoryInit()* API.

- FSP initializes memory and returns from *FspMemoryInit()* API.
  - If the *FspMemoryInit()* API returns the status code `FSP_STATUS_VARIABLE_REQUEST:`
  - The bootloader shall call the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseGetVariableRequestInfo* parameter to get the details of the requested non-volatile data access request.
  - Bootloader shall perform the access request and return the results to the FSP by calling the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter.
    - The call to the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter could return `FSP_STATUS_VARIABLE_REQUEST` again. In this scenario, the previous calling sequence of *EnumMultiPhaseGetVariableRequestInfo* followed by *EnumMultiPhaseCompleteVariableRequest* will be repeated until *EnumMultiPhaseCompleteVariableRequest* returns a status code other than `FSP_STATUS_VARIABLE_REQUEST`. Execution of the *FspMemoryInit()* API shall not be considered complete until a statuscode other than `FSP_STATUS_VARIABLE_REQUEST` is returned.

- Bootloader calls the FspMultiPhaseMemInit() API with the EnumMultiPhaseGetNumberOfPhases parameter to discover the number of memory initialization phases supported by the FSP.

- If the number of phases returned previously is greater than zero, the Bootloader must call the FspMultiPhaseMemInit() API with the EnumMultiPhaseExecutePhase parameter n times, where n is the number of phases returned previously. Bootloader may perform board specific code in between each phase as needed.
  - The number of phases, what is done during each phase, and anything the bootloader may need to do in between phases shall be described in the Integration Guide.
  - If the FspMultiPhaseMemInit() API returns the status code FSP_STATUS_VARIABLE_REQUEST:
    - The bootloader shall call the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseGetVariableRequestInfo* parameter to get the details of the requested non-volatile data access request.
    - Bootloader shall perform the access request and return the results to the FSP by calling the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter.
    - The call to the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter could return FSP_STATUS_VARIABLE_REQUEST again. In this scenario, the previous

calling sequence of *EnumMultiPhaseGetVariableRequestInfo* followed by *EnumMultiPhaseCompleteVariableRequest* will be repeated until *EnumMultiPhaseCompleteVariableRequest* returns a status code other than FSP_STATUS_VARIABLE_REQUEST. Execution of *EnumMultiPhaseExecutePhase* shall not be considered complete until a status code other than FSP_STATUS_VARIABLE_REQUEST is returned.

- Bootloader relocates itself to Memory.
- Bootloader calls *TempRamExit()* API. If Bootloader initialized the temporary memory in step 1)d)… this step and the next step are skipped.
- FSP returns from *TempRamExit()* API.
- Bootloader finds FSP-S and calls *FspSiliconInit()* API.
- FSP returns from *FspSiliconInit()* API.
    - If the *FspSiliconInit()* API returns the status code FSP_STATUS_VARIABLE_REQUEST:
        - The bootloader shall call the FspMultiPhaseSiInit() API with the EnumMultiPhaseGetVariableRequestInfo parameter to get the details of the requested non-volatile data access request.
        - Bootloader shall perform the access request and return the results to the FSP by calling the FspMultiPhaseSiInit() API with the EnumMultiPhaseCompleteVariableRequest parameter.
        - The call to the FspMultiPhaseSiInit() API with the EnumMultiPhaseCompleteVariableRequest parameter could return

          FSP_STATUS_VARIABLE_REQUEST again. In this scenario, the previous calling sequence of EnumMultiPhaseGetVariableRequestInfo followed by EnumMultiPhaseCompleteVariableRequest will be repeated until EnumMultiPhaseCompleteVariableRequest returns a status code other than FSP_STATUS_VARIABLE_REQUEST. Execution of the FspSiliconInit() API shall not be considered complete until a status code other than FSP_STATUS_VARIABLE_REQUEST is returned.
- Bootloader calls the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseGetNumberOfPhases* parameter to discover the number of silicon initialization phases supported by the bootloader.
- If the number of phases returned previously is greater than zero, the Bootloader must call the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseExecutePhase* parameter *n* times, where *n* is the number of phases returned previously. Bootloader may perform board specific code in between each phase as needed.
    - The number of phases, what is done during each phase, and anything the bootloader may need to do in between phases shall be described in the Integration Guide.
    - If the FspMultiPhase*SiInit()* API returns the status code FSP_STATUS_VARIABLE_REQUEST:
        - The bootloader shall call the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseGetVariableRequestInfo* parameter to get the details of the requested non-volatile data access request.
        - Bootloader shall perform the access request and return the results to the FSP by calling the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter.

        - The call to the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseCompleteVariableRequest* parameter could return `FSP_STATUS_VARIABLE_REQUEST` again. In this scenario, the previous calling sequence of *EnumMultiPhaseGetVariableRequestInfo* followed by *EnumMultiPhaseCompleteVariableRequest* will be repeated until *EnumMultiPhaseCompleteVariableRequest* returns a status code other than `FSP_STATUS_VARIABLE_REQUEST`. Execution of the *EnumMultiPhaseExecutePhase* shall not be considered complete until a status code other than `FSP_STATUS_VARIABLE_REQUEST` is returned.

- If the FSP includes the FSP-I component, bootloader finds FSP-I and calls

  FspSmmInit() API.

    - FSP-I copies its SMM code into SMRAM.
    - FSP programs the SMBASE register value for all the threads and programs the SMRR.
    - FSP can enable and handle SMI sources as required.
    - FSP dispatches bootloader provided SMM drivers.
    - FSP closes and locks SMRAM.
        - FSP returns to bootloader.

- Bootloader continues and device enumeration.
- Bootloader calls *NotifyPhase()* API with *AfterPciEnumeration* parameter.
- Bootloader calls *NotifyPhase()* API with *ReadyToBoot* parameter before transferring control to OS loader.
- When booting to a non-UEFI OS, Bootloader calls *NotifyPhase()* API with
- *EndOfFirmware* parameter immediately after *ReadyToBoot*.
- When booting to a UEFI OS, Bootloader calls NotifyPhase() with *EndOfFirmware* parameter during *ExitBootServices*.

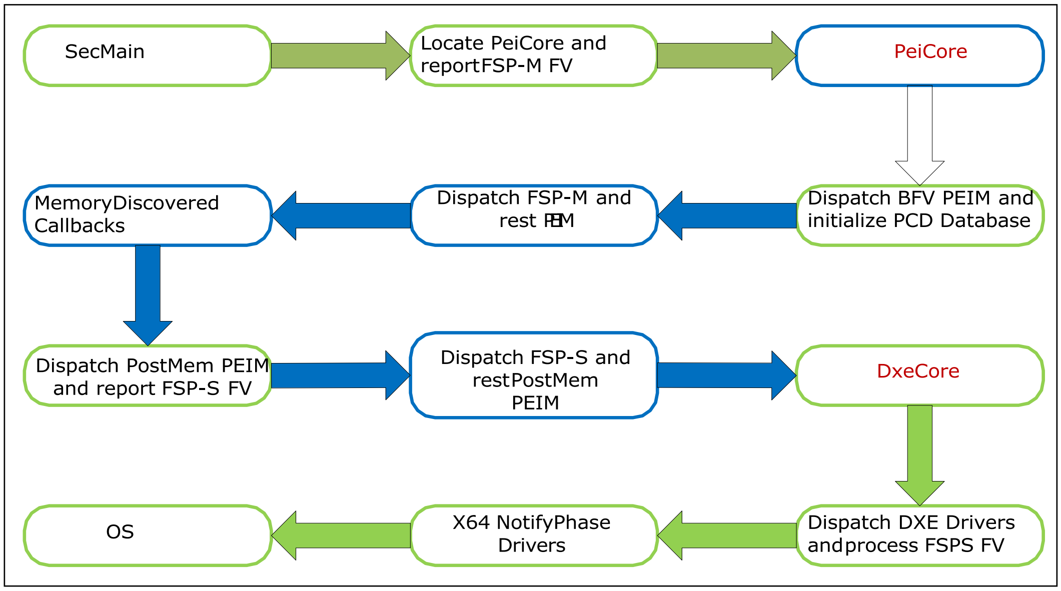
Note: If FSP returns the reset required status in any of the APIs', then bootloader performs the reset. Refer to the *Integration Guide* for more details on Reset Types.

7.2 Dispatch Mode Boot Flow

Dispatch mode is an optional boot flow intended to enable FSP to integrate well in to UEFI bootloader implementations. Implementation of this boot flow necessitates that the underlying FSP implementation uses the Pre-EFI Initialization (PEI) environment defined in the *PI Specification*. It is possible to implement an FSP without using PEI, so bootloaders must check that dispatch mode is available using the **FSP_INFO_HEADER**,see *Section 5.1.1* for further details. The *Integration Guide* will also specify if an FSP implements dispatch mode. Refer *Section 10.0* for a full description of dispatch mode.

7.2.1 High Level Overview

Figure 5. Dispatch Mode Boot Flow


SecMain
Locate PeiCore and reportFSP-M FV
PeiCore
Dispatch BFV PEIM and initialize PCD Database
Dispatch FSP-M and rest PEIM
MemoryDiscovered Callbacks
Dispatch PostMem PEIM and report FSP-S FV
Dispatch FSP-S and restPostMem PEIM
DxeCore
Dispatch DXE Drivers andprocess FSPS FV
X64 NotifyPhase Drivers
OS

Blue blocks are from the FSP binary and green blocks are from the bootloader. Blocks with mixed colors indicate that both bootloader and FSP modules are dispatched during that phase of the boot flow.

Dispatch mode is intended to implement a boot flow that is as close to a standard UEFI boot flow as possible. In dispatch mode, FSP exposes Firmware Volumes (FV) directly to the bootloader. The PEIM in these FV are executed directly in the context of the PEI environment provided by the boot loader. FSP-T, FSP-M, and FSP-S could contain one or multiple FVs. The exact FVs layout will be described in the Integration Guide. In dispatch mode, the PPI database, PCD database, and HOB list are shared between the boot loader and the FSP.

In dispatch mode, the NotifyPhase() API is not used. Instead, FSP-S contains DXE drivers that implement the native callbacks on equivalent events for each of the NotifyPhase() invocations.

7.2.2 Boot Flow Description

This boot flow assumes that the bootloader is a typical UEFI firmware implementationconforming to the *PI Specification*. Therefore, the bootloader will follow the standardfour phase PI boot flow progressing from SEC phase, to PEI phase, to DXE phase, to BDS phase.

- Bootloader provided SEC phase starts executing from Reset Vector.
    - Switches the mode to 32-bit mode.
    - If 64-bit mode support is indicated by FSP_INFO_HEADER.ImageAttribute[2], switch to x64 long mode and execute the remaining steps in this mode.
    - Initializes the early platform as needed.

- Finds FSP-T and calls the *TempRamInit()* API. SEC also has the option to initialize the temporary memory directly, in which case this step andstep 2 are skipped.

- FSP initializes temporary memory and returns from *TempRamInit()* API.
- SEC initializes the stack in temporary memory.
- SEC finds FSP-M and adds an instance of **EFI_PEI_CORE_FV_LOCATION_PPI**
- containing the address of FSP-M to the PpiList passed into PEI core.
- SEC calls the entry point for the PEI core inside FSP-M.
    - Boot loader passes the FSP-M PEI core a **EFI_SEC_PEI_HAND_OFF** data structure with the *BootFirmwareVolumeBase* and *BootFirmwareVolumeSize* members pointing to a FV provided by theplatform.
        - The bootloader provides the Boot Firmware Volume (BFV). Consequently, in FSP dispatch mode PEI core is not in the BFV unlike most UEFI firmware implementations.
- PEI core dispatches the PEIM in the BFV provided by the bootloader.
- Bootloader installs **FSPM_ARCH_CONFIG_PPI**.
- One of the PEIM provided by the bootloader installs a **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** for each FV contained in FSP-M.
    - The bootloader must not install the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI(s)** for FSP-M until the bootloader is ready for FSP-M to execute.
    - If FSP-M requires any DynamicEx PCD values, the bootloader must ensure those PCD contain valid data before installing the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI(s)** for FSP-M.
- PEI core will continue to dispatch PEIM. During the course of dispatch, PEIM included with FSP-M will be executed.
    - Some of the PEIM contained in FSP-M may require configuration datato be provided by the bootloader. If this is the case, the configuration data may be stored in either DynamicEx PCD or PPI.
        - If the configuration data is stored in PCD, then it is assumed that the PCD contains valid data before FSP-M begins execution.
        - If the configuration data is stored in PPI, then the needed PPI will either be in the PEIM's DEPEX, or the PEIM will register a callback for the needed PPI and not attempt to access the PPI until the callback is invoked by PEI core.
- FSP-M installs **FSP_TEMP_RAM_EXIT_PPI.**
- After dispatching the PEIM in FSP-M, memory will be initialized. Accordingly,FSP-M will call *(*PeiServices)->InstallPeiMemory()*.
    - PEI core shadows to main memory.
    - PEI core invokes TemporaryRamDone() from **EFI_PEI_TEMPORARY_RAM_DONE_PPI.** The implementation of **EFI_PEI_TEMPORARY_RAM_DONE_PPI** is provided by the bootloader.
    - The bootloader implementation of **EFI_PEI_TEMPORARY_RAM_DONE_PPI** calls TempRamExit() from **FSP_TEMP_RAM_EXIT_PPI.**
        - For platforms that use the SEC implementation in UefiCpuPkg, SEC core implements **EFI_PEI_TEMPORARY_RAM_DONE_PPI**. The

TemporaryRamDone() implementation in SEC core will call *SecPlatformDisableTemporaryMemory()*. This function would then locate **FSP_TEMP_RAM_EXIT_PPI** and call *TempRamExit()*.

        - If the bootloader did not call *TempRamInit()* in step 1.d) thenthe bootloader would not call *TempRamExit()*.

    - PEI core follows up with an installation of the **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI**. Refer to Volume 1 of the *PI Specification* for details.

- Post memory PEIM provided by the bootloader are now executed.
- One of the PEIM provided by the bootloader installs a

EFI_PEI_FIRMWARE_VOLUME_INFO_PPI for each FV contained in FSP-S.

    - The bootloader must not install the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI(s)** for FSP-S until the bootloader is ready for FSP-S to execute.
    - If FSP-S requires any DynamicEx PCD values, the bootloader must ensure those PCD contain valid data before installing the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI(s)** for FSP-S.

- PEI core will continue to dispatch PEIM. During the course of dispatch, PEIM included with FSP-S will be executed.
    - Some of the PEIM contained in FSP-S may require configuration datato be provided by the bootloader. If this is the case, the configuration data may be stored in either DynamicEx PCD or PPI.
        - If the configuration data is stored in PCD, then it is assumed that the PCD contain valid data before FSP-S begins execution.
        - If the configuration data is stored in PPI, then the needed PPI will either be in the PEIM's DEPEX, or the PEIM will register a callback for the needed PPI and not attempt to access the PPI until the callback is invoked by PEI core.
- If the FSP includes the FSP-I component and the bootloader chooses to useFSP SMM Model 2 (FSP owns SMRAM), bootloader finds FSP-I and calls *FspSmmInit()* API.
    - FSP-I copies its SMM code into SMRAM.
    - FSP programs the SMBASE register value for all the threads and programs the SMRR.
    - FSP can enable and handle SMI sources as required.
    - FSP dispatches bootloader provided SMM drivers.
    - FSP closes and locks SMRAM.
    - FSP returns to bootloader.
- If (1) the FSP includes the FSP-I component, (2) the bootloader chooses to use FSP SMM Model 3 (Bootloader provides the MM Foundation), and (3) the bootloader chooses to initialize the MM Foundation in post-memory PEI:
    - Bootloader copies its SMM code into SMRAM.
    - Bootloader programs the SMBASE register value for all the threads and programs the SMRR.
    - Bootloader enables and handles SMI sources as required.
    - Bootloader dispatches bootloader provided SMM drivers.
    - Bootloader dispatches FSP provided SMM drivers

  - Bootloader closes and locks SMRAM.
  - Bootloader returns to PEI.

- End of PEI is reached, and DXE begins execution.
- Any DXE drivers included in FSP-S are dispatched. These drivers may create events to be notified at different points in the boot flow. FSP shall use a subset of the events defined by the *PI Specification*, Refer *Section 10.3* for the full list ofevents the FSP may use.
- If (1) the FSP includes the FSP-I component, (2) the bootloader chooses to use FSP SMM Model 3 (Bootloader provides the MM Foundation), and (3) the bootloader chooses to initialize the MM Foundation in DXE:
  - Bootloader copies its SMM code into SMRAM.
  - Bootloader programs the SMBASE register value for all the threads and programs the SMRR.
  - Bootloader enables and handles SMI sources as required.
  - Bootloader dispatches bootloader provided SMM drivers.
  - Bootloader dispatches FSP provided SMM drivers
  - Bootloader closes and locks SMRAM.
  - Bootloader returns to DXE.
- DXE signals **EFI_END_OF_DXE_EVENT_GROUP_GUID** and transitions to BDSphase.

Note: The *PI Specification* does not require that Step 19 occurs beforeStep 21, however most implementations appear to use this order.

- BDS starts the PCI bus driver, which enumerates PCI devices. After enumeration, the PCI bus driver installs the **EFI_PCI_ENUMERATION_PROTOCOL**. DXE signals any applicable events.
- BDS signals **EFI_EVENT_GROUP_READY_TO_BOOT** immediately before loading the OS boot loader.
- BDS executes the OS boot loader. The OS boot loader loads the OS kernel into memory.
- The OS boot loader calls ExitBootServices(), DXE signals this event before shutting down the UEFI Boot Services.

7.2.3 Alternate Boot Flow Description

In some scenarios, the bootloader may wish to use a customized version of the PEI Foundation. For example, many software debugger implementations need to be linked with PEI core directly. For this reason, as an alternative to using the PEI core included with FSP-M, the bootloader may instead elect to use its own implementation of PEI core. In this case, the bootloader provided SEC will not produce the **EFI_PEI_CORE_FV_LOCATION_PPI**, and instead of calling the entry point for the PEI core inside FSP-M it shall call the entry point for the PEI core inside the BFV. Note that this will result in two copies of PEI core being present in the final image, one in the BFV and one in the FSP-M. If firmware storage space is under pressure, one may elect topost process FSP-M using Intel® FMMT to remove the PEI core included with FSP.

This is generally considered to be a debug feature, and is discouraged for use in a production environment as it deviates from the boot flow that receives the most validation. It is also inefficient due to the duplicate copy of PEI core it introduces.

- Bootloader provided SEC phase starts executing from Reset Vector.
    - Switches the mode to 32-bit mode.
    - If 64-bit mode support is indicated by **FSP_INFO_HEADER.ImageAttribute[2]**, switch to x64 long mode and execute the remaining steps in this mode.
    - Initializes the early platform as needed.
  - Finds FSP-T and calls the *TempRamInit()* API. SEC also has the option to initialize the temporary memory directly, in which case this step and step 2are skipped.
- FSP initializes temporary memory and returns from TempRamInit() API.
- SEC initializes the stack in temporary memory.
- SEC calls the entry point for the PEI core inside the Boot Firmware Volume (BFV).
- PEI core dispatches the PEIM in the BFV provided by the bootloader.
- Boot loader installs **FSPM_ARCH_CONFIG_PPI**.
- One of the PEIM provided by the bootloader installs a **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** for each FV contained in FSP-M.
    - The bootloader must not install the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI**(s) for FSP-M until the bootloader is ready for FSP-M to execute.
  - If FSP-M requires any DynamicEx PCD values, the bootloader must ensurethose PCD contain valid data before installing the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI**(s) for FSP-M.
- PEI core will encounter a second PEI core in FSP-M. Because it is not a PEIM, the dispatcher will skip it. PEI core will proceed to dispatch the PEIM in FSP-M.
- The boot flow proceeds the same as step 9 in the primary boot flow from here forwards.

§§

8.0 System Management Mode

System Management Mode (SMM) is a special-purpose operating mode provided for handling system-wide functions including certain system hardware control operationsduring system runtime. Such operations are required to address various situations fromsimple use cases like writing to the flash part to complex proprietary algorithms like initializing the memory controller for a memory online operation.

SMM provides a mechanism to run trusted firmware code during runtime to address these use cases and is intended for use only by system firmware.

The processor executes SMM code in a separate address space (SMRAM typically neartop of memory also known as TSEG) that can be made inaccessible from the other operating modes providing the necessary protection for trusted firmware code. Other SMM protections are out of scope for this document. Please refer to relevant platformspecific documentation for details.

In addition to FSP-T/M/S, an FSP binary may optionally include an FSP-I component.The FSP-I component is intended to provide SMM mode and other late silicon functionality that is executed when the system is in SMM. For example, services relatedruntime reliability/availability/serviceability like ECC error handling, error isolation to a specific DIMM module, etc. are required during runtime. These operations require that the system is in SMM and are typically platform dependent and the system firmwarebeing the platform specific component is expected to provide these services.

The FSP-I component includes three sub-components as below

“**Standalone SmmFoundation**” along with drivers providing MM Services necessary todispatch and support the SMM mode drivers.

“**Standalone SmmIpl**” that opens the SMRAM, loads SmmFoundation in SMRAM andcloses the SMRAM after the load process is complete.

“**Standalone SmmDrivers**” that handle specific SMI sources. The standalone SmmDrivers conform to the Platform Initialization (PI) Management Mode Core Interface Specification.

The FSP-I component is designed to allow various usage models as explained below.

8.1 Model 1 - No SMM

Certain platforms are designed for workloads that may require real time and predictable response time. SMM by its nature is transparent to the operating system and may cause jitter which cannot be tolerated by such platforms. Workloads requiringfunctional safety is another example of platforms that may decide to disable SMM.

System firmware for such platforms may decide to disable SMM and not use FSP-I. If anFSP binary provides the FSP-I component, usage of FSP-I can be mandatory for certain processors and chipsets. The *Integration Guide* will document if FSP-I is mandatory for specific FSP implementations. If an FSP binary provides the FSP-I component, then

FSP-I shall support model 2 (FSP owns SMRAM) at a minimum. If the FSP binary supports dispatch mode, then model 3 (Bootloader provides the MM Foundation) mustalso be supported in addition to model 2.

8.2 Model 2 – FSP Owns SMRAM

This model is applicable for **both** (1) bootloaders implementing PI specification (Dispatch Mode) as well as (2) bootloaders that don't implement PI specification (APIMode).

In this model, the FSP-I component manages the SMRAM independently without any bootloader involvement. Bootloader can provide SMM drivers for extensibility. The bootloader does this by providing a Firmware Volume (FV) to FSP-I via the BootloaderSmmFvBaseAddress and BootloaderSmmFvLength UPDs. FSP-I will createan **EFI_HOB_FIRMWARE_VOLUME** HOB for this bootloader provided FV and insert it into the HOB list provided to the MM Foundation. In some cases, FSP-I may require thebootloader to provide services in SMM. In this scenario, FSP-I will consume those services through MM protocol(s) installed by bootloader SMM drivers. Any required SMM services will be described in the *Integration Guide*.

8.3 Model 3 – Bootloader Provides MM Foundation (Dispatch Mode Only)

This model is applicable for bootloaders that implement the PI specification (DispatchMode) only. In this mode, the bootloader provides the MM Foundation. Any Firmware Volumes (FVs) contained in the FSP-I component are registered with the bootloader provided MM Foundation and any Standalone MM drivers included in those FVs are dispatched.

In Model 3, the bootloader provided MM Foundation can optionally support TraditionalMM drivers in addition to Standalone MM drivers if desired. This Traditional MM support would only be used by bootloader SMM drivers; FSP-I shall only contain Standalone MM drivers.

In some cases, FSP-I may require the bootloader to provide services in SMM. In this scenario, FSP-I will consume those services through MM protocol(s) installed by bootloader SMM drivers. Any required SMM services will be described in the *IntegrationGuide*.

FSP-I may require specific SMI sources to be enabled for proper operation. If this is thecase, those SMI sources and any APIs used by the FSP to register handlers for those SMI sources will be documented in the *Integration Guide*.

8.4 High Level Flow

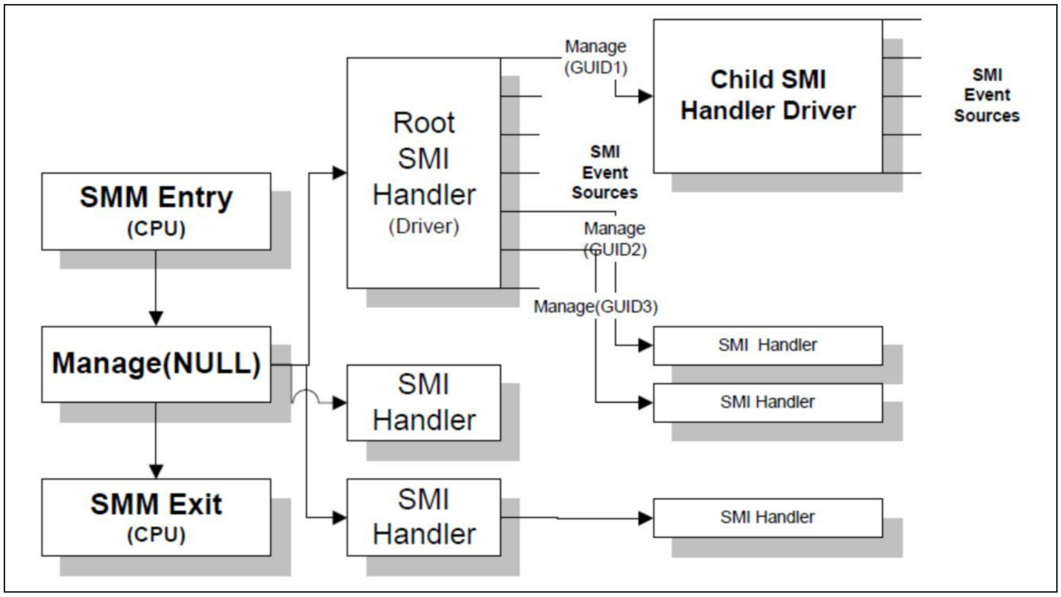
8.4.1 API Mode

A high-level boot flow involving FSP-I API mode is provided below. As the API mode isutilized by bootloaders not implementing the Platform Initialization (PI) specification, execution details have been abstracted as necessary.

1. Early initialization
2. Memory initialization including programming chipset registers reservingSMRAM memory
3. FspSiliconInit API is called by bootloader
4. Bootloader calls FspSmmInit entry point
5. FSP SMM module copies the SMM functionality code to the SMRAM
6. FSP SMM module programs the SMBASE register value for all the threads and programs the SMRR
7. FSP SMM module enables SMI sources as required
8. FSP SMM module closes and locks SMRAM
9. FSP SMM module returns to bootloader
10. Bootloader continues execution

When an SMI occurs, FSP SMM module services it, clears the SMI status bits, sets End ofSMI status, and returns from SMM to normal mode of operation.

Figure 6. SMM Drivers



8.4.2 Dispatch Mode

In model 2 (FSP owns SMRAM), the bootloader will include an FSP-I wrapper PEIM thatwill invoke the FspSmmInit entry point during post-memory PEI: after memory initialization is complete but before DXE IPL.

In model 3 (bootloader provides the MM Foundation), the bootloader will register anyFirmware Volumes (FVs) contained in FSP-I with the MM Foundation, causing SMM drivers contained in FSP-I to be dispatched along with bootloader provided SMM drivers. This can happen either in post-memory PEI or early DXE (before BDS).

§§

9.0 FSP API Mode Interface

9.1 Entry-Point Invocation Environment

There are some requirements regarding the operating environment for FSP execution. The bootloader is responsible to set up this operating environment before calling the FSP API. These conditions have to be met before calling any entry point (otherwise, thebehavior is not determined). These conditions include:

- Interrupts should be turned off.
- The FSP API should be called only by the system BSP, unless otherwise noted.
- Sufficient stack space should be available for the FSP API function to execute. Consult the Integration Guide for platform specific stack space requirements.

Specially for x86 32bit API mode:

- The system is in flat 32-bit mode.
- Both the code and data selectors should have full 4GB access range. Specially for x64 64bit API mode:
- The system is in 64-bit long mode with paging enabled.
- The full address space required by the FSP and bootloader execution shall be identity mapped (virtual address equals physical address), although the attributes of certain regions may not have all read, write, and execute attributes or be unmarked for purposes of platform protection. The mappings to other regions are undefined and may vary from implementation to implementation. Please refer to Integration Guide for page table address space range required by FSP execution.
- Selectors are set to flat and are otherwise not used.

Other requirements needed by individual FSP API will be covered in the respective sections.

9.2 Data Structure Convention

All data structure definitions should be packed using compiler provided directives suchas `#pragma pack(1)` to avoid alignment mismatch between the FSP and the bootloader.

9.3 Entry-Point Calling Convention

- All FSP APIs defined in the **FSP_INFO_HEADER** can be either 32-bit or 64-bit interface depending on FSP_INFO_HEADER.ImageAttribute BIT2 (64-bit support).

When FSP_INFO_HEADER.ImageAttribute[2] is 0, it indicates the FSP APIs providedby current FSP component only support 32-bit interfaces. Accordingly, when FSP_INFO_HEADER.ImageAttribute[2] is 1, it indicates the FSP APIs provided by current FSP component only support 64-bit interfaces.

- The FSP API 32-bit interface is similar to the default C_cdecl convention. Like the default C_cdecl convention, with the FSP API interface:
  - All parameters are pushed onto the stack in right-to-left order before the API iscalled.
- The FSP API 64-bit interface is similar to the EFIAPI calling convention defined by UEFI specification, with the FSP API interface:
  - The first 4 parameters are passed from left to right in RCX, RDX, R8 and R9 registers. The arguments five and above are passed onto the stack.
  - The 32 bytes shadow space is allocated on stack by caller before the API call.
  - A caller must always call with the stack 16-byte aligned.
- The calling function needs to clean the stack up after the API returns.
- The return value is returned in the EAX/RAX register. All the other registers including floating point registers are preserved, except as noted in the individual API descriptions below or in Integration Guide.

9.4 Return Status Code

All FSP API return a status code to indicate the API execution result. These return statuscodes are defined in *Section 13.2 Appendix A – EFI_STATUS.*

Sometimes for an initialization to take effect, a reset may be required. The FSP API mayreturn a status code indicating that a reset is required as documented in 13.2.2 OEM Status code.

When an FSP API returns one of the **FSP_STATUS_RESET_REQUIRED** codes, the bootloader can perform any required housekeeping tasks and issue the reset.

When an FSP API returns **FSP_STATUS_VARIABLE_REQUEST**, the bootloader shall perform an FSP variable access request. See *Section 9.6* for details.

9.5 FSP Events

FSP may optionally include the capability of generating events messages to aid in the debugging of firmware issues. These events fall under three categories: Error, Progress,and Debug. The event reporting mechanism follows the status code services described in Section 6 and 7 of the *PI Specification v1.7 Volume 3*.

The bootloader may provide an event handler to the FSP through the

FSPM_ARCH_UPD.FspEventHandlerand FSPS_ARCH_UPD.FspEventHandler

UPDs. Providing these event handlers is entirely optional. If the bootloader does notwish to handle FSP events, it may set these UPDs to `NULL`. FSP will only call **`FSPM_ARCH_UPD.FspEventHandler`** during FSP-M and **`FSPS_ARCH_UPD.FspEventHandler`** during FSP-S.

Due to the nature of early boot stages, FSP-T is mostly assembly code. Accordingly, FSP-T uses a simpler interface that only provides debug log messages using **`FSPT_ARCH_UPD.FspDebugHandler`**. Due to the need for a stack to be established tocall this handler, FSP-T can only call *FspDebugHandler()* after temporary memory is initialized. This may delay the output of debug log messages until later in the FSP-T flow.

The event handlers provided by the bootloader should not use more than 4KB of stackspace.

A similar feature is provided for dispatch mode, Refer *Section 10.4.12*.

9.5.1 PI Specification Architecturally Defined Status Codes

The *PI Specification* provides a rich set of status code classes and sub-classes, which may be used by the FSP. The bootloader may also parse these *PI Specification* definedstatus code events if desired.

If a bootloader chooses to implement the MIPI Sys-T specification, it is recommendedthat *PI Specification* architecturally defined status codes returned by the FSP be translated into human readable string descriptions and then output in either `MIPI_SYST_STRING_GENERIC` format or if the bootloader chooses to support `MIPI_SYST_TYPE_CATALOG` in catalog format. See Volume 3, Chapter 6 of the PI Specification for these descriptive strings. The bootloader should also provide a `MIPI_SYST_SEVERITY_*` value that is appropriate. Below is an example (but not required) mapping:

Table 5. EFI_STATUS_CODE_TYPE to MIPI_SYST_SEVERITY Mapping

Status Code Type	Status Code Severity	MIPI Sys-T Severity
EFI_DEBUG_CODE	N/A	MIPI_SYST_SEVERITY_DEBUG
EFI_PROGRESS_CODE	N/A	MIPI_SYST_SEVERITY_INFO
EFI_ERROR_CODE	EFI_ERROR_MINOR	MIPI_SYST_SEVERITY_WARNING
EFI_ERROR_CODE	EFI_ERROR_MAJOR	MIPI_SYST_SEVERITY_ERROR
EFI_ERROR_CODE	EFI_ERROR_UNRECOVERED	MIPI_SYST_SEVERITY_FATAL
EFI_ERROR_CODE	EFI_ERROR_UNCONTAINED	MIPI_SYST_SEVERITY_FATAL

9.5.2 Debug Log Messages

The FSP may use this event mechanism to provide debug log messages to the bootloader. When FSP-M or FSP-S provide debug log messages this way, the *Type* parameter's **EFI_STATUS_CODE_TYPE_MASK** will be set to **EFI_DEBUG_CODE** and the Data parameter shall contain a **EFI_STATUS_CODE_STRING_DATA** payload. Please see Section 6.6.2 of the *PI Specification v1.7 Volume 3* for details on **EFI_STATUS_CODE_STRING_DATA**. The FSP shall only pass a **EFI_STRING_TYPE** of **`EfiStringAscii`** for the purposes of debug log messages. The Instance parameter shall contain the *ErrorLevel*, Refer *Section 13.9* for details. The bootloader may parse these debug log events if desired.

If a bootloader chooses to implement the MIPI Sys-T specification, it is recommended that debug log messages provided in this way be output in `MIPI_SYST_STRING_GENERIC` format. The bootloader sets the `MIPI_SYST_SEVERITY_*`value for each message and can use the *ErrorLevel* values provided by the FSP to aid indeciding that value.

It should be noted that the strings for these log messages increase the binary size ofthe FSP considerably. Accordingly, FSP binaries intended for production use are unlikely includes debug log messages.

9.5.3 POST Progress Codes

The FSP may use this event mechanism to provide POST codes to the bootloader. IfFSP-M or FSP-S provide POST codes this way, the *Type* parameter's **EFI_STATUS_CODE_TYPE_MASK** will be set to **EFI_PROGRESS_CODE** and the *Value* parameter will have the upper 16-bits (**EFI_STATUS_CODE_CLASS_MASK** and **EFI_STATUS_CODE_SUBCLASS_MASK**) will be set to **`FSP_POST_CODE`**. The lower 16-bits (**EFI_STATUS_CODE_OPERATION_MASK**) will contain the POST code. The bootloader may parse these POST code events if desired.

9.5.4 MIPI Sys-T Catalog Debug Log Messages

The FSP may use this event mechanism to provide MIPI Sys-T catalog style debug messages. If FSP-M or FSP-S provide catalog debug messages this way, the *Type* parameter's **EFI_STATUS_CODE_TYPE_MASK** will be set to **EFI_DEBUG_CODE** and the *Value* parameter will have the upper 16-bits (**EFI_STATUS_CODE_CLASS_MASK** and **EFI_STATUS_CODE_SUBCLASS_MASK**) will be set to **FSP_CATALOG_MESSAGE**. The Instance parameter shall contain the *ErrorLevel*, please see *Section 13.9* for details. The MIPI Sys-T message's payload data will be provided via the Data parameter and will always be in MIPI_SYST_CATALOG_ID64_P64format, please see Section 9.1.4 of the MIPISys-T v1.0 Specification. The bootloader sets the MIPI_SYST_SEVERITY_* value for each message and can use the *ErrorLevel* values provided by the FSP to aid in deciding that value.

It should be noted that generating catalog style debug messages requires conversion of DebugLib style format strings to C99 style format strings. There is not a 1:1 mapping between these, and runtime data conversion is needed to generate a MIPI Sys-T message payload. The inclusion of these format strings and the code to parse and translate the variable argument list into a message payload increases the binary size of the FSP considerably and can have a substantial impact on boot performance.

Accordingly, FSP binaries intended for production use are unlikely to include catalog debug log messages.

9.5.5 Related Definitions

```
#define FSP_EVENT_CODE      0xF5000000
#define FSP_POST_CODE       (FSP_EVENT_CODE | 0x00F80000)
#define FSP_CATALOG_MESSAGE (FSP_EVENT_CODE | 0x00F90000)
```

Refer *Section* 13.10-13.11 *Appendix A – Data Structures* for the definitions of **EFI_STATUS_CODE_TYPE**, **EFI_STATUS_CODE_VALUE**, and **EFI_STATUS_CODE_DATA**.

9.5.6 FspEventHandler

Handler for FSP events, provided by the bootloader.

9.5.6.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_EVENT_HANDLER)(
 IN      EFI_STATUS_CODE_TYPE  Type,
 IN      EFI_STATUS_CODE_VALUE Value,
 IN      UINT32      Instance,
 IN OPTIONAL EFI_GUID       *CallerId,
 IN OPTIONAL EFI_STATUS_CODE_DATA  *Data
);
```

9.5.6.2 Parameters

Type	Indicates the type of event being reported. See *Section 13.10 Appendix A – Data Structures* for the definition of **EFI_STATUS_CODE_TYPE**.
Value	Describes the current status of a hardware or software entity. This includes information about the class and subclass that is used to classify the entity as well as an operation. For progress events, the operation is the current activity. For error events, it is the exception. For debug events, it is not defined at this time. See *Section 13.10 Appendix A – Data Structures* for thedefinition of **EFI_STATUS_CODE_VALUE**.
Instance	The enumeration of a hardware or software entity within the system. A system may contain multiple entities that match a class/subclass pairing. The instance differentiates between them. An instance of 0 indicates that instance information is unavailable, not meaningful, or not relevant. Valid instance numbers start with 1.
CallerId	This parameter can be used to identify the sub-module within the FSP generating the event. This parameter may be **NULL**.
Data	This optional parameter may be used to pass additional data. The contents can have event-specific data. For example, the FSP provides a **EFI_STATUS_CODE_STRING_DATA** instance to this parameter when sending debug messages. This parameter is **NULL** when no additional data is provided. See *Section 13.11 Appendix A – Data Structures* for thedefinition of **EFI_STATUS_CODE_STRING_DATA**.

9.5.6.3 Return Values

The return status will be passed back through the **EAX/RAX** register.

Table 6. Return Values - FspEventHandler()

EFI_SUCCESS	The event was handled successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_DEVICE_ERROR	The event handler failed.

9.5.7 FspDebugHandler

Handler for FSP-T debug log messages provided by the bootloader.

9.5.7.1 Prototype

```
typedef
 UINT32
(EFIAPI *FSP_DEBUG_HANDLER)(
 IN CHAR8*     DebugMessage,
 IN UINT32     MessageLength
);
```

9.5.7.2 Parameters

DebugMessage	A pointer to the debug message to be written to the log.
MessageLength	Number of bytes to written to the debug log.

9.5.7.3 Return Values

The return value will be passed back through the **EAX/RAX** register. The return valueindicates the number of bytes actually written to the debug log. If the return value is less than MessageLength, an error occurred.

9.6 FSP Variable Services

The FSP variable services enable the FSP to read and write non-volatile data. The method and implementation of non-volatile data storage can vary depending on chipset and platform design. Therefore, the FSP performs non-volatile data access indirectly through the bootloader. The bootloader exposes non-volatile data to the FSPusing an associative array abstract data type. The key-value pairs stored in this associative array shall have a key composed of a string (referred to as the variable name) and a GUID. The GUID is used to establish a namespace, so that in the case where bootloader data and FSP data is stored in a shared space, name collisions are a non-issue. The value of each key-value pair is an opaque byte array. These key-value pairs are referred to as *variables*. The FSP can read and write an arbitrary number of these key-value pairs (aka variables) during the *FspMemoryInit()* and *FspSiliconInit()* API.The FSP accesses this associative array through a set of four variable services provided by the bootloader:

Table 7. List of FSP Variable Services

GetVariable	Retrieves a variable's value using its name and GUID.
GetNextVariableName	This service is called multiple times to retrieve the name and GUID of all variables currently available in the associative array.
SetVariable	Stores a new value to the variable with the given name andGUID.
QueryVariableInfo	This service informs the FSP of how much non-volatile storage space is allocated for the storage of variables, how much is remaining, and what the maximum allowable size isfor each variable. The minimum amount of storage space required by the FSPwill be mentioned in the *Integration Guide*.

9.6.1 Variable Store Contents

The associative array that the bootloader uses to provide non-volatile data storage tothe FSP can be initially empty or can contain data for the bootloader's private use. The FSP shall not assume that any of variables used by the FSP exist. In the case where pre-existing variables do not exist, the FSP shall gracefully enter a "first boot" flow that creates any variables that the FSP needs and initializes them with appropriate data.

This will likely increase the FSP's execution time as these non-volatile data must be regenerated from scratch.

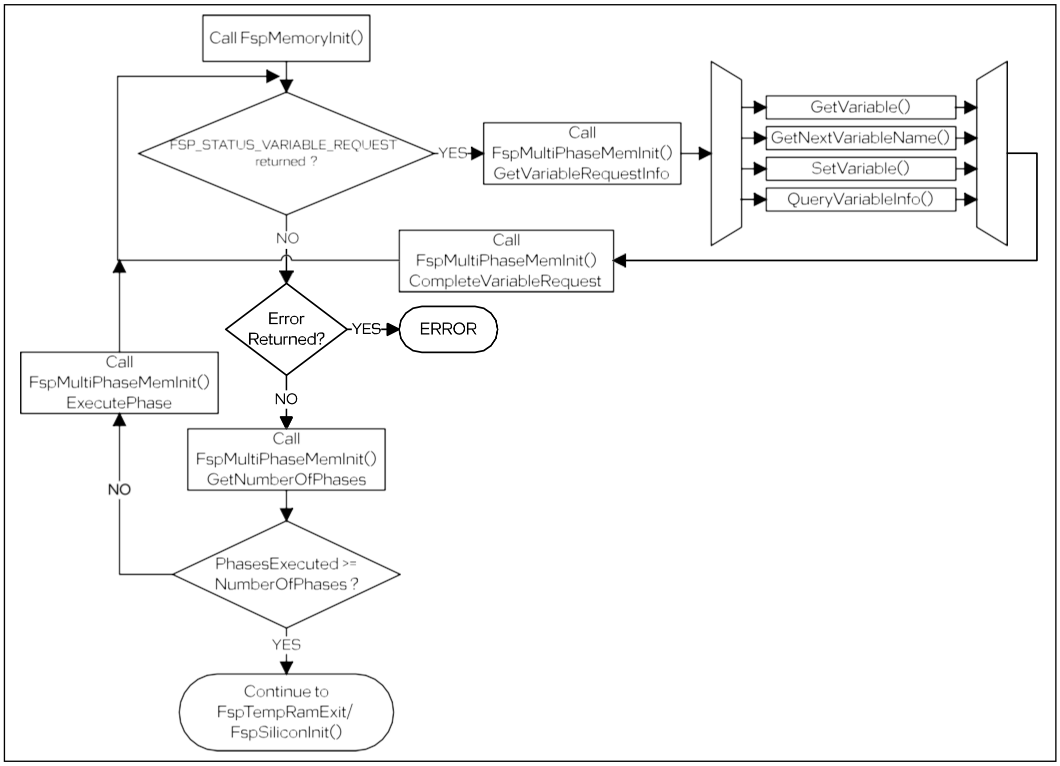
If the platform is resuming from S3, then the FSP can require that non-volatile data from the initial S5 resume does exist; if it does not then the FSP can request a reset,converting the S3 resume into an S5 resume.

The minimum amount of storage space required by the FSP will be mentioned in the Integration Guide.

9.6.2 API Mode Variable Sequence

Because access to non-volatile data can sometimes be critical to successfully completing the boot sequence, access to variables is not done through an optional function pointer like FSP events. Instead, the FSP halts execution indicates to the bootloader that a variable access request is pending and must be completed before theboot flow can continue.

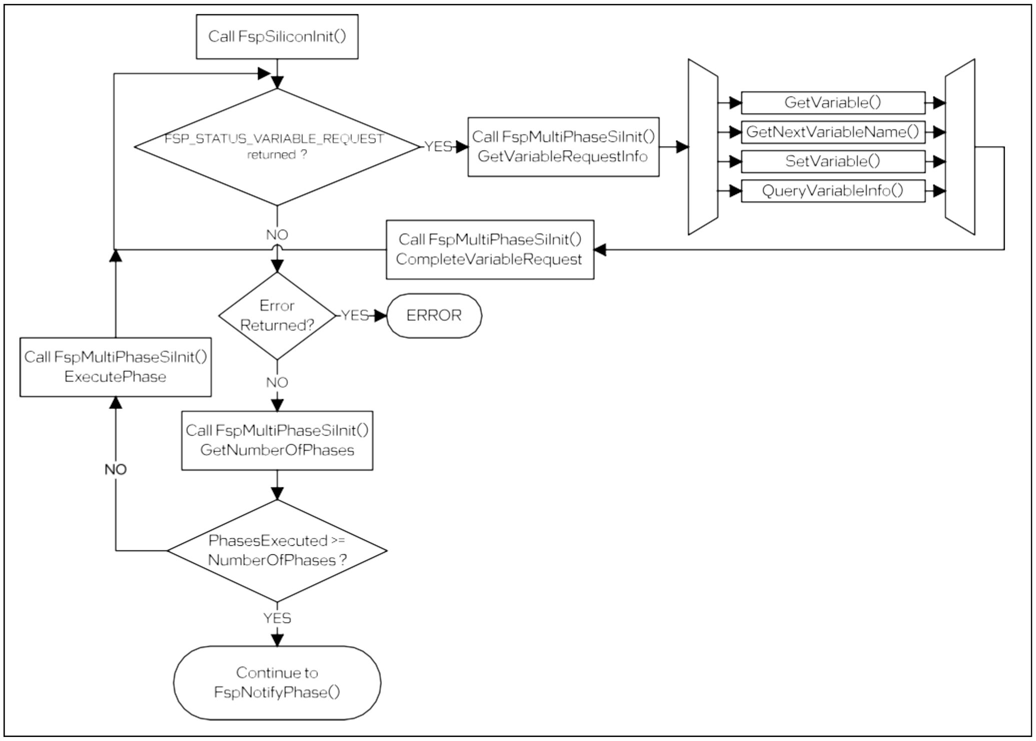
Figure 7. FspMemoryInit() Variable Services Invocation Sequence


Call FspMemoryInit()
FSP_STATUS_VARIABLE_REQUEST returned ?
YES
Call FspMultiPhaseMemInit() GetVariableRequestInfo
GetVariable()
GetNextVariableName()
SetVariable()
QueryVariableInfo()
NO
Call FspMultiPhaseMemInit() CompleteVariableRequest
Error Returned?
YES
ERROR
NO
Call FspMultiPhaseMemInit() ExecutePhase
Call FspMultiPhaseMemInit() GetNumberOfPhases
NO
PhasesExecuted >= NumberOfPhases ?
YES
Continue to FspTempRamExit/ FspSiliconInit()

The *FspMultiPhaseMemInit()* API is used to transfer information about the variable request back to the bootloader and to inform the FSP of when the variable access is complete. If *FspMemoryInit()* returns the status code FSP_STATUS_VARIABLE_REQUEST, then the *FspMultiPhaseMemInit()* API shall be called by the bootloader with the *EnumMultiPhaseGetVariableRequestInfo* action given.The FSP will populate a **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure given by the bootloader and return (see *Section 9.11*.) The bootloader parses the given data and performs the variable access. Once the access is complete, the bootloader calls the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* action to indicate to the FSP that the I/O is done and that the FSP can continue execution. FSP_STATUS_VARIABLE_REQUEST canalso be returned by *FspMultiPhaseMemInit()* when either the *EnumMultiPhaseCompleteVariableRequest* or *EnumMultiPhaseExecutePhase* actions are given. The bootloader shall be prepared to handle variable access requests in in these scenarios as well.

The variable services invocation flow above applies for FspSiliconInit()/FspMultiPhaseSiInit() as well.

Figure 8. FspSiliconInit() Variable Services Invocation Sequence


Call FspSiliconInit()
FSP_STATUS_VARIABLE_REQUEST returned ?
YES
Call FspMultiPhaseSiInit() GetVariableRequestInfo
GetVariable()
GetNextVariableName()
SetVariable()
QueryVariableInfo()
NO
Call FspMultiPhaseSiInit() CompleteVariableRequest
Error Returned?
YES
ERROR
Call FspMultiPhaseSiInit() ExecutePhase
NO
Call FspMultiPhaseSiInit() GetNumberOfPhases
NO
PhasesExecuted >= NumberOfPhases ?
YES
Continue to FspNotifyPhase()

9.6.3 Variable Service Descriptions

When FSP_STATUS_VARIABLE_REQUEST is returned, the bootloader shall invoke FspMultiPhaseMem/SiInit() with the *EnumMultiPhaseGetVariableRequestInfo* action given. The *VariableRequest* member of the **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure returned by this API indicates which variable service should be invoked.

9.6.3.1 GetVariable

This service retrieves a variable's value using its name and GUID.

When the *VariableRequest* member of the **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure contains *EnumFspVariableRequestGetVariable*, the FSP is requesting this service.

9.6.3.1.1 Parameters

The members of **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** are usedin the following manner when the GetVariable service is requested:

VariableRequest	Shall be set to EnumFspVariableRequestGetVariable by theFSP.
VariableName	A pointer to an FSP provided buffer containing a null-terminated string that is the variable's name.

VariableNameSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.
VariableGuid	A pointer to an FSP provided buffer containing an **EFI_GUID**that is the variable's GUID. The combination of *VariableGuid*and *VariableName* must be unique.
Attributes	If non-**NULL**, a pointer to an FSP provided buffer that the bootloader shall set this buffer to the variable's attributes before invoking *EnumMultiPhaseCompleteVariableRequest*. If NULL, the bootloader does not return the variable's attributes.
DataSize	On entry, points to an FSP provided buffer that indicates the size in bytes of the FSP provided buffer pointed to by the *Data*member. The bootloader shall set *DataSize* to the size of the data written into the *Data* buffer before invoking *EnumMultiPhaseCompleteVariableRequest*.
Data	Points to an FSP provided buffer which will hold the returned variable value. May be **NULL** with a zero DataSize in order to determine the size of the buffer needed. If non-**NULL** and the buffer size (indicated by *DataSize*) is large enough to hold the variable's value, the bootloader shall copy the variable's valueto this buffer before invoking *EnumMultiPhaseCompleteVariableRequest*.
MaximumVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shall set it to NULL.
RemainingVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.
MaximumVariableSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.

9.6.3.1.2 Description

Reads the specified variable from non-volatile storage. If the *Data* buffer is too small tohold the contents of the variable, the error **EFI_BUFFER_TOO_SMALL** is returned and*DataSize* is set to the required buffer size to obtain the data.

9.6.3.1.3 Return Values

Once the variable is read, the bootloader calls the *FspMultiPhaseMemInit()* API with the

EnumMultiPhaseCompleteVariableRequest action to indicate to the FSP that the variable read is complete. When invoking *EnumMultiPhaseCompleteVariableRequest*,the bootloader shall provide an **FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS** structure. The

VariableRequestStatus member of this structure will be set to one of the following values:

Table 8. Return Values – GetVariable Service

EFI_SUCCESS	The variable was read successfully.
EFI_NOT_FOUND	The variable was not found.

EFI_BUFFER_TOO_SMALL	The DataSize is too small for the resulting data. DataSize is updated with the size required for the specified variable.
EFI_INVALID_PARAMETER	VariableName, VariableGuid, DataSize or Data is NULL.
EFI_DEVICE_ERROR	The variable could not be retrieved because of a device error.

9.6.3.2 GetNextVariableName

This service is called multiple times to retrieve the name and GUID of all variables currently available.

When the *VariableRequest* member of the **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure contains *EnumFspVariableRequestGetNextVariableName*, the FSP is requesting this service.

9.6.3.2.1 Parameters

The members of **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** are usedin the following manner when the GetNextVariableName service is requested:

VariableRequest	Shall be set to EnumFspVariableRequestGetNextVariableName by the FSP.
VariableName	A pointer to an FSP provided buffer containing a null-terminated string that is the current variable's name. If the buffer size (indicated by VariableNameSize) is large enough to hold the next variable’s name, the bootloader shall copy the next variable’s name to this buffer before invoking EnumMultiPhaseCompleteVariableRequest.
VariableNameSize	A pointer to an FSP provided buffer containing the size of the buffer pointed to by VariableName. The bootloader shall copy the size of the buffer needed to contain the next variable’s name to this buffer before invoking EnumMultiPhaseCompleteVariableRequest.
VariableGuid	A pointer to an FSP provided buffer containing an **EFI_GUID**that is the current variable's GUID. The bootloader shall copythe next variable’s GUID to this buffer before invoking *EnumMultiPhaseCompleteVariableRequest*.
Attributes	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
DataSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
Data	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
MaximumVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
RemainingVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.

MaximumVariableSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.

9.6.3.2.2 Description

This service is called multiple times to retrieve the *VariableName* and *VariableGuid* of all variables currently available in the system. On each call, the previous results are passed into the interface, and, on return, the interface returns the data for the next variable. To get started, *VariableName* should initially contain `L"\0"` and the bufferpointed to by *VariableNameSize* should contain `sizeof(CHAR16)`. When the entirevariable list has been returned, **EFI_NOT_FOUND** is returned.

9.6.3.2.3 Return Values

Once the next variable name is read, the bootloader calls the *FspMultiPhaseMemInit()*API with the *EnumMultiPhaseCompleteVariableRequest* action to indicate to the FSP that reading the next variable name is complete. When invoking *EnumMultiPhaseCompleteVariableRequest*, the bootloader shall provide an **FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS** structure. The

VariableRequestStatus member of this structure will be set to one of the following values:

Table 9. Return Values – GetNextVariableName Service

EFI_SUCCESS	The next variable name was read successfully.
EFI_NOT_FOUND	All variables have been enumerated.
EFI_BUFFER_TOO_SMALL	The VariableNameSize is too small for the resulting data. VariableNameSize is updated with the size required for the specified variable.
EFI_INVALID_PARAMETER	VariableName, VariableGuid, or VariableNameSize is NULL.
EFI_DEVICE_ERROR	The variable name could not be retrieved because of a device error.

9.6.3.3 SetVariable

This service stores a new value to the variable with the given name and GUID.

When the *VariableRequest* member of the **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure contains EnumFspVariableRequestSetVariable, the FSP is requesting this service.

9.6.3.3.1 Parameters

The members of **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** are usedin the following manner when the SetVariable service is requested:

VariableRequest	Shall be set to EnumFspVariableRequestSetVariable by the FSP.

VariableName	A pointer to an FSP provided buffer containing a null-terminated string that is the name of the variable. Each VariableName is unique for each VariableGuid. VariableName must contain 1 or more characters. If VariableName is an empty string, then **EFI_INVALID_PARAMETER** is returned.
VariableNameSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.
VariableGuid	A pointer to an FSP provided buffer containing an **EFI_GUID** that is the variable's GUID.
Attributes	A pointer to an FSP provided buffer containing the attributes bitmask for the variable.
DataSize	A pointer to an FSP provided buffer containing the size in bytes of the Data buffer. Unless the **EFI_VARIABLE_APPEND_WRITE** attribute is set, a size of zero causes the variable to be deleted. When the **EFI_VARIABLE_APPEND_WRITE** attribute is set, then a SetVariable() call with a DataSize of zero will not cause any change to the variable value.
Data	A pointer to an FSP provided buffer containing the new data for the variable.
MaximumVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.
RemainingVariableStorageSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.
MaximumVariableSize	Unused; bootloader shall ignore this value and the FSP shall set it to **NULL**.

9.6.3.3.2 Description

This service stores a new value to the variable with the given name and GUID. If a variable with the given name and GUID does not exist and *DataSize* is not zero, then anew variable is created. If *DataSize* is set to zero, the **EFI_VARIABLE_APPEND_WRITE**attribute is **not** set, and an existing variable with the given name and GUID exists, thenthat variable is deleted.

9.6.3.3.3 Return Values

Once the variable is written, the bootloader calls the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* action to indicate to the FSP that the variable write is complete. When invoking *EnumMultiPhaseCompleteVariableRequest*, the bootloader shall provide an **FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS** structure. The *VariableRequestStatus* member of this structure will be set to one of the following values:

Table 10. Return Values – SetVariable Service

EFI_SUCCESS	The bootloader has successfully stored the variable and its data as defined by the Attributes.

EFI_INVALID_PARAMETER	An invalid combination of attribute bits, name, and GUID wassupplied, or the *DataSize* exceeds the maximum allowed.
EFI_INVALID_PARAMETER	*VariableName* is an empty string.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be stored because of a hardware error.
EFI_WRITE_PROTECTED	The variable is read-only.
EFI_WRITE_PROTECTED	The variable cannot be deleted.
EFI_SECURITY_VIOLATION	The variable could not be written due to **EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS, or EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS,** or **EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** being set. The FSP is forbidden from writing to authenticated variables. This feature is only relevant for UEFI Secure Boot and the FSP does not require the bootloader to implement UEFI Secure Boot.
EFI_NOT_FOUND	The variable trying to be updated or deleted was not found.

9.6.3.4 QueryVariableInfo

This service informs the FSP of how much nonvolatile storage space is allocated for thestorage of variables, how much is remaining, and what the maximum allowable size is for each variable.

When the *VariableRequest* member of the **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** structure contains *EnumFspVariableRequestQueryVariableInfo*, the FSP is requesting this service.

9.6.3.4.1 Parameters

The members of **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** are usedin the following manner when the QueryVariableInfo service is requested:

VariableRequest	Shall be set to EnumFspVariableRequestQueryVariableInfo by the FSP.
VariableName	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
VariableNameSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
VariableGuid	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
Attributes	A pointer to an FSP provided buffer containing the Attributes bitmask that specifies the type of variables on which to return information.
DataSize	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.

Data	Unused; bootloader shall ignore this value and the FSP shallset it to **NULL**.
MaximumVariableStorageSize	A pointer to an FSP provided buffer which the bootloader shall set to the maximum size of the storage space available for variables associated with the *Attributes* specified before invoking *EnumMultiPhaseCompleteVariableRequest*.
RemainingVariableStorageSize	A pointer to an FSP provided buffer which the bootloader shall set to the remaining size of the storage space available for variables associated with the *Attributes* specified before invoking *EnumMultiPhaseCompleteVariableRequest*.
MaximumVariableSize	A pointer to an FSP provided buffer which the bootloader shall set to the maximum size of an individual variable associated with the attributes specified before invoking *EnumMultiPhaseCompleteVariableRequest*.

9.6.3.4.2 Description

This service informs the FSP of how much non-volatile storage space is allocated forthe storage of variables, how much is remaining, and what the maximum allowable sizeis for each variable.

The minimum amount of storage space required by the FSP will be mentioned in the Integration Guide.

9.6.3.4.3 Return Values

Once the storage utilization data is ready, the bootloader calls the *FspMultiPhaseMemInit()* API with the *EnumMultiPhaseCompleteVariableRequest* actionto indicate to the FSP that these data are available. When invoking

EnumMultiPhaseCompleteVariableRequest, the bootloader shall provide an **FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS** structure. The *VariableRequestStatus* member of this structure will be set to one of the following values:

Table 11. Return Values – QueryVariableInfo Service

EFI_SUCCESS	The usage of non-volatile storage was determined successfully.
EFI_INVALID_PARAMETER	An invalid combination of *Attribute* bits was supplied
EFI_UNSUPPORTED	The given Attribute bitmask is not supported on this platform, and the MaximumVariableStorageSize, RemainingVariableStorageSize, MaximumVariableSize are undefined.

9.7 TempRamInit API

This FSP API is called after coming out of reset and typically performs the following functions - loads the microcode update, enables code caching for a region specified by the bootloader and sets up a temporary memory area to be used prior to main memorybeing initialized.

The *TempRamInit()* API should be called using the same entry point calling conventiondescribed in the previous section. However, platform limitations, such as the unavailability of a stack, may require steps as mentioned below:

A hardcoded stack must be set up with the following values:

1. The return address where the *TempRamInit()* API returns control.
2. A pointer to the input parameter structure for *TempRamInit()* API when this API isin 32-bit mode. When this API is in 64-bit mode, the pointer to the input parameterstructure will be passed by **RCX** register instead of stack.

The **ESP/RSP** register must be initialized to point to this hardcoded stack.

Since the stack may not be writeable, this API cannot be called using the “call” instruction, but needs to be jumped to directly.

The *TempRamInit()* API preserves the following general purpose registers **EBX/RBX**, **EDI/RDI**, **ESI/RSI**, **EBP/RBP** and the following floating point registers **MM0**, **MM1**. In addition, for 64-bit FSP API mode, the preserved list will be extended to include general purpose registers from **R12** to **R15** and following floating point registers from**XMM6** to **XMM15.**The bootloader can use these registers to save data across the *TempRamInit()* API call. Refer to *Integration Guide* for other register usage.

Calling this API may be optional. Refer to the Integration Guide for any prerequisitesbefore directly calling *FspMemoryInit()* API.

If the bootloader uses this API, then it should be called only once after the system comes out the reset, and it must be called before any other FSP API.

9.7.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_INIT) (
 IN VOID      *FsptUpdDataPtr
);
```

9.7.2 Parameters

FsptUpdDataPtr	Pointer to the FSPT_UPD data structure. If NULL, FSPwill use the defaults from FSP-T component. Refer tothe *Integration Guide* for the structure definition.

9.7.3 Return Values

If this function is successful, the FSP initializes the **ECX/RCX** and **EDX/RDX** registers to point to a temporary but writeable memory range available to the bootloader. Register **ECX/RCX** points to the start of this temporary memory range and **EDX/RDX** points to the end of the range [ECX/RCX, EDX/RDX], where ECX/RCX is inclusive and EDX/RDX isexclusive in the range. The bootloader is free to use the whole range described.

Typically, the bootloader can reload the **ESP/RSP** register to point to the end of thisreturned range so that it can be used as a standard stack.

Table 12. Return Values - TempRamInit() API

EFI_SUCCESS	Temporary RAM was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	Temp RAM initialization failed.

9.7.4 Description

After the bootloader completes its initial steps, it finds the address of the **FSP_INFO_HEADER** and then from the **FSP_INFO_HEADER** finds the offset of the *TempRamInit()* API. It then converts the offset to an absolute address by adding the base of the FSP component and invokes the *TempRamInit()* API.

The temporary memory range returned by this API is intended to be primarily used bythe bootloader as a stack. After this stack is available, the bootloader can switch to using C functions. This temporary stack should be used to do only the minimal initialization that needs to be done before memory can be initialized by the next call into the FSP.

Refer to the *Integration Guide* for details on **FSPT_UPD** parameters.

9.8 FspMemoryInit API

This FSP API initializes the system memory. This FSP API accepts a pointer to a datastructure that will be platform-dependent and defined for each FSP binary.

FspMemoryInit() API initializes the memory subsystem, initializes the pointer to the HobListPtr, and returns to the bootloader from where it was called. Since the systemmemory has been initialized in this API, the bootloader must migrate its stack and datafrom temporary memory to system memory after this API.

9.8.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_MEMORY_INIT) (
 IN VOID        *FspmUpdDataPtr,
 OUT VOID       **HobListPtr
);
```

9.8.2 Parameters

FspmUpdDataPtr	Pointer to the **FSPM_UPD** data structure. If NULL, FSP will use the default from FSP-M component. Refer to the *Integration Guide* for structure definition.
HobListPtr	Pointer to receive the address of the HOB list as defined inthe *Section 13.7 - Appendix A – Data Structures*

9.8.3 Return Values

The *FspMemoryInit()* API will preserve all the general-purpose registers except **EAX/RAX**. The return status will be passed back through the **EAX/RAX** register.

Table 13. Return Values - FspMemoryInit() API

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	FSP memory initialization failed.
EFI_OUT_OF_RESOURCES	Stack range requested by FSP is not met.
FSP_STATUS_RESET_REQUIRED_*	A reset is required. These status codes will not be returned during S3. See Section 13.2.2 for details.
FSP_STATUS_VARIABLE_REQUEST	A FSP variable access is required. See Section 9.6 for details.

9.8.4 Description

When *FspMemoryInit()* API is called, the FSP requires a stack available for its use. Beforecalling the *FspMemoryInit()* API, the bootloader should setup a stack of required size as mentioned in Integration Guide and initialize the **FSPM_ARCH_UPD.StackBase** and **FSPM_ARCH_UPD.StackSize** parameters. FSP consumes this stack region only inside this API.

A set of parameters that the FSP may need to initialize memory under special circumstances, such as during an S3 resume or during fast boot mode, are returned by the FSP to the bootloader during a normal boot. The bootloader is expected to store these parameters in a non-volatile memory such as SPI flash and return a pointer to this structure through **FSPM_ARCH_UPD.NvsBufferPtr** when it is requesting the FSPto initialize the silicon under these special circumstances. Refer to *Section 11.2* **FSP_NON_VOLATILE_STORAGE_HOB2** and *Section 11.3* **FSP_NON_VOLATILE_STORAGE_HOB** for the details on how to get the returned NVS data from FSP.

This API should be called only once before system memory is initialized. This API will produce a HOB list and update the ***HobListPtr*** output parameter. The HOB list will contain a number of Memory Resource Descriptor HOB which the bootloader can use to understand the system memory map. The bootloader should not expect a completeHOB list after the FSP returns from this API. It is recommended for the bootloader to save this **HobListPtr** returned from this API and parse the full HOB list after the *FspSiliconInit()* API.

When this API returns, the bootloader data and stack are still in temporary memory. It is the responsibility of the bootloader to

- Migrate any data from temporary memory to system memory
- Setup a new bootloader stack in system memory

If an initialization step requires a reset to take effect, the *FspMemoryInit()* API will returnone of the **FSP_STATUS_RESET_REQUIRED** statuses as described in *Section 9.4*. ThisAPI will not request a reset during S3 resume flow.

9.9 TempRamExit API

This FSP API is called after *FspMemoryInit()* API. This FSP API tears down the temporary memory set up by *TempRamInit()* API. This FSP API accepts a pointer to a data structurethat will be platform dependent and defined for each FSP binary.

TempRamExit() API provides bootloader an opportunity to get control after system memory is available and before the temporary memory is torn down.

This API is an optional API, refer to Integration Guide for prerequisites before directlycalling *FspSiliconInit()* API.

9.9.1 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_EXIT) (
 IN VOID        *TempRamExitParamPtr
);
```

9.9.2 Parameters

TempRamExitParamPtr	Pointer to the TempRamExit parameters structure. This structure is normally defined in the *Integration Guide*. If it isnot defined in the *Integration Guide*, pass **NULL**.

9.9.3 Return Values

The *TempRamExit()* API will preserve all the general-purpose registers except **EAX/RAX**. The return status will be passed back through the **EAX/RAX** register.

Table 14. Return Values - TempRamExit() API

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.

EFI_DEVICE_ERROR	Temporary memory exit.

9.9.4 Description

This API should be called only once after the *FspMemoryInit()* API and before

FspSiliconInit() API.

This API tears down the temporary memory area set up in the cache and returns the cache to normal mode of operation. After the cache is returned to normal mode of operation, any data that was in the temporary memory is destroyed. It is therefore expected that the bootloader migrates any bootloader specific data that it might havehad in the temporary memory area and also set up a stack in the system memory before calling *TempRamExit()* API.

After the *TempRamExit()* API returns, the bootloader is expected to set up the BSP MTRRs to enable caching. The bootloader can collect the system memory map information by parsing the HOB data structures and use this to set up the MTRR andenable caching.

9.10 FspSiliconInit API

This FSP API initializes the processor and the chipset including the IO controllers in thechipset to enable normal operation of these devices.

This API should be called only once after the system memory has been initialized, datafrom temporary memory migrated to system memory and cache configuration has been initialized.

9.10.1 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *FSP_SILICON_INIT) (
 IN VOID        *FspsUpdDataPtr
);
```

9.10.2 Parameters

FspsUpdDataPtr	Pointer to the **FSPS_UPD** data structure. If **NULL**, FSP will use the default parameters. Refer to the *Integration Guide* for structure definition.

9.10.3 Return Values

The FspSiliconInit API will preserve all the general-purpose registers except **EAX/RAX**.The return status will be passed back through the **EAX/RAX** register.

Table 15. Return Values – FspSiliconInit() API

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	FSP silicon initialization failed.
FSP_STATUS_RESET_REQUIRED_*	A reset is required. These status codes will not be returned during S3.
FSP_STATUS_VARIABLE_REQUEST	A FSP variable access is required. See Section 9.6 for details.

9.10.4 Description

This API should be called only once after the *FspMemoryInit()* API (if the bootloader isnot using *TempRamExit()* API) or the *TempRamExit()* API.

This FSP API accepts a pointer to a data structure that will be platform dependent anddefined for each FSP binary. This will be documented in the *Integration Guide*.

This API adds HOBs to the HobListPtr to pass more information to the bootloader. To obtain the additional information, the bootloader must parse the HOB list again after the FSP returns from this API.

If an initialization step requires a reset to take effect, the *FspSiliconInit()* API will returnan **FSP_STATUS_RESET_REQUIRED** as described in *Section 9.4*. This API will not request a reset during S3 resume flow.

9.11 FspMultiPhaseMem/SiInit API

This FSP API provides multi-phase memory and silicon initialization, which brings greater modularity to the existing *FspMemoryInit()* and *FspSiliconInit()* API. Increased modularity is achieved by adding an extra API to FSP-M and FSP-S. This allows the bootloader to add board specific initialization steps throughout the MemoryInit and SiliconInit flows as needed. The *FspMemoryInit()* API is always called before *FspMultiPhaseMemInit()*; it is the first phase of memory initialization. Similarly, the *FspSiliconInit()* API is always called before *FspMultiPhaseSiInit()*; it is the first phase of silicon initialization. After the first phase, subsequent phases are invoked by calling the*FspMultiPhaseMem/SiInit()* API.

The *FspMultiPhaseMemInit()* API may only be called after the *FspMemoryInit()* API and before the *FspSiliconInit()* API; or in the case that FSP-T is being used, before the *TempRamExit()* API. The *FspMultiPhaseSiInit()* API may only be called after the *FspSiliconInit()* API and before *NotifyPhase()* API; or in the case that FSP-I is being used, before the *FspSmmInit()* API. The multi-phase APIs may not be called at any other time.

9.11.1 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *FSP_MULTI_PHASE_INIT) (
 IN FSP_MULTI_PHASE_PARAMS   *MultiPhaseInitParamPtr
);
```

9.11.2 Parameters

MultiPhaseInitParamPtr	Pointer to the **FSP_MULTI_PHASE_PARAMS** data structure.

9.11.3 Related Definitions

```
typedef enum {
 EnumMultiPhaseGetNumberOfPhases            = 0x0,
 EnumMultiPhaseExecutePhase                 = 0x1,
 EnumMultiPhaseGetVariableRequestInfo       = 0x2,
 EnumMultiPhaseCompleteVariableRequest      = 0x3
} FSP_MULTI_PHASE_ACTION;

typedef struct {
 IN    FSP_MULTI_PHASE_ACTION MultiPhaseAction;
 IN    UINT32                  PhaseIndex;
 IN OUT VOID                  *MultiPhaseParamPtr;
} FSP_MULTI_PHASE_PARAMS;
typedef struct {
 UINT32       NumberOfPhases;
 UINT32       PhasesExecuted;
} FSP_MULTI_PHASE_GET_NUMBER_OF_PHASES_PARAMS;

typedef enum {
 EnumFspVariableRequestGetVariable          = 0x0,
 EnumFspVariableRequestGetNextVariableName  = 0x1,
 EnumFspVariableRequestSetVariable          = 0x2,
 EnumFspVariableRequestQueryVariableInfo    = 0x3
} FSP_VARIABLE_REQUEST_TYPE;
typedef struct {
 IN FSP_VARIABLE_REQUEST_TYPE VariableRequest;
 IN OUT CHAR16              *VariableName;
 IN OUT UINT64              *VariableNameSize;
 IN OUT EFI_GUID            *VariableGuid;
 IN OUT UINT32              *Attributes;
 IN OUT UINT64              *DataSize;
 IN OUT VOID                *Data;
 OUT   UINT64                  *MaximumVariableStorageSize;
 OUT   UINT64                  *RemainingVariableStorageSize;
 OUT   UINT64                  *MaximumVariableSize;
} FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS;
typedef struct {
 EFI_STATUS                    VariableRequestStatus;
} FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS;
```

EnumMultiPhaseGetNumberOfPhases

This action returns the number of MemoryInit or SiliconInit phases that the FSP supports. This indicates the maximum number of times the *FspMultiPhaseMem/SiInit()*API may be called by the bootloader with the *EnumMultiPhaseExecutePhase* action given.

When this action is called, the bootloader must set *PhaseIndex* to zero and provide aninstance of **FSP_MULTI_PHASE_GET_NUMBER_OF_PHASES_PARAMS** to the *MultiPhaseParamPtr*. The *NumberOfPhases* value inside this instance will be used to return the number of phases to the bootloader. The *PhasesExecuted* value inside this instance informs the bootloader of how many of those phases have already been executed thus far. If the bootloader has not yet executed any phases, then the *PhasesExecuted* integer will be set to `0x0`.

The *EnumMultiPhaseGetNumberOfPhases* action can be invoked by the bootloader asmany times as desired. It only retrieves the current status; it does not modify it.

EnumMultiPhaseExecutePhase

This action executes the memory or silicon initialization phase provided by the *PhaseIndex* parameter. The *MultiPhaseParamPtr* shall be `NULL`. Note that *PhaseIndex* isa one-based index, not a zero-based index. On the first call, *PhaseIndex* shall be `0x1`; setting *PhaseIndex* to `0x0` will result in `EFI_INVALID_PARAMETER` being returned.

EnumMultiPhaseGetVariableRequestInfo

This action provides information to the bootloader about a pending non-volatile I/O request being made by the FSP. When `FSP_STATUS_VARIABLE_REQUEST` is returned,the bootloader shall invoke *FspMultiPhaseMem/SiInit()* with the *EnumMultiPhaseGetVariableRequestInfo* action given.

When this action is called, the bootloader must set *PhaseIndex* to zero and provide aninstance of **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** to the *MultiPhaseParamPtr*. The FSP will copy data detailing its pending non-volatile I/O request into this bootloader provided buffer. The bootloader will then use this data to

service the FSP’s access request. Please see *Section 9.6* for a detailed description of thiscalling sequence.

EnumMultiPhaseCompleteVariableRequest

This action informs the FSP that the variable access request is complete.

When this action is called, the bootloader must set *PhaseIndex* to zero and provide aninstance of **FSP_MULTI_PHASE_COMPLETE_VARIABLE_REQUEST_PARAMS** to the *MultiPhaseParamPtr*. The *VariableRequestStatus* value inside this instance shall be setto indicate to the FSP whether the variable access request was successful or not according to the return values provided in *Section 9.6.3*.

In the case where the bootloader must return data to the FSP, the bootloader must write any relevant data into the buffer(s) provided by the FSP via **FSP_MULTI_PHASE_VARIABLE_REQUEST_INFO_PARAMS** before invoking this action. This action will allow the FSP will continue execution where it left off. Please see *Section 9.6* for a detailed description of this calling sequence.

9.11.4 Return Values

The FspMultiPhaseMem/SiInit API will preserve all the general-purpose registers except **EAX/RAX**. The return status will be passed back through the **EAX/RAX** register.

Table 16. Return Values – FspMultiPhaseSiInit() API

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	FSP silicon initialization failed.
FSP_STATUS_RESET_REQUIRED_*	A reset is required. These status codes will not be returned during S3. This status code can only be givenwhen either the *EnumMultiPhaseCompleteVariableRequest* or *EnumMultiPhaseExecutePhase* actions are given.
FSP_STATUS_VARIABLE_REQUEST	A FSP variable access is required. See *Section 9.6* fordetails. This status code can only be given when eitherthe *EnumMultiPhaseCompleteVariableRequest* or *EnumMultiPhaseExecutePhase* actions are given.

9.11.5 Description

This API may only be called after the *FspSiliconInit()* API and before *NotifyPhase()* API,and may not be called at any other time.

An FSP binary may optionally implement multi-phase silicon initialization. When usingmulti-phase silicon initialization, the *FspSiliconInit()* API is always called first; it is the first phase of silicon initialization. After the first phase, subsequent phases are invoked by calling the *FspMultiPhaseSiInit()* API. When single-phase silicon initialization is used,only the *FspSiliconInit()* API is called.

If the *FspMultiPhaseSiInitEntryOffset* field in **FSP_INFO_HEADER** is non-zero, the FSP includes support for multi-phase SiliconInit, see *Section 5.1.1* for further details. To enable multi-phase, the bootloader must set `FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit` to a non-zero value.

If **FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit** is set to a non-zero value, then the bootloader must invoke the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseExecutePhase* parameter *n* times, where *n* == *NumberOfPhases* returned by *EnumMultiPhaseGetNumberOfPhases*. The bootloader must invoke the *FspMultiPhaseSiInit()* API with the *EnumMultiPhaseExecutePhase* parameter in the correct sequence; *PhaseIndex* must be set to **1** on the first call, **2** on the second call, and so on. The bootloader must complete the multi-phase sequence by invoking the*FspMultiPhaseSiInit()* API with *PhaseIndex* == *NumberOfPhases* before invoking the *NotifyPhase()* API with the *AfterPciEnumeration* parameter.

If **FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit** is set to a zero or if the *FspMultiPhaseSiInitEntryOffset* field in **FSP_INFO_HEADER** is zero, then the bootloader must not invoke the *FspMultiPhaseSiInit()* API at all.

The breakdown of which silicon initialization steps are implemented in which phase may vary for different processor and the chipset designs and will be detailed in the *Integration Guide*.

This API may add HOBs to the HobListPtr to pass more information to the bootloader.To obtain the additional information, the bootloader must parse the HOB list again after the FSP returns from this API.

If an initialization step requires a reset to take effect, the *FspMultiPhaseSiInit()* API will return an **FSP_STATUS_RESET_REQUIRED** as described in *Section 9.4*. This API will not request a reset during S3 resume flow.

9.12 FspSmmInit API

This FSP API initializes SMM and provides any OS runtime silicon services; including Reliability, Availability, and Serviceability (RAS) features implemented by the CPU.

9.12.1 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *FSP_SMM_INIT) (
 IN VOID       *FspiUpdDataPtr
);
```

9.12.2 Parameters

FspiUpdDataPtr	Pointer to instance of FSPI_UPD structure.

9.12.3 Return Values

The *FspSmmInit()* API will preserve all the general-purpose registers except **RAX**. Thereturn status will be passed back through the **RAX** register.

Table 17. Return Values - FspSmmInit() API

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The API calling conditions were not met.

9.12.4 Description

This API should only be called once after the *FspSiliconInit()* API. It may only be calledon the boot strap processor (BSP).

This FSP API accepts a pointer to a data structure that will be platform dependent anddefined for each FSP binary. This will be documented in the *Integration Guide*.

9.13 NotifyPhase API

This FSP API is used to notify the FSP about the different phases in the boot process. This allows the FSP to take appropriate actions as needed during different initializationphases. The phases will be platform dependent and will be documented with the FSP release. The current FSP specification supports three notify phases:

- Post PCI enumeration
- Ready to Boot
- End of Firmware

9.13.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_NOTIFY_PHASE) (
 IN NOTIFY_PHASE_PARAMS     *NotifyPhaseParamPtr
);
```

9.13.2 Parameters

NotifyPhaseParamPtr	Address pointer to the **NOTIFY_PHASE_PARAMS**

9.13.3 Related Definitions

```
typedef enum {
 EnumInitPhaseAfterPciEnumeration = 0x20,
 EnumInitPhaseReadyToBoot        = 0x40,
 EnumInitPhaseEndOfFirmware       = 0xF0
} FSP_INIT_PHASE;
typedef struct {
 FSP_INIT_PHASE Phase;
} NOTIFY_PHASE_PARAMS;
```

EnumInitPhaseAfterPciEnumeration

This stage is notified when the bootloader completes the PCI enumeration and the resource allocation for the PCI devices is complete.

EnumInitPhaseReadyToBoot

This stage is notified just before the bootloader hand-off to the OS loader.

EnumInitPhaseEndOfFirmware

This stage is notified just before the firmware/Preboot environment transfers management of all system resources to the OS or next level execution environment.

When booting to non-UEFI OS, this stage is notified immediately after the *EnumInitPhaseReadyToBoot*. When booting to UEFI OS this stage is notified at *ExitBootServices* callback from OS.

9.13.4 Return Values

The *NotifyPhase()* API will preserve all the general purpose registers except **EAX/RAX**. The return status will be passed back through the **EAX/RAX** register.

Table 18. Return Values – NotifyPhase() API

EFI_SUCCESS	The notification was handled successfully.
EFI_UNSUPPORTED	The notification was not called in the proper order.
EFI_INVALID_PARAMETER	The notification code is invalid.
FSP_STATUS_RESET_REQUIRED_*	A reset is required. These status codes will not be returned during S3.

9.13.5 Description

EnumInitPhaseAfterPciEnumeration

FSP will use this notification to do some specific initialization for processor and chipsetthat requires PCI resource assignments to have been completed.

This API must be called before executing 3rd party code, including PCI Option ROM, forsecure design reasons.

On S3 resume path this API must be called before the bootloader hand-off to the OSresume vector.

EnumInitPhaseReadyToBoot

FSP will perform required configuration by the BWG / BIOS Specification when it is notified that the bootloader is ready to transfer control to the OS loader.

On S3 resume path this API must be called after *EnumInitPhaseAfterPciEnumeration*

notification and before the bootloader hand-off to the OS resume vector.

EnumInitPhaseEndOfFirmware

FSP can use this notification to perform some handoff of the system resources beforetransferring control to the OS.

When booting to non-UEFI OS this stage is notified immediately after the *EnumInitPhaseReadyToBoot*. When booting to UEFI OS this stage is notified at *ExitBootServices* callback from OS.

On the S3 resume path this API must be called after *EnumInitPhaseReadyToBoot*

notification and before the bootloader hand-off to the OS resume vector.

After this phase, the whole FSP flow is considered to be complete and the results of any further FSP API calls are undefined.

If an initialization step requires a reset to take effect, the *NotifyPhase()* API will return an`FSP_STATUS_RESET_REQUIRED` as described in *Section 9.4*. This API will not request areset during S3 resume flow.

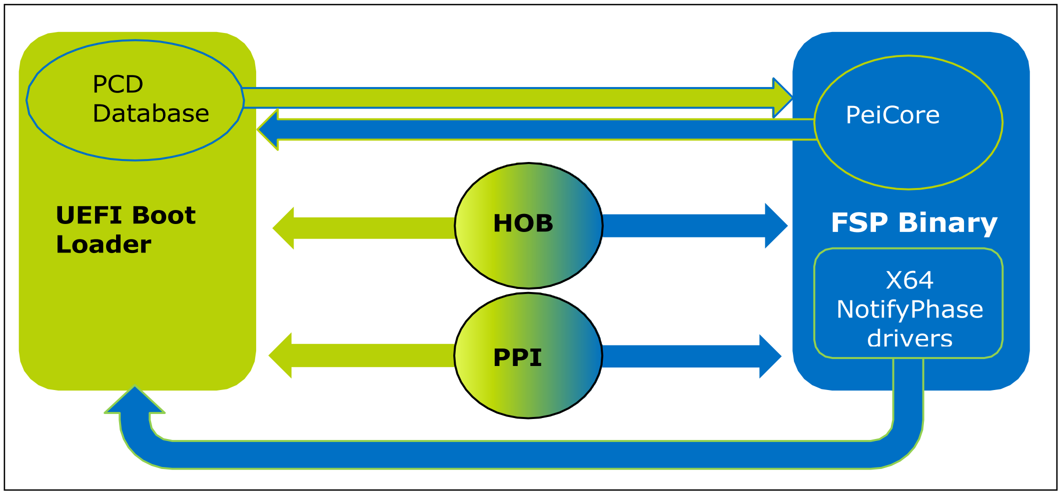
§§

10.0 FSP Dispatch Mode Interface

Dispatch mode is an optional boot flow intended to enable FSP to integrate well in toUEFI bootloader implementations. The **FSP_INFO_HEADER** indicates if an FSP implements dispatch mode, Refer *Section* 5.1.1 for further details.

10.1 Dispatch Mode Design

Figure 9. Dispatch Mode Design



Dispatch mode is intended to enable a boot flow that is as close to a standard UEFI boot flow as possible. FSP dispatch mode fully conforms to the *PI Specification* and assumes the boot loader will follow the standard four phase PI boot flow progressing from SEC phase to PEI phase, to DXE phase, and to BDS phase. It is recommended thatthe reader have knowledge of the contents of the *PI Specification* before continuing.

In dispatch mode, FSP-T, FSP-M, FSP-S, and FSP-I (in FSP SMM model 3) are containers that expose firmware volumes (FVs) directly to the bootloader. The PEIMs in these FVs are executed directly in the context of the PEI environment provided by the bootloader.FSP-T, FSP-M, FSP-S, and FSP-I could contain one or multiple FVs. The exact number ofFVs contained in FSP-T, FSP-M, FSP-S, and FSP-I will be described in the *Integration Guide*. In dispatch mode, the PPI database, PCD database, and HOB list are shared between the bootloader and the FSP.

UPDs are not needed to provide a mechanism to pass configuration data from the bootloader to the FSP. Instead, configuration data is communicated to the FSP using PCDs and PPIs. These mechanisms are native to bootloader implementations conforming to the *PI Specification* and constitute a more natural method of supplyingconfiguration data to the FSP. These PCDs and PPIs are platform specific. The *FSP Distribution Package* will contain source code definitions of the configuration data structures consumed by the FSP. The configuration data structures will also bedescribed by the *Integration Guide*.

The bootloader must provide the PCD database implementation. Any dynamic PCDs consumed by the FSP must be included in the PCD database provided by the bootloader. The *FSP Distribution Package* will contain a DSC file which defines all PCDsused by the FSP. The recommended method of including these PCDs is to use the `!include` directive in the bootloader's top-level platform DSC file. Because the FSP is a pre-compiled binary, all dynamic PCDs consumed by the FSP must be of the DynamicEx type. Refer to *MdeModulePkg/Universal/PCD/Pei/Pcd.inf* for more details on platform token numbers. In addition to the DSC file included in the *FSP Distribution Package*, the *Integration Guide* will also list the PCDs (along with TokenSpace GUID and TokenNumber) consumed by the FSP.

In dispatch mode, the *NotifyPhase*() API is not used. Instead, FSP-S contains DXE drivers that implement the native callbacks on equivalent events for each of the *NotifyPhase()* invocations. The inclusion of DXE drivers allows dispatch mode to provide capabilities that would not be possible in API mode.

10.2 PEI Phase Requirements

PEIMs contained in FSP firmware volumes are intended to be executed within the processor context and calling conventions defined by the *PI Specification, Volume 1* foreither the IA-32 or x64 platforms. The exact target platform will be specified in the *Integration Guide*.

PEIMs contained in the FSP shall use a subset of the API provided by the PEI Foundation. Specifically, PEIMs contained in FSP firmware volumes should not use thefollowing architecturally defined PPIs:

- **EFI_PEI_READ_ONLY_VARIABLE2_PPI**

If BIT3 (Variable Support) of the *ImageAttribute* field in the **FSP_INFO_HEADER** is set, the FSP shall use the **EDKII_PEI_VARIABLE_PPI** to access NV storage. As **EFI_PEI_READ_ONLY_VARIABLE2_PPI** only supports reads, it is considered legacy, andshould not be used. If BIT3 is not set, variable access from PEIMs contained in FSP firmware volumes is forbidden.

10.3 DXE and BDS Phase Requirements

DXE drivers contained in FSP firmware volumes are intended to be executed within the processor context and calling conventions defined by the *PI Specification, Volume 2* forx64 platforms.

DXE drivers contained in the FSP shall use a subset of the API provided by the DXE Foundation. Specifically, DXE drivers contained in FSP firmware volumes shall **not** usethe following UEFI services:

- ExitBootServices()
- SetWatchdogTimer()
- SetTime()
- SetWakeupTime()
- UpdateCapsule()
- QueryCapsuleCapabilities()

If BIT3 (Variable Support) of the *ImageAttribute* field in the **FSP_INFO_HEADER** is **not**set, then DXE drivers contained in FSP firmware volumes shall **not** use the following UEFI services:

- GetVariable()
- GetNextVariableName()
- SetVariable()
- QueryVariableInfo()

The FSP may use the following *PI Specification* defined events during DXE phase:

1. **EFI_END_OF_DXE_EVENT_GROUP_GUID** – The *PI Specification* requires the bootloader to signal this event prior to invoking any UEFI drivers or applications that are not from the platform manufacturer or connecting consoles.
2. **EFI_PCI_ENUMERATION_PROTOCOL** – The *PI Specification* requires the bootloader to install this protocol after PCI enumeration is complete.
3. **EFI_EVENT_GROUP_READY_TO_BOOT** – The *PI Specification* requires the bootloader to signal this event when it is about to load and execute a boot option.

Create an event to be notified when *ExitBootServices()* is invoked using **EVT_SIGNAL_EXIT_BOOT_SERVICES.**

DXE drivers may use other events for platform specific use cases. Any additional eventsbeyond those described above will be documented in the *Integration Guide*.

10.4 Dispatch Mode API

FSP dispatch mode fully conforms to the *PI Specification*. Accordingly, dispatch modedoes not require many FSP specific API definitions since the *PI Specification* already defines most API. This section therefore only describes FSP specific extensions to the *PI Specification*. Most FSP API will be platform specific and therefore documented in the *Integration Guide*.

10.4.1 TempRamInit API

The *PI Specification* defines a code module format for PEI and DXE (PEIMs and DXE drivers, respectively). However, the *PI Specification* does not define a module format forSEC phase. Temporary RAM must be initialized during the SEC phase. Therefore, in dispatch mode FSP-T uses the same API defined in *Section 9.7* to provide *TempRamInit()* to the bootloader SEC implementation.

10.4.2 EFI PEI Core Firmware Volume Location PPI

If the boot flow described in *Section 7.2.2* is followed, the PEI Foundation does not reside in the Boot Firmware Volume (BFV). In compliance with the *PI Specification v1.7*, SEC must pass the **EFI_PEI_CORE_FV_LOCATION_PPI** as a part of the PPI list provided to the PEI Foundation Entry Point. Refer Section 6.3.9 of the *PI Specification v1.7* Volume 1 for more details on this PPI. If the alternate boot flow described in Section 7.2.3 is followed, then the PEI Foundation resides in the BFV. Accordingly, this PPI should not be produced.

10.4.3 FSP Temporary RAM Exit PPI

FSP_TEMP_RAM_EXIT_PPI

10.4.3.1 Summary

Tears down the temporary memory set up by *TempRamInit()* API.

10.4.3.2 GUID

```
#define FSP_TEMP_RAM_EXIT_GUID \
 {0xbc1cfbdb, 0x7e50, 0x42be, \
 {0xb4, 0x87, 0x22, 0xe0, 0xa9, 0x0c, 0xb0, 0x52}}
```

10.4.3.3 Prototype

```
typedef struct {
 FSP_TEMP_RAM_EXIT  TempRamExit;
} FSP_TEMP_RAM_EXIT_PPI;
```

10.4.3.4 Parameters

TempRamExit	Tears down the temporary memory set up by TempRamInit() API.

10.4.3.5 Description

This PPI provides the equivalent functionality as the *TempRamExit()* function defined in *Section 9.9* to bootloaders that use the FSP in dispatch mode. The *TempRamExit()* function defined in this PPI tears down the temporary memory set up by *TempRamInit()*API. Bootloaders that use dispatch mode must not use the *TempRamExit()* API defined in *Section 9.9*, they must use this PPI instead.

10.4.4 FSP_TEMP_RAM_EXIT_PPI.TempRamExit ()

10.4.4.1 Summary

Tears down the temporary memory set up by *TempRamInit()* API.

10.4.4.2 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_EXIT) (
 IN VOID       *TempRamExitParamPtr
);
```

10.4.4.3 Parameters

TempRamExitParamPtr	Pointer to the TempRamExit parameters structure. This structure is normally defined in the *Integration Guide*. Ifit is not defined in the *Integration Guide*, pass **NULL**.

10.4.4.4 Description

This API is intended to be used by the bootloader's implementation of **EFI_PEI_TEMPORARY_RAM_DONE_PPI**. This API tears down the temporary memory set up by the *TempRamInit()* API. This API accepts a pointer to a data structure that willbe platform dependent and defined for each FSP binary.

The *FSP_TEMP_RAM_EXIT_PPI->TempRamExit()* API provides the bootloader an opportunity to get control after system memory is available and before the temporarymemory is torn down. Therefore, is the boot loader's responsibility to call *FSP_TEMP_RAM_EXIT_PPI->TempRamExit()* when ready.

This API is an optional API, refer to the *Integration Guide* for prerequisites before installing the **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** instances to begin dispatch of PEIMs in FSP-S firmware volume(s).

Implementation Note: The UefiCpuPkg in EDK2 provides a reference implementationof SEC phase. If the boot loader elects to use this, at time of writing the UefiCpuPkg implementation of SEC core produces the **EFI_PEI_TEMPORARY_RAM_DONE_PPI**. The*TemporaryRamDone()* implementation in SEC core will call *SecPlatformDisableTemporaryMemory()*, this function is implemented by the boot loader. The boot loader implementation of this function would then locate **FSP_TEMP_RAM_EXIT_PPI** and call *TempRamExit()* when ready.

10.4.4.5 Return Values

Table 19. Return Values - TempRamExit() PPI

EFI_SUCCESS	FSP execution environment was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	Temporary memory exit.

10.4.5 FSP-M Architectural Configuration PPI

FSPM_ARCH_CONFIG_PPI

10.4.5.1 Summary

Architectural configuration data for FSP-M.

10.4.5.2 GUID

```
#define FSPM_ARCH_CONFIG_GUID \
 {0x824d5a3a, 0xaf92, 0x4c0c, \
 {0x9f, 0x19, 0x19, 0x52, 0x6d, 0xca, 0x4a, 0xbb}}
```

10.4.5.3 Prototype

```
typedef struct {
 UINT8                  Revision;
 UINT8                  Reserved[3]
 VOID                   *NvsBufferPtr;
 UINT32                 BootLoaderTolumSize;
 UINT8                  Reserved1[4];
} FSPM_ARCH_CONFIG_PPI;
```

10.4.5.4 Parameters

Revision	Revision of the structure is 1 for this version of the specification.
NvsBufferPtr	This value is deprecated starting with v2.4 of this specification and will be removed in an upcoming version of this specification. If BIT3 (Variable Support) in the *ImageAttribute* field of the **FSP_INFO_HEADER** is set, then this value is unused and must be set to **NULL**. In this case, the FSP shall use the FSP variable services described in*Section 10.4.6* instead. Pointer to the non-volatile storage (NVS) data buffer. If it is**NULL** it indicates the NVS data is not available. Refer to *Section 11.2* and *11.3* for more details.
BootloaderTolumSize	Size of memory to be reserved by FSP below "top of low usable memory" for bootloader usage. Refer to *Section 11.4*for more details.

10.4.5.5 Description

This PPI contains architectural configuration data that is needed by PEIMs in FSP-M and/or FSP-S. It is the responsibility of the bootloader to install this PPI. The bootloader must be able to provide these data within the pre-memory PEI timeframe. In adherence with the weak ordering requirement for PEIMs, any PEIM contained in FSPthat uses this PPI shall either include this PPI in its DEPEX or shall register a callback using *(*PeiServices)->NotifyPpi ()* and refrain from accessing these data until the callback is invoked by the PEI Foundation.

As a performance optimization, it is recommended (but not required) that the boot loader install this PPI before installing **EFI_PEI_FIRMWARE_VOLUME_INFO_PPI** instances for the firmware volume(s) contained in FSP-M. This will reduce the numberof times the PEI Dispatcher will need to loop in order to complete PEI phase.

10.4.6 EDK II PEI Variable PPI

EDKII_PEI_VARIABLE_PPI

10.4.6.1 Summary

The EDKII PEI Variable PPI provides access to the FSP Variable Services.

10.4.6.2 GUID

```
#define EDKII_PEI_VARIABLE_PPI_GUID \
 {0xe7b2cd04, 0x4b14, 0x44c2, \
 {0xb7, 0x48, 0xce, 0xaf, 0x2b, 0x66, 0x4a, 0xb0}}
```

10.4.6.3 Prototype

```
typedef struct {
 EDKII_PEI_GET_VARIABLE           GetVariable;
 EDKII_PEI_GET_NEXT_VARIABLE_NAME GetNextVariableName;
 EDKII_PEI_SET_VARIABLE           SetVariable;
 EDKII_PEI_QUERY_VARIABLE_INFO     QueryVariableInfo;
} EDKII_PEI_VARIABLE_PPI;
```

10.4.6.4 Parameters

GetVariable	Retrieves a variable's value using its name and GUID.
GetNextVariableName	This service is called multiple times to retrieve the name and GUID of all variables currently available.
SetVariable	Stores a new value to the variable with the given name andGUID.
QueryVariableInfo	This service informs the FSP of how much nonvolatile storage space is allocated for the storage of variables, how much is remaining, and what the maximum allowable size isfor each variable.

10.4.6.5 Description

The EDKII PEI Variable PPI provides access to the FSP variable services described in *Section 9.6*. The bootloader is required to publish this PPI in dispatch mode. In dispatchmode, the FSP calls this PPI directly instead of using the Multi-Phase invocation sequence described in *Section 9.6.1*. Generally this PPI provides access to the UEFI variable services, but other implementations are possible.

10.4.7 EDKII_PEI_VARIABLE_PPI.GetVariable ()

10.4.7.1 Summary

This service retrieves a variable's value using its name and GUID.

10.4.7.2 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *EDKII_PEI_GET_VARIABLE) (
 IN CONST  EDKII_PEI_VARIABLE_PPI   *This,
 IN CONST  CHAR16                  *VariableName,
 IN CONST  EFI_GUID                *VariableGuid,
 OUT       UINT32                  *Attributes, OPTIONAL
 IN OUT    UINTN                   *DataSize,
 OUT   VOID                        *Data        OPTIONAL
);
```

10.4.7.3 Parameters

This	A pointer to this instance of the **EDKII_PEI_VARIABLE_PPI**.
VariableName	A pointer to a null-terminated string that is the variable's name.
VariableGuid	A pointer to an **EFI_GUID** that is the variable's GUID. The combination of *VariableGuid* and *VariableName* must be unique.
Attributes	If non-NULL, on return, contains the variable's attributes.
DataSize	On entry, points to the size in bytes of the Data buffer. On return, points to the size of the data returned in Data.
Data	Points to the buffer which will hold the returned variable value.May be NULL with a zero DataSize in order to determine the size of the buffer needed.

10.4.7.4 Description

Reads the specified variable from non-volatile storage. If the *Data* buffer is too small tohold the contents of the variable, the error **EFI_BUFFER_TOO_SMALL** is returned and*DataSize* is set to the required buffer size to obtain the data.

10.4.7.5 Return Values

Table 20. Return Values - GetVariable()

EFI_SUCCESS	The variable was read successfully.
EFI_NOT_FOUND	The variable was not found.
EFI_BUFFER_TOO_SMALL	The DataSize is too small for the resulting data. DataSize is updated with the size required for the specified variable.
EFI_INVALID_PARAMETER	VariableName, VariableGuid, DataSize or Data is NULL.
EFI_DEVICE_ERROR	The variable could not be retrieved because of a device error.

10.4.8 EDKII_PEI_VARIABLE_PPI.GetNextVariableName ()

10.4.8.1 Summary

This service is called multiple times to retrieve the name and GUID of all variables currently available.

10.4.8.2 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *EDKII_PEI_GET_NEXT_VARIABLE_NAME) (
  IN   CONST     EDKII_PEI_VARIABLE_PPI    *This,
  IN   OUT       UINTN                     *VariableNameSize,
  IN   OUT       CHAR16                    *VariableName,
  IN   OUT       EFI_GUID                  *VariableGuid

);
```

10.4.8.3 Parameters

This	A pointer to this instance of the **EDKII_PEI_VARIABLE_PPI**.
VariableNameSize	On entry, points to the size of the buffer pointed to by *VariableName*. On return, the size of the buffer needed to contain the next variable's name.
VariableName	A pointer to a buffer containing a null-terminated string that isthe variable's name. On entry, the buffer contains the current variable name. On return, the buffer contains the next variable's name. If the buffer size (indicated by *VariableNameSize*) is large enough to hold the next variable's name, the bootloader shallcopy the next variable's name to this buffer.
VariableGuid	A pointer to a buffer containing an **EFI_GUID** that is the currentvariable's GUID. On return, the bootloader shall copy the next variable's GUID to this buffer.

10.4.8.4 Description

Return the next variable name and GUID.

This function is called multiple times to retrieve the *VariableName* and *VariableGuid* ofall variables currently available in the system. On each call, the previous results are passed into the interface, and, on return, the interface returns the data for the next variable. To get started, *VariableName* should initially contain `L"\0"` and the buffer pointed to by *VariableNameSize* should contain **`sizeof(CHAR16)`**. When the entire variable list has been returned, **EFI_NOT_FOUND** is returned.

10.4.8.5 Return Values

Table 21. Return Values - GetNextVariableName()

EFI_SUCCESS	The next variable name was read successfully.
EFI_NOT_FOUND	All variables have been enumerated.
EFI_BUFFER_TOO_SMALL	The *VariableNameSize* is too small for the resulting data. *VariableNameSize* is updated with the size required for the specified variable.
EFI_INVALID_PARAMETER	VariableName, VariableGuid, or VariableNameSize is **NULL**.
EFI_DEVICE_ERROR	The variable name could not be retrieved because of a deviceerror.

10.4.9 EDKII_PEI_VARIABLE_PPI.SetVariable ()

10.4.9.1 Summary

Stores a new value to the variable with the given name and GUID.

10.4.9.2 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *EDKII_PEI_SET_VARIABLE) (
 IN    CONST    EDKII_PEI_VARIABLE_PPI    *This,
 IN             CHAR16                    *VariableName,
 IN             EFI_GUID                  *VariableGuid,
 IN             UINT32                    Attributes,
 IN             UINTN                     DataSize,
 IN             VOID                      *Data
);
```

10.4.9.3 Parameters

This	A pointer to this instance of the **EDKII_PEI_VARIABLE_PPI**.
VariableName	A Null-terminated string that is the name of the variable. Each*VariableName* is unique for each *VariableGuid*. *VariableName* must contain 1 or more characters. If *VariableName* is an empty string, then **EFI_INVALID_PARAMETER** is returned.
VariableGuid	A pointer to an **EFI_GUID** that is the variable's GUID.
Attributes	Attributes bitmask for the variable.

DataSize	The size in bytes of the Data buffer. Unless the **EFI_VARIABLE_APPEND_WRITE** attribute is set, a size of zero causes the variable to be deleted. When the **EFI_VARIABLE_APPEND_WRITE** attribute is set, then a SetVariable() call with a DataSize of zero will not cause any change to the variable value.
Data	The contents for the variable.

10.4.9.4 Description

Stores a new value to the variable with the given name and GUID. If a variable with thegiven name and GUID does not exist and *DataSize* is not zero, then a new variable is created. If *DataSize* is set to zero, the **EFI_VARIABLE_APPEND_WRITE** attribute is **not**set, and an existing variable with the given name and GUID exists, then that variable isdeleted.

10.4.9.5 Return Values

Table 22. Return Values - SetVariable()

EFI_SUCCESS	The bootloader has successfully stored the variable and its data as defined by the Attributes.
EFI_INVALID_PARAMETER	An invalid combination of attribute bits, name, and GUID wassupplied, or the *DataSize* exceeds the maximum allowed.
EFI_INVALID_PARAMETER	*VariableName* is an empty string.
EFI_OUT_OF_RESOURCES	Not enough storage is available to hold the variable and its data.
EFI_DEVICE_ERROR	The variable could not be stored because of a hardware error.
EFI_WRITE_PROTECTED	The variable is read-only.
EFI_WRITE_PROTECTED	The variable cannot be deleted.
EFI_SECURITY_VIOLATION	The variable could not be written due to **EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS, or EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS,** or **EFI_VARIABLE_ENHANCED_AUTHENTICATED_ACCESS** being set. The FSP is forbidden from writing to authenticated variables. This feature is only relevant for UEFI Secure Boot and the FSP does not require the bootloader to implement UEFI Secure Boot.
EFI_NOT_FOUND	The variable trying to be updated or deleted was not found.

10.4.10 EDKII_PEI_VARIABLE_PPI.QueryVariableInfo ()

10.4.10.1 Summary

This service informs the FSP of how much nonvolatile storage space is allocated for thestorage of variables, how much is remaining, and what the maximum allowable size is for each variable.

10.4.10.2 Prototype

```
typedef
 EFI_STATUS
(EFIAPI *EDKII_PEI_QUERY_VARIABLE_INFO) (
 IN CONST EDKII_PEI_VARIABLE_PPI *This,
 IN     UINT32          *Attributes,
 OUT    UINT64          *MaximumVariableStorageSize,
 OUT    UINT64          *RemainingVariableStorageSize,
 OUT    UINT64          *MaximumVariableSize
);
```

10.4.10.3 Parameters

This	A pointer to this instance of the **EDKII_PEI_VARIABLE_PPI**.
Attributes	Attributes bitmask to specify the type of variables on which toreturn information.
MaximumVariableStorageSize	Returns the maximum size of the storage space available forvariables associated with the *Attributes* specified.
RemainingVariableStorageSize	Returns the remaining size of the storage space available forvariables associated with the *Attributes* specified.
MaximumVariableSize	Returns the maximum size of an individual variable associatedwith the *Attributes* specified.

10.4.10.4 Description

This service informs the FSP of how much non-volatile storage space is allocated forthe storage of variables, how much is remaining, and what the maximum allowable sizeis for each variable.

The minimum amount of storage space required by the FSP will be mentioned in the Integration Guide.

10.4.10.5 Return Values

Table 23. Return Values - QueryVariableInfo()

EFI_SUCCESS	The usage of non-volatile storage was determined successfully.
EFI_INVALID_PARAMETER	An invalid combination of *Attribute* bits was supplied

EFI_UNSUPPORTED	The given Attribute bitmask is not supported on this platform, and the MaximumVariableStorageSize, RemainingVariableStorageSize, MaximumVariableSize areundefined.

10.4.11 FSP Error Information

FSP_ERROR_INFO

10.4.11.1 Summary

Notifies the bootloader of a fatal error occurring during the execution of the FSP.

10.4.11.2 GUID

```
#define STATUS_CODE_DATA_TYPE_FSP_ERROR_GUID \
 {0x611e6a88, 0xadb7, 0x4301, \
 {0x93, 0xff, 0xe4, 0x73, 0x04, 0xb4, 0x3d, 0xa6}}
```

10.4.11.3 Prototype

```
typedef struct {
 EFI_STATUS_CODE_DATA DataHeader;
 EFI_GUID            ErrorType;
 EFI_STATUS          Status;
} FSP_ERROR_INFO;
```

10.4.11.4 Parameters

DataHeader	The data header identifying the data. ***DataHeader.HeaderSize*** shall be **sizeof(EFI_STATUS_CODE_DATA).** ***DataHeader.Size*** shall be **sizeof (FSP_ERROR_INFO) - HeaderSize**. Finally, ***DataHeader.Type*** shall be **STATUS_CODE_DATA_TYPE_FSP_ERROR_GUID.**
ErrorType	A GUID identifying the nature of the fatal error. ThisGUID is platform specific. A listing of all possible GUIDs shall be provided by the *Integration Guide*.
Status	A code describing the error encountered. Please see *Section 13.2* for a listing of possible error codes.

10.4.11.5 Description

In the case of a fatal error occurring during the execution of the FSP, it may not be possible for the FSP to continue. If a fatal error that prevents the successful completionof the FSP occurs, the FSP may use **FSP_ERROR_INFO** to report this error to the bootloader. During PEI phase, *(*PeiServices)-> ReportStatusCode ()* shall be used to transmit this error notification to the bootloader. During DXE phase, **EFI_STATUS_CODE_PROTOCOL.ReportStatusCode ()** shall be used to transmit this error notification to the bootloader. The bootloader must ensure that *ReportStatusCode ()* services are available before FSP-M begins execution. When the FSP calls *ReportStatusCode ()*, the Type parameter's **EFI_STATUS_CODE_TYPE_MASK**must be **EFI_ERROR_CODE** with the **EFI_STATUS_CODE_SEVERITY_MASK** >= **EFI_ERROR_UNRECOVERED**. The Value and Instance parameters must be 0. The CallerId parameter should be a GUID that identifies the PEIM or DXE driver which was executing at the time of the error.

The bootloader must register a listener for this status code. This listener should check if **DataHeader.Type == STATUS_CODE_DATA_TYPE_FSP_ERROR_GUID** to detect an **FSP_ERROR_INFO** notification. If an **FSP_ERROR_INFO** notification is encountered, the bootloader should assume that normal operation is no longer possible. In debug scenarios, this notification should be considered an ASSERT. In a production environment the most simple and least effective method of handling this error is a CPU dead loop, which effectively causes a bricked system. A more robust and recommendedsolution would be to initiate a firmware recovery. If the bootloader does not handle this notification, the PEIM or DXE driver that called *ReportStatusCode ()* will immediately return back to the dispatcher with an **EFI_STATUS** return code matching the ***Status*** field in **FSP_ERROR_INFO**. Continuing to dispatch FSP PEIMs or DXE Drivers after this will result in undefined behavior. The bootloader should initiate recovery flows instead of continuing with normal dispatch.

10.4.12 FSP Debug Messages

FSP may optionally include the capability of generating log messages to aid in the debugging of firmware issues. When technically feasible, these log messages will be broadcast to the bootloader from the FSP by calling *(*PeiServices)-> ReportStatusCode ()* in PEI phase or **EFI_STATUS_CODE_PROTOCOL.ReportStatusCode ()** in DXE phase.

The format of messages provided through *ReportStatusCode()* shall match those usedfor FSP Events, please see *Section 9.5* for details. All the message types (Debug Messages, POST codes, etc.) described in *Section 9.5* are also applicable to Dispatch Mode.

It should be noted that the strings for these log messages increase the binary size ofthe FSP considerably. Accordingly, FSP binaries intended for production use are unlikely includes debug log messages.

§§

11.0 FSP Output

The FSP builds a series of data structures called the Hand Off Blocks (HOBs). These data structures conform to the HOB format as described in the *Platform Initialization(PI) Specification - Volume 3: Shared Architectural Elements* specification as referenced in *Section 1.3 Related Documentation*. The user of the FSP binary is strongly encouraged to go through the specification mentioned above to understand the HOB details and create a simple infrastructure to parse the HOB list, because the same infrastructure can be reused with different FSP across different platforms.

The bootloader developer must decide on how to consume the information passed through the HOB produced by the FSP. The *PI Specification* defines a number of HOBand most of this information may not be relevant to a particular bootloader. For example, to generate system memory map, bootloader needs to parse the resource descriptor HOBs produced by FSP-M.

In addition to the *PI Specification* defined HOB, the FSP produces a number of FSP architecturally defined GUID types HOB. The following sections describe the GUID andstructure of these FSPs defined HOB.

Additional platform-specific HOB may be defined in the *Integration Guide*.

11.1 FSP_RESERVED_MEMORY_RESOURCE_HOB

The FSP optionally reserves some memory for its internal use and a descriptor for this memory region used by the FSP is passed back through a HOB. This is a generic resource HOB, but the owner field of the HOB identifies the owner as FSP. **This FSP reserved memory region must be preserved by the bootloader and must be reportedas reserved memory to the OS.**

This HOB follows the **EFI_HOB_RESOURCE_DESCRIPTOR** format with the owner GUID defined as below.

```
#define FSP_RESERVED_MEMORY_RESOURCE_HOB_GUID \
{ 0x69a79759, 0x1373, 0x4367, { 0xa6, 0xc4, 0xc7, 0xf5,0x9e,
0xfd, 0x98, 0x6e }}
```

This HOB is valid after *FspMemoryInit()* API.

11.2 FSP_NON_VOLATILE_STORAGE_HOB2

This HOB has been replaced by FSP variable services and is considered deprecated. IfBIT3 (Variable Support) of the *ImageAttribute* field in the **FSP_INFO_HEADER** is set, then the FSP will not produce this HOB, nor will it use the *NvsBufferPtr* field in **FSPM_ARCH2_UPD**, **FSPM_ARCH_UPD**, or **FSP_ARCH_CONFIG_PPI**. Bootloaders should provide FSP variable services and only search for this HOB if BIT3 is not set.

The Non-Volatile Storage (NVS) HOB version 2 provides a mechanism for FSP to request the bootloader to save the platform configuration data into non-volatile storage so that it can be reused in special cases, such as S3 resume or fast boot.

One of the limitations of the HOB format is the 16-bit length field limits the amount ofdata that can be stored in a single HOB to approximately 64KB. Version 2 of this HOB allows >64KB of NVS data to be stored by specifying a pointer to the NVS data.

This HOB follows the **EFI_HOB_GUID_TYPE** format with the name GUID and contentdefined as below:

```
#define FSP_NON_VOLATILE_STORAGE_HOB2_GUID \
{ 0x4866788f, 0x6ba8, 0x47d8, { 0x83, 0x6, 0xac, 0xf7,
0x7f,0x55, 0x10, 0x46 }}

typedef struct {
 EFI_HOB_GUID_TYPE          GuidHob;
 EFI_PHYSICAL_ADDRESS       NvsDataPtr;
 UINT64                     NvsDataLength;
} FSP_NON_VOLATILE_STORAGE_HOB2;
```

GuidHob	The GUID HOB header identifying the data. ***GuidHob.Name*** shall be **FSP_NON_VOLATILE_STORAGE_HOB2_GUID**.
NvsDataPtr	Pointer to the non-volatile storage (NVS) data buffer. If it is **NULL** it indicates the NVS data was not produced, bootloader should continue to pass the existing NVS data to FSP during next boot.
NvsDataLength	The total number of bytes in the non-volatile storage (NVS)data buffer.

The bootloader needs to parse the HOB list to see if such a GUID HOB exists after memory is initialized. The HOB(s) shall be populated after FSP-M is complete. If it exists, the bootloader should extract the NVS data from the buffer specified by **FSP_NON_VOLATILE_STORAGE_HOB2.NvsDataPtr** and then save it into a platform-specific NVS device, such as flash, EEPROM, etc. On subsequent boots, the bootloadershould load the data block back from the NVS device to temporary memory and populate the buffer pointer into **FSPM_ARCH_UPD.NvsBufferPtr** field before calling *FspMemoryInit()* in API mode or **FSPM_ARCH_CONFIG_PPI.NvsBufferPtr** before installing **FSPM_ARCH_CONFIG_PPI** in dispatch mode. If the NVS device is memory mapped, the bootloader can initialize the buffer pointer directly to the buffer.

In API mode, the NVS data buffer shall be contained within the FSP reserved memory region defined by **FSP_RESERVED_MEMORY_RESOURCE_HOB**. In dispatch mode, the NVS data buffer will be contained in a memory region reserved via a Memory AllocationHOB (**EFI_HOB_MEMORY_ALLOCATION**) with **EFI_HOB_MEMORY_ALLOCATION.AllocDescriptor.MemoryType set to** EfiBootServicesData.

If **FSP_INFO_HEADER.SpecVersion** >= 0x23, then the FSP should produce **FSP_NON_VOLATILE_STORAGE_HOB2** instead of

FSP_NON_VOLATILE_STORAGE_HOB. Bootloaders should practice defensive programming and not explicitly check the valueof **FSP_INFO_HEADER.SpecVersion** to determine which type of HOB to search for. Instead, bootloaders should first search for **FSP_NON_VOLATILE_STORAGE_HOB2**, andonly search for **FSP_NON_VOLATILE_STORAGE_HOB** if the former is not found in the HOB list.

This HOB must be parsed after *FspMemoryInit()* in API mode or when a PPI notification for EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI with the EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH type is invoked in dispatch mode (EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK type will be invoked too early.)

If this HOB is not produced in S3 or fast boot, bootloader should continue to pass the existing NVS data to FSP during next boot.

11.3 FSP_NON_VOLATILE_STORAGE_HOB

This HOB has been replaced by FSP variable services and is considered deprecated. IfBIT3 (Variable Support) of the *ImageAttribute* field in the **FSP_INFO_HEADER** is set, then the FSP will not produce this HOB, nor will it use the *NvsBufferPtr* field in **FSPM_ARCH2_UPD**, **FSPM_ARCH_UPD**, or **FSP_ARCH_CONFIG_PPI**. Moreover, this HOB was replaced by **FSP_NON_VOLATILE_STORAGE_HOB2** before FSP variable services were added. If a bootloader wishes to retain backwards compatibility back to FSP v2.0, the bootloader should provide FSP variable services, then search for **FSP_NON_VOLATILE_STORAGE_HOB2**, and only search for **FSP_NON_VOLATILE_STORAGE_HOB** if the former is not found in the HOB list.

The Non-Volatile Storage (NVS) HOB provides a mechanism for FSP to request the bootloader to save the platform configuration data into non-volatile storage so that itcan be reused in special cases, such as S3 resume or fast boot.

This HOB follows the **EFI_HOB_GUID_TYPE** format with the name GUID defined as below:

```
#define FSP_NON_VOLATILE_STORAGE_HOB_GUID \
{ 0x721acf02, 0x4d77, 0x4c2a, { 0xb3, 0xdc, 0x27, 0xb,
0x7b,0xa9, 0xe4, 0xb0 }}
```

The bootloader needs to parse the HOB list to see if such a GUID HOB exists after memory is initialized. The HOB shall be populated either after returning from

FspMemoryInit() in API mode or after all notification call backs for **EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI** are completed in dispatch mode. If it exists, the bootloader should extract the data portion from the HOB structure and then save it into a platform-specific NVS device, such as flash, EEPROM, etc. On the followingboot flow the bootloader should load the data block back from the NVS device to temporary memory and populate the buffer pointer into **FSPM_ARCH_UPD.NvsBufferPtr** field before calling *FspMemoryInit()* in API mode or **FSPM_ARCH_CONFIG_PPI.NvsBufferPtr** before installing **FSPM_ARCH_CONFIG_PPI** in dispatch mode. If the NVS device is memory mapped, the bootloader can initialize thebuffer pointer directly to the buffer.

This HOB must be parsed after *FspMemoryInit()* in API mode or when a PPI notification for EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI with the

EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH type is invoked in dispatch mode (EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK type will be invoked too early.)

If this HOB is not produced in S3 or fast boot, bootloader should continue to pass the existing NVS data to FSP during next boot.

11.4 FSP_BOOTLOADER_TOLUM_HOB

The FSP can reserve some memory below "top of low usable memory" for bootloader usage. The size of this region is determined by **FSPM_ARCH_UPD.BootLoaderTolumSize** when in API mode or **FSPM_ARCH_CONFIG_PPI.BootLoaderTolumSize** when in dispatch mode. The FSP reserved memory region will be placed below this region.

This HOB will only be published when the **BootLoaderTolumSize** is valid and non-zero.

This HOB follows the **EFI_HOB_RESOURCE_DESCRIPTOR** format with the owner GUID defined as below:

```
#define FSP_BOOTLOADER_TOLUM_HOB_GUID \
{ 0x73ff4f56, 0xaa8e, 0x4451, { 0xb3, 0x16, 0x36, 0x35,
0x36,
0x67, 0xad, 0x44 }}
```

This HOB is valid after FspMemoryInit() in API mode or when a PPI notification for EFI_PEI_PERMANENT_MEMORY_INSTALLED_PPI with EFI_PEI_PPI_DESCRIPTOR_NOTIFY_DISPATCH priority is invoked in dispatch mode (EFI_PEI_PPI_DESCRIPTOR_NOTIFY_CALLBACK priority is too early.)

11.5 EFI_PEI_GRAPHICS_INFO_HOB

If BIT0 (Graphics Support) of the ImageAttribute field in the **FSP_INFO_HEADER** is set, the FSP includes graphics initialization capabilities. To complete the initialization of thegraphics system, FSP may need some platform specific configuration data which wouldbe documented in the *Integration Guide*.

When graphics capability is included in FSP and enabled as documented in *IntegrationGuide*, FSP produces a **EFI_PEI_GRAPHICS_INFO_HOB** as described in the *PI Specification* as referenced in *Section 1.3 Related Documents,* which provides information about the graphics mode and framebuffer.

```
#define EFI_PEI_GRAPHICS_INFO_HOB_GUID \
{ 0x39f62cce, 0x6825, 0x4669, { 0xbb, 0x56, 0x54, 0x1a,
0xba,0x75, 0x3a, 0x07 }}
```

It is to be noted that the **FrameBufferAddress** address in **EFI_PEI_GRAPHICS_INFO_HOB** will reflect the value assigned by the FSP. A bootloader consuming this HOB should be aware that a generic PCI enumeration logic could reprogram the temporary resources assigned by the FSP and it is the responsibility ofthe bootloader to update its internal data structures with the new framebuffer address after the enumeration is complete.

In API mode, if FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit == 0 then this HOB is valid after *FspSiliconInit()*. If FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit != 0, then this HOB is valid after completing the multi-phase SiliconInit sequence by invoking the *FspMultiPhaseSiInit()* API with PhaseIndex == (NumberOfPhases – 1).

In dispatch mode, this HOB is valid after EFI_PEI_END_OF_PEI_PHASE_PPI is installed.

11.6 EFI_PEI_GRAPHICS_DEVICE_INFO_HOB

If BIT0 (Graphics Support) of the ImageAttribute field in the **FSP_INFO_HEADER** is set, the FSP includes graphics initialization capabilities. To complete the initialization of thegraphics system, FSP may need some platform specific configuration data which wouldbe documented in the *Integration Guide*.

When graphics capability is included in FSP and enabled as documented in *IntegrationGuide*, FSP produces a **EFI_PEI_GRAPHICS_DEVICE_INFO_HOB** as described in the *PISpecification* as referenced in *Section 1.3 Related Documents,* which provides information about the graphics hardware which produces the framebuffer supplied by **EFI_PEI_GRAPHICS_INFO_HOB**.

```
#define EFI_PEI_GRAPHICS_DEVICE_INFO_HOB_GUID \
{ 0xe5cb2ac9, 0xd35d, 0x4430, { 0x93, 0x6e, 0x1d, 0xe3,
0x32,0x47, 0x8d, 0xe7 }}
```

Together, **EFI_PEI_GRAPHICS_INFO_HOB** and **EFI_PEI_GRAPHICS_DEVICE_INFO_HOB** provide sufficient information to implement a basic graphics driver.

In API mode, if FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit == 0 then this HOB is valid after FspSiliconInit(). If FSPS_ARCH_UPD.EnableMultiPhaseSiliconInit != 0,

then this HOB is valid after completing the multi-phase SiliconInit sequence by invoking the FspMultiPhaseSiInit() API with PhaseIndex == (NumberOfPhases – 1).

In dispatch mode, this HOB is valid after EFI_PEI_END_OF_PEI_PHASE_PPI is installed.

11.7 FSP_ERROR_INFO_HOB

In the case of an error occurring during the execution of the FSP, the FSP may optionally produce an **FSP_ERROR_INFO_HOB** which describes the error in more detail. This HOB is only produced in API mode. In dispatch mode, *ReportStatusCode ()* isused as described in *Section* *10.4.6.*

```
#define FSP_ERROR_INFO_HOB_GUID \
 {0x611e6a88, 0xadb7, 0x4301, \
 {0x93, 0xff, 0xe4, 0x73, 0x04, 0xb4, 0x3d, 0xa6}}
```

```
typedef struct {
 EFI_HOB_GUID_TYPE           GuidHob;
 EFI_STATUS_CODE_TYPE        Type;
 EFI_STATUS_CODE_VALUE       Value;
 UINT32                      Instance;
 EFI_GUID                    CallerId;
 EFI_GUID                    ErrorType;
 UINT32                      Status;
} FSP_ERROR_INFO_HOB;
```

GuidHob	The GUID HOB header identifying the data.***GuidHob.Name*** shall be **FSP_ERROR_INFO_HOB_GUID**.
Type	A *ReportStatusCode()* type identifier. The Type's **EFI_STATUS_CODE_TYPE_MASK** must be **EFI_ERROR_CODE** with the **EFI_STATUS_CODE_SEVERITY_MASK** >= **EFI_ERROR_UNRECOVERED**. See Section 6 of the PI Specification v1.7 Volume 3.
Value	A *ReportStatusCode()* Value. Used to determine status code class and sub-class, see Section 6 of the*PI Specification v1.7 Volume 3*. This field shall be setto zero (0).
Instance	A *ReportStatusCode()* Instance number. See Section6 of the *PI Specification v1.7 Volume 3*. This field shall be set to zero (0).
CallerId	An optional GUID which may be used to identify which internal component of the FSP was executing at the time of the error. If the FSP does not implement this CallerId shall be zero (0).
ErrorType	A GUID identifying the nature of the fatal error. ThisGUID is platform specific. A listing of all possible GUIDs shall be provided by the *Integration Guide*.
Status	A code describing the error encountered. Refer *Section* 13.2 for a listing of possible error codes.

If an **FSP_ERROR_INFO_HOB** is found, the bootloader should assume that normal operation is no longer possible. In debug scenarios, this notification should be considered an ASSERT. In a production environment the most simple and least effective method of handling this error is a CPU dead loop, which effectively causes a bricked system. A more robust and recommended solution would be to initiate a firmware recovery. If a **FSP_ERROR_INFO_HOB** is produced after an FSP API call, the bootloader should not call any of the subsequent FSP APIs (if any) and should instead initiate recovery flows.

11.8 FSP_SMM_BOOTLOADER_FV_CONTEXT_HOB

In the scenario where FSP owns SMRAM (FSP SMM Model 2), the bootloader can choose to provide a firmware volume containing any desired platform specific SMM drivers. In this case, the bootloader may need to communicate platform specific configuration data to these SMM drivers. The FSP SMM bootloader FV context HOB provides a mechanism for the bootloader to provide these data.

This HOB follows the **EFI_HOB_GUID_TYPE** format with the name GUID defined as below:

```
#define FSP_SMM_BOOTLOADER_FV_CONTEXT_HOB_GUID \
{ 0xf9f1dbb9, 0x1be5, 0x4c3d, { 0xb8, 0x17, 0xe6, 0xd8,
0xd,0xb5, 0x24, 0x3 }}
```

The contents of this HOB will be the data provided by **FSPI_ARCH_UPD.BootloaderSmmFvContextData** and **FSPI_ARCH_UPD.BootloaderSmmFvContextDataLength.**

This HOB will be present in the MM foundation’s HOB list. The start of the HOB list is found in the MmConfigurationTable array of the **EFI_MM_SYSTEM_TABLE** provided to entry point of all bootloader SMM drivers. Please see the PI Specification for details.

§§

12.0 Other Host Bootloader Considerations

12.1 ACPI

ACPI is an independent component of the bootloader and is not provided by the FSP inAPI mode. In dispatch mode, DXE drivers included with the FSP may optionally use the **EFI_ACPI_TABLE_PROTOCOL** to install ACPI tables.

12.2 Bus Enumeration

FSP will initialize the processor and the chipset to a state in which all bus topologies can be discovered by the host bootloader. However, it is the responsibility of the bootloader to enumerate the bus topology.

12.3 Security

FSP will follow the BWG / BIOS Specification to lock the necessary silicon specific registers. However, platform features like measured boot, verified, and authenticatedboot are responsibilities of the bootloader.

§§

Appendix A Data Structures

The declarations/definitions provided here were derived from the EDK2 source available for download at https://github.com/tianocore/edk2

A.1 BOOT_MODE

PiBootMode.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiBootMode.h

```
#define BOOT_WITH_FULL_CONFIGURATION                  0x00
#define BOOT_WITH_MINIMAL_CONFIGURATION               0x01
#define BOOT_ASSUMING_NO_CONFIGURATION_CHANGES        0x02
#define BOOT_ON_S4_RESUME                             0x05
#define BOOT_ON_S3_RESUME                             0x11
#define BOOT_ON_FLASH_UPDATE                          0x12
#define BOOT_IN_RECOVERY_MODE                         0x20
```

A.2 EFI_STATUS

UefiBaseType.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Base.h
https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Uefi/UefiBaseType.h

For x86 32-bit FSP API interface:

```
#define MAX_BIT     0x80000000
```

For x64 64-bit FSP API interface:

```
#define MAX_BIT     0x8000000000000000ULL
```

The following FSP return status are defined.

```
#define ENCODE_ERROR(StatusCode)  \
        ((EFI_STATUS)(MAX_BIT | (StatusCode)))
#define EFI_SUCCESS                          0
#define EFI_INVALID_PARAMETER                ENCODE_ERROR(2)
#define EFI_UNSUPPORTED                      ENCODE_ERROR(3)
#define EFI_NOT_READY                        ENCODE_ERROR(6)
#define EFI_DEVICE_ERROR                     ENCODE_ERROR(7)
#define EFI_OUT_OF_RESOURCES                 ENCODE_ERROR(9)
#define EFI_VOLUME_CORRUPTED                 ENCODE_ERROR(10)
#define EFI_NOT_FOUND                        ENCODE_ERROR(14)
```

```
#define EFI_TIMEOUT                             ENCODE_ERROR(18)
#define EFI_ABORTED                             ENCODE_ERROR(21)
#define EFI_INCOMPATIBLE_VERSION                ENCODE_ERROR(25)
#define EFI_SECURITY_VIOLATION                  ENCODE_ERROR(26)
#define EFI_CRC_ERROR                           ENCODE_ERROR(27)
#define EFI_COMPROMISED_DATA                    ENCODE_ERROR(33)

typedef UINT64              EFI_PHYSICAL_ADDRESS;
```

OEM Status Code

The range of status code that has the highest bit clear and the next to highest bit setare reserved for use by OEMs.

The FSP will use the following status to indicate that an API is requesting that a reset isrequired.

```
#define ENCODE_RESET_REQUEST(ResetType)   \
              ((EFI_STATUS)((MAX_BIT >> 1) | (ResetType)))
#define FSP_STATUS_RESET_REQUIRED_COLD ENCODE_RESET_REQUEST(1)
#define FSP_STATUS_RESET_REQUIRED_WARM ENCODE_RESET_REQUEST(2)
#define FSP_STATUS_RESET_REQUIRED_3    ENCODE_RESET_REQUEST(3)
#define FSP_STATUS_RESET_REQUIRED_4    ENCODE_RESET_REQUEST(4)
#define FSP_STATUS_RESET_REQUIRED_5    ENCODE_RESET_REQUEST(5)
#define FSP_STATUS_RESET_REQUIRED_6    ENCODE_RESET_REQUEST(6)
#define FSP_STATUS_RESET_REQUIRED_7    ENCODE_RESET_REQUEST(7)
#define FSP_STATUS_RESET_REQUIRED_8    ENCODE_RESET_REQUEST(8)
#define FSP_STATUS_VARIABLE_REQUEST   ENCODE_RESET_REQUEST(10)
```

A.3 EFI_PEI_GRAPHICS_INFO_HOB

GraphicsInfoHob.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Guid/GraphicsInf oHo b.h

```
typedef struct {
 EFI_PHYSICAL_ADDRESS       FrameBufferBase;
 UINT32                     FrameBufferSize;
 EFI_GRAPHICS_OUTPUT_MODE_INFORMATION GraphicsMode;
} EFI_PEI_GRAPHICS_INFO_HOB;
```

A.4 EFI_PEI_GRAPHICS_DEVICE_INFO_HOB

GraphicsInfoHob.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Guid/GraphicsInfoHo b.h

```
typedef struct {
 UINT16                      VendorId;
 UINT16                      DeviceId;
 UINT16                      SubsystemVendorId;
 UINT16                      SubsystemId;
 UINT8                       RevisionId;
 UINT8                       BarIndex;
} EFI_PEI_GRAPHICS_DEVICE_INFO_HOB;
```

A.5 EFI_GUID

Base.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Base.h

```
typedef struct {
 UINT32 Data1;
 UINT16 Data2;
 UINT16 Data3;
 UINT8  Data4[8];
} GUID;
```

UefiBaseType.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Uefi/UefiBaseType.h

```
typedef GUID  EFI_GUID;
```

A.6 EFI_MEMORY_TYPE

UefiMultiPhase.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Uefi/UefiMultiPhase.h

```
///

/// Enumeration of memory types.

///
```

```
typedef enum {
 EfiReservedMemoryType,
 EfiLoaderCode,
 EfiLoaderData,
 EfiBootServicesCode,
 EfiBootServicesData,
 EfiRuntimeServicesCode,
 EfiRuntimeServicesData,
 EfiConventionalMemory,
 EfiUnusableMemory,
 EfiACPIReclaimMemory,
 EfiACPIMemoryNVS,
 EfiMemoryMappedIO,
 EfiMemoryMappedIOPortSpace,
 EfiPalCode,
 EfiPersistentMemory,
 EfiMaxMemoryType
} EFI_MEMORY_TYPE;
```

A.7 Hand Off Block (HOB)

PiHob.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiHob.h

```
typedef UINT32 EFI_RESOURCE_TYPE;
typedef UINT32 EFI_RESOURCE_ATTRIBUTE_TYPE;

//
// Value of ResourceType in EFI_HOB_RESOURCE_DESCRIPTOR.
//
#define EFI_RESOURCE_SYSTEM_MEMORY          0x00000000
#define EFI_RESOURCE_MEMORY_MAPPED_IO       0x00000001
#define EFI_RESOURCE_IO                     0x00000002
#define EFI_RESOURCE_FIRMWARE_DEVICE        0x00000003
#define EFI_RESOURCE_MEMORY_MAPPED_IO_PORT 0x00000004
#define EFI_RESOURCE_MEMORY_RESERVED             0x00000005
#define EFI_RESOURCE_IO_RESERVED            0x00000006
#define EFI_RESOURCE_MAX_MEMORY_TYPE             0x00000007

//
// These types can be ORed together as needed.
// The first three enumerations describe settings
//
#define EFI_RESOURCE_ATTRIBUTE_PRESENT           0x00000001
#define EFI_RESOURCE_ATTRIBUTE_INITIALIZED       0x00000002
#define EFI_RESOURCE_ATTRIBUTE_TESTED       0x00000004
```

```
//
// The rest of the settings describe capabilities
//
#define EFI_RESOURCE_ATTRIBUTE_SINGLE_BIT_ECC    0x00000008
#define EFI_RESOURCE_ATTRIBUTE_MULTIPLE_BIT_ECC  0x00000010
#define EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_1    0x00000020
#define EFI_RESOURCE_ATTRIBUTE_ECC_RESERVED_2    0x00000040
#define EFI_RESOURCE_ATTRIBUTE_READ_PROTECTED    0x00000080
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTED   0x00000100
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTED 0x00000200
#define EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE       0x00000400
#define EFI_RESOURCE_ATTRIBUTE_WRITE_COMBINEABLE 0x00000800
#define EFI_RESOURCE_ATTRIBUTE_WRITE_THROUGH_CACHEABLE 0x00001000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_BACK_CACHEABLE 0x00002000
#define EFI_RESOURCE_ATTRIBUTE_16_BIT_IO    0x00004000
#define EFI_RESOURCE_ATTRIBUTE_32_BIT_IO    0x00008000
#define EFI_RESOURCE_ATTRIBUTE_64_BIT_IO    0x00010000
#define EFI_RESOURCE_ATTRIBUTE_UNCACHED_EXPORTED 0x00020000
#define EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTED 0x00040000
#define EFI_RESOURCE_ATTRIBUTE_READ_PROTECTABLE  0x00100000
#define EFI_RESOURCE_ATTRIBUTE_WRITE_PROTECTABLE 0x00200000
#define EFI_RESOURCE_ATTRIBUTE_EXECUTION_PROTECTABLE 0x00400000
#define EFI_RESOURCE_ATTRIBUTE_READ_ONLY_PROTECTABLE 0x00800000
#define EFI_RESOURCE_ATTRIBUTE_PERSISTABLE  0x01000000
#define EFI_RESOURCE_ATTRIBUTE_MORE_RELIABLE     0x02000000


//
// HobType of EFI_HOB_GENERIC_HEADER.
//
#define EFI_HOB_TYPE_MEMORY_ALLOCATION      0x0002
#define EFI_HOB_TYPE_RESOURCE_DESCRIPTOR    0x0003
#define EFI_HOB_TYPE_GUID_EXTENSION         0x0004
#define EFI_HOB_TYPE_UNUSED                 0xFFFE
#define EFI_HOB_TYPE_END_OF_HOB_LIST        0xFFFF

///
/// Describes the format and size of the data inside the HOB.
/// All HOBs must contain this generic HOB header.
///
typedef struct {
 UINT16 HobType;
 UINT16 HobLength;
 UINT32  Reserved;
} EFI_HOB_GENERIC_HEADER;

///
```

```
/// Describes various attributes of logical memory
allocation.
///
typedef struct {
 EFI_GUID           Name;
 EFI_PHYSICAL_ADDRESS MemoryBaseAddress;
 UINT64             MemoryLength;
 EFI_MEMORY_TYPE    MemoryType;
 UINT8              Reserved[4];
} EFI_HOB_MEMORY_ALLOCATION_HEADER;


///
/// Describes all memory ranges used during the HOB producer
/// phase that exist outside the HOB list. This HOB type
/// describes how memory is used, not the physical attributes
/// of memory.
///
typedef struct {
 EFI_HOB_GENERIC_HEADER     Header;
 EFI_HOB_MEMORY_ALLOCATION_HEADER AllocDescriptor;
} EFI_HOB_MEMORY_ALLOCATION;


///
/// Describes the resource properties of all fixed,
/// nonrelocatable resource ranges found on the processor
/// host bus during the HOB producer phase.
///
typedef struct {
 EFI_HOB_GENERIC_HEADER     Header;
 EFI_GUID                   Owner;
 EFI_RESOURCE_TYPE          ResourceType;
 EFI_RESOURCE_ATTRIBUTE_TYPE ResourceAttribute;
 EFI_PHYSICAL_ADDRESS       PhysicalStart;
 UINT64                     ResourceLength;
} EFI_HOB_RESOURCE_DESCRIPTOR;

///
/// Allows writers of executable content in the HOB producer
/// phase to maintain and manage HOBs with specific GUID.
///
typedef struct {
 EFI_HOB_GENERIC_HEADER     Header;
 EFI_GUID                   Name;
} EFI_HOB_GUID_TYPE;
```

```
///
/// Union of all the possible HOB Types.
///
typedef union {
 EFI_HOB_GENERIC_HEADER     *Header;
 EFI_HOB_MEMORY_ALLOCATION *MemoryAllocation;
 EFI_HOB_RESOURCE_DESCRIPTOR *ResourceDescriptor;
 EFI_HOB_GUID_TYPE          *Guid;
 UINT8                      *Raw;
} EFI_PEI_HOB_POINTERS;
```

A.8 Firmware Volume and Firmware Filesystem

Refer to PiFirmwareVolume.h and PiFirmwareFile.h from EDK2 project fororiginal source.

PiFirmwareVolume.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiFirmwareVolume.h

```
///
/// EFI_FV_FILE_ATTRIBUTES
///
typedef UINT32 EFI_FV_FILE_ATTRIBUTES;

///
/// type of EFI FVB attribute
///
typedef UINT32 EFI_FVB_ATTRIBUTES_2;

typedef struct {
 UINT32 NumBlocks;
 UINT32 Length;
} EFI_FV_BLOCK_MAP_ENTRY;

///
/// Describes the features and layout of the firmware volume.
///
typedef struct {
 UINT8        ZeroVector[16];
 EFI_GUID     FileSystemGuid;
 UINT64             FvLength;
 UINT32             Signature;
 EFI_FVB_ATTRIBUTES_2   Attributes;
 UINT16             HeaderLength;
 UINT16             Checksum;
 UINT16             ExtHeaderOffset;
 UINT8               Reserved[1];
 UINT8               Revision;
 EFI_FV_BLOCK_MAP_ENTRY  BlockMap[1];
} EFI_FIRMWARE_VOLUME_HEADER;

#define EFI_FVH_SIGNATURE SIGNATURE_32 ('_', 'F', 'V', 'H')

///
/// Firmware Volume Header Revision definition
///
#define EFI_FVH_REVISION 0x02

///
/// Extension header pointed by ExtHeaderOffset of volume header.
///
typedef struct {
 EFI_GUID FvName;
 UINT32  ExtHeaderSize;
} EFI_FIRMWARE_VOLUME_EXT_HEADER;

///
/// Entry struture for describing FV extension header
///
typedef struct {
 UINT16 ExtEntrySize;
 UINT16  ExtEntryType;
} EFI_FIRMWARE_VOLUME_EXT_ENTRY;

#define EFI_FV_EXT_TYPE_OEM_TYPE 0x01

///
/// This extension header provides a mapping between a GUID
/// and an OEM file type.
///
typedef struct {
 EFI_FIRMWARE_VOLUME_EXT_ENTRY Hdr;
 UINT32  TypeMask;
} EFI_FIRMWARE_VOLUME_EXT_ENTRY_OEM_TYPE;

#define EFI_FV_EXT_TYPE_GUID_TYPE 0x0002

///
/// This extension header EFI_FIRMWARE_VOLUME_EXT_ENTRY_GUID_TYPE
/// provides a vendor specific GUID FormatType type which
/// includes a length and a successive series of data bytes.
///
typedef struct {
 EFI_FIRMWARE_VOLUME_EXT_ENTRY   Hdr;
 EFI_GUID     FormatType;
} EFI_FIRMWARE_VOLUME_EXT_ENTRY_GUID_TYPE;
```

PiFirmwareFile.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiFirmwareFile.h

```
///
/// Used to verify the integrity of the file.
///
typedef union {
 struct {
  UINT8  Header;
  UINT8  File;
} Checksum;
 UINT16  Checksum16;
} EFI_FFS_INTEGRITY_CHECK;

///
/// FFS_FIXED_CHECKSUM is the checksum value used when the
/// FFS_ATTRIB_CHECKSUM attribute bit is clear.
///
#define FFS_FIXED_CHECKSUM 0xAA

typedef UINT8 EFI_FV_FILETYPE;
typedef UINT8 EFI_FFS_FILE_ATTRIBUTES;
typedef UINT8 EFI_FFS_FILE_STATE;

///
/// File Types Definitions
///
#define EFI_FV_FILETYPE_FREEFORM            0x02

///
/// FFS File Attributes.
///
#define FFS_ATTRIB_LARGE_FILE               0x01
#define FFS_ATTRIB_FIXED                    0x04
#define FFS_ATTRIB_DATA_ALIGNMENT           0x38
#define FFS_ATTRIB_CHECKSUM                 0x40

///
/// FFS File State Bits.
///
#define EFI_FILE_HEADER_CONSTRUCTION        0x01
#define EFI_FILE_HEADER_VALID               0x02
#define EFI_FILE_DATA_VALID                 0x04
#define EFI_FILE_MARKED_FOR_UPDATE          0x08
```

```
#define EFI_FILE_DELETED                    0x10
#define EFI_FILE_HEADER_INVALID             0x20

///
/// Each file begins with the header that describe the
/// contents and state of the files.
///
typedef struct {
 EFI_GUID                   Name;
 EFI_FFS_INTEGRITY_CHECK    IntegrityCheck;
 EFI_FV_FILETYPE            Type;
 EFI_FFS_FILE_ATTRIBUTES    Attributes;
 UINT8                      Size[3];
 EFI_FFS_FILE_STATE         State;
} EFI_FFS_FILE_HEADER;

typedef struct {
 EFI_GUID                   Name;
 EFI_FFS_INTEGRITY_CHECK    IntegrityCheck;
 EFI_FV_FILETYPE            Type;
 EFI_FFS_FILE_ATTRIBUTES    Attributes;
 UINT8                      Size[3];
 EFI_FFS_FILE_STATE         State;
 UINT32                     ExtendedSize;
} EFI_FFS_FILE_HEADER2;

#define IS_FFS_FILE2(FfsFileHeaderPtr) \
  ((((EFI_FFS_FILE_HEADER *) (UINTN) FfsFileHeaderPtr)-
>Attributes) & FFS_ATTRIB_LARGE_FILE) == FFS_ATTRIB_LARGE_FILE)

#define FFS_FILE_SIZE(FfsFileHeaderPtr) \
  ((UINT32) (*((UINT32 *) ((EFI_FFS_FILE_HEADER *) (UINTN)
FfsFileHeaderPtr)->Size) & 0x00ffffff))

#define FFS_FILE2_SIZE(FfsFileHeaderPtr) \
  (((EFI_FFS_FILE_HEADER2 *) (UINTN) FfsFileHeaderPtr)-
>ExtendedSize)

typedef UINT8 EFI_SECTION_TYPE;
#define EFI_SECTION_RAW     0x19

///
/// Common section header.
///
typedef struct {
 UINT8        Size[3];
 EFI_SECTION_TYPE Type;
} EFI_COMMON_SECTION_HEADER;
```

```
typedef struct {
 UINT8        Size[3];
 EFI_SECTION_TYPE Type;
 UINT32       ExtendedSize;
} EFI_COMMON_SECTION_HEADER2;

///
/// The leaf section which contains an array of zero or more
/// bytes.
///
typedef EFI_COMMON_SECTION_HEADER EFI_RAW_SECTION;
typedef EFI_COMMON_SECTION_HEADER2 EFI_RAW_SECTION2;

#define IS_SECTION2(SectionHeaderPtr) \
  ((UINT32) (*((UINT32 *) ((EFI_COMMON_SECTION_HEADER *) (UINTN)
SectionHeaderPtr)->Size) & 0x00ffffff) == 0x00ffffff)

#define SECTION_SIZE(SectionHeaderPtr) \
  ((UINT32) (*((UINT32 *) ((EFI_COMMON_SECTION_HEADER *) (UINTN)
SectionHeaderPtr)->Size) & 0x00ffffff))

#define SECTION2_SIZE(SectionHeaderPtr) \
  (((EFI_COMMON_SECTION_HEADER2 *) (UINTN) SectionHeaderPtr)-
>ExtendedSize)
```

A.9 Debug Error Level

Refer to DebugLib.h from the EDK2 project for the original source.

DebugLib.h

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Library/DebugLib.h

```
//
// Declare bits for PcdDebugPrintErrorLevel and the
ErrorLevelparameter of DebugPrint()
//
#define DEBUG_INIT   0x00000001 // Initialization
#define DEBUG_WARN   0x00000002 // Warnings
#define DEBUG_LOAD   0x00000004 // Load events
#define DEBUG_FS     0x00000008 // EFI File system
#define DEBUG_POOL   0x00000010 // Alloc & Free (pool)
#define DEBUG_PAGE   0x00000020 // Alloc & Free (page)
#define DEBUG_INFO   0x00000040 // Informational debug
messages#define DEBUG_DISPATCH 0x00000080 // PEI/DXE/SMM
Dispatchers #define DEBUG_VARIABLE 0x00000100 // Variable
```

```
#define DEBUG_BM    0x00000400 // Boot Manager
#define DEBUG_BLKIO 0x00001000 // BlkIo Driver
#define DEBUG_NET   0x00004000 // Network Io Driver
#define DEBUG_UNDI  0x00010000 // UNDI Driver #define
DEBUG_LOADFILE 0x00020000 // LoadFile
#define DEBUG_EVENT 0x00080000 // Event messages
#define DEBUG_GCD   0x00100000 // Global Coherency Database
changes
#define DEBUG_CACHE 0x00200000 // Memory range cachability
changes
#define DEBUG_VERBOSE 0x00400000 // Detailed debug messages
thatmay
              // significantly impact boot performance
#define DEBUG_ERROR 0x80000000 // Error

//
// Aliases of debug message mask bits
//
#define EFI_D_INIT  DEBUG_INIT
#define EFI_D_WARN  DEBUG_WARN
#define EFI_D_LOAD  DEBUG_LOAD
#define EFI_D_FS    DEBUG_FS
#define EFI_D_POOL  DEBUG_POOL
#define EFI_D_PAGE  DEBUG_PAGE
#define EFI_D_INFO  DEBUG_INFO
#define EFI_D_DISPATCH DEBUG_DISPATCH
#define EFI_D_VARIABLE DEBUG_VARIABLE
#define EFI_D_BM    DEBUG_BM
#define EFI_D_BLKIO DEBUG_BLKIO
#define EFI_D_NET   DEBUG_NET
#define EFI_D_UNDI  DEBUG_UNDI
#define EFI_D_LOADFILE DEBUG_LOADFILE
#define EFI_D_EVENT DEBUG_EVENT
#define EFI_D_VERBOSE  DEBUG_VERBOSE
#define EFI_D_ERROR DEBUG_ERROR
```

A.10 Event Code Types

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiStatusCode.h

```
typedef UINT32 EFI_STATUS_CODE_TYPE;

#define EFI_STATUS_CODE_TYPE_MASK     0x000000FF
#define EFI_STATUS_CODE_SEVERITY_MASK 0xFF000000
#define EFI_STATUS_CODE_RESERVED_MASK  0x00FFFF00

#define EFI_PROGRESS_CODE   0x00000001
#define EFI_ERROR_CODE      0x00000002
```

```
#define EFI_DEBUG_CODE      0x00000003

#define EFI_ERROR_MINOR                     0x40000000
#define EFI_ERROR_MAJOR                     0x80000000
#define EFI_ERROR_UNRECOVERED               0x90000000
#define EFI_ERROR_UNCONTAINED               0xA0000000

typedef UINT32 EFI_STATUS_CODE_VALUE;

#define EFI_STATUS_CODE_CLASS_MASK          0xFF000000
#define EFI_STATUS_CODE_SUBCLASS_MASK       0x00FF0000
#define EFI_STATUS_CODE_OPERATION_MASK      0x0000FFFF
#define EFI_SOFTWARE                        0x03000000
```

A.11 EFI_STATUS_CODE_STRING_DATA

https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Guid/StatusCodeDat aTypeId.h

```
#define EFI_STATUS_CODE_DATA_TYPE_STRING_GUID \
 { 0x92D11080, 0x496F, 0x4D95,
 { 0xBE, 0x7E, 0x03, 0x74, 0x88, 0x38, 0x2B, 0x0A }}
typedef struct {
 UINT16 HeaderSize;
 UINT16  Size;
 EFI_GUID Type;
} EFI_STATUS_CODE_DATA;

typedef enum {
 EfiStringAscii,
 EfiStringUnicode,
 EfiStringToken
} EFI_STRING_TYPE;

typedef union {
 CHAR8              *Ascii;
 CHAR16             *Unicode;
 EFI_STATUS_CODE_STRING_TOKEN  Hii;
} EFI_STATUS_CODE_STRING;

typedef struct {
 EFI_STATUS_CODE_DATA                       DataHeader;
 EFI_STRING_TYPE            StringType;
 EFI_STATUS_CODE_STRING     String;
} EFI_STATUS_CODE_STRING_DATA;
```

§§

Appendix B Acronyms

ACPI	Advanced Configuration and Power Interface
BCT	Binary Configuration Tool
BIOS	Basic Input Output System
BSP	Boot Strap Processor
BSF	Boot Setting File
BWG	BIOS Writer’s Guide a.k.a. BIOS Specification a.k.a. IA FW Specification
FDF	Flash Description File
FSP	Firmware Support Package(s)
FSP API	Firmware Support Package Interface(s)
FV	Firmware Volume
GUI	Graphical User Interface
GUID	Globally Unique IDentifier(s)
HOB	Hand Off Block(s)
PI	Platform Initialization
PIC	Position Independent Code
RAM	Random Access Memory
ROM	Read Only Memory
SMM	System Management Mode
SOC	System-On-Chip(s)
TOLUM	Top of low usable memory
TPM	Trusted Platform Module
UEFI	Unified Extensible Firmware Interface
UPD	Updatable Product Data

§§