Intel® Firmware Support Package

External Architecture Specification

November 2015

Version 1.1a

Document Number: 332394-002US

Contents

Figures

Tables

Revision History

Date	Revision	Description
November 2015	002	• Specification version 1.1a • Section 2.2.1 – Added Data Structure Descriptions • Section 5.1.1 and 6.6: Made FspInit API optional • Section 5.1.5: Added FSP Patch Table (FSPP) • Section 6.5.4: Added clarification for the return range • Section 7.3: Updated to parse the FSP_NON_VOLATILE_STORAGE_HOB after FspMemoryInit instead FspSiliconInit API • Section 7.5: Updated EFI_PEI_GRAPHICS_INFO_HOB will not be produced in S3 boot path • Section 8.2: Defined MemoryInitUpdOffset and SiliconInitUpdOffset in the UPD standard fields • Section 10: Added GitHub links to the sample files. • Section 10.1.1: Added additional Boot Mode values
April 2015	001	• Specification version 1.1 • Added FspMemoryInit, TempRamExit and FspSiliconInit API • Added FSP_INFO_EXTENDED_HEADER • FSP_INFO_HEADER changes — Updated HeaderRevision from 1 to 2 — Update ImageRevision format to Major.Minor.Rev.Build — Define BIT0 for Display support in ImageAttribute — Updated ApiEntryNum from 3 to 6 • Updated Boot Flow • Added EFI_PEI_GRAPHICS_INFO_HOB • Added FSP_INIT_RT_COMMON_BUFFER.BootLoaderTolumSize and FSP_BOOTLOADER_TOLUM_HOB • Added Data Structure definitions • Added FSP description file information • Added Microcode Region layout, HOB and other clarifications
April 2014	1.0	• Specification version 1.0 • Initial publication

§

1 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).

1.2 Intended Audience

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

1.3 Related Documents

- *Intel® Firmware Support Package (FSP) External Architecture Specification v1.0* http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/fsp-architecture-spec.pdf
- *Intel® Firmware Support Package (FSP) External Architecture Specification v1.1* http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/fsp-architecture-spec-v1-1.pdf
- *Unified Extensible Firmware Interface (UEFI) Specification* located at http://www.uefi.org/specifications
- *Platform Initialization (PI) Specification* v1.4 located at http://www.uefi.org/sites/default/files/resources/PI_1_4.zip
- *Binary Configuration Tool (BCT) for Intel® Firmware Support Package* located *at* http://www.intel.com/fsp.
- *Boot Specification File (BSF) Specification*

§

2 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 proprietary information 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 the key 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 Writers Guide (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 strucutures 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.

§

3 FSP Integration

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

Below are some required steps for the integration:

- **Customizing**

  The FSP has some sets of configuration parameters that are part of the FSP binary and can be customized by external tools provided by Intel.

- **Rebasing**

  The FSP is not Position Independent Code (PIC) and the whole FSP 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
- Vital Product Data (VPD)/Updatable Product Data (UPD) Data structure definitions
- Boot Settings File (BSF)

The FSP configuration utility called Binary Configuration Tool (BCT) will be available as a separate package.

3.2 FSP Image ID and Revision

The **FSP_INFO_HEADER** structure contained within the FSP binary contains an Image Identifier 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 the FSP as the FSP configuration data could change over different FSP Image identifiers and revisions.

§

4 Boot Flows

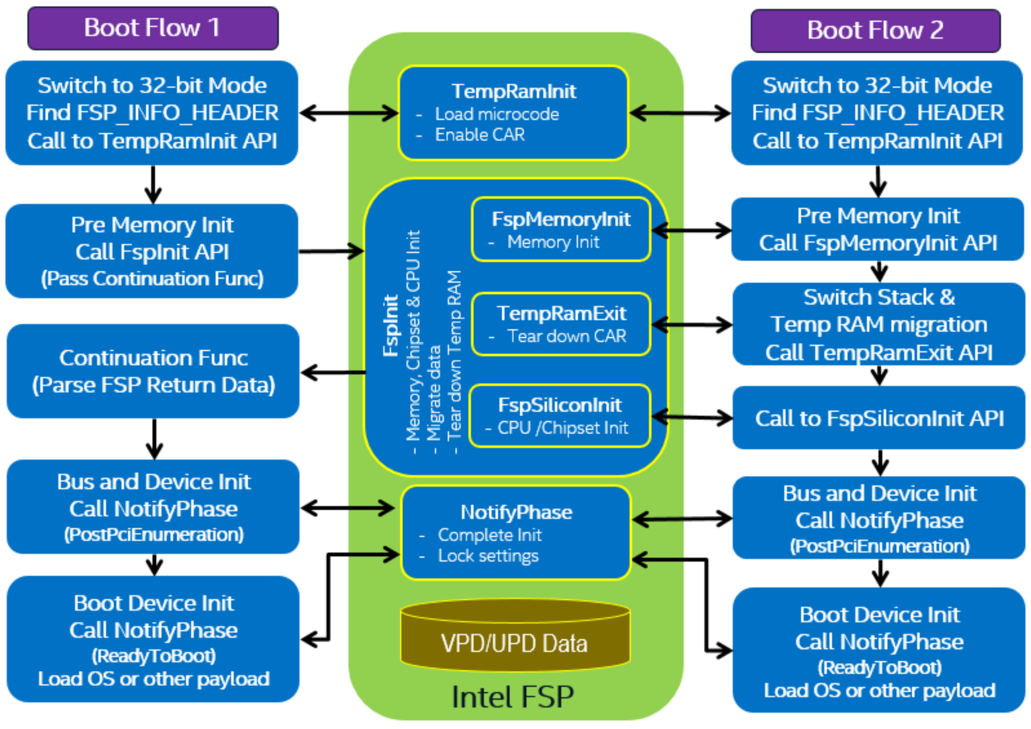
FSP supports two boot flows. The first boot flow is simpler for the bootloader. The second boot flow increases flexibility and control for the bootloader. This chapter describes the boot flows using the FspInit API (Boot Flow 1) and FspMemoryInit, TempRamExit and FspSiliconInit API (Boot Flow 2).

Boot Flow 2 is recommended for new bootloader implementations. Boot Flow 1 is supported for existing bootloader implementations.

These two boot flows are mutually exclusive, i.e. a bootloader can choose one or the other, but not both.

The figure below shows both boot flows from the reset vector to the OS hand-off for a typical bootloader. The APIs are described in more detail in the following sections.

Figure 1. Boot Flows



§

5 FSP Binary Format

The FSP is distributed in a binary format. The FSP binary contains:

a) **FSP_INFO_HEADER** structure providing information about FSP,

b) Initialization code and data needed by the Intel silicon supported, and a

c) Configuration region that allows the bootloader developer to customize some of the settings.

5.1 FSP Information tables

The FSP binary must always have an **FSP_INFO_HEADER** table and may optionally have additional tables as described below.

All FSP tables must have a 4-byte aligned base address and a size that is a multiple of 4 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.

5.1.1 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 value for this field is 72.
8	3	Reserved	Reserved bytes for future.
11	1	HeaderRevision	Revision of the header. The current value for this field is 2.

Byte Offset	Size in Bytes	Field	Description
12	4	ImageRevision	Revision of the FSP binary. Major.Minor.Revision.Build The ImageRevision can be decoded as follows: 0 : 7 - Build Number 8 : 15 - Revision 16 : 23 - Minor Version 24 : 31 - Major Version
16	8	ImageId	8 ASCII character byte signature string that will help match the FSP binary to a supported hardware configuration. BootLoader should not assume null-terminated.
24	4	ImageSize	Size of the entire FSP binary in bytes.
28	4	ImageBase	FSP binary preferred base address. If the FSP binary is located at the address different from the preferred address, the FSP binary needs to be rebased.
32	4	ImageAttribute	Attributes of the FSP binary. • Bit 0: GRAPHICS_SUPPORT – Set to 1 when FSP supports enabling Graphics Display. • Bits 1:31 - Reserved for future use.
36	4	CfgRegionOffset	Offset of the configuration region (VPD). This offset is relative to the FSP binary base address.
40	4	CfgRegionSize	Size of the configuration region (VPD).
44	4	ApiEntryNum	Number of API entries this FSP supports. The current design supports 6 API.
48	4	TempRamInitEntryOffset	Offset for the API to setup a temporary stack till the memory is initialized.
52	4	FspInitEntryOffset	Offset for the API to initialize the processor and the chipset (SOC). FspInitEntry API encapsulates the functionality of FspMemoryInit, TempRamExit and FspSiliconInit API. If the value is set to 0x00000000, then this API is not supported. Instead use the boot flow 2 as described in Boot Flows
56	4	NotifyPhaseEntryOffset	Offset for the API to inform the FSP about the different stages in the boot process.
60	4	FspMemoryInitEntryOffset	Offset for the API to initialize the Memory.
64	4	TempRamExitEntryOffset	Offset for the API to tear down the temporary memory.
68	4	FspSiliconInitEntryOffset	Offset for the API to initialize the processor and chipset.

5.1.2 FSP_INFO_EXTENDED_HEADER

The **FSP_INFO_EXTENDED_HEADER** structure conveys additional information about the FSP binary. 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.

5.1.3 Finding FSP_INFO_HEADER

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.

The 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 FV.

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 the firmware volume. All firmware files will have a GUID that can be used to identify the files, including the **FSP_INFO_HEADER** file. The **FSP_INFO_HEADER** file GUID is **FSP_FFS_INFORMATION_FILE_GUID**

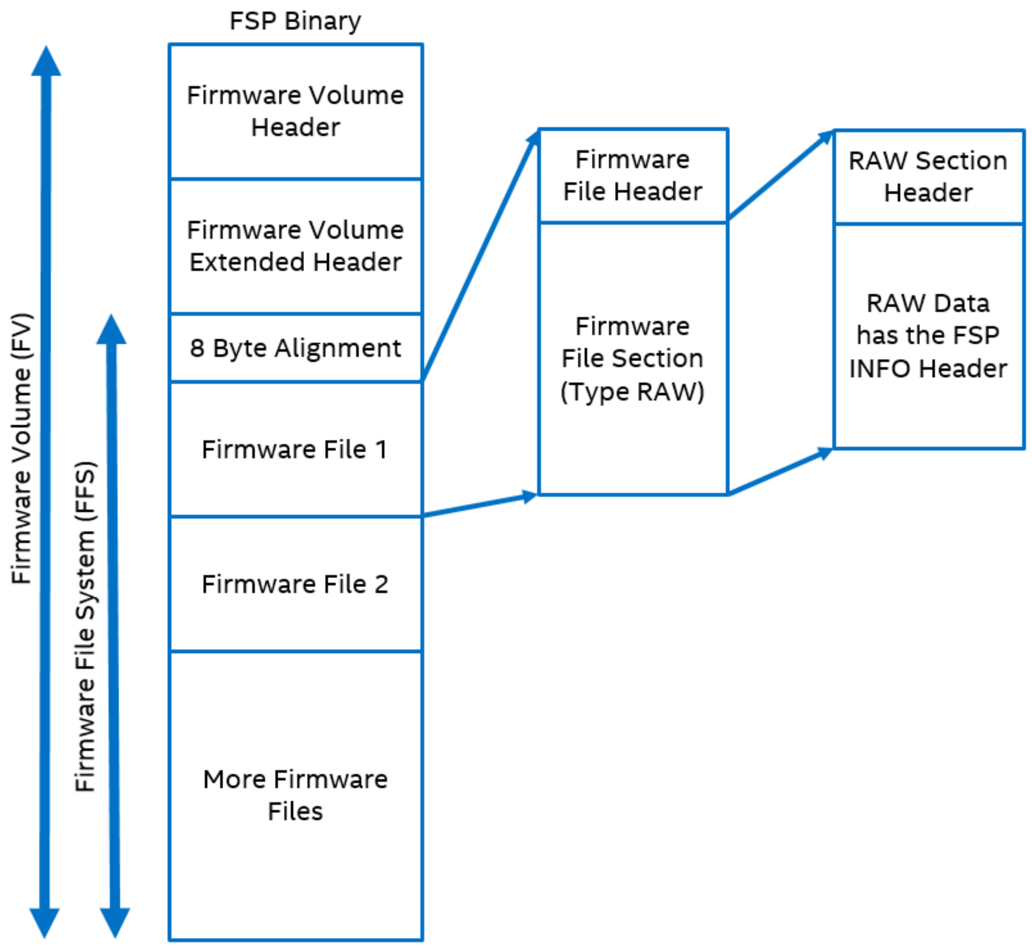
```
#define FSP_FFS_INFORMATION_FILE_GUID \
{ 0x912740be, 0x2284, 0x4734, { 0xb9, 0x71, 0x84, 0xb0, 0x27,
0x35, 0x3f, 0x0c }};
```

The bootloader can find the offset of the **FSP_INFO_HEADER** within the FSP binary 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** is located 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 below.

Figure 2. Data Structures



5.1.4 FSP Description File

The FSP binary may optionally include an FSP description file. This file will provide information about the FSP including information about different silicon revisions the FSP supports. 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 as shown 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
}
```

5.1.5 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 to be patched accordingly.

```
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.

5.1.5.1 Example

Let's assume the FSP image size is 0x38000. And we need to rebase the FSP base from 0xFFFC0000 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 entry:

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

The image offset to patch in the FSP image is indicated by BIT[23:0], 0xFFFFFC. Since BIT[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 new value should be:

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

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

§

6 FSP Interface (FSP API)

6.1 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, the behavior is not determined). These conditions include:

- The system is in flat 32-bit mode.
- Both the code and data selectors should have full 4GB access range.
- Interrupts should be turned off.
- The FSP API should be called only by the system BSP, unless otherwise noted.

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

6.2 Data Structure Convention

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

6.3 Entry-Point Calling Convention

All FSP APIs defined in the **FSP_INFO_HEADER** are 32-bit only. The FSP API 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 is called.
- The calling function needs to clean the stack up after the API returns.
- The return value is returned in the **EAX** register. All the other registers including floating point registers are preserved, except as noted in the individual API descriptions below or in *Integration Guide*.

There are, however, a couple of notable exceptions with the FSP API interface convention. Refer to individual API descriptions for any special notes and these exceptions.

6.4 Return Status Code

All FSP API return a status code to indicate the API execution result. These return status codes are defined in the Appendix A – Data Structures, Section 10.2

6.5 TempRamInit API

This FSP API is called soon after coming out of reset and before memory and stack are available. This FSP API loads the microcode update, enables code caching for a region specified by the bootloader and sets up a temporary stack to be used prior to main memory being initialized.

To invoke this API, 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 this API.

The **ESP** 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.

This API should be called only once after the system comes out the reset, and it must be called before any other FSP API. Otherwise, unexpected results may occur.

The TempRamInit API preserves the following general purpose registers **EBX**, **EDI**, **ESI**, **EBP** and the following floating point registers **MM0**, **MM1**. The bootloader can use these registers to save data across the TempRamInit API call. No other registers are preserved.

6.5.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_TEMP_RAM_INIT) (
  IN  FSP_TEMP_RAM_INIT_PARAMS     *TempRamInitParamPtr
);
```

6.5.2 Parameters

TempRamInitParamPtr

Address pointer to the **FSP_TEMP_RAM_INIT_PARAMS** structure. The structure definition is provided below under *Related Definitions*.

6.5.3 Related Definitions

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

`MicrocodeRegionBase`	Base address of the microcode region. This address must be 16 byte aligned.
`MicrocodeRegionLength`	Length of the microcode region. The length must be total size of all patches or 0xFFFFFFFF if auto size detection is desired.
`CodeRegionBase`	Base address of the cacheable flash region.
`CodeRegionLength`	Length of the cacheable flash region. A size of 0 indicates that no code caching is desired.

6.5.4 Return Values

If this function is successful, the FSP initializes the **ECX** and **EDX** registers to point to a temporary but writeable memory range available to the bootloader. Register **ECX** points to the start of this temporary memory range and **EDX** points to the end of the range [ECX, EDX], where ECX is inclusive and EDX is exclusive in the range. The bootloader is free to use the whole range described. Typically, the bootloader can reload the **ESP** register to point to the end of this returned range so that it can be used as a standard stack.

Note: This returned range is a sub-region of the whole temporary memory initialized. The FSP maintains and consumes the remaining temporary memory. The bootloader must not access the temporary memory beyond the returned boundary. The bootloader must not assume that this range is initialized with zeros.

Table 4. Return Values - TempRamInit API

EFI_SUCCESS	Temporary RAM was initialized successfully.
EFI_INVALID_PARAMETER	Input parameters are invalid.
EFI_NOT_FOUND	A valid microcode was not loaded in the processor.
EFI_UNSUPPORTED	The FSP calling conditions were not met.
EFI_DEVICE_ERROR	Temp RAM initialization failed.

6.5.5 Description

The entry to this function is in a stackless/memoryless environment. 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 function. It then converts the offset to an absolute address by adding the base of the FSP binary and jumps to the TempRamInit function as described in 6.3 Entry-Point Calling Convention and 6.6 FspInit API

The temporary memory range returned by this API is intended to be primarily used by the 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.

The input parameter structure describes a microcode region. The microcode region may have multiple microcode patches starting at a 16 byte boundary and packed together one after the other. The FSP will attempt to load the latest revision of the appropriate microcode patch based on CPUID and the microcode patch header contents.

The microcode region may have a defined length or not. In either case, the FSP stops looking for additional microcode patches if either:

- A valid microcode header is not found on the subsequent 16 byte aligned address
- If the size of the microcode patch region is exceeded

The microcode region is required even if the hardware or bootloader load the microcode patch before calling TempRamInit API.

The microcode region needs to remain in the same address across all FSP API calls.

The code caching region is optional. Valid code caching regions may be limited by the FSP implementation or the hardware, as specified in the *Integration Guide*.

6.6 FspInit API

This FSP API is called after TempRamInit. This FSP API initializes the memory, the processor and the chipset to enable normal operation of these devices. This FSP API accepts a pointer to a data structure that will be platform dependent and defined for each FSP binary. This will be documented with each FSP release in the *Integration Guide*.

The bootloader provides a continuation function as a parameter when calling FspInit. After FspInit completes its execution, it does not return to the bootloader from where it was called but instead returns control to the bootloader by calling the continuation function which is passed to FspInit as an argument.

The FspMemoryInit, TempRamExit and FspSiliconInit API provide an alternate method to complete the silicon initialization and provides the bootloader the opportunity to get control after system memory is available and before the temporary memory is torn down.

This API should be called only once after the TempRamInit API.

Use of this API is mutually exclusive to the FspMemoryInit, TempRamExit and FspSilicon API.

When HeaderRevision in FSP_INFO_TABLE is >=2, this API is optional. If this API is not implemented then the FspInitEntryOffset in FSP_INFO_TABLE is 0x00000000.

6.6.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_INIT) (
  IN OUT  FSP_INIT_PARAMS       *FspInitParamPtr
);
```

6.6.2 Parameters

FspInitParamPtr	Address pointer to the **FSP_INIT_PARAMS** structure.

6.6.3 Related Definitions

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

`NvsBufferPtr`	Pointer to the non-volatile storage (NVS) data buffer. If it is **`NULL`**, it indicates the NVS data is not available.
`RtBufferPtr`	Pointer to the runtime data buffer **`FSP_INIT_RT_BUFFER`**. This buffer contains FSP configuration data that will be used during the platform initialization. The detailed structure layout is described in 6.6.3.1.
`ContinuationFunc`	Pointer to a continuation function provided by the bootloader.

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

`Status`	Status of the FspInit API.
`HobListPtr`	Pointer to the HOB data structure defined in Section 10, Appendix A – Data Structures.

6.6.3.1 FSP_INIT_RT_BUFFER

This structure contains a common configuration data structure FSP_INIT_RT_COMMON_BUFFER defined below, followed by platform specific-data that will be defined in the *Integration Guide*.

```
typedef struct {
  FSP_INIT_RT_COMMON_BUFFER    Common;
  ..... // Optional platform specific data structure
} FSP_INIT_RT_BUFFER;
```

```
typedef struct {
  UINT32                  *StackTop;
  UINT32                  BootMode;
  VOID                    *UpdDataRgnPtr;
  UINT32                  BootLoaderTolumSize;
  UINT32                  Reserved[6];
} FSP_INIT_RT_COMMON_BUFFER;
```

`StackTop`	Points to the desired bootloader stack top location in memory after memory is initialized.
`BootMode`	Current boot mode. Possible bit values definitions are defined in Section 10, Appendix A - Data Structures.
`UpdDataRgnPtr`	Pointer to an updatable platform configuration data structure **`UPD_DATA_REGION`** defined in Integration Guide. This structure contains options that can be overridden by the bootloader at runtime. If this pointer is **`NULL`**, it indicates the default built-in values in the FSP binary will be used. Refer to Section 8, FSP Configuration Firmware File for details.
`BootLoaderTolumSize`	The size of memory to be reserved below the top of low usable memory (TOLUM) for bootloader usage. This is optional and value can be zero. If non-zero, the size must be a multiple of 4KB.
`Reserved`	Reserved fields. Must be set to 0.

6.6.4 Return Values

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

Table 5. Return Values - FspInit 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 initialization failed.

6.6.5 Description

One important piece of data that will be part of the **FSP_INIT_RT_BUFFER** structure is the **StackTop**. This passes the address of the stack top where the bootloader wants to establish the stack after memory is initialized and available for use.

Note that the FspInit API initializes the permanent memory and switches the stack from the temporary memory to the permanent memory as specified by **FSP_INIT_RT_COMMON_BUFFER.StackTop**. Sometimes switching the stack in a function can cause some unexpected execution results because the compiler is not aware of the stack change during runtime and the precompiled code may still refer to the old stack for data and pointers. A stack switch therefore requires assembly code to go patch the data for the new stack location, which may lead to compatibility issues. To avoid such possible compatibility issues introduced by different compilers and to ease the integration of FSP with a bootloader, the API uses the **ContinuationFunction** parameter to continue the bootloader execution flow rather than return as a normal C function.

ContinuationFunc is a function entry point that will be jumped to at the end of the FspInit API to transfer control back to the bootloader. FSP will setup the stack at **FSP_INIT_RT_COMMON_BUFFER.StackTop** when calling **ContinuationFunction.** The FSP needs to get some parameters from the bootloader when it is initializing the silicon. These parameters are passed from the bootloader to the FSP through the **FSP_INIT_RT_BUFFER** structure pointer. Refer to the related FSP *Integration Guide* for the detailed structure definitions.

The FSP produces a series of data structures, called HOB, as it initializes the silicon which provides information about the silicon configuration. This information is passed to the bootloader **ContinuationFunction** through the HobListPtr. More details are provided in Section 7, FSP Output in this document.

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 **NvsBufferPtr**) when it is requesting the FSP to initialize the silicon under these special circumstances. Refer to Section 7.3 *FSP_NON_VOLATILE_STORAGE_HOB* for the details on how to get the returned NVS data from FSP.

6.7 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 initialization phases. The phases will be platform dependent and will be documented with the FSP release. The current FSP supports two notify phases:

- Post PCI enumeration
- Ready To Boot

6.7.1 Prototype

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

6.7.2 Parameters

NotifyPhaseParamPtr	Address pointer to the **NOTIFY_PHASE_PARAMS**

6.7.3 Related Definitions

```
typedef enum {
  EnumInitPhaseAfterPciEnumeration = 0x20,
  EnumInitPhaseReadyToBoot         = 0x40
} 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. FSP will use it to do some specific initialization for processor and chipset that requires PCI resource assignments to have been completed.

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

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

`EnumInitPhaseReadyToBoot`

This stage is notified just before the bootloader hand-off to the OS loader. FSP will use it to do some specific initialization for processor and chipset that is required before control is transferred to the OS.

On the S3 resume path this API must be called after EnumInitPhaseAfterPciEnumeration notification and before the bootloader hand-off to the OS resume vector.

6.7.4 Return Values

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

Table 6. 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.

6.7.5 Description

The FSP will lock the configuration registers to enhance security as required by the BWG / BIOS Specification when it is notified that the bootloader is ready to transfer control to the operating system.

Therefore, this API should only be called after the FspInit or FspSiliconInit API and each notification code should be called only once in the predefined order. For example, the **`EnumInitPhaseAfterPciEnumeration`** notification needs to be called before the **`EnumInitPhaseReadyToBoot`** notification. Once the **`EnumInitPhaseReadyToBoot`** is notified, the whole FSP flow is considered to be completed and the results of any further FSP API calls are undefined.

6.8 FspMemoryInit API

This FSP API is called after TempRamInit and initializes the memory. This FSP API accepts a pointer to a data structure that will be platform-dependent and defined for each FSP binary. This will be documented in *Integration Guide* with each FSP release.

FspMemoryInit API initializes the memory subsystem, initializes the pointer to the HobListPtr, and returns to the bootloader from where it was called. Since the system memory has been initialized in this API, the bootloader must migrate it's stack and data from temporary memory to system memory after this API.

FspMemoryInit, TempRamExit and FspSiliconInit API provide an alternate method to complete the silicon initialization and provides bootloader an opportunity to get control after system memory is available and before the temporary memory is torn down.

This API must not be called if FspInit API has been called.

6.8.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_MEMORY_INIT) (
  IN OUT  FSP_MEMORY_INIT_PARAMS   *FspMemoryInitParamPtr
);
```

6.8.2 Parameters

FspMemoryInitParamPtr	Address pointer to the **FSP_MEMORY_INIT_PARAMS** structure.

6.8.3 Related Definitions

```
typedef struct {
  VOID                 *NvsBufferPtr;
  VOID                 *RtBufferPtr;
  VOID                 **HobListPtr;
} FSP_MEMORY_INIT_PARAMS;
```

NvsBufferPtr	Pointer to the non-volatile storage (NVS) data buffer. If it is **NULL** it indicates the NVS data is not available.
RtBufferPtr	Pointer to the runtime data buffer **FSP_INIT_RT_BUFFER**. This buffer contains various FSP configuration data that will be used during the platform initialization. The detailed structure layout is described in 6.6.3.1.
HobListPtr	Pointer to receive the address of the HOB list as defined in the 10, Appendix A – Data Structures

6.8.4 Return Values

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

Table 7. 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.

6.8.5 Description

FspMemoryInit API will use the **FSP_INIT_RT_COMMON_BUFFER** structure as its **RtBufferPtr** parameter, but field **FSP_INIT_RT_COMMON_BUFFER.StackTop** will not be used and must be initialized to 0.

The FSP needs to get some parameters from the bootloader when it is initializing the silicon. These parameters are passed from the bootloader to the FSP through the **FSP_INIT_RT_COMMON_BUFFER** structure pointer.

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 **NvsBufferPtr**) when it is requesting the FSP to initialize the silicon under these special circumstances. Refer to section 7.3 **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 after the TempRamInit API. This API will produce a HOB list and update the **HobListPtr** 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 complete HOB 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. This API must NOT tear down the temporary memory. Temporary memory setup by TempRamInit API will be torn down by TempRamExit API. It is the responsibility of the bootloader to

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

6.9 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 structure that will be platform dependent and defined for each FSP binary. This will be documented in the *Integration Guide*.

FspMemoryInit, TempRamExit and FspSiliconInit API provide an alternate method to complete the silicon initialization and provides bootloader an opportunity to get control after system memory is available and before the temporary memory is torn down.

This API must not be called if FspInit API has been called.

6.9.1 Prototype

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

6.9.2 Parameters

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

6.9.3 Return Values

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

Table 8. 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.

6.9.4 Description

This API should be called only once after the FspMemoryInit 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 data that it might have had 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 and enable caching.

FspMemoryInit, TempRamExit and FspSiliconInit API provide an alternate method to complete the silicon initialization and provides bootloader an opportunity to get control after system memory is available and before the temporary memory is torn down.

6.10 FspSiliconInit API

This FSP API is called after TempRamExit API. FspMemoryInit, TempRamExit and FspSiliconInit API provide an alternate method to complete the silicon initialization.

This API must not be called if FspInit API has been called.

6.10.1 Prototype

```
typedef
EFI_STATUS
(EFIAPI *FSP_SILICON_INIT) (
  IN OUT VOID          *FspSiliconInitParamPtr
);
```

6.10.2 Parameters

FspSiliconInitParamPtr	Pointer to the Silicon Init parameters structure. This structure is normally defined in the *Integration Guide*. If it is not defined in the *Integration Guide*, pass **NULL**.

6.10.3 Return Values

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

Table 9. 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.

6.10.4 Description

This API should be called only once after the TempRamExit API.

This FSP API initializes the processor and the chipset including the IO controllers in the chipset to enable normal operation of these devices. This FSP API accepts a pointer to a data structure that will be platform dependent and defined 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.

§

7 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 Related Documents. 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 HOB and 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 FspInit and FspMemoryInit API.

In addition to the *PI Specification* defined HOB, the FSP produces a number of FSP architecturally defined GUID type HOB. The sections below describes the GUID and the structure of these FSP defined HOB.

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

7.1 FSP_BOOTLOADER_TEMP_MEMORY_HOB

As described in the FspInit API, the system memory is initialized and the whole temporary memory is destroyed during this API call. However, the subregion of the temporary memory returned in the TempRamInit API may still contain bootloader specific data which might be useful to the bootloader after the FspInit call.

Before destroying the temporary memory, the FSP copies the subregion into a HOB in permanent memory and adds that to the HOB list. The bootloader can use this HOB to access the data saved in the temporary memory after FspInit API if necessary. If the bootloader does not care about the previous data, this HOB can be ignored.

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

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

This HOB is only published and applicable when using FspInit API.

7.2 FSP_RESERVED_MEMORY_RESOURCE_HOB

The FSP 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 reported as 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 FspInit or FspMemoryInit API.

7.3 FSP_NON_VOLATILE_STORAGE_HOB

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 it can be reused in special cases, such as S3 resume.

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 returning from the FspInit or FspMemoryInit API. 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 following boot flow the bootloader should load the data block back from the NVS device to temporary memory and populate the buffer pointer into **FSP_INIT_PARAMS.NvsBufferPtr** or **FSP_MEMORY_INIT_PARAMS.NvsBufferPtr** field before calling into the FspInit or FspMemoryInit API, respectively. If the NVS device is memory mapped, the bootloader can initialize the buffer pointer directly to the buffer.

This HOB must be parsed after FspInit or FspMemoryInit API.

7.4 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 **FSP_INIT_RT_COMMON_BUFFER.BootLoaderTolumSize.** The FSP reserved memory region will be placed below this region.

This HOB will only be published when the **FSP_INIT_RT_COMMON_BUFFER.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 FspInit or FspMemoryInit API.

7.5 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 the graphics system, FSP may need some platform specific configuration data which would be documented in the *Integration Guide*.

When graphics capability is included in FSP and enabled as documented in *Integration Guide*, FSP produces a **EFI_PEI_GRAPHICS_INFO_HOB** as described in the *PI Specification* as referenced in 1.3, 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 of the bootloader to update its internal data structures with the new framebuffer address after the enumeration is complete.

This HOB is valid after FspInit or FspSiliconInit API. This HOB is not produced in S3 boot path i.e., when `FSP_INIT_RT_COMMON_BUFFER.BootMode` is set to BOOT_ON_S3_RESUME.

8 FSP Configuration Firmware File

The FSP binary contains a configurable data region which will be used by the FSP during initialization.

The configurable data region has two sets of data:

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

Both the VPD and the UPD parameters can be statically customized using a separate tool. There will be a Boot Setting File (BSF) provided along with FSP binary to describe the configuration options within the FSP.

In addition to static configuration, the UPD data can be overridden by the bootloader during runtime. The UPD data is organized as a structure. The FspInit() and FspMemoryInit() API parameter includes an **FSP_INIT_RT_COMMON_BUFFER.UpdDataRgnPtr** pointer which can be initialized to point to the UPD data structure. If this pointer is initialized to NULL when calling the FspInit() or FspMemoryInit() API, the FSP will use the default built-in UPD configuration data in the FSP binary. However, if the bootloader wishes to override any of the UPD parameters, it has to copy the whole UPD structure from flash to memory, override the parameters and initialize the **FSP_INIT_RT_COMMON_BUFFER.UpdDataRgnPtr** pointer to the address of the UPD structure with updated data in memory and call FspInit() or FspMemoryInit() API. The FSP will use this data structure instead of the default configuration region data for platform initialization. The UPD data structure pointed by pointer **FSP_INIT_RT_COMMON_BUFFER.UpdDataRgnPtr** is a project specific structure. Please refer to 8.2 and the *Integration Guide* for the details of this structure.

Both the VPD and the UPD structure definitions will be provided as part of the FSP distribution package. To update these configuration options statically using the BCT, a BSF file will be required. This file contains the detailed information on all configurable options, including description, help information, valid value range and the default value. The BSF file will also be provided with the FSP distribution package.

8.1 VPD Standard Fields

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

Table 10. VPD Standard Fields

Offset	Field
0x00 – 0x07	VPD Region Signature. FSP specific signature described in the *Integration Guide*. This field is used by BCT to verify if .bsf is valid for FSP binary. If the HeaderRevision field in the FSP_INFO_HEADER is > 1, then this signature should match the 8 byte Image Id in the FSP_INFO_HEADER.
0x08 – 0x0B	Image Revision. Should match the revision in the FSP_INFO_HEADER
0x0C – 0x0F	UPD Region offset
0x10 – 0x13	UPD Region size
0x14 – 0x1F	Reserved

8.2 UPD Standard Fields

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

Table 11. UPD Standard Fields

Offset	Field
0x00 – 0x07	UPD Region Signature.FSP specific signature described in the *Integration Guide*. This field is used by BCT to verify if .BSF is valid for FSP binary.
0x08	Revision
0x09 – 0x0F	ReservedUpd0[7]
0x10 – 0x13	MemoryInitUpdOffset
0x14 – 0x17	SiliconInitUpdOffset
0x18 – 0x1F	ReservedUpd1

§

9 Other Host BootLoader Considerations

9.1 Power Management

FSP does not provide power management functions besides making power management features available to the host bootloader. ACPI is an independent component of the bootloader, and it will not be included in the FSP.

9.2 Bus Enumeration

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

9.3 Security

FSP will follow the BWG / BIOS Specification to set the necessary registers for security concerns. However, some security features, such as secure boot, are not necessarily covered by the FSP.

Examples include, but are not limited to, SMM, discrete TPM, measured boot, verified, and authenticated boot.

§

10 Appendix A – Data Structures

The declarations/definitions provided here were derived from the EDK2 source available for download at https://github.com/tianocore/edk2. The GitHub links point to the latest version of the files and may be newer than the version seen in this document.

10.1 BOOT_MODE

10.1.1 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
```

10.2 EFI_STATUS

10.2.1 UefiBaseType.h

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

```
#define EFI_SUCCESS                         0x00000000
#define EFI_INVALID_PARAMETER               0x80000002
#define EFI_UNSUPPORTED                     0x80000003
#define EFI_NOT_READY                       0x80000006
#define EFI_DEVICE_ERROR                    0x80000007
#define EFI_OUT_OF_RESOURCES                0x80000009
#define EFI_VOLUME_CORRUPTED                0x8000000A
#define EFI_NOT_FOUND                       0x8000000E
#define EFI_TIMEOUT                         0x80000012
#define EFI_ABORTED                         0x80000015
#define EFI_INCOMPATIBLE_VERSION            0x80000019
#define EFI_SECURITY_VIOLATION              0x8000001A
#define EFI_CRC_ERROR                       0x8000001B

typedef UINT64                   EFI_PHYSICAL_ADDRESS;
```

10.3 EFI_PEI_GRAPHICS_INFO_HOB

10.3.1 GraphicsInfoHob.h

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

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

10.4 EFI_GUID

10.4.1 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;
```

10.4.2 UefiBaseType.h

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

```
typedef GUID                         EFI_GUID;
```

10.5 EFI_MEMORY_TYPE

10.5.1 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,
  EfiMaxMemoryType
} EFI_MEMORY_TYPE;
```

10.6 Hand Off Block (HOB)

10.6.1 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
```

```
//
// 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;
```

10.7 Firmware Volume and Firmware Filesystem

Please refer to PiFirmwareVolume.h and PiFirmwareFile.h from EDK2 project for original source.

10.7.1 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;
```

10.7.2 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)
```

11 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
VPD	Vital Product Data

§