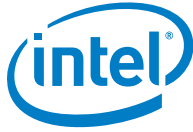


Intel[®] Firmware Support Package

External Architecture Specification v2.0

May 2016



You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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

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.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting: <http://www.intel.com/design/literature.htm>.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at <http://www.intel.com/> or from the OEM or retailer.

No computer system can be absolutely secure.

Intel include any Intel trademarks which are used in this document] and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

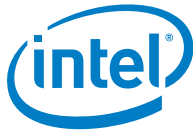
*Other names and brands may be claimed as the property of others.

Copyright © 2016, Intel Corporation. All rights reserved.



Contents

1	Introduction	7
	1.1 Purpose	7
	1.2 Intended Audience	7
	1.3 Related Documents	7
2	FSP Overview	8
	2.1 Design Philosophy	8
	2.2 Technical Overview	8
	2.2.1 Data Structure Descriptions	8
3	FSP Integration	9
	3.1 FSP Distribution Package	9
4	FSP Binary Format	10
	4.1.1 FSP-T: Temp RAM initialization phase.....	10
	4.1.2 FSP-M: Memory initialization phase.....	10
	4.1.3 FSP-S: Silicon initialization phase	10
	4.1.4 OEM Components (FSP-O)	11
	4.2 FSP Component Identification	11
	4.2.1 FSP Image ID and Revision	11
	4.2.2 FSP Component Layout	11
5	FSP Information tables	13
	5.1.1 FSP_INFO_HEADER.....	13
	5.1.2 FSP_INFO_EXTENDED_HEADER.....	15
	5.1.3 Locating FSP_INFO_HEADER.....	16
	5.1.4 FSP Description File	18
	5.1.5 FSP Patch Table (FSPP)	18
	5.1.5.1 Example	19
6	FSP Configuration Data.....	20
	6.1 UPD Standard Fields.....	21
	6.1.1 FSP-T UPD Structure	21
	6.1.2 FSP-M UPD Structure	22
	6.1.3 FSP-S UPD Structure.....	23
7	Boot Flow	24
	7.1.1 Bootflow Description	24
8	FSP Interface (FSP API)	26
	8.1 Entry-Point Invocation Environment	26
	8.2 Data Structure Convention	26
	8.3 Entry-Point Calling Convention	26
	8.4 Return Status Code	26
	8.5 TempRamInit API.....	28
	8.5.1 Prototype	28
	8.5.2 Parameters	28
	8.5.3 Return Values	29
	8.5.4 Description	29
	8.6 FspMemoryInit API.....	30



- 8.6.1 Prototype 30
- 8.6.2 Parameters 30
- 8.6.3 Return Values 30
- 8.6.4 Description 31
- 8.7 TempRamExit API 32
 - 8.7.1 Prototype 32
 - 8.7.2 Parameters 32
 - 8.7.3 Return Values 32
 - 8.7.4 Description 33
- 8.8 FspSiliconInit API 33
 - 8.8.1 Prototype 33
 - 8.8.2 Parameters 33
 - 8.8.3 Return Values 34
 - 8.8.4 Description 34
- 8.9 NotifyPhase API 34
 - 8.9.1 Prototype 35
 - 8.9.2 Parameters 35
 - 8.9.3 Related Definitions 35
 - 8.9.4 Return Values 36
 - 8.9.5 Description 36
- 9 FSP Output 37
 - 9.1 FSP_RESERVED_MEMORY_RESOURCE_HOB 38
 - 9.2 FSP_NON_VOLATILE_STORAGE_HOB 38
 - 9.3 FSP_BOOTLOADER_TOLUM_HOB 39
 - 9.4 EFI_PEI_GRAPHICS_INFO_HOB 39
- 10 Other Host BootLoader Considerations 40
 - 10.1 ACPI 40
 - 10.2 Bus Enumeration 40
 - 10.3 Security 40
- 11 Appendix A – Data Structures 41
 - 11.1 BOOT_MODE 41
 - 11.1.1 PiBootMode.h 41
 - 11.2 EFI_STATUS 42
 - 11.2.1 UefiBaseType.h 42
 - 11.2.2 OEM Status Code 42
 - 11.3 EFI_PEI_GRAPHICS_INFO_HOB 44
 - 11.3.1 GraphicsInfoHob.h 44
 - 11.4 EFI_GUID 44
 - 11.4.1 Base.h 44
 - 11.4.2 UefiBaseType.h 44
 - 11.5 EFI_MEMORY_TYPE 45
 - 11.5.1 UefiMultiPhase.h 45
 - 11.6 Hand Off Block (HOB) 46
 - 11.6.1 PiHob.h 46
 - 11.7 Firmware Volume and Firmware Filesystem 49
 - 11.7.1 PiFirmwareVolume.h 49
 - 11.7.2 PiFirmwareFile.h 50
- 12 Appendix B – Acronyms 54

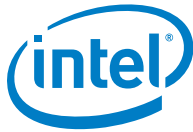


Figures

Figure 1: FSP Component Logical View	10
Figure 2: FSP Component Layout View	12
Figure 3: FSP Component Headers	17
Figure 4: Boot Flow.....	24

Tables

Table 1.	FSP_INFO_HEADER.....	13
Table 2.	FSP_INFO_EXTENDED_HEADER.....	15
Table 3.	FSPP – PatchData Encoding.....	18



Revision History

Date	Revision	Description
May 2016	001	<ul style="list-style-type: none">• FSP EAS v2.0• Based on FSP EAS v1.1a – Removed compatibility with v1.x• Updated FSP Binary format with FSP component information, layout, parsing and identification• FSP_INFO_HEADER changes<ul style="list-style-type: none">◦ Updated <i>HeaderRevision</i> from 2 to 3◦ Reduced <i>ImageAttribute</i> field from 4 to 2 bytes◦ Defined new <i>ComponentAttribute</i> field and defined <i>ComponentType</i> (Bits15:12)◦ Defined Bit0 and Bit1 in <i>ComponentAttribute</i> for Debug/Release & Test/Official respectively◦ Renamed <i>Reserved</i> to <i>Reserved1</i>◦ Renamed <i>ApiEntryNum</i> to <i>Reserved2</i>◦ Renamed <i>FspInitEntryOffset</i> to <i>Reserved3</i>◦ Added <i>SpecVersion</i> at offset 11• Removed VPD configuration data and updated UPD configuration data & UPD common header structure• Added Reset Request status return types• Updated API sections to clarify optional API and calling order of API• Updated the input parameters of <i>TempRamInit()</i>, <i>FspMemoryInit()</i>, <i>TempRamExit()</i>, <i>FspSiliconInit()</i> and <i>NotifyPhase()</i> APIs• <i>TempRamInit()</i><ul style="list-style-type: none">◦ Stack usage/stack allocation to bootloader clarified◦ Calling convention exception clarified◦ Removed parameter structure/description.◦ Updated API parameters to use FSPT_UPD• <i>FspMemoryInit()</i><ul style="list-style-type: none">◦ Simplified the API and remove the parameter structures◦ Minor clarification related to stack base and size and cleanup◦ Defined Arch UPDs for FSP-M component FSPM_ARCH_UPD• <i>TempRamExit()</i> - Updated API parameters• <i>NotifyPhase()</i> - Added <i>EndOfFirmware</i> phase• Clarified NVS HOB Fast Boot / S3 path• Updated BootFlow diagram and added description

§



1 Introduction

1.1 Purpose

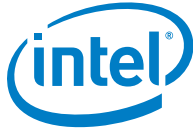
The purpose of this document is to describe the external architecture and interfaces provided in the Intel® Firmware Support Package (FSP). Implementation specific details are outside the scope of this document. Refer to *Integration Guide* for details.

1.2 Intended Audience

This document is targeted at all platform and system developers who need to generate or 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® FSP EAS version 1.1a
<http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/fsp-architecture-spec-v1-1a.pdf>
- Boot Specification File (BSF) Specification
https://firmware.intel.com/sites/default/files/BSF_1_0.pdf
- Unified Extensible Firmware Interface (UEFI) Specification
<http://www.uefi.org/specifications>
- Platform Initialization (PI) Specification v1.4
http://www.uefi.org/sites/default/files/resources/PI_1_4.zip
- Binary Configuration Tool (BCT) for Intel® Firmware Support Package - available at
<http://www.intel.com/fsp>



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

4 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).

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: Temp 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 APIs

- *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 APIs

- *FspMemoryInit()*
- *TempRamExit()*

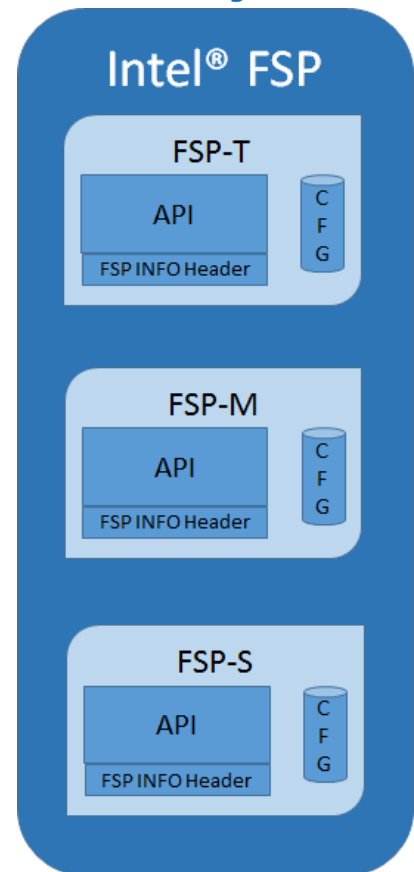
4.1.3 FSP-S: Silicon initialization phase

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

This phase consists of below FSP APIs

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

Figure 1: FSP Component Logical View





4.1.4 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 attribute field 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 first Firmware Volume (FV). The **FSP_INFO_HEADER** will have an attribute field that can be used to identify that component as an FSP-T/FSP-M/FSP-S/FSP-O component.

There can be only one instance of the FSP-T / FSP-M / FSP-S 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 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.

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

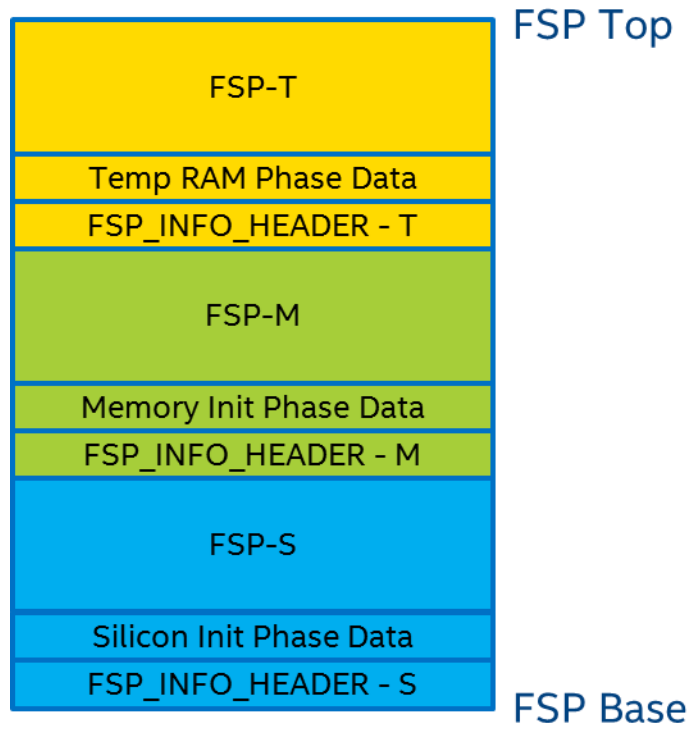
4.2.2 FSP Component Layout

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

Further more, if there are multiple Firmware Volume(s) inside the FSP component, they are also packaged back to back. Also, these components can be packaged in any order inside FSP binary.



Figure 2: FSP Component Layout View





5 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 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.

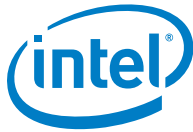
A generic table search algorithm for additional tables can be implemented with a signature search algorithm until a terminator signature 'FSPP' is found.

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	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 v2.0 the value will be 0x20
11	1	HeaderRevision	Revision of the header. The current value for this field is 3.
12	4	ImageRevision	Revision of the FSP binary. Major.Minor.Revision.Build The ImageRevision can be decoded as follows: 7 : 0 - Build Number 15 : 8 - Revision 23 : 16 - Minor Version 31 : 24 - 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 this component in bytes.



Byte Offset	Size in Bytes	Field	Description
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. <ul style="list-style-type: none"> • Bit 0: Graphics Support – Set to 1 when FSP supports enabling Graphics Display. • Bits 15:1 - Reserved
34	2	ComponentAttribute	Attributes of the FSP Component <ul style="list-style-type: none"> • Bit 0 – Build Type <ul style="list-style-type: none"> 0 – Debug Build 1 - Release Build • Bit 1 – Release Type <ul style="list-style-type: none"> 0 - Test Release 1 - Official Release • Bit 11:2 - Reserved • Bits 15:12 – Component Type <ul style="list-style-type: none"> 0000 – Reserved 0001 – FSP-T 0010 – FSP-M 0011 – FSP-S 0100 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 Section 6 for details.
40	4	CfgRegionSize	Size of the UPD configuration region. Please refer Section 6 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.



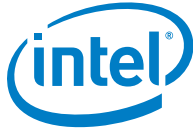
Byte Offset	Size in Bytes	Field	Description
60	4	FspMemoryInitEntryOffset	Offset for the API to initialize the Memory. If the value is set to 0x00000000, then this API is not available in this component.
64	4	TempRamExitEntryOffset	Offset for the API to tear down the temporary memory. If the value is 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. If the value is set to 0x00000000, then this API is not available in this component.

5.1.2 FSP_INFO_EXTENDED_HEADER

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



5.1.3 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 FSP component's first Firmware Volume (FV). 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 }};
```

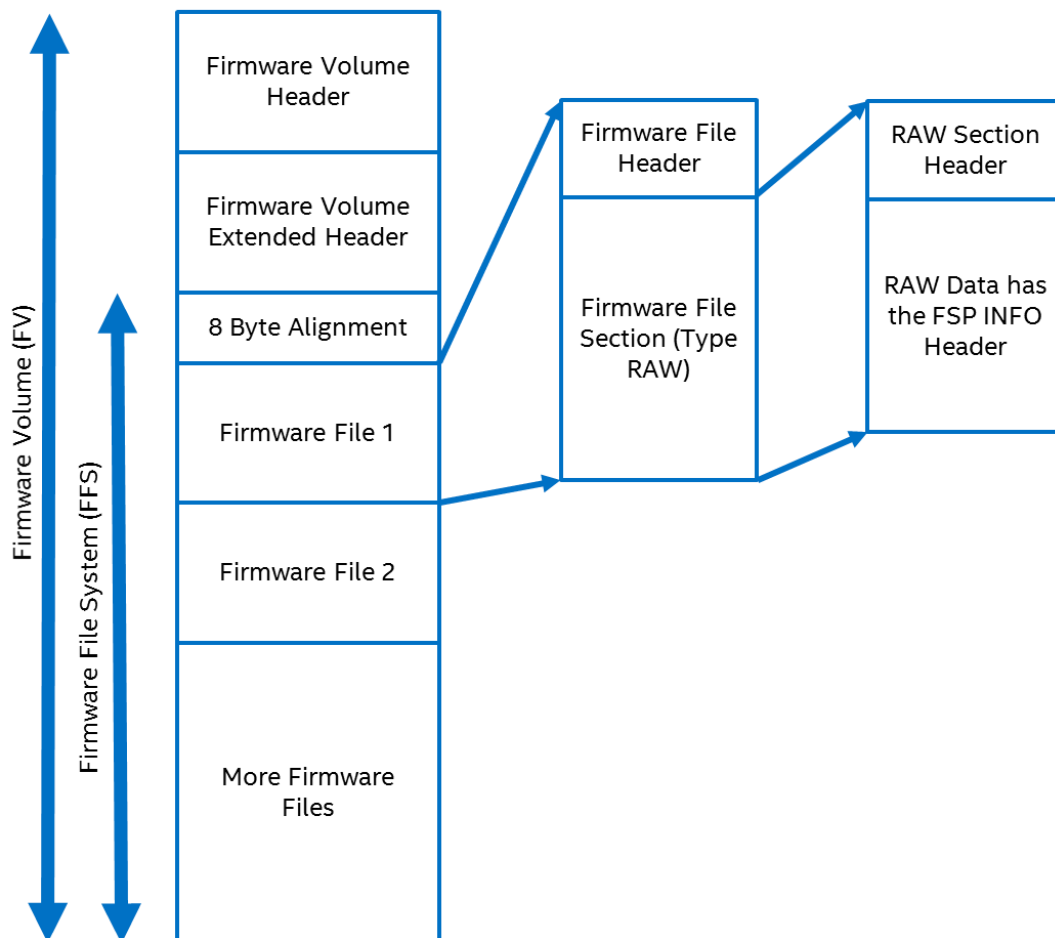



The bootloader can find the offset of the **FSP_INFO_HEADER** within the FSP component's first Firmware Volume (FV) by the following steps described below:

- 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** 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 3: FSP Component Headers





5.1.4 FSP Description File

An FSP component 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.

A PatchEntryNum of 0 is valid and indicates that there are no entries in the patch table and 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]	<p>0: The FSP image offset to patch is determined by Bits[23:0]</p> <p>1: The FSP image offset to patch is calculated by (ImageSize - (0x1000000 - Bits[23:0]))</p> <p>If the FSP image offset to patch is greater than the ImageSize in the FSP_INFO_HEADER, then this patch entry should be ignored.</p>
----------	--

5.1.5.1 Example

Let's assume the FSP image size is 0x38000. And we need to rebase the FSP base from 0xFFFFC0000 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;
    {
        0xFFFFFFFFFC    ///< Patch FVBASE at end of FV
    }
};
```

Looking closer at the patch table entry:

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

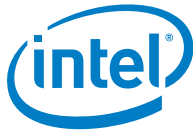
The image offset to patch in the FSP image is indicated by BIT[23:0], 0xFFFFFC. Since BIT[31] is 1, the actual FSP image offset to patch should be:

$$\text{ImageSize} - (0x1000000 - 0xFFFFFC) = 0x38000 - 4 = 0x37FFC$$

If the DWORD at offset 0x37FFC in the original FSP image is 0xFFFFC0000, then the new value should be:

$$\text{OldValue} + (\text{NewBase} - \text{OldBase}) = 0xFFFFC0000 + (0xFFF00000 - 0xFFFFC0000) = 0xFFF00000$$

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



6 FSP Configuration Data

Each FSP module contains a configurable data region which will 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 the FSP initialization.

The UPD parameters can be statically customized using a separate Binary Configuration Tool (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 in addition 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 API, 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 parameters and 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 Where XXXXXX is an 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 of the FSP release package and documented in the integration guide as well.

```
typedef struct {
    FSP_UPD_HEADER          UpdHeader;

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



6.1.2 FSP-M UPD Structure

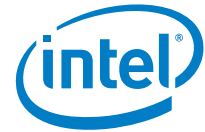
The UPD data structure definition for the FSP-M component will be provided as part of the FSP release package and documented in the integration guide as well.

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

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

typedef struct {
    UINT8                         Revision;
    UINT8                         Reserved[3];
    VOID                          *NvsBufferPtr;
    VOID                          *StackBase;
    UINT32                        StackSize;
    UINT32                        BootloaderTolumSize;
    UINT32                        BootMode;
    UINT8                         Reserved1[8];
} FSPM_ARCH_UPD;
```

Revision	Revision of the structure is 1 for this version of the specification.
NvsBufferPtr	Pointer to the non-volatile storage (NVS) data buffer. If it is NULL it indicates the NVS data is not available.
StackBase	Pointer to the temporary stack base address to be consumed inside <i>FspMemoryInit()</i> API.
StackSize	Temporary stack size to be consumed inside <i>FspMemoryInit()</i> API. Refer to <i>Integration Guide</i> for the minimum required stack size.
BootloaderTolumSize	Size of memory to be reserved by FSP below "top of low usable memory" for bootloader usage. Refer to Section 9.3 for more details.
BootMode	Current boot mode. Values are defined in <i>Section 11.1 Appendix A – Data Structures</i> . Refer to the <i>Integration Guide</i> for supported boot modes.



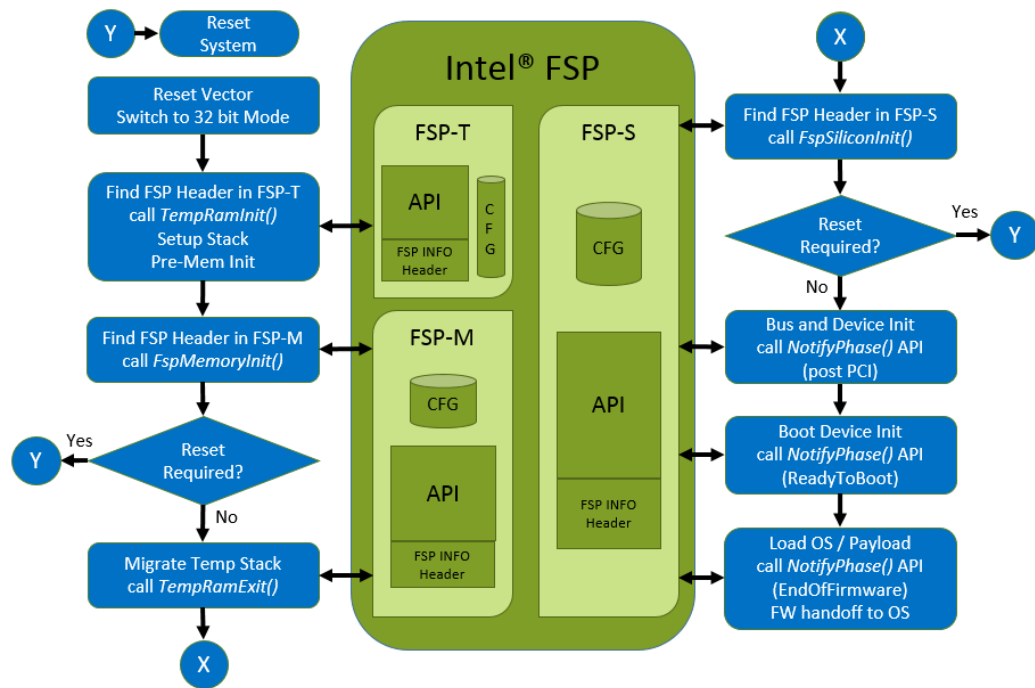
6.1.3 FSP-S UPD Structure

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

```
typedef struct {  
    FSP_UPD_HEADER          UpdHeader;  
  
    /**  
     * Platform specific parameters  
     **/  
    ...  
} FSPS_UPD;
```

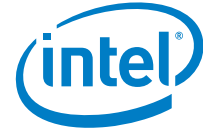
7 Boot Flow

Figure 4: Boot Flow



7.1.1 Bootflow Description

1. Bootloader starts executing from Reset Vector
 - a) Switches the mode to 32-bit mode
 - b) Initializes the early platform as needed
 - c) Finds FSP-T and calls *TempRamInit()* API. If Bootloader initializes the temporary memory this step and step2 can be skipped.
2. FSP initializes temporary memory and returns from *TempRamInit()* API
3. Bootloader initializes the stack in temporary memory
 - a) Initializes the platform as needed
 - b) Finds FSP-M and calls the *FspMemoryInit()* API



4. FSP initializes memory and returns from *FspMemoryInit()* API
5. Bootloader relocates itself to Memory
6. Bootloader calls *TempRamExit()* API. If Bootloader initialized the temporary memory, this step and next step can be skipped
7. FSP returns from *TempRamExit()* API
8. Bootloader finds FSP-S and calls *FspSiliconInit()* API
9. FSP returns from *FspSiliconInit()* API
10. Bootloader continues and device enumeration
11. Bootloader calls *NotifyPhase()* API with *AfterPciEnumeration* parameter
12. Bootloader calls *NotifyPhase()* API with *ReadyToBoot* parameter before transferring control to OS loader
13. When booting to non-UEFI OS, Bootloader calls *NotifyPhase()* API with *EndOfFirmware* parameter immediately after *ReadyToBoot*
14. When booting to UEFI OS, Bootloader calls *NotifyPhase()* with *EndOfFirmware* parameter during *ExitBootServices*

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



8 FSP Interface (FSP API)

8.1 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.

8.2 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.

8.3 Entry-Point Calling Convention

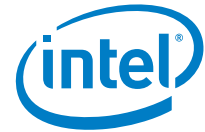
All FSP APIs defined in the **FSP_INFO_HEADER** are 32-bit only. The FSP API interface is similar to the default C `__cdecl` convention. Like the default C `__cdecl` convention, with the FSP API interface:

- 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*.

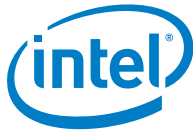
8.4 Return Status Code

All FSP API return a status code to indicate the API execution result. These return status codes are defined in Section 11.2.1 *Appendix A - EFI_STATUS*.

Sometimes for an initialization to take effect, a reset may be required. The FSP API may return a status code indicating that a reset is required as documented in 11.2.2 OEM Status code.



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



8.5 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 memory being initialized.

The *TempRamInit()* API should be called using the same entry point calling convention described in the previous section. However platform limitations like 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 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.

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. Refer to *Integration Guide* for other register usage.

Calling this API may be optional. Refer to the Integration Guide for any prerequisites before 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.

8.5.1 Prototype

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

8.5.2 Parameters

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



8.5.3 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.

Table 7. 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.

8.5.4 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 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.

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



8.6 FspMemoryInit API

This FSP API initializes the system memory. This FSP API accepts a pointer to a data structure 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 system memory has been initialized in this API, the bootloader must migrate its stack and data from temporary memory to system memory after this API.

8.6.1 Prototype

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

8.6.2 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 in the *Section 11.6 - Appendix A – Data Structures*

8.6.3 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 9. 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_REQUIREDx	A reset is required. These status codes will not be returned during S3.



8.6.4 Description

When *FspMemoryInit()* API is called, the FSP requires a stack available for its use. Before calling 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.

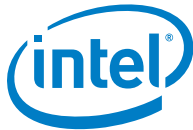
A set of parameters that the FSP may need to initialize memory under special circumstances, such as during an S3 resume or during fast boot mode, are returned by the FSP to the bootloader during a normal boot. The bootloader is expected to store these parameters in a non-volatile memory such as SPI flash and return a pointer to this structure through **FSPM_ARCH_UPD.NvsBufferPtr** when it is requesting the FSP to initialize the silicon under these special circumstances. Refer to section 9.2 **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 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. 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 return one of the **FSP_STATUS_RESET_REQUIRED** status as described in section 8.4. This API will not request a reset during S3 resume flow.



8.7 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.

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 directly calling *FspSiliconInit()* API.

8.7.1 Prototype

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

8.7.2 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**.

8.7.3 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 10. 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.



8.7.4 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 migrate any bootloader specific 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.

8.8 FspSiliconInit 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 API should be called only once after the system memory has been initialized, data from temporary memory migrated to system memory and cache configuration has been initialized

8.8.1 Prototype

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

8.8.2 Parameters

FspUpdDataPtr	Pointer to the FSPS_UPD data structure. If NULL, FSP will use the default parameters. Refer to the <i>Integration Guide</i> for structure definition.
----------------------	---



8.8.3 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 11. 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_REQUIREDx	A reset is required. These status codes will not be returned during S3.

8.8.4 Description

This API should be called only once after the *FspMemoryInit()* API (if the bootloader is not using *TempRamExit()* API) or the *TempRamExit()* 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*.

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 return an **FSP_STATUS_RESET_REQUIRED** as described in section 8.4. This API will not request a reset during S3 resume flow.

8.9 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 specification supports three notify phases:

- Post PCI enumeration
- Ready To Boot
- End Of Firmware



8.9.1 Prototype

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

8.9.2 Parameters

NotifyPhaseParamPtr Address pointer to the
NOTIFY_PHASE_PARAMS

8.9.3 Related Definitions

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

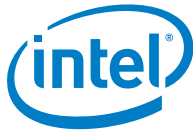
typedef struct {
    FSP_INIT_PHASE    Phase;
} NOTIFY_PHASE_PARAMS;
```

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.



8.9.4 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 8. 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.

8.9.5 Description

EnumInitPhaseAfterPciEnumeration

FSP will use this notification 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 S3 resume path this API must be called before the bootloader hand-off to the OS resume 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 before transferring 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.



9 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 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 *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*.



9.1 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 `FspMemoryInit()` API.

9.2 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 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 returning from the `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 `FSPM_ARCH_UPD.NvsBufferPtr` field before calling into the `FspMemoryInit()` API. If the NVS device is memory mapped, the bootloader can initialize the buffer pointer directly to the buffer.

This HOB must be parsed after `FspMemoryInit()` API.

This HOB is produced only when new NVS data is generated. For example, 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.



9.3 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**. The FSP reserved memory region will be placed below this region.

This HOB will only be published when the **FSPM_ARCH_UPD.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()* API.

9.4 EFI_PEI_GRAPHICS_INFO_HOB

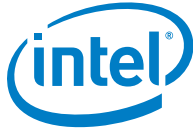
If BIT0 (Graphics Support) of the ImageAttribute field in the **FSP_INFO_HEADER** is set, the FSP includes graphics initialization capabilities. To complete the initialization of the graphics system, FSP may need some platform specific configuration data which would be documented in the *Integration 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 *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 of the bootloader to update its internal data structures with the new framebuffer address after the enumeration is complete.

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



10 Other Host BootLoader Considerations

10.1 ACPI

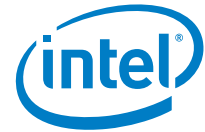
ACPI is an independent component of the bootloader, and it will not be included in the FSP.

10.2 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.

10.3 Security

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



11 Appendix A – Data Structures

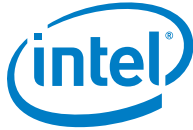
The declarations/definitions provided here were derived from the EDK2 source available for download at <https://github.com/tianocore/edk2>

11.1 BOOT_MODE

11.1.1 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
```



11.2 EFI_STATUS

11.2.1 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;
```

11.2.2 OEM Status Code

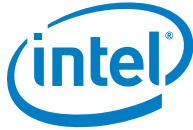
The range of status code that have the highest bit clear and the next to highest bit set are reserved for use by OEMs.

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

Appendix A – Data Structures



```
#define FSP_STATUS_RESET_REQUIRED_COLD           0x40000001
#define FSP_STATUS_RESET_REQUIRED_WARM          0x40000002
#define FSP_STATUS_RESET_REQUIRED_3            0x40000003
#define FSP_STATUS_RESET_REQUIRED_4            0x40000004
#define FSP_STATUS_RESET_REQUIRED_5            0x40000005
#define FSP_STATUS_RESET_REQUIRED_6            0x40000006
#define FSP_STATUS_RESET_REQUIRED_7            0x40000007
#define FSP_STATUS_RESET_REQUIRED_8            0x40000008
```



11.3 EFI_PEI_GRAPHICS_INFO_HOB

11.3.1 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;
```

11.4 EFI_GUID

11.4.1 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;
```

11.4.2 UefiBaseType.h

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

```
typedef GUID          EFI_GUID;
```



11.5 EFI_MEMORY_TYPE

11.5.1 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,  
    EfiPersistentMemory,  
    EfiMaxMemoryType  
} EFI_MEMORY_TYPE;
```



11.6 Hand Off Block (HOB)

11.6.1 PiHob.h

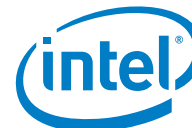
<https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Pi/PiHob.h>

```
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
#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               0xFFFFE
#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;
```




11.7 Firmware Volume and Firmware Filesystem

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

11.7.1 PiFirmwareVolume.h

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

```

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

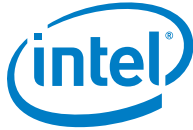
///
/// type of EFI FVB attribute
///
typedef UINT32  EFI_FVB_ATTRIBUTES_2;

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
    
```

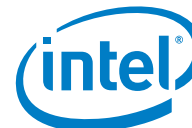


```
///  
/// Extension header pointed by ExtHeaderOffset of volume header.  
///  
typedef struct {  
    EFI_GUID    FvName;  
    UINT32      ExtHeaderSize;  
} EFI_FIRMWARE_VOLUME_EXT_HEADER;  
  
///  
/// Entry structure for describing FV extension header  
///  
typedef struct {  
    UINT16      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;
```

11.7.2 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;

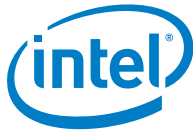
///
/// File Types Definitions
///
#define EFI_FV_FILETYPE_FREEFORM 0x02

///
/// 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

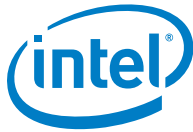
///
/// Common section header.
///
typedef struct {
    UINT8           Size[3];
    EFI_SECTION_TYPE Type;
} EFI_COMMON_SECTION_HEADER;

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)
```



12 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
FDL	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